Warehouse Robot Synchronization Simulation

Boussaid Abderrazag
Mohamed Ben Hadj Nasr

University of Tunis El Manar — Higher Institute of Computer Science
March 11, 2025

Introduction

This mini-project demonstrates a fundamental concept in concurrent programming: resource synchronization. It simulates multiple warehouse robots attempting to access a shared shelf, illustrating the problems that can occur without proper synchronization and how those problems can be solved using locks.

1 The Problem: Race Conditions in Concurrent Access

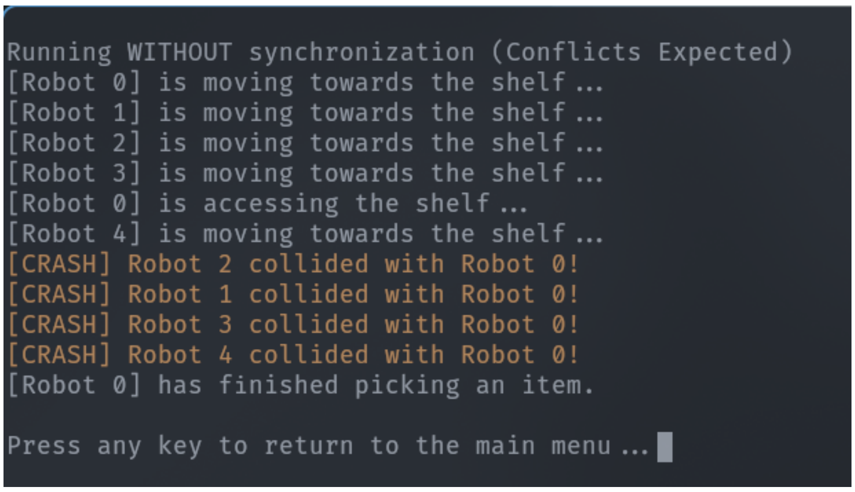
1.1 Scenario Overview

In the warehouse simulation, multiple robot threads simultaneously attempt to access a single shelf. Without proper synchronization, this creates a race condition - a situation where the behavior of the system depends on the relative timing of events, which can lead to unpredictable and undesirable outcomes.

1.2 Specific Issues Observed

When running the simulation without synchronization, the following problems occur:

- Resource Corruption: Multiple robots might believe the shelf is available simultaneously.
- Conflicting Access: Robots attempt to use the shelf while another robot is already using it.
- Collision Risk: Physical robots would crash into each other at the shelf location.
- Inconsistent State: The shelf_available and current_robot variables are read and modified by multiple threads without coordination.



```
Running WITHOUT synchronization (Conflicts Expected)
[Robot 0] is moving towards the shelf...
[Robot 1] is moving towards the shelf...
[Robot 2] is moving towards the shelf...
[Robot 3] is moving towards the shelf...
[Robot 0] is accessing the shelf...
[Robot 4] is moving towards the shelf...
[CRASH] Robot 2 collided with Robot 0!
[CRASH] Robot 1 collided with Robot 0!
[CRASH] Robot 3 collided with Robot 0!
[CRASH] Robot 4 collided with Robot 0!
[Robot 0] has finished picking an item.

Press any key to return to the main menu...
```

Figure 1: showcase How robots crash

1.3 Code Analysis of the Unsynchronized Implementation

Main.py

```
def access_shelf_without_lock(robot_id):
    global shelf_available, current_robot
    if shelf_available:
        shelf_available = False
        current_robot = robot_id
        time.sleep(random.uniform(1.0, 2.0))
        shelf_available = True
        current_robot = None
```

The critical issue occurs in the sequence between checking if shelf_available is True and setting it to False. If two robots check the condition almost simultaneously:

1. Robot A checks shelf_available (finding it True)
2. Before Robot A sets shelf_available = False, Robot B also checks shelf_available (also finding it True)
3. Both robots now believe they have exclusive access to the shelf
4. Both robots proceed to access the shelf simultaneously, causing a collision

This demonstrates a classic race condition where the outcome depends on the precise timing of thread execution.

