

Infinite Buffer Implementation with Dynamic Memory Allocation

1 Project Summary

Our project, **Infinite Buffer Implementation with Dynamic Memory Allocation**, aims to solve the classical Producer–Consumer problem. Traditional fixed-size buffers are often limited by memory constraints and can block producers when full, thereby reducing throughput and scalability. We will be creating an infinite buffer that dynamically allocates memory to accommodate an unbounded number of items.

The buffer is implemented using a **singly linked list with dynamic memory allocation**, allowing it to grow as needed. To ensure integrity of the linked list, synchronization mechanisms like **mutexes** and **condition variables** are employed. Our solution also supports **multiple producers and consumers**.

In addition to the infinite buffer, we have also implemented a **fixed-size buffer** using a circular linked list for performance comparison. Logging and performance statistics are maintained for both implementations. Through this, we keep track of key metrics such as memory usage, producer/consumer wait times, and throughput. These metrics will allow us to do a comparison between the two variants.

2 Introduction

2.1 Problem Statement

The problem statement given to us was:

Implement an infinite buffer that dynamically allocates memory as needed to handle an unbounded number of items. Use synchronization mechanisms to manage producer and consumer access to the buffer.

- Implement dynamic memory allocation for the buffer to handle an unbounded number of items.
- Use synchronization mechanisms (e.g., semaphores or monitors) to ensure proper access to the buffer.
- Add support for multiple producers and consumers.
- Implement logging to track buffer operations (e.g., insertions, deletions, and memory usage).
- Compare the performance of the infinite buffer implementation with a fixed-size buffer.

2.2 Objectives

- To design and implement an **infinite buffer** using dynamic memory allocation.
- To support **multiple producers and consumers** concurrently along with ensuring that our data structure integrity is not damaged. To do this, we use proper synchronization mechanisms.
- To develop a **fixed-size buffer** implementation for comparison.
- To incorporate **logging and metrics tracking** for operations and memory usage. This will allow us to do a comparison between the infinite and fixed-size buffer implementations.
- To evaluate the trade-offs between the infinite buffer implementation and the fixed buffer implementation.

2.3 Scope of the Project

This project is implemented in C++ using the standard threading and synchronization primitives (`std::thread`, `std::mutex`, `std::condition_variable`). The scope of our project includes:

- Design and implementation of both infinite and fixed-size buffer systems.
- Support for parallel producer and consumer threads.
- Comparison using performance metrics.
- Discussion of applicability in real-world scenarios like logging systems and real-time systems.

The overview of the two implementations that we have done is given below:

Infinite Buffer

- Dynamically grows using a singly linked list data structure.
- Allows unbounded data insertion by producers without blocking.
- Uses dual mutex locks (one for producers and one for consumers) and a condition variable to synchronize consumer access.

Fixed-Size Buffer

- Implemented using a circular linked list with a predefined capacity.
- Uses memory in a restricted way.
- Uses condition variables to manage access and block threads as needed.

3 Design Overview

3.1 Infinite Buffer Design

The infinite buffer is implemented using a **singly linked list** data structure with dynamic memory allocation. It starts with a dummy node. This simplifies the coding part. Producers add items by filling the current head node and then creating a new empty node and shifting the head pointer. Consumers consume items from the tail and delete nodes after use and free the memory.

This architecture for the infinite buffer case ensures:

- The buffer size is only limited by available system memory. This is an approximation of an infinite buffer.
- Separate head and tail pointers enable concurrent producer and consumer operations. Due to this the producers do not have to wait for consumers to free the buffer memory before they can produce the next item.
- Dual mutex locks (one for producers and one for consumers) also ensure that producers do not have to wait for consumers to free the buffer memory before they can produce the next item.

3.2 Fixed Buffer Design

The fixed buffer uses a **circular linked list** with a pre-defined number of nodes. The size of the linked list stays constant here unlike the infinite buffer. Producers have to wait when the buffer is full, and consumers have to wait when it is empty.

This architecture for the finite buffer case ensures:

- No dynamic allocation after initialization. This leads to only a constant amount of memory being used at a given time.
- Since we have only a finite amount of space, producers have to wait for consumers to free the buffer memory before they can produce the next item.
- But on the other hand, constant memory usage helps in making the system predictable. So, the process is restricted from taking up a very large chunk of the memory.

3.3 Data Structures Used

<u>Case</u>	<u>Data Structure</u>	<u>Node Structure</u>	<u>Pointers Used</u>
Infinite Buffer	Singly Linked List	int data, bool filled, Node* next	head, tail
Fixed Buffer	Circular Linked List	int data, bool filled, Node* next	head, tail

In both cases, we have designed the node class to have a filled flag to signal if the data is ready for consumption. This approach allows producers and consumers to operate independently without the need for large synchronization implementation.

4 Implementation Details

4.1 Buffer Implementation

4.1.1 Infinite Buffer

Producer Code:

```
void produce(int item, int producer_id) {  
    // Mark the time when the producer starts trying to acquire the lock.  
    auto request_lock_time = chrono::steady_clock::now();  
  
    // Acquiring the producer lock to ensure mutual exclusion while modifying the buffer  
    ticket_lock_producer.lock();  
    auto acquired_lock_time = chrono::steady_clock::now();  
  
    // Calculating how long the thread waited for the lock.  
    auto wait_duration = acquired_lock_time - request_lock_time;  
  
    // Store the produced item in the current node and mark it as filled.  
    head->data = item;  
    // Dynamically allocate a new node and link it to the current node.  
    // This ensures the buffer can grow indefinitely as new items are produced.  
    Node* new_node = new Node();  
    head->next = new_node;  
    head->filled = true;  
    head = new_node;  
  
    // Generating a timestamp relative to the start of buffer operation  
    auto now = chrono::steady_clock::now();  
    long long timestamp = chrono::duration_cast<chrono::microseconds>(now - start_time).count();  
  
    // Converting the wait time into milliseconds for logging  
    double waited_ms = chrono::duration_cast<chrono::duration<double, milli>>(wait_duration).count();  
  
    // Logging the producer event  
    logEvent("[ " + to_string(timestamp) + "us] Producer " + to_string(producer_id) + " produced: " + to_string(item) + " | Waited: " + to_string(waited_ms) + "ms");  
  
    // Releasing the producer lock so that other producers can enter the critical section.  
    ticket_lock_producer.unlock();  
    // Notifying one of the waiting consumer threads that a new item is available in the buffer  
    cv_not_empty.notify_one();  
  
    // Marking the time when production ends (after releasing the lock and notifying)  
    auto end = chrono::steady_clock::now();  
  
    // Updating cumulative production time (including wait + actual work),  
    // ensuring thread-safe access to statistics using stat mutex.  
    lock_guard<mutex> stats_lock(prod_stat_mutex);  
    total_produce_time += (end - request_lock_time);  
}
```

