ASSIGNMENT No. 3

Complex Computing Problem

Hafsa Hafeez Siddiqui
02-136212-026

Operating System
CSC-320

Scenario:

Let's consider the following scenario:

1. An operating system provides an online ticket reservation system for a concert that has only one seat left.
2. User A and User B both access the reservation system concurrently.
3. User A's request to reserve the last seat is received by the operating system
4. User B's request to reserve the last seat is also received by the operating system.
5. The operating system checks the availability of the seat and finds that it is still available forboth User A and User B since neither reservation has been processed yet.
6. The operating system processes User A's request and reserves the seat for User A.
7. The operating system then processes User B's request and reserves the seat for User B as well, unaware that User A has already reserved the seat.
8. As a result, both User A and User B are assigned the same seat, leading to a conflict and potentially causing disruption during the event

Analysis and Solution:

- **Initially discuss the concept of concurrency and context switching concept in it.**
- **Then discuss the concept of race condition in concurrency and its solution.**
- **Lastly, solve the values for User A and User B**

The above-mentioned scenario, the race condition arises when both User A and User B attempt to reserve the last seat concurrently and context switching occurs when the operating system switches between User A's and User B's reservation requests, leading to conflicts and potential disruptions during the event.

The operating system can use atomic operations to perform the seat reservation. Atomic operations and locks ensure that a series of operations are treated as a single, indivisible unit. In this case, the reservation process would involve checking the availability of the seat and reserving it in a single atomic operation. This ensures that only one user can successfully complete the reservation.

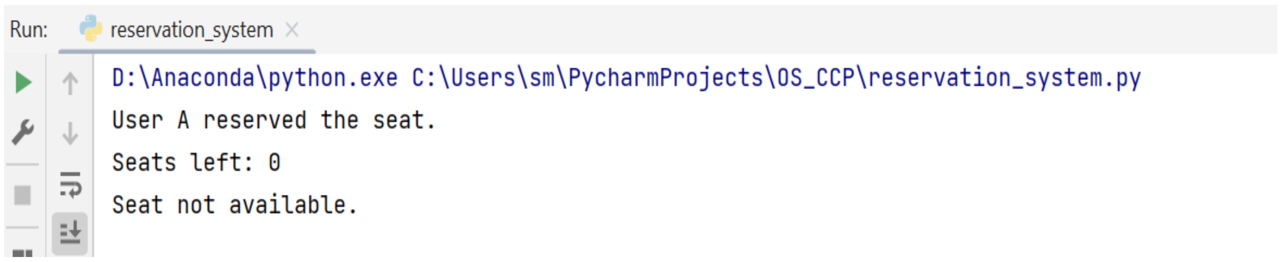
Pseudocode**:**

```
// Declare atomic variable for seat availability
AtomicBoolean seatAvailable = new AtomicBoolean(true)

// Function for reserving a seat
function reserveSeat(user):
    if seatAvailable.compareAndSet(true, false):
        // Seat is available, reserve it for the user
        print "Seat reserved for User ", user
    else:
        // Seat is already reserved by another user
        print "Seat not available for User ", user

// Create threads for User A and User B
Thread userA = new Thread(function() {
    reserveSeat("A")
})
```

```
Thread userB = new Thread(function() {
   reserveSeat("B")
})

userA.start()
userB.start()

userA.join()
userB.join()
```

Source Code:

```python
from threading import Thread, Lock

seats_left = 1
lock = Lock()

# Function for reserving seat
def reserve_seat(user):
    global seats_left
    global lock

    lock.acquire()  # Acquire the lock before accessing the shared resource
    if seat_available():
        seats_left = seats_left - 1
        print(f"User {user} reserved the seat.")
        print(f"Seats left: {str(seats_left)}")
    else:
        print("Seat not available.")
    lock.release()  # Release the lock after accessing the shared resource

# Function to check seat availability
def seat_available():
    global seats_left
    # Check seat availability logic
    return True if seats_left > 0 else False

# Create threads for User A and User B
thread_a = Thread(target=reserve_seat, args=("A",))
thread_b = Thread(target=reserve_seat, args=("B",))

# Start the threads
thread_a.start()
thread_b.start()

# Wait for the threads to finish
thread_a.join()
thread_b.join()
```



```
D:\Anaconda\python.exe C:\Users\sm\PycharmProjects\OS_CCP\reservation_system.py
User A reserved the seat.
Seats left: 0
Seat not available.
```

Comparative Approach:

Analyze the given conditions and identify which User will reserve seats first, keep in viewrace condition

Without extra information or specific synchronisation procedures in place, it is impossible to predict who will reserve the seat first given the current circumstances and the possibility of a race condition. When a system's outcome depends on the relative timing or interleaving of several concurrent activities, it is said to be in a race condition.

In this case, there is no specific synchronisation mechanism indicated to prevent race conditions, and User A and User B are both simultaneously attempting to reserve the remaining seat. Without adequate synchronisation, many factors, such as the speed of execution, system scheduling, and thread interleaving, might affect the sequence in which User A and User B's reservation requests are handled and the seat is reserved. It is not deterministic, and many system runs may provide various outcomes.

It is conceivable for User A and User B to both notice that the seat is still accessible in the absence of synchronisation methods like locks, atomic operations, or transactions, resulting in a race condition where they both book the seat. Due to the possibility of both users being given the same seat, this can lead to disputes and interruptions during the event.

It is essential to utilise appropriate synchronisation strategies that permit only one user to successfully reserve the seat while restricting concurrent access and conflicting actions on shared resources in order to guarantee a deterministic and consistent conclusion and avoid race circumstances.

Give a comparative approach using the concept of atomic variable that provides a solution to race conditions in critical sections along with the explanation of process synchronization in OS.

Atomic variables provide a solution to race conditions in critical sections by ensuring that operations on shared variables are atomic, indivisible, and non-interruptible. Atomic operations are performed on these variables, guaranteeing that only one thread can access and modify the variable at a time. When it comes to race conditions in critical sections, atomic variables can be used to prevent conflicts between concurrent threads accessing shared resources.

By using atomic operations such as compare-and-swap or test-and-set, concurrent threads can atomically check the state of a variable and modify it in a single, uninterrupted operation. This ensures that only one thread can successfully update the variable while other threads are prevented from accessing it simultaneously.

Process Synchronization refers to the techniques and mechanisms used to coordinate and control the execution of multiple processes or threads to avoid race conditions and ensure the correct order of operations. It involves coordination between processes. The operating system provides various synchronization primitives and mechanisms to achieve process synchronization.

Locks, such as mutex locks, provide mutual exclusion by allowing only one process or thread to acquire the lock at a time. This ensures that critical sections of code are executed by a single process while others wait until the lock is released.

In summary, atomic variables provide a solution to race conditions in critical sections by guaranteeing atomic operations on shared variables. Process synchronization mechanisms in operating systems, such as locks, ensure orderly execution and coordination between processes or threads, preventing conflicts and maintaining data integrity.