2 The Solution: Mutex Locks for Synchronization

2.1 Synchronization Technique: Mutex Lock

The solution implements a mutex (mutual exclusion) lock using Python's threading.Lock(). A mutex ensures that only one thread can access a critical section of code at any given time, preventing race conditions.

2.2 Implementation Details

Main.py

```
def access_shelf_with_lock(robot_id):
    global shelf_available, current_robot
    time.sleep(random.uniform(0.5, 1.0))
    with lock:
        shelf_available = False
        current_robot = robot_id
        time.sleep(random.uniform(1.0, 2.0))
        shelf_available = True
        current_robot = None
```

The key improvements in this implementation are:

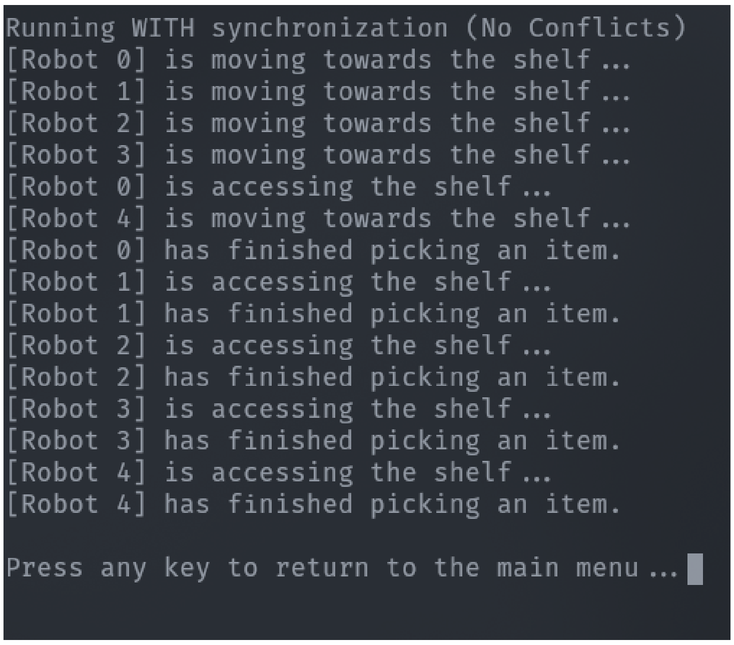
The "with lock:" statement creates a critical section that only one thread can enter at a time. When a robot thread enters this section, it acquires the lock, preventing other robots from entering until it releases the lock. The entire sequence of checking availability, marking the shelf as occupied, performing the task, and marking it as available again is protected within this critical section. The lock is automatically released when the robot thread exits the with block, even if an exception occurs.

2.3 How the Lock Resolves the Race Condition

When Robot A acquires the lock:

1. Robot A enters the critical section and sets shelf_available = False
2. Robot B attempts to acquire the lock but must wait
3. Robot A completes its task, sets shelf_available = True, and releases the lock
4. Only then can Robot B acquire the lock and access the shelf

This sequential access eliminates the possibility of collisions and ensures data consistency.



```
Running WITH synchronization (No Conflicts)
[Robot 0] is moving towards the shelf...
[Robot 1] is moving towards the shelf...
[Robot 2] is moving towards the shelf...
[Robot 3] is moving towards the shelf...
[Robot 0] is accessing the shelf...
[Robot 4] is moving towards the shelf...
[Robot 0] has finished picking an item.
[Robot 1] is accessing the shelf...
[Robot 1] has finished picking an item.
[Robot 2] is accessing the shelf...
[Robot 2] has finished picking an item.
[Robot 3] is accessing the shelf...
[Robot 3] has finished picking an item.
[Robot 4] is accessing the shelf...
[Robot 4] has finished picking an item.

Press any key to return to the main menu...
```

3 Technical Benefits and Considerations

3.1 Advantages of the Lock-Based Solution

- Deterministic Behavior: The system now behaves predictably regardless of thread timing.
- Data Integrity: Shared variables are protected from simultaneous access.
- Safety: In a physical implementation, robot collisions would be prevented.
- Simplicity: The threading.Lock provides a straightforward synchronization mechanism.