Consumer Code:

```

int consume(int consumer_id) {
    // consumer starts trying to acquire the lock
    auto request_lock_time = chrono::steady_clock::now();

    // Acquiring the consumer lock to ensure mutual exclusion while reading from the buffer.
    unique_lock<mutex> lock(mutex_consumer);

    // Blocking the consumer thread until an item is available (tail->filled == true).
    // This avoids busy waiting and efficiently synchronizes with the producer.
    cv_not_empty.wait(lock, [&]() {
        return tail->filled;
    });
    auto acquired_lock_time = chrono::steady_clock::now();

    // Measuring how long the thread waited (for the lock and data availability).
    auto wait_duration = acquired_lock_time - request_lock_time;

    // Retrieve the item from the buffer and mark the node as empty
    int item = tail->data;
    tail->filled = false;

    // Recording the timestamp of the consume event, relative to the global start time
    auto now = chrono::steady_clock::now();
    long long timestamp = chrono::duration_cast<chrono::microseconds>(now - start_time).count();

    // Converting the wait time to milliseconds for better logs
    double waited_ms = chrono::duration_cast<chrono::duration<double, milli>>(wait_duration).count();

    // Logging the consumer event
    logEvent("[ " + to_string(timestamp) + " us] Consumer " + to_string(consumer_id) + " consumed: " + to_string(item) + " | Waited: " + to_string(waited_ms) + " ms");

    // Freeing up the memory dynamically
    Node* temp = tail;
    tail = tail->next;
    delete temp;

    // Unlocking the buffer so other consumers can proceed
    lock.unlock();

    // Update cumulative consumption time using a thread-safe mutex
    auto end = chrono::steady_clock::now();
    lock_guard<mutex> stats_lock(cons_stat_mutex);
    total_consume_time += (end - request_lock_time);
}

```

- The buffer is implemented as a singly linked list.
- Our list starts at the dummy node. This allows us to avoid using complex logic to avoid the head of the list being deleted.
- Each producer inserts items at the head of the list, marks the node as filled, allocates a new node, and moves the head pointer forward.
- Each consumer reads from the tail, moves the pointer forward, deletes the node, and moves to the next node.

4.1.2 Fixed-Size Buffer

Producer Code:

```

void produce(int item, int producer_id) {
    auto request_lock_time = chrono::steady_clock::now(); // Time before acquiring lock

    // Ensuring fairness
    ticket_lock_producer.lock();
    // Acquiring the producer lock to ensure mutual exclusion while modifying the buffer
    unique_lock<mutex> lock(mutex_producer);

    // Wait until the node at head is free to be filled
    cv_not_full.wait(lock, [this] { return !head->filled; });

    auto acquired_lock_time = chrono::steady_clock::now(); // Time after acquiring lock

    // Calculating how long the thread waited for the lock and buffer to have space
    auto wait_duration = acquired_lock_time - request_lock_time;

    // Fill the buffer node with item
    head->data = item;
    Node* temp = head->next;
    head->filled = true;
    head = temp; // Move to next node in circle

    // Generating a timestamp relative to the start of buffer operation
    auto now = chrono::steady_clock::now();
    long long timestamp = chrono::duration_cast<chrono::microseconds>(now - start_time).count();

    // Converting the wait time into milliseconds for logging
    double waited_ms = chrono::duration_cast<chrono::duration<double, milli>>(wait_duration).count();

    // Logging the producer event
    logEvent("[ " + to_string(timestamp) + "us] Producer " + to_string(producer_id) + " produced: " + to_string(item) + " | Waited: " + to_string(waited_ms) + "ms");

    // Releasing the producer lock so that other producers can enter the critical section.
    lock.unlock();
    // Notifying one of the waiting consumer threads that a new item is available in the buffer
    cv_not_empty.notify_one();

    ticket_lock_producer.unlock();

    // Marking the time when production ends (after releasing the lock and notifying)
    auto end = chrono::steady_clock::now();

    // Updating cumulative production time (including wait + actual work),

```

Consumer Code:

```

int consume(int consumer_id) {
    // consumer starts trying to acquire the lock
    auto request_lock_time = chrono::steady_clock::now();

    // Acquiring the consumer lock to ensure mutual exclusion while reading from the buffer.
    unique_lock<mutex> lock(mutex_consumer);

    // Blocking the consumer thread until an item is available (tail->filled == true).
    // This avoids busy waiting and efficiently synchronizes with the producer.
    cv_not_empty.wait(lock, [this] { return tail->filled; });
    auto acquired_lock_time = chrono::steady_clock::now();

    // Measuring how long the thread waited (for the lock and data availability).
    auto wait_duration = acquired_lock_time - request_lock_time;

    // Retrieve the item from the buffer and mark the node as empty
    int item = tail->data;
    tail->filled = false;

    // Recording the timestamp of the consume event, relative to the global start time
    auto now = chrono::steady_clock::now();
    long long timestamp = chrono::duration_cast<chrono::microseconds>(now - start_time).count();

    // Converting the wait time to milliseconds for better logs
    double waited_ms = chrono::duration_cast<chrono::duration<double, milli>>(wait_duration).count();

    // Logging the consumer event
    logEvent("[ " + to_string(timestamp) + "us] Consumer " + to_string(consumer_id) + " consumed: " + to_string(item) + " | Waited: " + to_string(waited_ms) + "ms");

    Node* temp = tail;
    tail = tail->next; // Move to next node in circle

    // Unlocking the buffer so other consumers can proceed
    lock.unlock();
    // Notifying producers that space is available
    cv_not_full.notify_one();

    // Update cumulative consumption time using a thread-safe mutex
    auto end = chrono::steady_clock::now();
    lock_guard<mutex> stats_lock(cons_stat_mutex);
    total_consume_time += (end - request_lock_time);

    return item;
}

```

- The buffer is structured as a circular linked list with a constant number of nodes.
- Here, the head pointer points to the next node where a produced item should be stored and the tail pointer points to the next node from where the next item should be consumed.

- Producers have to wait if the node at the head is already filled. The producers will unblock when the consumer has consumed the data at that node.
- Consumers also have to wait if the node at the tail is empty. The unblock when it has an item added by a producer.

