

Operating Systems Lab Assignment: Thread-Safe Data Structures

Jonathan Ross

October 19, 2025

1 Introduction

This report documents the implementations and analyses for the thread-safe data structures lab assignment, covering thread-safe queue and stack implementations, Producer-Consumer and Undo-Redo use cases, and additional exercises.

2 Exercise 1: Thread-Safe Queue and Stack

```
#include <iostream>
#include <queue>
#include <stack>
#include <mutex>
#include <thread>
#include <vector>
#include <string>
#include <chrono>
#include <atomic>

// Thread-safe Queue class
template <typename T>
class ThreadSafeQueue {
private:
    std::queue<T> queue;
    std::mutex mtx;

public:
    ThreadSafeQueue() {}

    void push(T value) {
        std::lock_guard<std::mutex> lock(mtx);
        queue.push(value);
    }

    bool pop(T& value) {
        std::lock_guard<std::mutex> lock(mtx);
        if (queue.empty()) {
            return false;
        }
        value = queue.front();
        queue.pop();
        return true;
    }

    bool empty() {
        std::lock_guard<std::mutex> lock(mtx);
        return queue.empty();
    }
}
```

```

        size_t size() {
            std::lock_guard<std::mutex> lock(mtx);
            return queue.size();
        }
};

// Thread-safe Stack class
template <typename T>
class ThreadSafeStack {
private:
    std::stack<T> stack;
    std::mutex mtx;

public:
    ThreadSafeStack() {}

    void push(T value) {
        std::lock_guard<std::mutex> lock(mtx);
        stack.push(value);
    }

    bool pop(T& value) {
        std::lock_guard<std::mutex> lock(mtx);
        if (stack.empty()) {
            return false;
        }
        value = stack.top();
        stack.pop();
        return true;
    }

    bool empty() {
        std::lock_guard<std::mutex> lock(mtx);
        return stack.empty();
    }

    size_t size() {
        std::lock_guard<std::mutex> lock(mtx);
        return stack.size();
    }
};

// Problem 1: Producer-Consumer Simulation
void producerConsumerProblem() {
    ThreadSafeQueue<std::string> messageQueue;
    const int NUM_PRODUCERS = 3;
    const int NUM_CONSUMERS = 2;
    const int MESSAGES_PER_PRODUCER = 5;
    std::atomic<int> messages_produced(0);
    std::vector<std::thread> producers;
    std::vector<std::thread> consumers;

    auto producer = [&](ThreadSafeQueue<std::string>& queue, std::atomic<int>&
        produced_count, int id) {
        for (int i = 0; i < MESSAGES_PER_PRODUCER; ++i) {
            std::string message = "Producer_" + std::to_string(id) + "_Message_"
                + std::to_string(i);
            queue.push(message);
            produced_count++;
            std::cout << "Produced:_" << message << std::endl;
            std::this_thread::sleep_for(std::chrono::milliseconds(100));
        }
    };

```

```

};

auto consumer = [&](ThreadSafeQueue<std::string>& queue, std::atomic<int>&
produced_count, int id) {
    int consumed = 0;
    while (consumed < NUM_PRODUCERS * MESSAGES_PER_PRODUCER) {
        std::string message;
        if (queue.pop(message)) {
            std::cout << "Consumer_" << id << "_processed:" << message <<
                std::endl;
            consumed++;
        } else if (produced_count == NUM_PRODUCERS * MESSAGES_PER_PRODUCER)
        {
            break;
        }
        std::this_thread::sleep_for(std::chrono::milliseconds(150));
    }
};

for (int i = 0; i < NUM_PRODUCERS; ++i) {
    producers.emplace_back(producer, std::ref(messageQueue), std::ref(
        messages_produced), i + 1);
}
for (int i = 0; i < NUM_CONSUMERS; ++i) {
    consumers.emplace_back(consumer, std::ref(messageQueue), std::ref(
        messages_produced), i + 1);
}

for (auto& t : producers) {
    t.join();
}
for (auto& t : consumers) {
    t.join();
}
}

// Problem 2: Undo-Redo System
void undoRedoProblem() {
    ThreadSafeStack<std::string> undoStack;
    ThreadSafeStack<std::string> redoStack;
    std::string currentText = "";

    auto clearRedoStack = [&]() {
        while (!redoStack.empty()) {
            std::string temp;
            redoStack.pop(temp);
        }
    };

    auto editText = [&](const std::string& newText) {
        undoStack.push(currentText);
        if (!redoStack.empty()) {
            std::string temp;
            while(redoStack.pop(temp));
        }

        currentText = newText;
        std::cout << "Text_updated_to:" << currentText << std::endl;
    };

    auto undo = [&]() {
        std::string prevText;
        if (undoStack.pop(prevText)) {

