Airline Booking System: Critical Section Analysis
Based on Centrum Air Overbooking Incident
Student Name: Sharapova Elizabet
Course: Operating Systems CS2-2024
Date: December 29, 2025

1. Executive Summary

This report analyzes the Centrum Air overbooking incident that occurred on September 26, where two passengers were left stranded at Istanbul Airport due to double-booking of seats. This real-world problem demonstrates a classic **Critical Section** issue in Operating Systems - specifically a **race condition** that occurs when multiple processes access shared resources without proper synchronization.
The report explains: - Why overbooking happens (technical root cause) - How it relates to OS concepts (critical sections, race conditions) - How to prevent it (synchronization mechanisms) - Implementation of both unsafe and safe booking systems

2. The Centrum Air Incident

2.1 What Happened?

On September 26, 2024, Centrum Air sold two tickets for the **same seat** on the Istanbul-Namangan route. This resulted in: - Two passengers arriving at the airport with valid tickets - Not enough physical seats on the aircraft - Passengers stranded at Istanbul Airport - Forced rebooking to Istanbul-Tashkent route after 24 hours

2.2 Root Cause

The technical reason for this overbooking is a **race condition** in the booking system.

3. Critical Section Problem Explained

3.1 What is a Critical Section?

A **critical section** is a segment of code where: 1. **Shared resources** are accessed (e.g., available_seats variable) 2. Only **one process** should execute at a time 3. Other processes must **wait** until the critical section is free

3.2 Critical Section in Booking System

In our booking system, the critical section includes:

```
# CRITICAL SECTION BEGINS
current_available = self.available_seats  # READ
if current_available >= num_tickets:
    self.available_seats -= num_tickets   # WRITE
    # Create booking record
# CRITICAL SECTION ENDS
```

Why is this critical? - Multiple threads can read available_seats simultaneously - If they all see enough seats, they all proceed to book - Multiple threads write to available_seats without coordination - Result: More tickets sold than seats available

3.3 Requirements for Critical Section Solution

Any solution must satisfy these properties:

Mutual Exclusion: Only one process in critical section at a time

Progress: If no process is in critical section, one waiting process should be allowed to enter
Bounded Waiting: There must be a limit on how long a process waits
No Assumptions: Solution shouldn't depend on CPU speed or number of processes

4. Race Condition Analysis
4.1 What is a Race Condition?
A **race condition** occurs when: - Multiple processes access shared data concurrently - The final result depends on the **timing** of their execution - Different executions can produce different results
4.2 Race Condition in Centrum Air Case
Scenario: 1 seat left, 2 customers booking simultaneously
Thread A (User 1):
1. Read available_seats = 1 ✓
2. [Context switch - Thread B starts]
Thread B (User 2):
1. Read available_seats = 1 ✓ (Still sees 1 seat!)
2. Check: 1 >= 1 → True ✓
3. Book seat: available_seats = 1 - 1 = 0
Thread A (resumes):
3. Check: 1 >= 1 → True ✓ (Used old data!)
4. Book seat: available_seats = 0 - 1 = -1 ❌
Result: 2 tickets sold, but only 1 seat existed!
4.3 Why Network Delays Make It Worse
```
# Read shared variable
current_available = self.available_seats

# Network delay or processing time
time.sleep(0.1)  # During this time, OTHER threads can also read!

# Write to shared variable (now using STALE data)
self.available_seats -= num_tickets
```
The delay between READ and WRITE creates a **window of vulnerability** where race conditions occur.

5. Read/Write Conflicts
5.1 Types of Conflicts
In concurrent systems, we have three types of operations:
Read-Read: No conflict (multiple reads are safe)
Read-Write: Conflict (reader gets inconsistent data)
Write-Write: Conflict (data corruption)
5.2 Booking System Conflicts
Our booking system has **Read-Write conflicts**:
Process A
Process B

Problem
Read seats = 10
-
A reads 10
-
Read seats = 10
B reads 10 (same value)
Write seats = 7
-
A books 3 seats
-
Write seats = 7
B books 3 seats (overwrites A's update!)
Expected result: 10 - 3 - 3 = 4 seats
Actual result: 7 seats (one booking lost!)
5.3 Solution: Atomic Operations
To prevent conflicts, we need **atomic operations** - operations that cannot be interrupted:

```
with lock:  # Atomic section begins
    read_value = shared_variable
    modify_value = read_value - amount
    shared_variable = modify_value
# Lock released, atomic section ends
```

6. Producer-Consumer Analogy
6.1 Classic Producer-Consumer Problem
The Producer-Consumer problem is a classic synchronization problem: - **Producers** create items and add them to a buffer - **Consumers** remove items from the buffer - **Problem:** Prevent buffer overflow/underflow
6.2 Booking System as Producer-Consumer
Our booking system is similar:
Producers = Ticket Cancellations - Add available seats back to the pool - Increase available_seats
Consumers = Ticket Bookings - Remove seats from the available pool - Decrease available_seats
Shared Buffer = Available Seats
6.3 Similarities
Producer-Consumer
Booking System
Buffer overflow
Negative seat count
Buffer underflow
Overbooking
Bounded buffer
Limited seat capacity
Synchronization needed

Mutex/Semaphore needed
Both problems require: 1. **Mutual exclusion** when accessing shared resource 2. **Condition checking** before operations 3. **Synchronization primitives** (locks, semaphores)

