Dungeon LFG System

Baccay, Dominic Luis M.

STDISCM S12

Deadlock and Starvation Risks

Potential Deadlock Scenarios:

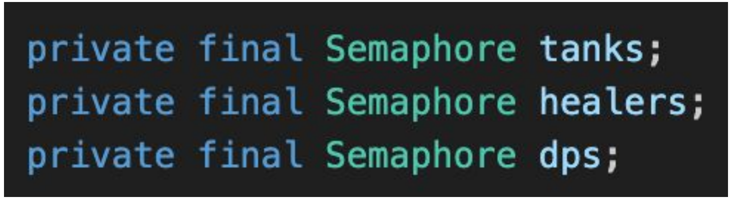
- Party formation could deadlock if tank/healer/DPS resources are acquired incompletely
- Dungeon instances waiting for parties while parties wait for available instances
- Threads waiting indefinitely on synchronization objects that are never released

Starvation Risks:

- Parties might never form if resources are continually acquired by competing threads
- Dungeon instances might remain idle if party formation logic favors certain instances
- The main thread could wait indefinitely if CountDownLatch is never fully counted down

Synchronization Mechanisms - Resource Management

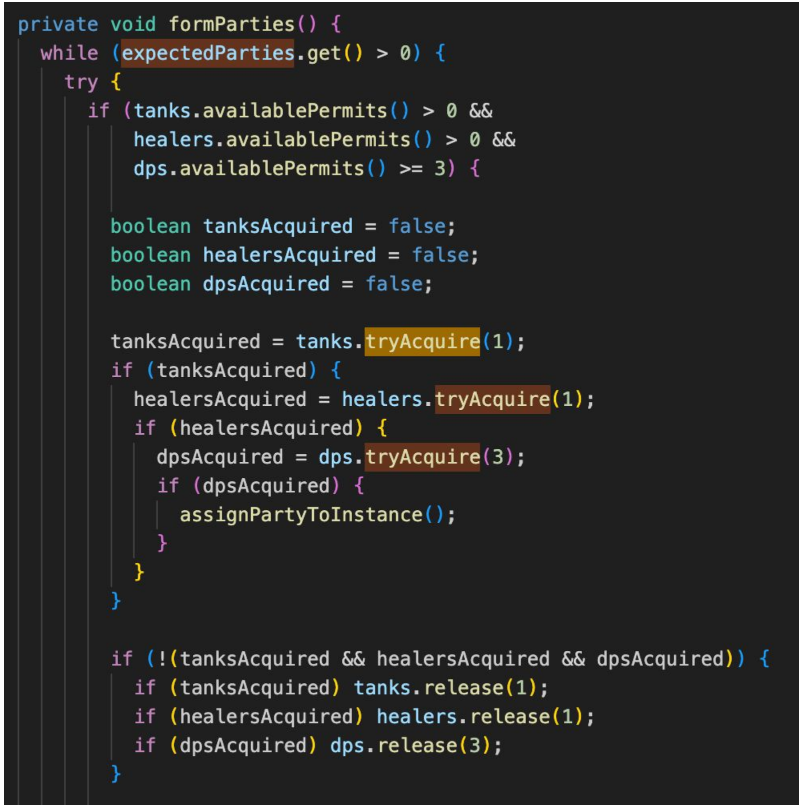
Controlling Player Resource Access with Semaphores



```
private final Semaphore tanks;
private final Semaphore healers;
private final Semaphore dps;
```

Semaphores are used to control access to limited player resources. They allow multiple threads to acquire resources up to a fixed limit, making them ideal for managing role-based player pools, such as tanks, healers, and DPS, with concurrent access.

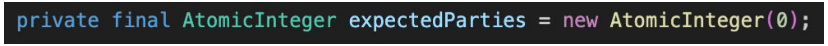
The implementation strategy involves using the **tryAcquire()** method for atomic resource acquisition seen in the formParties() function in DungeonManager. Incomplete acquisitions are explicitly released to prevent deadlock. The system will then attempt to form complete parties by ensuring there is 1 tank, 1 healer, and 3 DPS roles.



```
private void formParties() {
  while (expectedParties.get() > 0) {
    try {
      if (tanks.availablePermits() > 0 &&
          healers.availablePermits() > 0 &&
          dps.availablePermits() >= 3) {

        boolean tanksAcquired = false;
        boolean healersAcquired = false;
        boolean dpsAcquired = false;

        tanksAcquired = tanks.tryAcquire(1);
        if (tanksAcquired) {
          healersAcquired = healers.tryAcquire(1);
          if (healersAcquired) {
            dpsAcquired = dps.tryAcquire(3);
            if (dpsAcquired) {
              assignPartyToInstance();
            }
          }
        }

        if (!(tanksAcquired && healersAcquired && dpsAcquired)) {
          if (tanksAcquired) tanks.release(1);
          if (healersAcquired) healers.release(1);
          if (dpsAcquired) dps.release(3);
        }
```

Synchronization Mechanisms - Party Management

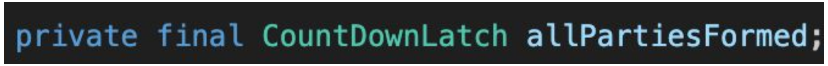
AtomicInteger for Expected Parties:



```
private final AtomicInteger expectedParties = new AtomicInteger(0);
```

The **AtomicInteger** is used to track how many parties can still be formed. It provides thread-safe operations without the need for explicit locking. The key operations include using decrementAndGet() to decrease the count when a party is formed and getAndSet() to atomically update the expected number of parties when expectations change.