4.2 Synchronization Mechanisms

4.2.1 Infinite Buffer

Dual Mutexes:

Since the buffer is unbounded, producers never have to wait for space. This allows us to have two independent mutexes for producers and consumers.

- TicketLock: Ensures mutual exclusion with FIFO fairness for producer operations (adding items to the buffer).
- mutex_consumer: Ensures mutual exclusion for consumer operations (removing items from the buffer)

TicketLock Here we have implemented our own synchronization primitive TicketLock.

Its mechanism is shown below:

Ticket Assignment:

```
int my_ticket = next_ticket++; // Atomic increment
```

Each producer receives a unique ticket on lock request.

Wait for Turn:

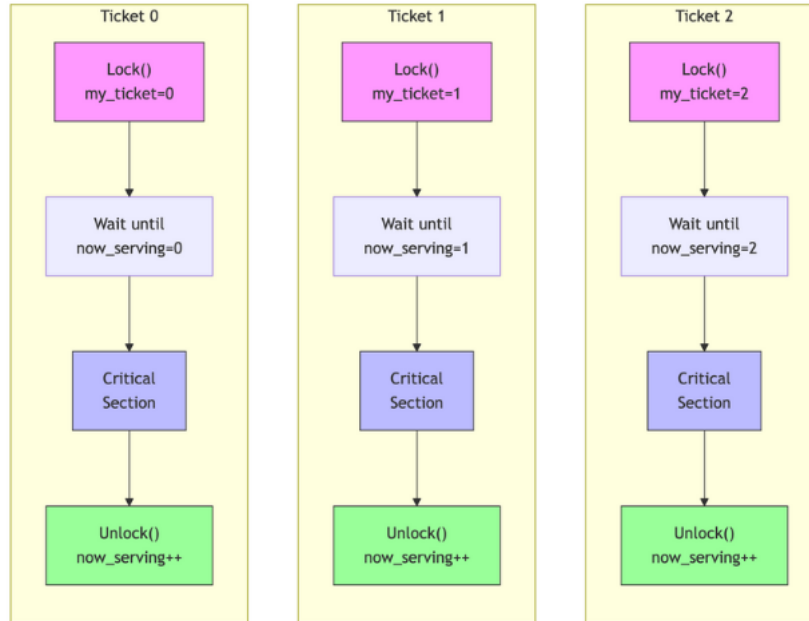
```
while (now_serving != my_ticket) {
    this_thread::yield(); // Avoids busy-wait
}
```

Producers wait until their ticket is called:

```
now_serving++; // Atomic increment
```

Unlocks the next producer in line.

A diagram showing this is given below:



Condition Variable (`cv_not_empty`): Used by consumers to wait if the buffer is empty.

Mutexes used for logging operations (`prod_stat_mutex`, `cons_stat_mutex`): Used to safely record performance metrics without interfering with buffer operations.

4.2.2 Fixed Buffer

Mutexes: Here also we use separate mutexes for producer and consumer operations. We are able to do this because we use condition variables to ensure that the produce and consume operations happen only when they are legal.

- `TicketLock`: `TicketLock` ensures fair FIFO ordering of producers.
- `mutex_producer`: Ensures that only one producer among all producers can access and modify the buffer at a time.
- `mutex_consumer`: Ensures that only one consumer among all consumers can access and modify the buffer at a time.

Two Condition Variables:

- `cv_not_empty`: Ensures consumers wait until data is available.
- `cv_not_full`: Ensures producers wait until there is space to insert data.

Mutexes used for logging operations (`prod_stat_mutex`, `cons_stat_mutex`): Used to safely record performance metrics without interfering with buffer operations.

4.3 Concurrency Safeguards

4.3.1 Zero Busy Waiting:

- Busy Waiting: Threads actively check again and again for a condition (e.g., checking `tail->filled` in a loop), wasting CPU cycles.
- Avoidance Mechanism: Threads sleep until notified, zero CPU spinning.
- Condition Variables (`cv_not_empty`, `cv_not_full`):
- Predicate Checks: Threads only wake when their required condition is met (e.g., `tail->filled` for consumers).
- Fair Notifications: `notify_one()` ensures at least one waiting thread progresses, preventing indefinite starvation.
- Example:
- `cv_not_empty.wait(lock, [this] { return tail->filled; });` // Consumer sleeps until data exists
- Predicate Checks: Eliminate spurious wakeups, ensuring threads only proceed when valid.
- Progress Guarantee: OS scheduler fairness + predicate checks guarantee eventual thread progress.

4.3.2 Monitors

- Implementation:
 - Encapsulated Buffer Class: `LinkedListBuffer` acts as a monitor, managing shared data (`head/tail`) and synchronization logic.
 - Mutexes and other Locks:
 - * `TicketLock`: Guards producer-critical sections (`head` updates).
 - * `mutex_consumer`: Guards consumer critical sections (`tail` updates).
 - * `stat_mutex`: Protects performance metric operations.
- Partial Monitor Pattern: Unlike traditional single-lock monitors, dual mutexes allow parallel producer/consumer access while maintaining mutual exclusion.
- Locking Strategy: `std::unique_lock`: Automatically releases locks during waits (e.g., `cv_not_full.wait(...)`).
- Non-Nested Locks: No thread holds multiple locks simultaneously.
- `TicketLock`: Ensures mutual exclusion with FIFO fairness for producer operations (adding items to the buffer).

Deadlock Free

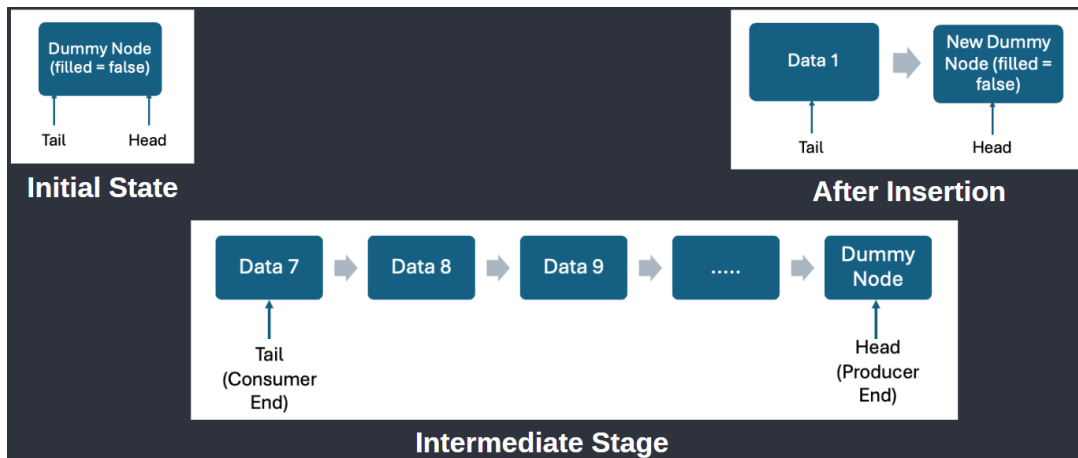
- No Circular Waits: Producers only use `mutex_producer`; consumers only use `mutex_consumer`.
- No overlap in lock acquisition order.
- Condition Variable Design: Threads release locks before waiting, allowing others to proceed.
- Strict Lock Boundaries: Producers never access tail; consumers never access head.

