ASSIGNMENT4: THREADS

Evaluate each synchronization mechanism and determine the most appropriate one to ensure thread safety for the class. Provide insights and justification in a PDF or text document.

Synchronization Mechanism Selection for Ensuring Thread Safety

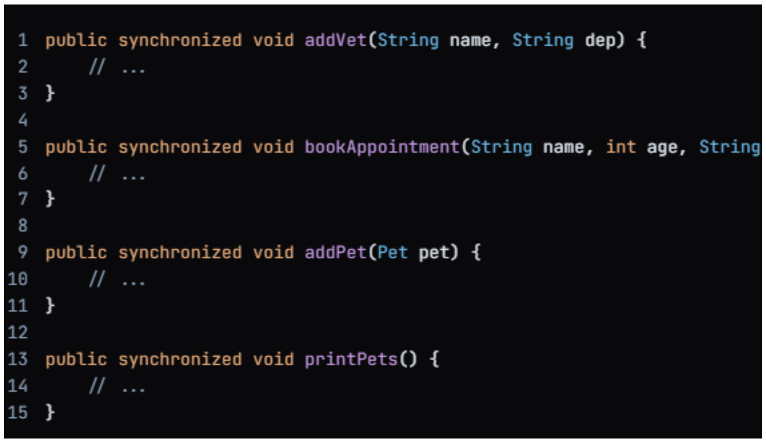
In the context of multithreaded programming, ensuring thread safety is crucial to prevent data inconsistencies and unexpected behavior. Given the identified segments lacking thread safety: Synchronized Methods or Blocks, Locks (ReentrantLock), and Concurrent Collections.

Race Condition

A race condition occurs when multiple processes access and process the same data concurrently, and the outcome depends on the order in which the access takes place [2]. A race condition is a situation that may occur inside a critical section. This happens when the result of multiple thread execution in the critical section differs according to the order in which the threads execute. Race conditions in critical sections can be avoided if the critical section is treated as an atomic instruction. Also, proper thread synchronization using locks or atomic variables can prevent race conditions.

Synchronized Methods or Blocks

Using the **`synchronized`** keyword in Java ensures that only one thread can execute a synchronized method or block at a time. This mechanism provides i**ntrinsic locking**, simplifying the implementation. However, it may lead to potential contention if many threads access the same synchronized block frequently.



```java
public synchronized void addVet(String name, String dep) {
    // ...
}

public synchronized void bookAppointment(String name, int age, String
    // ...
}

public synchronized void addPet(Pet pet) {
    // ...
}

public synchronized void printPets() {
    // ...
}
```

These methods are synchronized to ensure that only one thread can execute them at a time, preventing concurrent access issues.

Pros:

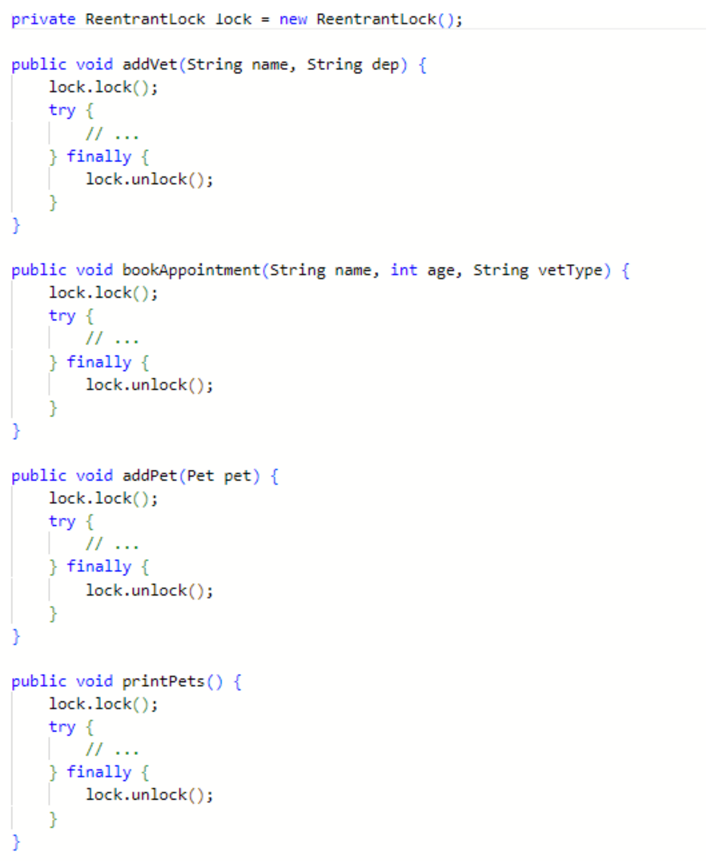
1. **Easy to implement**: Synchronized methods or blocks are straightforward to implement, requiring minimal code changes.
2. **Intrinsic locking**: The JVM handles locking and unlocking, reducing the risk of manual locking errors.

Cons:

3. **Coarse-grained locking**: Synchronized methods or blocks lock the entire object, which can lead to performance bottlenecks.
4. **Limited flexibility**: Synchronized methods or blocks do not provide explicit control over locking behavior.
5. **Potential contention**: Frequent access to synchronized blocks can cause thread contention, leading to performance degradation.