CountDownLatch for Completion Signaling:

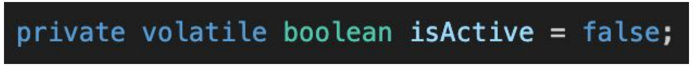


```
private final CountDownLatch allPartiesFormed;
```

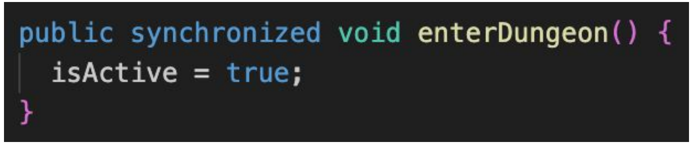
The **CountDownLatch** is used to allow the main thread to wait until all possible parties have been formed. It is ideal for this scenario because it is a one-way signaling mechanism, perfect for "wait until done" situations. The latch prevents deadlock by dynamically adjusting the countdown whenever expectations change, ensuring synchronization and proper flow.

Synchronization Mechanisms - Instance Management

Volatile and Synchronized for Instance Status:



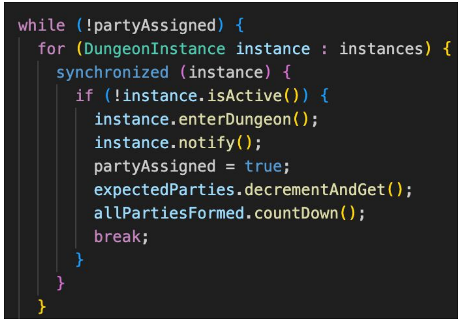
```java
private volatile boolean isActive = false;
```



```java
public synchronized void enterDungeon() {
  isActive = true;
}
```

The **volatile** keyword is used to ensure that changes to the isActive state are visible across all threads. The synchronized method, enterDungeon(), is used to safely manage transitions of the instance's state, ensuring that changes to the isActive variable happen without race conditions. This combination ensures that state changes are properly handled in a multithreaded environment.

Wait/Notify Pattern:

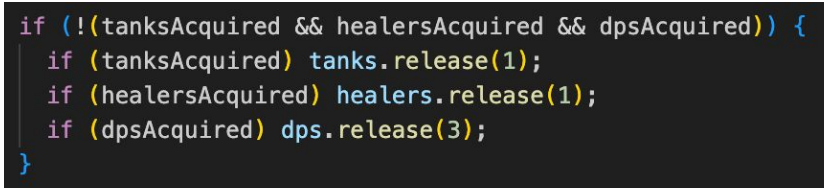


```java
while (!partyAssigned) {
  for (DungeonInstance instance : instances) {
    synchronized (instance) {
      if (!instance.isActive()) {
        instance.enterDungeon();
        instance.notify();
        partyAssigned = true;
        expectedParties.decrementAndGet();
        allPartiesFormed.countDown();
        break;
      }
    }
  }
```

The **wait/notify** is used to signal instance threads when parties are assigned. The synchronized block ensures that the instance’s state is checked and modified atomically, and notify() is called to wake up any waiting threads. This approach prevents starvation by giving each instance a fair chance to process parties, ensuring efficient thread coordination.

Failure Handling

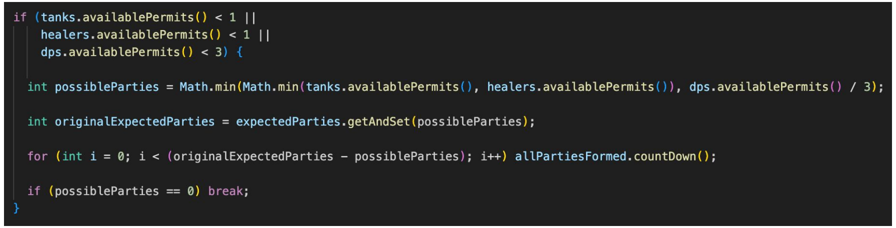
Resource Acquisition with Rollback:



```
if (!(tanksAcquired && healersAcquired && dpsAcquired)) {
  if (tanksAcquired) tanks.release(1);
  if (healersAcquired) healers.release(1);
  if (dpsAcquired) dps.release(3);
}
```

If a complete set of resources (1 tank, 1 healer, 3 DPS) cannot be acquired, any successfully acquired resources are released. This ensures no partial parties form, preventing resource leaks and deadlocks. It ensures all-or-nothing acquisition, maintaining resource integrity and mimicking database transaction rollbacks.

Dynamic Party Expectation Adjustment:



```
if (tanks.availablePermits() < 1 ||
    healers.availablePermits() < 1 ||
    dps.availablePermits() < 3) {

  int possibleParties = Math.min(Math.min(tanks.availablePermits(), healers.availablePermits()), dps.availablePermits() / 3);

  int originalExpectedParties = expectedParties.getAndSet(possibleParties);

  for (int i = 0; i < (originalExpectedParties - possibleParties); i++) allPartiesFormed.countDown();

  if (possibleParties == 0) break;
}
```

The number of possible parties is dynamically recalculated based on resource availability. This update adjusts expectations and signals waiting threads when parties can't be formed, preventing deadlock. The **CountDownLatch** is used efficiently to track party formation and impossibilities, ensuring atomic state updates with getAndSet().