```

```

        redoStack.push(currentText);
        currentText = prevText;
        std::cout << "Undo-Current text:" << currentText << std::endl;
    }
};

auto redo = [&]() {
    std::string nextText;
    if (redoStack.pop(nextText)) {
        undoStack.push(currentText);
        currentText = nextText;
        std::cout << "Redo-Current text:" << currentText << std::endl;
    }
};

editText("Hello");
editText("HelloWorld");
editText("HelloUniverse");
undo();
redo();
editText("HelloGalaxy");
undo();
undo();
redo();
}

int main() {
    std::cout << "Problem1: Producer-Consumer Simulation\n";
    producerConsumerProblem();

    std::cout << "\nProblem2: Undo-Redo System\n";
    undoRedoProblem();

    return 0;
}

```

Explanation: In both classes, a lockguard locks the mutex in every method, creating a critical section that serializes access to the internal data structure. This locking prevents race conditions, enabling the ThreadSafeQueue to safely coordinate producer/consumer messages and the ThreadSafeStack to correctly manage undo/redo history.

Analysis: The mutex provides mutual exclusion, forcing any thread to block and wait if another thread is already accessing the data. Consumers terminate only after a pop fails and an atomic counter confirms all messages have been produced, guaranteeing no work is missed.

Screenshot: Include a screenshot of compiling and running exercise1.cpp.

```

● @JonRoss7 → /workspaces/Thread_Safe_DS (main) $ g++ exercise1.cpp -pthread -std=c++11 -o exercise1
● @JonRoss7 → /workspaces/Thread_Safe_DS (main) $ ./exercise1
Problem 1: Producer-Consumer Simulation
Produced: Producer 1 Message 0
Produced: Producer 2 Message 0
Produced: Producer 3 Message 0
Consumer 1 processed: Producer 1 Message 0
Consumer 2 processed: Producer 2 Message 0
Produced: Produced: Producer 1 Message 1Producer 2 Message 1

Produced: Producer 3 Message 1
Consumer 2 processed: Producer 3 Message 0
Consumer 1 processed: Producer 2 Message 1
Produced: Producer 1 Message 2
Produced: Producer 2 Message 2
Produced: Producer 3 Message 2
Produced: Producer 1 Message 3
Produced: Producer 2 Message 3
Produced: Producer 3 Message 3
Consumer 2 processed: Producer 1 Message 1
Consumer 1 processed: Producer 3 Message 1
Produced: Producer 2 Message 4
Produced: Producer 3 Message 4
Produced: Producer 1 Message 4
Consumer 2 processed: Producer 1 Message 2
Consumer 1 processed: Producer 2 Message 2
Consumer Consumer 12 processed: Producer 1 Message 3 processed: Producer 3 Message 2

Consumer 2 processed: Producer 3 Message 3Consumer 1 processed: Producer 2 Message 3

Consumer 1 processed: Producer 2 Message 4
Consumer 2 processed: Producer 3 Message 4
Consumer 1 processed: Producer 1 Message 4

Problem 2: Undo-Redo System
Text updated to: Hello
Text updated to: Hello World
Text updated to: Hello Universe
Undo - Current text: Hello World
Redo - Current text: Hello Universe
Text updated to: Hello Galaxy
Undo - Current text: Hello Universe
Undo - Current text: Hello World
Redo - Current text: Hello Universe
● @JonRoss7 → /workspaces/Thread_Safe_DS (main) $