Locks (ReentrantLock)

ReentrantLocks offers more flexibility than synchronized methods or blocks. They allow for explicit locking and unlocking, enabling finer-grained control over locking behavior.



```java
private ReentrantLock lock = new ReentrantLock();

public void addVet(String name, String dep) {
    lock.lock();
    try {
        // ...
    } finally {
        lock.unlock();
    }
}

public void bookAppointment(String name, int age, String vetType) {
    lock.lock();
    try {
        // ...
    } finally {
        lock.unlock();
    }
}

public void addPet(Pet pet) {
    lock.lock();
    try {
        // ...
    } finally {
        lock.unlock();
    }
}

public void printPets() {
    lock.lock();
    try {
        // ...
    } finally {
        lock.unlock();
    }
}
```

In the above screenshot, a ReentrantLock is used to provide finer-grained control over locking behavior. The lock is acquired before accessing shared data structures and released afterwards to prevent concurrent access issues.

Pros:

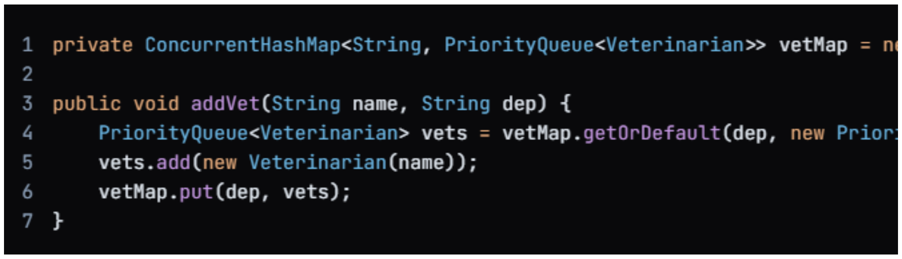
6. **Finer-grained control**: ReentrantLocks provide explicit control over locking behavior, enabling more efficient locking strategies.
7. **Flexibility**: ReentrantLocks supports features like fairness policies and timeout mechanisms, allowing for customized locking behavior.
8. **Performance**: ReentrantLocks can outperform synchronized methods or blocks in certain scenarios.

Cons:

9. **More complex implementation**: ReentrantLocks requires manual locking and unlocking, increasing the risk of manual locking errors.
10. **Error-prone**: Incorrect usage of ReentrantLocks can lead to deadlocks or other synchronization issues.

Concurrent Collections

Concurrent collections, such as `ConcurrentHashMap` or `ConcurrentLinkedQueue`, provide built-in thread safety for concurrent access. They internally handle synchronization to ensure thread safety without explicit locking from the user.



```java
private ConcurrentHashMap<String, PriorityQueue<Veterinarian>> vetMap = ne

public void addVet(String name, String dep) {
    PriorityQueue<Veterinarian> vets = vetMap.getOrDefault(dep, new Prior
    vets.add(new Veterinarian(name));
    vetMap.put(dep, vets);
}
```

As shown above: A ConcurrentHashMap is used to store the vetMap, which provides built-in thread safety for concurrent access. This eliminates the need for explicit synchronization or locking.

Pros:

11. **Built-in thread safety**: Concurrent collections provide thread safety without requiring explicit locking or synchronization.
12. **High-performance**: Concurrent collections are optimized for concurrent access, providing high-performance and low-latency operations.
13. **Easy to use**: Concurrent collections are simple to use, requiring minimal code changes.

Cons:

14. **Limited customization**: Concurrent collections provide a fixed locking strategy, limiting customization options.
15. **Overhead**: Concurrent collections may introduce additional overhead due to internal synchronization mechanisms.

Selection Criteria

When selecting a synchronization mechanism, consider the following criteria:

16. **Thread safety**: The mechanism must ensure thread safety for the class.
17. **Performance**: The mechanism should minimize performance overhead and contention.
18. **Flexibility**: The mechanism should provide flexibility in locking behavior and customization options.

19. **Ease of implementation**: The mechanism should be easy to implement and maintain.

Suggestions for Implementation:

1. **Synchronized Blocks for Critical Sections:**
    - Use synchronized blocks to ensure thread safety when accessing shared data structures like `vetMap` and `pets`.
    - Wrap critical sections of code with synchronized blocks to prevent concurrent access issues.

2. **ReentrantLock for Fine-grained Control:**
    - Consider using ReentrantLocks for more complex synchronization requirements or scenarios where finer-grained locking is necessary.
    - Utilize try-final blocks to ensure proper unlocking even in case of exceptions.
3. **Concurrent Collections for Simplified Management:**
    - Replace standard collections with concurrent counterparts like `ConcurrentHashMap` for `vetMap`.
    - Concurrent collections provide built-in thread safety, reducing the need for explicit synchronization.

References:

[1] Process Synchronization. (2023, October 4). https://www.geeksforgeeks.org/process-synchronization/

[2] locks and synchronization.https://web.mit.edu/6.005/www/fa15/classes/23-locks/