Table of Contents

Old Bridge Problem Implementation in C++

Introduction

Bridge Problem

- Classic synchronization problem
- Single-lane bridge with capacity limits
- Models resource sharing in concurrent systems

OS Significance

- Thread synchronization demonstration
- Mutual exclusion concept
- Resource management example

Implementation Goals

- C++ threads usage
- Constraints enforcement
- Performance analysis

Real-world concurrency problem with practical OS concepts

Problem Statement

Bridge Constraints

1. At most 3 cars on bridge at a time
2. All cars go in same direction
3. Empty bridge + waiting car = immediate entry
4. Non-full bridge + same direction = allow entry

Synchronization Requirements

5. One thread accesses shared state at a time
6. Fairness between directions
7. Prevent deadlock & starvation

Implementation Approach

Data Structures

- **Bridge Class** with shared state
- **Direction flags** & counters
- **Time tracking** for fairness

Synchronization

- **Mutex** for exclusive access
- **Condition variables** for waiting
- **Fairness algorithm** for direction

Thread Management

- **Car threads** with random direction
- **Bridge logic** for entry/exit
- **Timing simulation** for crossing

Synchronization ensures all constraints are met while maximizing bridge throughput

Code Analysis <>

{} Bridge Class

```
class Bridge { private: mutex mtx;
condition_variable cv; int cars_on_bridge; int
waiting[2]; int current_dir; public: Bridge():
cars_on_bridge(0), waiting{false,false},
current_dir(-1) {} };
```

Σ Key Functions

```
get_lock() - Exclusive access
can_enter_unlock() - Entry condition
enter_unlocked() - Bridge entry
exit_unlocked() - Bridge exit
arrive_and_across() - Main thread
```

→ Entry Logic

```
auto lock = bridge.get_lock();
bridge.mark_wating_unlock(dir); cv.wait(lock,
[&]() { return bridge.can_enter_unlock(dir); });
bridge.enter_unlocked(dir);
```

← Exit Logic

```
// Simulate crossing time
this_thread::sleep_for( chrono::milliseconds(500
+ (id % 5) * 200) ); auto lock2 =
bridge.get_lock(); bridge.exit_unlocked();
bridge.notify_all();
```

Results and Discussion

Bridge Crossing Simulation

```
$ ./bridge_simulation
car1 enter the bridge side left | on_bridge=1
car2 enter the bridge side left | on_bridge=2
car3 enter the bridge side left | on_bridge=3
[EXIT] Car 1 dir=LEFT | on_bridge=2
car4 enter the bridge side left | on_bridge=3
[EXIT] Car 2 dir=LEFT | on_bridge=2
```

Constraints Met

- **Capacity Limit**: Never exceeds 3 cars
- **Direction Consistency**: All cars in same direction
- **Fairness**: Alternates between directions when empty

Thread Behavior

- **Blocking**: Cars wait when conditions not met
- **Notification**: cv.notify_all() wakes waiting threads
- **Locking**: Exclusive access to shared state

Conclusion

Implementation Summary

- All 5 constraints implemented
- Mutex & condition_variable used
- Fairness mechanism added

Learning Outcomes

- Thread synchronization concepts
- C++ threading library experience
- Concurrent programming skills

Technical Insights

- Proper locking prevents race conditions
- Wait/notify patterns for coordination
- Performance vs correctness balance

Future Enhancements

- Priority-based crossing system
- Visualization of bridge traffic
- Performance metrics collection

This implementation demonstrates effective synchronization techniques