```

Figure 1: Compilation and execution of exercise1.cpp

3 Exercise 3: Thread-Safe Priority Queue

```

#include <iostream>
#include <queue>
#include <mutex>
#include <thread>
#include <vector>
#include <chrono>
#include <cstdlib>

// Thread-safe Priority Queue class
template <typename T>
class ThreadSafePriorityQueue {
private:
    std::priority_queue<T> pq;
    std::mutex mtx;

public:
    void push(T value) {
        std::lock_guard<std::mutex> lock(mtx);
        pq.push(value);
    }

    bool pop(T& value) {
        std::lock_guard<std::mutex> lock(mtx);

```

```

        if (pq.empty()) return false;
        value = pq.top();
        pq.pop();
        return true;
    }

    bool empty() {
        std::lock_guard<std::mutex> lock(mtx);
        return pq.empty();
    }

    size_t size() {
        std::lock_guard<std::mutex> lock(mtx);
        return pq.size();
    }
};

void priorityQueueTest() {
    ThreadSafePriorityQueue<int> pq;
    std::vector<std::thread> threads;
    const int NUM_THREADS = 4;

    auto pusher = [&pq](int id) {
        for (int i = 0; i < 5; ++i) {
            int priority = rand() % 100;
            pq.push(priority);
            std::cout << "Thread_" << id << "_pushed:" << priority << "\n";
            std::this_thread::sleep_for(std::chrono::milliseconds(100));
        }
    };

    auto popper = [&pq]() {
        for (int i = 0; i < 20; ++i) {
            int value;
            if (pq.pop(value)) {
                std::cout << "Popped:" << value << "\n";
            }
            std::this_thread::sleep_for(std::chrono::milliseconds(150));
        }
    };

    for (int i = 0; i < NUM_THREADS; ++i) {
        threads.emplace_back(pusher, i);
    }
    threads.emplace_back(popper);

    for (auto& t : threads) {
        t.join();
    }
}

int main() {
    priorityQueueTest();
    return 0;
}

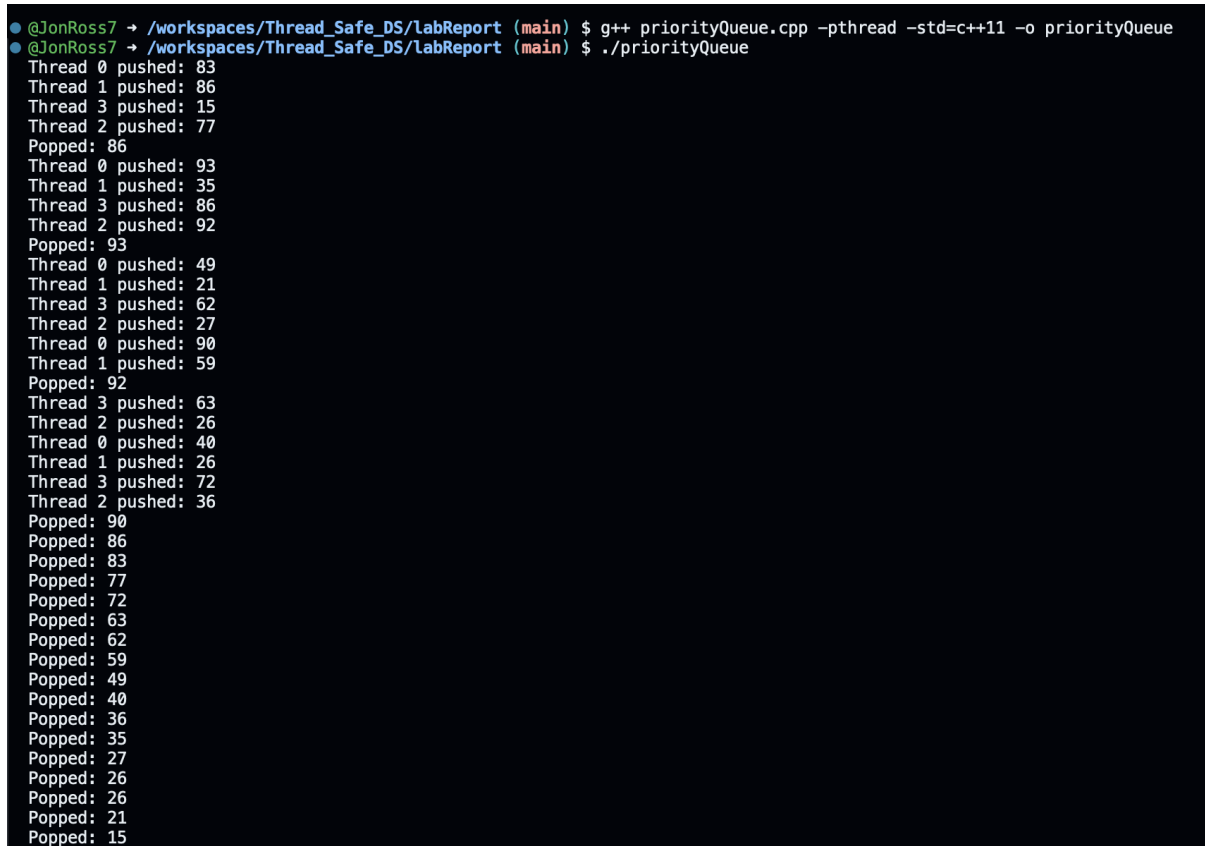
```