7. Solution Implementation
7.1 Solution 1: Mutex (Mutual Exclusion Lock)
How Mutex Works:

```
class SafeBookingSystem:
    def __init__(self):
        self.lock = threading.Lock()  # Create mutex
        self.available_seats = 100

    def book_ticket(self, num_tickets):
        with self.lock:  # Acquire lock (only one thread enters)
            # CRITICAL SECTION - Protected!
            if self.available_seats >= num_tickets:
                self.available_seats -= num_tickets
                return True
            return False
        # Lock automatically released
```

Why This Works: 1. threading.Lock() creates a mutex object 2. with self.lock: acquires the lock 3. Only ONE thread can hold the lock at a time 4. Other threads WAIT until lock is released 5. No race condition possible!
Advantages: - Simple to implement - Guarantees mutual exclusion - Prevents all race conditions
Disadvantages: - Can cause performance bottleneck - Risk of deadlock if not used carefully
7.2 Solution 2: Semaphore
How Semaphore Works:

```
class SemaphoreBookingSystem:
    def __init__(self, total_seats):
        self.seat_semaphore = threading.Semaphore(total_seats)
        self.booking_lock = threading.Lock()

    def book_ticket(self, num_tickets):
        # Try to acquire N semaphore permits
        acquired = []
        for i in range(num_tickets):
            if self.seat_semaphore.acquire(blocking=False):
                acquired.append(i)
            else:
                # Not enough seats, release what we got
                for _ in acquired:
                    self.seat_semaphore.release()
```

```
            return False

        # Successfully acquired all needed permits
        with self.booking_lock:
            # Create booking record
            pass
        return True
```

Why This Works: - Semaphore initialized with seat count - Each booking "consumes" semaphore permits - When count reaches 0, no more bookings possible - Releasing permits makes seats available again

Advantages: - Natural representation of resources - Can track multiple resources - Good for resource pools

Disadvantages: - More complex than mutex - Need to handle partial acquisition

7.3 Solution 3: Database Transactions

For real-world systems, use database-level synchronization:

```
BEGIN TRANSACTION;

-- Lock the row for update
SELECT available_seats FROM flights
WHERE flight_id = 123
FOR UPDATE;

-- Check and update atomically
UPDATE flights
SET available_seats = available_seats - 2
WHERE flight_id = 123
AND available_seats >= 2;

-- Check if update succeeded
IF (ROW_COUNT() > 0) THEN
    INSERT INTO bookings (flight_id, passenger, tickets)
    VALUES (123, 'Passenger A', 2);
    COMMIT;
ELSE
    ROLLBACK;
END IF;
```

Why This Works: - FOR UPDATE locks the row - Other transactions wait - WHERE available_seats >= 2 ensures atomic check - Transaction guarantees all-or-nothing

8. Comparative Analysis

8.1 Test Results

Running our implementation with 10 seats and 6 concurrent users:

Unsafe System (No Protection):

Total Seats: 10

Bookings Made: 6

Total Tickets Sold: 16
OVERBOOKING: 6 seats! ❌
Safe System (With Mutex):
Total Seats: 10
Bookings Made: 4
Total Tickets Sold: 10
No overbooking ✓
2 customers rejected (fair) ✓
8.2 Performance Considerations
Aspect
Unsafe System
Safe System
Speed
Faster (no waiting)
Slightly slower (locking overhead)
Correctness
❌ Wrong results
✓ Correct results
Reliability
❌ Unpredictable
✓ Predictable
Data Integrity
❌ Corrupted
✓ Protected
Conclusion: The slight performance cost of synchronization is worth it for correctness!

9. Real-World Solutions
9.1 How Airlines Actually Handle This
Modern booking systems use multiple layers of protection:

- **Database Transactions** (primary protection)
- **Optimistic Locking** (version checking)
- **Seat Reservations** (temporary hold before payment)
- **Queue Systems** (serialize booking requests)
- **Intentional Overbooking** (statistical models)

9.2 Optimistic Locking Example

```
class OptimisticBooking:
    def book_ticket(self, flight_id, num_tickets):
        while True:
            # Read current state with version
            current = db.query(
                "SELECT available_seats, version FROM flights WHERE id = ?",
                flight_id
            )
```

```
        if current.available_seats < num_tickets:
            return False

        # Try to update with version check
        updated = db.execute(
            """UPDATE flights
               SET available_seats = available_seats - ?,
                   version = version + 1
               WHERE id = ? AND version = ?""",
            num_tickets, flight_id, current.version
        )

        if updated.rowcount > 0:
            return True  # Success!
        else:
            # Version changed, retry
            continue
```

This approach: - Doesn't lock rows - Detects conflicts at write time - Retries if conflict detected - Better performance under low contention

10. Conclusion

10.1 Summary of Findings

The Centrum Air overbooking incident is a perfect example of a **critical section problem** in real-world systems. The root cause was a **race condition** where multiple booking processes accessed shared seat inventory without proper synchronization.

12. Appendix: Running the Code

A. Python Backend

```
# Install Python 3.x if not installed
python3 --version

# Run the demonstration
python3 booking_system.py

# Expected output:
# - Unsafe system shows overbooking
# - Safe system prevents overbooking
# - Semaphore system also prevents overbooking
```

B. HTML Visualization

```
# Simply open in browser
firefox index.html
# or
chrome index.html
```

```
# Or use simple HTTP server
python3 -m http.server 8000
# Then visit: http://localhost:8000
```

C. Test Different Scenarios

Modify these parameters to test:

```
# In booking_system.py
system = SafeBookingSystem(total_seats=5)   # Try different seat counts
passengers = [("User1", 3), ("User2", 3)]   # Try different booking requests

# In index.html
# Use the input fields to change:
# - Total seats
# - Number of concurrent users
```