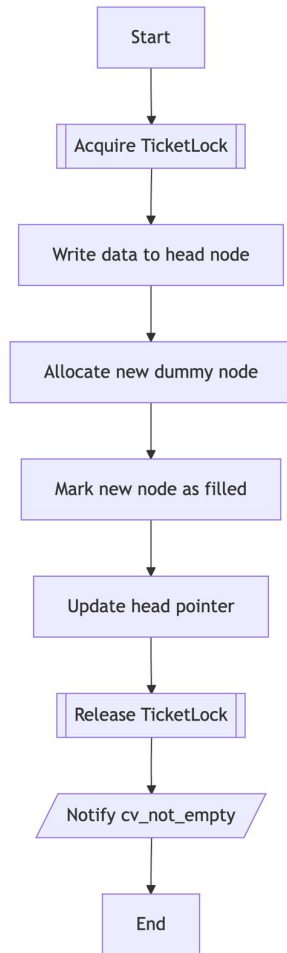
4.3.3 Stat Measurement Isolation

- Dedicated Mutexes: Stats tracking uses dedicated mutexes (`prod_stat_mutex`, `cons_stat_mutex`), separate from buffer operation locks (`mutex_producer`, `mutex_consumer`).
- Benefits:
- Non-Blocking: Stats updates don't block producers/consumers during core operations.
- Low Overhead: Stats are recorded after releasing primary locks, minimizing contention.

4.4 Producer Consumer Workflow

4.4.1 Infinite Buffer

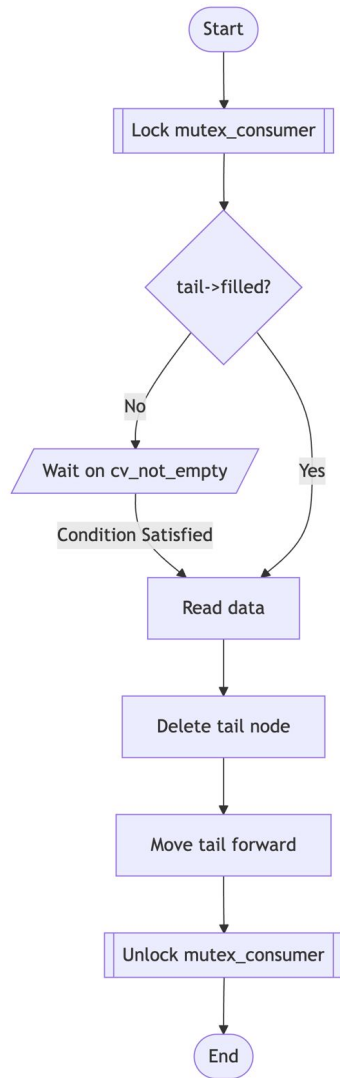




Producer Workflow

1. Acquire TicketLock (FIFO Order)
2. Write data to current head node
3. Allocate + link new dummy node
4. Mark node as filled
5. Move head forward
6. Release TicketLock + notify consumers

Key Feature: No capacity checks – always expands.

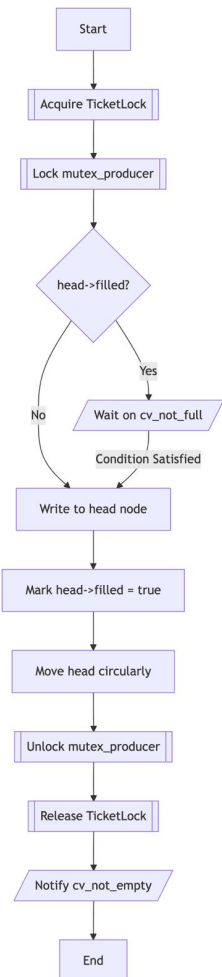
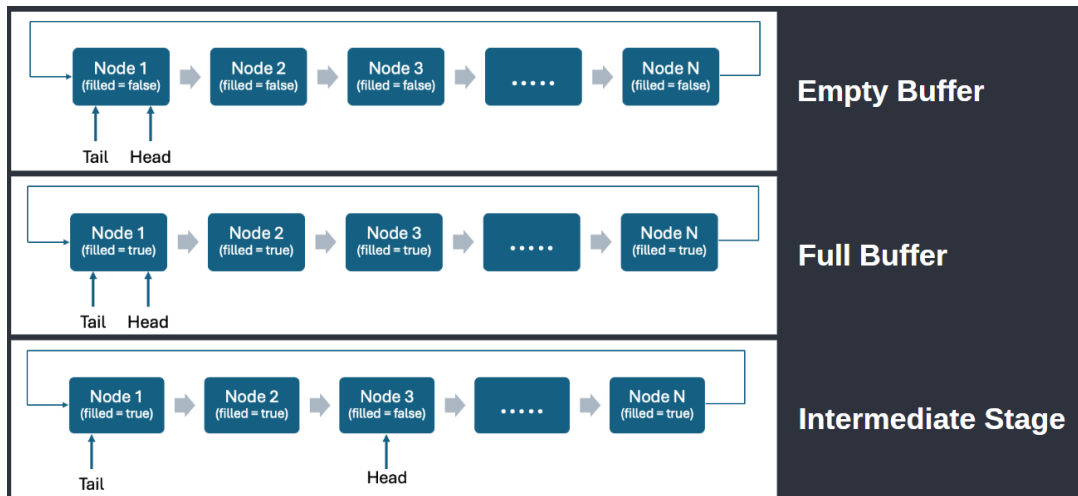


Consumer Workflow

1. Lock consumer mutex
2. Wait via `cv_not_empty` until `tail→filled`
3. Read `tail→data`
4. Mark `tail→filled = false`
5. Move tail forward
6. Delete old node
7. Unlock

Safety: Consumers never access uninitialized nodes.