Explanation: The ThreadSafePriorityQueue ensures thread safety by using a lockguard to lock its mutex in every method, which serializes access and prevents race conditions. This locking protects the internal priority queue, allowing it to correctly maintain order so that pop operations always remove the highest-priority element, even when multiple threads are pushing and popping simultaneously.

Analysis: Concurrent push and pop operations are serialized by the mutex, meaning only one thread can modify the priority queue at any given time. While thread scheduling makes the order of

operations non-deterministic, the pop thread is always guaranteed to receive the highest-priority item that is currently in the queue at the exact moment it acquires the lock.

Screenshot: Include a screenshot of compiling and running `priorityQueue.cpp`.



```
@JonRoss7 → /workspaces/Thread_Safe_DS/LabReport (main) $ g++ priorityQueue.cpp -pthread -std=c++11 -o priorityQueue
@JonRoss7 → /workspaces/Thread_Safe_DS/LabReport (main) $ ./priorityQueue
Thread 0 pushed: 83
Thread 1 pushed: 86
Thread 3 pushed: 15
Thread 2 pushed: 77
Popped: 86
Thread 0 pushed: 93
Thread 1 pushed: 35
Thread 3 pushed: 86
Thread 2 pushed: 92
Popped: 93
Thread 0 pushed: 49
Thread 1 pushed: 21
Thread 3 pushed: 62
Thread 2 pushed: 27
Thread 0 pushed: 90
Thread 1 pushed: 59
Popped: 92
Thread 3 pushed: 63
Thread 2 pushed: 26
Thread 0 pushed: 40
Thread 1 pushed: 26
Thread 3 pushed: 72
Thread 2 pushed: 36
Popped: 90
Popped: 86
Popped: 83
Popped: 77
Popped: 72
Popped: 63
Popped: 62
Popped: 59
Popped: 49
Popped: 40
Popped: 36
Popped: 35
Popped: 27
Popped: 26
Popped: 26
Popped: 21
Popped: 15
```

Figure 2: Compilation and execution of `priorityQueue.cpp`

4 Exercise 4: Thread-Safe Circular Buffer

```
#include <iostream>
#include <mutex>
#include <condition_variable>
#include <thread>
#include <vector>
#include <chrono>
#include <cstdlib>

#define BUFFER_SIZE 5

class ThreadSafeCircularBuffer {
private:
    int buffer[BUFFER_SIZE];
    int in = 0, out = 0, count = 0;
    std::mutex mtx;
    std::condition_variable not_full, not_empty;

public:
    void push(int value) {
        std::unique_lock<std::mutex> lock(mtx);

        not_full.wait(lock, [this] { return count < BUFFER_SIZE; });
```

```

        buffer[in] = value;
        in = (in + 1) % BUFFER_SIZE;
        count++;

        lock.unlock();
        not_empty.notify_one();
    }

    void pop(int& value) {
        std::unique_lock<std::mutex> lock(mtx);

        not_empty.wait(lock, [this] { return count > 0; });

        value = buffer[out];
        out = (out + 1) % BUFFER_SIZE;
        count--;

        lock.unlock();
        not_full.notify_one();
    }

    bool empty() {
        std::lock_guard<std::mutex> lock(mtx);
        return count == 0;
    }

    bool full() {
        std::lock_guard<std::mutex> lock(mtx);
        return count == BUFFER_SIZE;
    }
};

void circularBufferTest() {
    ThreadSafeCircularBuffer cb;
    std::vector<std::thread> producers, consumers;

    const int NUM_PRODUCERS = 2;
    const int NUM_CONSUMERS = 2;
    const int ITEMS_PER_PRODUCER = 5;
    const int ITEMS_PER_CONSUMER = 5;

    auto producer = [&cb]() {
        for (int i = 0; i < ITEMS_PER_PRODUCER; ++i) {
            int value = rand() % 100;
            cb.push(value);
            std::cout << "Produced:␣" << value << "\n";
            std::this_thread::sleep_for(std::chrono::milliseconds(100));
        }
    };

    auto consumer = [&cb]() {
        for (int i = 0; i < ITEMS_PER_CONSUMER; ++i) {
            int value;
            // Updated to match the new void pop()
            cb.pop(value);
            std::cout << "Consumed:␣" << value << "\n";
            std::this_thread::sleep_for(std::chrono::milliseconds(150));
        }
    };
}

```



```

};

for (int i = 0; i < NUM_PRODUCERS; ++i) {
    producers.emplace_back(producer);
}

for (int i = 0; i < NUM_CONSUMERS; ++i) {
    consumers.emplace_back(consumer);
}

for (auto& t : producers) t.join();
for (auto& t : consumers) t.join();
}

int main() {
    circularBufferTest();
    return 0;
}

```

Explanation: Condition variables allow threads to efficiently block and wait for a specific condition instead of spinning in a loop. A thread waits on the condition, which atomically unlocks the mutex and puts the thread to sleep. Another thread then uses `notifyOne` to wake the waiting thread after it changes the condition.

Analysis: The `ThreadSafeQueue` from Exercise 1 was unbounded and used only a mutex, which forced consumers to "busy-wait" by repeatedly trying to pop from an empty queue. This `ThreadSafeCircularBuffer` is bounded and uses condition variables to make threads wait when the buffer is full or empty, which is far more efficient as it eliminates busy-waiting entirely.

Screenshot: Include a screenshot of compiling and running `circularBuffer.cpp`.



```

@JonRoss7 → /workspaces/Thread_Safe_DS/LabReport (main) $ g++ circularBuffer.cpp -pthread -std=c++11 -o circularBuffer
@JonRoss7 → /workspaces/Thread_Safe_DS/LabReport (main) $ ./circularBuffer
Produced: 83
Consumed: 86
Produced: Consumed: 83
86
Produced: 77
Produced: 15
Consumed: Consumed: 1577

Produced: 93
Produced: 35
Consumed: 93
Consumed: 35
Produced: 86
Produced: 92
Produced: 49
Produced: 21
Consumed: 86
Consumed: 92
Consumed: 49
Consumed: 21
@JonRoss7 → /workspaces/Thread_Safe_DS/LabReport (main) $

```

Figure 3: Compilation and execution of `circularBuffer.cpp`

5 Exercise 5: Thread-Safe Deque

```

#include <iostream>
#include <deque>
#include <mutex>
#include <thread>
#include <vector>
#include <chrono>
#include <string>

template <typename T>
class ThreadSafeDeque {