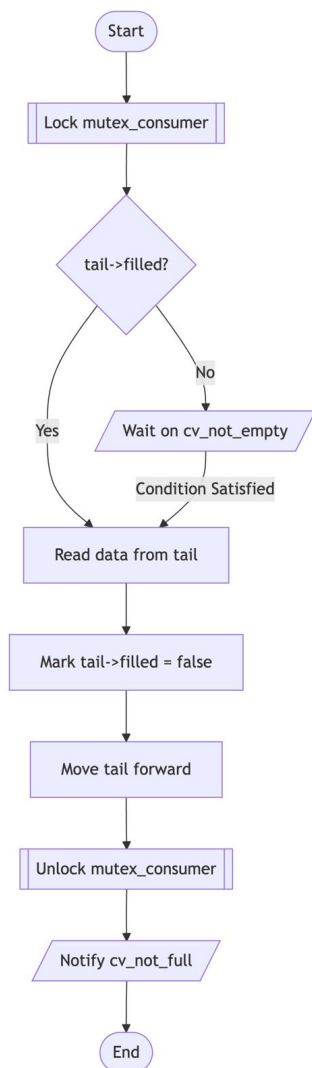
4.4.2 Fixed Buffer



Producer Workflow

1. Acquire ticket_lock_producer (fair queuing)
2. Lock PRODUCER_MUTEX
3. Wait on cv_not_full until head→filled == false
4. Write data to head→data
5. Set head→filled = true
6. Move head = head→next (circular wrap)
7. Unlock PRODUCER_MUTEX
8. Notify cv_not_empty (signal consumers)
9. Release ticket_lock_producer

Blocking Behavior: Producers stall if buffer is full (all nodes filled).



Consumer Workflow

1. Lock CONSUMER_MUTEX
2. Wait on cv_not_empty until tail→filled == true
3. Read tail→data
4. Set tail→filled = false
5. Move tail = tail→next (circular wrap)
6. Unlock CONSUMER_MUTEX
7. Notify cv_not_full (signal producers)

Safety Guarantee: Consumers never read uninitialized/unfilled nodes.

5 Logging and Monitoring

5.1 Logging Operations

In our project we have performed logging in two ways: Log file, and Standard Console Output: -

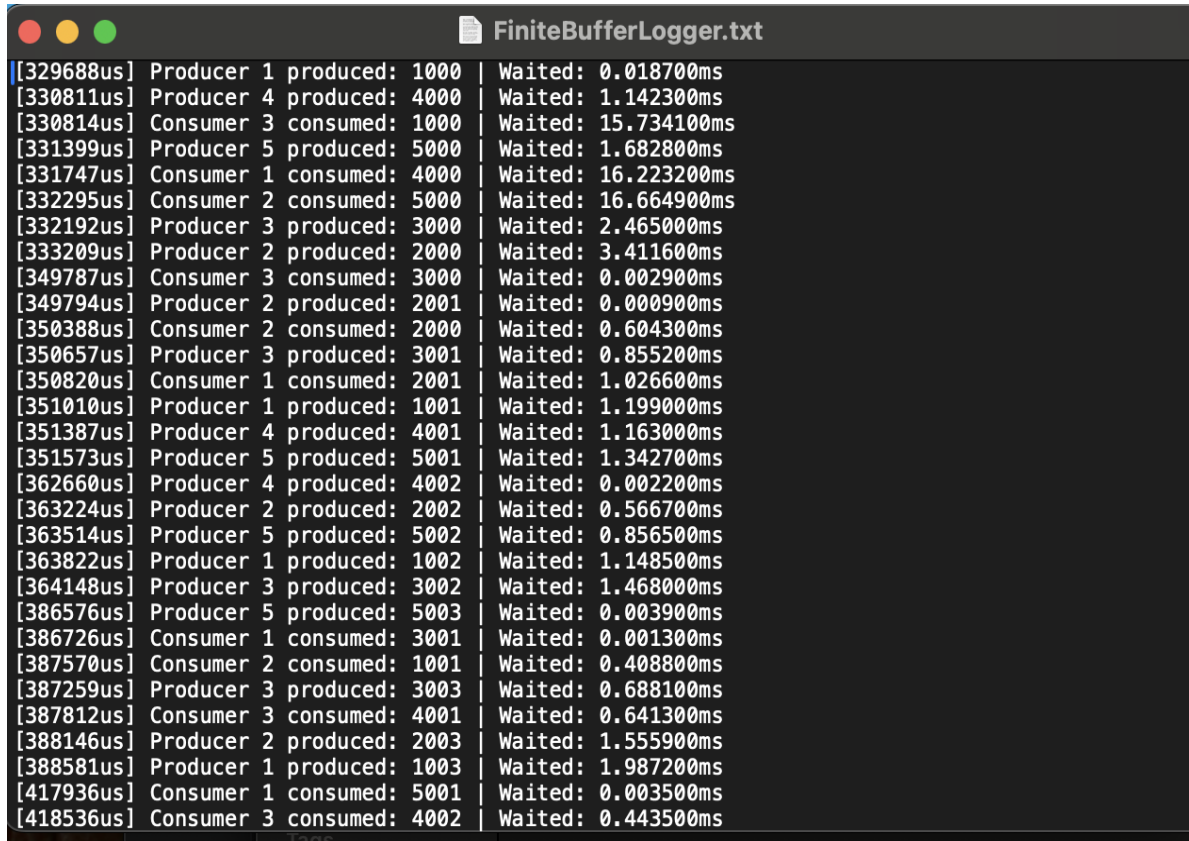
5.1.1 Log file:

- On running the code for Infinite and Finite Buffer, it generates 2 log files namely InfiniteBufferLogger.txt and FiniteBufferLogger.txt respectively.
- In these files, each event (production and consumption) is being recorded with the following details:
 1. Timestamp of the event (in microseconds from the start)
 2. Producer or Consumer ID
 3. Produced/Consumed item
 4. Wait time for the producer or consumer due to mutual exclusion (milliseconds)