```

```

private:
    std::deque<T> deque;
    std::mutex mtx;

public:
    void push_front(T value) {
        std::lock_guard<std::mutex> lock(mtx);
        deque.push_front(value);
    }

    void push_back(T value) {
        std::lock_guard<std::mutex> lock(mtx);
        deque.push_back(value);
    }

    bool pop_front(T& value) {
        std::lock_guard<std::mutex> lock(mtx);
        if (deque.empty()) return false;
        value = deque.front();
        deque.pop_front();
        return true;
    }

    bool pop_back(T& value) {
        std::lock_guard<std::mutex> lock(mtx);
        if (deque.empty()) return false;
        value = deque.back();
        deque.pop_back();
        return true;
    }

    bool empty() {
        std::lock_guard<std::mutex> lock(mtx);
        return deque.empty();
    }

    size_t size() {
        std::lock_guard<std::mutex> lock(mtx);
        return deque.size();
    }
};

void dequeTest() {
    ThreadSafeDeque<int> dq;
    std::vector<std::thread> threads;

    auto push_front = [&dq](int id) {
        for (int i = 0; i < 5; ++i) {
            dq.push_front(id * 100 + i);
            std::cout << "Thread_" << id << " pushed front:_" << (id * 100 + i)
                << "\n";
            std::this_thread::sleep_for(std::chrono::milliseconds(100));
        }
    };

    auto push_back = [&dq](int id) {
        for (int i = 0; i < 5; ++i) {
            dq.push_back(id * 100 + i);
            std::cout << "Thread_" << id << " pushed back:_" << (id * 100 + i)
                << "\n";
            std::this_thread::sleep_for(std::chrono::milliseconds(100));
        }
    };
};

```

```

auto pop_front = [&dq]() {
    for (int i = 0; i < 10; ++i) {
        int value;
        if (dq.pop_front(value)) {
            std::cout << "Popped front: " << value << "\n";
        }
        std::this_thread::sleep_for(std::chrono::milliseconds(150));
    }
};

auto pop_back = [&dq]() {
    for (int i = 0; i < 10; ++i) {
        int value;
        if (dq.pop_back(value)) {
            std::cout << "Popped back: " << value << "\n";
        }
        std::this_thread::sleep_for(std::chrono::milliseconds(150));
    }
};

threads.emplace_back(push_front, 1);
threads.emplace_back(push_back, 2);
threads.emplace_back(pop_front);
threads.emplace_back(pop_back);

for (auto& t : threads) t.join();
}

int main() {
    dequeTest();
    return 0;
}

```

Explanation: The main challenge is that operations on opposite ends don't inherently conflict, but operations on the same end or on an empty deque do create race conditions. This implementation uses a single, coarse-grained mutex that locks the entire deque for every operation, which solves the problem simply but creates a performance bottleneck by serializing all access, even non-conflicting ones.

Analysis: Concurrent operations on the deque are fully serialized because the single mutex locks the entire data structure for every method. This "coarse-grained" lock effectively prevents all race conditions, but it also creates a performance bottleneck by forcing even non-conflicting operations to wait for each other instead of running in parallel.

Screenshot: Include a screenshot of compiling and running threadSafeDeque.cpp.

```

@JonRoss7 → /workspaces/Thread_Safe_DS/LabReport (main) $ g++ threadSafeDeque.cpp -pthread -std=c++11 -o threadSafeDeque
@JonRoss7 → /workspaces/Thread_Safe_DS/LabReport (main) $ ./threadSafeDeque
Thread 1 pushed front: 100
Thread 2 pushed back: 200
Popped front: 100
Popped back: 200
Thread 2 pushed back: 201
Thread 1 pushed front: 101
Popped front: 101
Popped back: 201
Thread Thread 12 pushed back: 202 pushed front: 102
Popped back: Thread 202
1 pushed front: 103
Thread 2 pushed back: 203
Popped front: 102
Thread 1 pushed front: 104
Thread 2 pushed back: 204
Popped back: 204
Popped front: 104
Popped back: 203
Popped front: 103
@JonRoss7 → /workspaces/Thread_Safe_DS/LabReport (main) $

```

Figure 4: Compilation and execution of threadSafeDeque.cpp

6 Exercise 6: Thread-Safe Linked List

```
#include <iostream>
#include <mutex>
#include <thread>
#include <vector>
#include <chrono>

template <typename T>
class ThreadSafeLinkedList {
private:
    struct Node {
        T data;
        Node* next;
        Node(T value) : data(value), next(nullptr) {}
    };
    Node* head = nullptr;
    std::mutex mtx;

public:
    ~ThreadSafeLinkedList() {
        while (!empty()) {
            T value;
            pop_front(value);
        }
    }

    void push_front(T value) {
        std::lock_guard<std::mutex> lock(mtx);
        Node* newNode = new Node(value);
        newNode->next = head;
        head = newNode;
    }

    bool pop_front(T& value) {
        std::lock_guard<std::mutex> lock(mtx);
        if (!head) return false;
        Node* temp = head;
        value = temp->data;
        head = head->next;
        delete temp;
        return true;
    }

    bool empty() {
        std::lock_guard<std::mutex> lock(mtx);
        return head == nullptr;
    }

    size_t size() {
        std::lock_guard<std::mutex> lock(mtx);
        size_t count = 0;
        Node* current = head;
        while (current) {
            count++;
            current = current->next;
        }
        return count;
    }
};

void linkedListTest() {
    ThreadSafeLinkedList<int> list;
```

```

std::vector<std::thread> threads;

auto pusher = [&list](int id) {
    for (int i = 0; i < 5; ++i) {
        list.push_front(id * 100 + i);
        std::cout << "Thread_" << id << "_pushed:_" << (id * 100 + i) << "\n";
        std::this_thread::sleep_for(std::chrono::milliseconds(100));
    }
};

auto popper = [&list]() {
    for (int i = 0; i < 10; ++i) {
        int value;
        if (list.pop_front(value)) {
            std::cout << "Popped:_" << value << "\n";
        }
        std::this_thread::sleep_for(std::chrono::milliseconds(150));
    }
};

threads.emplace_back(pusher, 1);
threads.emplace_back(pusher, 2);
threads.emplace_back(popper);

for (auto& t : threads) t.join();
}

int main() {
    linkedListTest();
    return 0;
}

```

Explanation: The main challenge is that list operations like `pushFront` are not atomic. Instead, they involve multiple steps. If two threads execute these steps at the same time, they could read the same head value, and one thread's update would be lost, corrupting the list. This implementation uses a single, coarse-grained mutex to lock the entire list, which serializes all access and prevents these race conditions but also creates a bottleneck, as even non-conflicting operations are forced to wait.

Analysis: Thread safety is achieved using a single, coarse-grained mutex that locks the entire list for every operation, which simply and effectively prevents all race conditions. The major performance consideration is that this lock serializes all access, creating a bottleneck that limits scalability, as only one thread can operate on the list at a time, even for non-conflicting operations.

Screenshot: Include a screenshot of compiling and running `threadSafeLinkedList.cpp`.

```

@JonRoss7 → /workspaces/Thread_Safe_DS/labReport (main) $ g++ threadSafeLinkedList.cpp -pthread -std=c++11 -o threadSafeLinkedList
@JonRoss7 → /workspaces/Thread_Safe_DS/labReport (main) $ ./threadSafeLinkedList
Thread 1 pushed: 100
Popped: 200
Thread 2 pushed: 200
Thread 1 pushed: 101
Thread 2 pushed: 201
Popped: 201
Thread 1 pushed: 102
Thread 2 pushed: 202
Popped: 202
Thread 1 pushed: 103
Thread 2 pushed: 203
Thread 1 pushed: 104
Thread 2 pushed: 204
Popped: 204
Popped: 104
Popped: 203
Popped: 103
Popped: 102
Popped: 101
Popped: 100
@JonRoss7 → /workspaces/Thread_Safe_DS/labReport (main) $

```

Figure 5: Compilation and execution of `threadSafeLinkedList.cpp`