InfiniteBufferLogger: -

```
InfiniteBufferLogger.txt
[332253us] Producer 1 produced: 1000 | Waited: 0.009700ms
[332747us] Producer 2 produced: 2000 | Waited: 0.486100ms
[332755us] Consumer 1 consumed: 1000 | Waited: 18.222700ms
[333041us] Producer 5 produced: 5000 | Waited: 0.802000ms
[333225us] Consumer 2 consumed: 2000 | Waited: 18.556200ms
[333401us] Producer 3 produced: 3000 | Waited: 1.162500ms
[333559us] Consumer 3 consumed: 5000 | Waited: 18.802700ms
[333729us] Producer 4 produced: 4000 | Waited: 1.429800ms
[346120us] Producer 3 produced: 3001 | Waited: 0.002600ms
[346639us] Producer 2 produced: 2001 | Waited: 0.517500ms
[346950us] Producer 1 produced: 1001 | Waited: 0.816600ms
[347282us] Producer 4 produced: 4001 | Waited: 1.100600ms
[348020us] Producer 5 produced: 5001 | Waited: 1.833200ms
[372255us] Producer 2 produced: 2002 | Waited: 0.001400ms
[372255us] Consumer 2 consumed: 3000 | Waited: 0.001600ms
[372679us] Producer 3 produced: 3002 | Waited: 0.426100ms
[373247us] Consumer 1 consumed: 4000 | Waited: 0.983900ms
[373613us] Producer 1 produced: 1002 | Waited: 1.340500ms
[373807us] Consumer 3 consumed: 3001 | Waited: 1.535900ms
[373990us] Producer 5 produced: 5002 | Waited: 1.689900ms
[374379us] Producer 4 produced: 4002 | Waited: 2.072900ms
[387806us] Producer 2 produced: 2003 | Waited: 0.002600ms
[403788us] Producer 3 produced: 3003 | Waited: 0.001900ms
[404205us] Producer 2 produced: 2004 | Waited: 0.422600ms
[404461us] Producer 4 produced: 4003 | Waited: 0.674200ms
[404676us] Producer 1 produced: 1003 | Waited: 0.877300ms
[404862us] Producer 5 produced: 5003 | Waited: 1.056200ms
[419429us] Consumer 2 consumed: 2001 | Waited: 0.002400ms
[419461us] Producer 5 produced: 5004 | Waited: 0.000600ms
[420086us] Consumer 1 consumed: 1001 | Waited: 0.660000ms
```

FiniteBufferLogger: -



```
[329688us] Producer 1 produced: 1000 | Waited: 0.018700ms
[330811us] Producer 4 produced: 4000 | Waited: 1.142300ms
[330814us] Consumer 3 consumed: 1000 | Waited: 15.734100ms
[331399us] Producer 5 produced: 5000 | Waited: 1.682800ms
[331747us] Consumer 1 consumed: 4000 | Waited: 16.223200ms
[332295us] Consumer 2 consumed: 5000 | Waited: 16.664900ms
[332192us] Producer 3 produced: 3000 | Waited: 2.465000ms
[333209us] Producer 2 produced: 2000 | Waited: 3.411600ms
[349787us] Consumer 3 consumed: 3000 | Waited: 0.002900ms
[349794us] Producer 2 produced: 2001 | Waited: 0.000900ms
[350388us] Consumer 2 consumed: 2000 | Waited: 0.604300ms
[350657us] Producer 3 produced: 3001 | Waited: 0.855200ms
[350820us] Consumer 1 consumed: 2001 | Waited: 1.026600ms
[351010us] Producer 1 produced: 1001 | Waited: 1.199000ms
[351387us] Producer 4 produced: 4001 | Waited: 1.163000ms
[351573us] Producer 5 produced: 5001 | Waited: 1.342700ms
[362660us] Producer 4 produced: 4002 | Waited: 0.002200ms
[363224us] Producer 2 produced: 2002 | Waited: 0.566700ms
[363514us] Producer 5 produced: 5002 | Waited: 0.856500ms
[363822us] Producer 1 produced: 1002 | Waited: 1.148500ms
[364148us] Producer 3 produced: 3002 | Waited: 1.468000ms
[386576us] Producer 5 produced: 5003 | Waited: 0.003900ms
[386726us] Consumer 1 consumed: 3001 | Waited: 0.001300ms
[387570us] Consumer 2 consumed: 1001 | Waited: 0.408800ms
[387259us] Producer 3 produced: 3003 | Waited: 0.688100ms
[387812us] Consumer 3 consumed: 4001 | Waited: 0.641300ms
[388146us] Producer 2 produced: 2003 | Waited: 1.555900ms
[388581us] Producer 1 produced: 1003 | Waited: 1.987200ms
[417936us] Consumer 1 consumed: 5001 | Waited: 0.003500ms
[418536us] Consumer 3 consumed: 4002 | Waited: 0.443500ms
```

5.1.2 Standard Console Output:

- After the simulation, a summary report is printed as standard output in the console with the following details:
 - Total items Produced
 - Total items Consumed
 - Final Buffer Size (0 in case of infinite buffer and equal to the fixed size in case of finite buffer)
 - Peak Buffer Size (It is used to indicate the memory usage)
 - Total Runtime (For comparing performance of both types of Buffers)
 - Total Produce time
 - Total Consume time
- Producer Stats: Total Wait Time, Average Wait Time, and Maximum Wait Time
- Consumer Stats: same parameters as in Producers Stats

- Producer Fairness (Starvation-free algorithm):

It is printed to show that our algorithm offers a fair chance to each of the producers. It is seen that the average waiting time and more importantly the maximum waiting time for each of the producers is almost the same and none of the producers has to face starvation during the production of items. Thus, our algorithm is starvation-free for the producers.

However, we have intentionally not made it starvation-free at the consumer's side.

– **Why Not Fairness:**

* **Task Agnostic:** Order of consumption irrelevant (e.g., batch processing).

– **Throughput Focus:** Avoid TicketLock's 15-20% latency penalty.

For Infinite Buffer: -

```
===== LOG ANALYSIS REPORT =====
Total Items Produced      : 150
Total Items Consumed      : 150
Final Buffer Size         : 0
Peak Buffer Size (Nodes)  : 102

--- Runtime ---
Total Runtime              : 1.688 seconds
Total Produce Time (just to produce in buffer including lock acquiring time and writing time) : 0.194 seconds
Total Consume Time (just to consume from buffer including lock acquiring time and reading time): 0.158 seconds

--- Producer Stats ---
Total Wait Time           : 135.516 ms
Average Wait Time         : 0.903 ms
Maximum Wait Time         : 4.057 ms

--- Consumer Stats ---
Total Wait Time           : 90.145 ms
Average Wait Time         : 0.601 ms
Maximum Wait Time         : 11.749 ms

--- Producer Fairness (by Avg Wait Time) ---
Producer 4 | Produced: 30 | Avg Wait Time: 1.046 ms | Max Wait Time: 2.079 ms
Producer 5 | Produced: 30 | Avg Wait Time: 0.912 ms | Max Wait Time: 1.704 ms
Producer 2 | Produced: 30 | Avg Wait Time: 0.869 ms | Max Wait Time: 2.250 ms
Producer 3 | Produced: 30 | Avg Wait Time: 0.663 ms | Max Wait Time: 4.057 ms
Producer 1 | Produced: 30 | Avg Wait Time: 1.027 ms | Max Wait Time: 2.937 ms

=====

All tasks done. Check InfiniteBufferLogger.txt for logs.
```

For Finite Buffer: -

```

===== LOG ANALYSIS REPORT =====
Total Items Produced      : 150
Total Items Consumed      : 150
Final Buffer Size         : 10
Peak Buffer Size (Nodes)  : 10

--- Runtime ---
Total Runtime              : 1.648 seconds
Total Produce Time (just to produce in buffer including lock acquiring time and writing time) : 4.911 seconds
Total Consume Time (just to consume from buffer including lock acquiring time and reading time) : 0.146 seconds

--- Producer Stats ---
Total Wait Time            : 4862.041 ms
Average Wait Time          : 32.414 ms
Maximum Wait Time          : 50.581 ms

--- Consumer Stats ---
Total Wait Time            : 93.894 ms
Average Wait Time          : 0.626 ms
Maximum Wait Time          : 15.278 ms

--- Producer Fairness (by Avg Wait Time) ---
Producer 5 | Produced: 30 | Avg Wait Time: 31.481 ms | Max Wait Time: 50.277 ms
Producer 2 | Produced: 30 | Avg Wait Time: 33.425 ms | Max Wait Time: 50.581 ms
Producer 4 | Produced: 30 | Avg Wait Time: 33.406 ms | Max Wait Time: 48.527 ms
Producer 1 | Produced: 30 | Avg Wait Time: 32.403 ms | Max Wait Time: 48.793 ms
Producer 3 | Produced: 30 | Avg Wait Time: 31.353 ms | Max Wait Time: 48.643 ms

=====
All tasks done. Check FiniteBufferLogger.txt for logs.

```

5.2 Memory Usage Tracking

In order to track memory usage, we have calculated the peak buffer size as it will indicate the maximum memory used during the execution of the program.

The Infinite Buffer dynamically allocates memory as a new node whenever a producer needs to produce an item and deallocates the memory as a node is deleted whenever a consumer needs to consume an item.

The Finite Buffer has a fixed memory usage which is decided by the fixed size of the buffer.

6 Performance Analysis

6.1 Experimental Setup

- Language: C++
- # Producers: 5
- # Consumers: 3
- Workload: 30 items/ producer and 50 items/ consumer
- Delays: 10ms between productions and 18ms between consumptions
- Graphics library: SFML (Simple and Fast Multimedia Library) is used for real-time visualization of producer-consumer events.

6.2 Results: Infinite Buffer vs Fixed Buffer

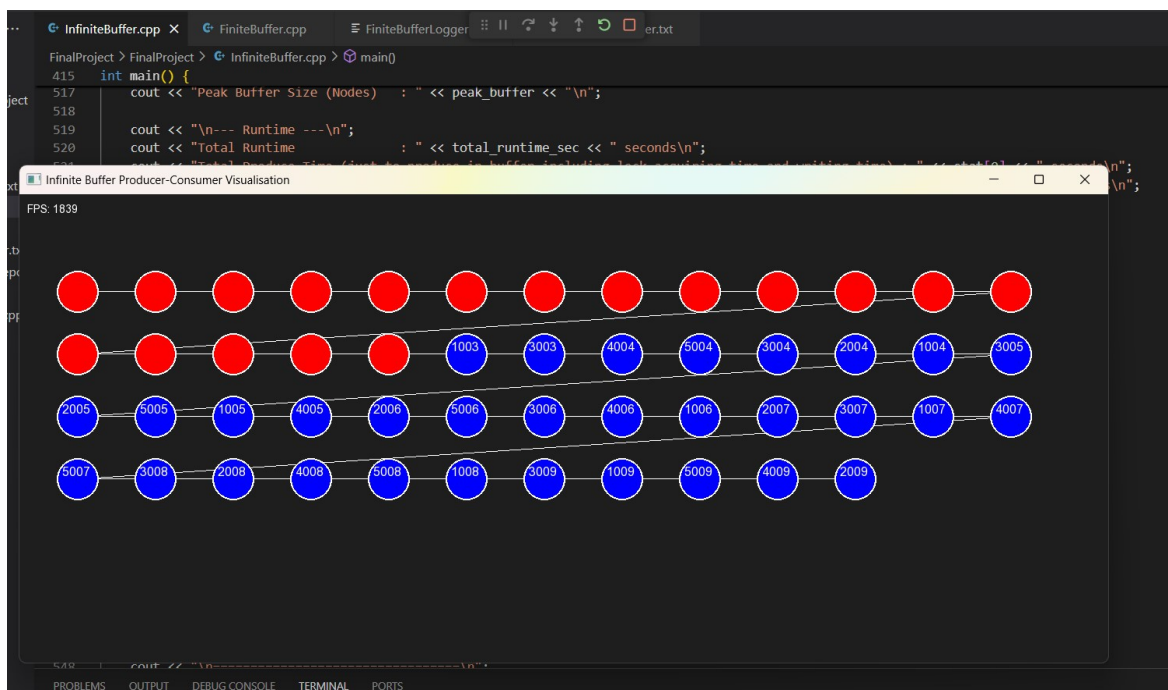
Metric	Infinite Buffer	Finite Buffer
Blocking on Full Buffer	<u>No</u> Since there is no fixed size of the buffer, the producers can produce indefinitely.	<u>Yes</u> As soon as the buffer becomes full, the consumers have to wait until any consumer consumes an item.
Blocking on Empty Buffer	<u>Yes</u> If the buffer is empty then producers have to wait.	<u>Yes</u> If the buffer is empty then producers have to wait.
Memory Allocation	<u>Dynamic</u> Each node is dynamically allocated memory as soon as the producer wants to produce the next item. When a consumer consumes an item then that node is also dynamically deallocated making that memory available for future use.	<u>Static</u> Memory is statically allocated and remains fixed during the production-consumption process.
Peak Memory Usage	<u>High (up to 102 nodes)</u> Memory usage is high in this case as the producers can keep on producing without any restrictions.	<u>Constant (10 nodes)</u> Memory usage is comparatively lower due to memory size being fixed.
Producer Wait Time	<u>Low</u> Average and Maximum producer wait time is low and is independent of the rates of production and consumption.	<u>Moderate to High</u> Average and Maximum producer wait time is relatively moderate to higher and it depends upon the relative rate of production and consumption.

6.3 Observations

- Infinite buffer handles production surges extremely well without blocking producers.
- Finite buffer, while memory efficient, experiences producer blocking when full.
- SFML visualization helps in understanding the dynamic behavior of buffers.
- Ticket locks in the infinite buffer ensure fairness among producers by maintaining order.

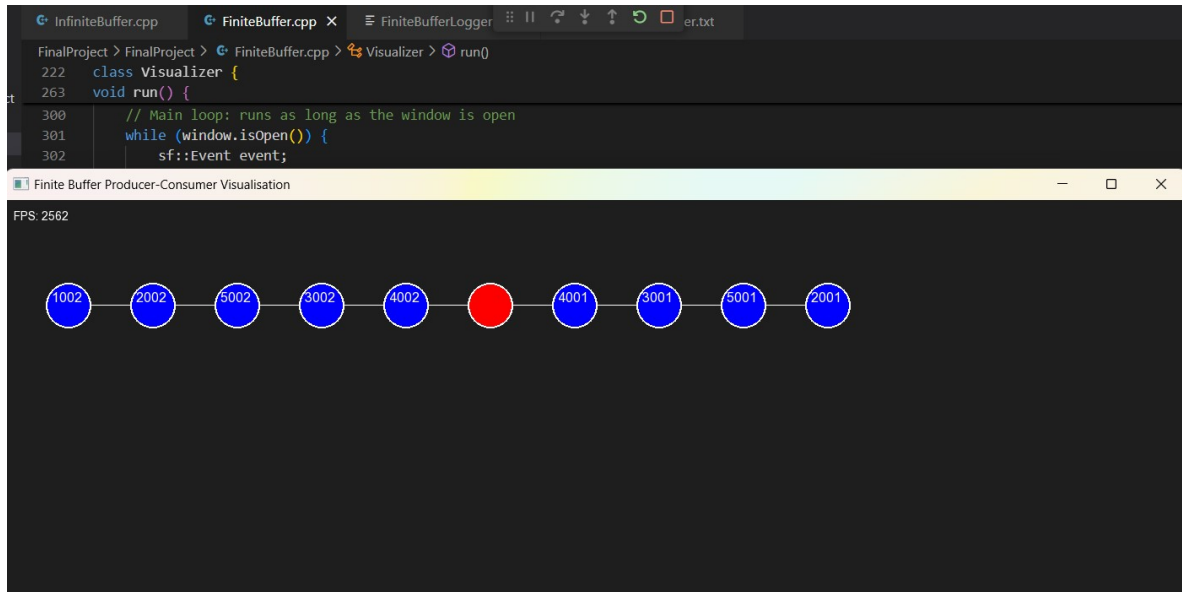
- The finite buffer provides deterministic memory usage, making it ideal for embedded or resource-constrained systems.
- Infinite buffer requires careful memory management to prevent system overload.
- Peak buffer size tracking reveals the maximum memory usage.
- Console and file logs enable detailed post-run performance analysis.
- Dynamic memory allocation in infinite buffer introduces slight overhead compared to static allocation in finite buffer.

Infinite Buffer Visualization using SFML: -



- Here red nodes represent those nodes which have been consumed and deleted.

Finite Buffer Visualization using SFML: -



- Here red nodes represent those nodes which are consumed. Here, at any point number of blue nodes does not exceed 10 (as we have set that as the buffer capacity).

7 Applications

7.1 Infinite Buffer is Better When:

7.1.1 Bursty/Unpredictable Workloads

- Real-time analytics: Sudden spikes in social media activity.
- Log aggregation: Handling massive log floods during system failures.

7.1.2 Continuous Data Streams

- Video streaming services: Buffering frames to avoid playback interruptions.
- IoT telemetry: Storing sensor data until cloud processing is available.

7.1.3 High Producer Throughput

- Big data ingestion: Producers (e.g., APIs) outpace consumers (e.g., batch processors).
- AI/ML training pipelines: Collecting training data asynchronously.

7.2 Finite Buffer is Better When:

7.2.1 Resource-Constrained Systems

- Embedded systems (e.g., IoT devices) with limited memory.
- Real-time systems requiring predictable memory usage.

7.2.2 Rate-Limited Workloads

- Manufacturing assembly lines: Fixed buffer zones for parts to avoid overflow.
- Load balancing: Throttling incoming requests to prevent server overload.

7.2.3 Stability-Critical Systems

- Automotive Anti-lock Braking Systems (ABS) processing wheel speed sensor data.

8 Conclusions

8.1 High Producer Wait Time in Fixed Buffer

- Explanation: Producers frequently wait on `cv_not_full` when the buffer is full, as they cannot proceed until consumers free space.
- Cause: Fixed buffer size creates contention during high producer throughput.
- Implication: Throughput is limited by consumer speed, making it unsuitable for bursty producer workloads.

8.2 High Memory Usage in Infinite Buffer

- Explanation: Dynamic node allocation allows unbounded growth, consuming memory proportional to unprocessed items.
- Cause: No node recycling – each insertion creates a new node, deleted only after consumption.
- Implication: Risk of out-of-memory errors in long-running systems with slow consumers.

8.3 Total Producer Produce Time Exceeds Total Runtime

- Explanation: Cumulative time across all producer threads exceeds wall-clock runtime due to parallel execution.
- Cause: Multiple producers work concurrently (e.g., 5 threads \times 30 items each = 150 produce operations).
- Implication: Highlights effective concurrency.

9 References

- A. Silberschatz, P. B. Galvin and G. Gagne, Operating System Concepts
- A. Williams, C++ Concurrency in Action: Practical Multithreading
- Lecture slides by Professor Peddoju Sateesh Kumar, Department of Computer Science, IIT Roorkee

- GeeksforGeeks –Tutorials on producer–consumer problem, mutexes, condition variables
- SFML Library: sfml-dev.org
- C++ Reference: cppreference.com

Team Details

Ansh Jain (23115017)
Aryan Laroia (23118014)
Nandini (23112066)
Shorya Panwar (23125032)
Yash Jain (23125040)