PROBLEM SET **NO. 2**

CAASI S24

STDISCM AY 2024-2025 | DLSU

requirements

- Only ***n*** dungeon instances active concurrently.
- Standard party composition: **1 tank**, **1 healer**, **3 DPS**
- Handle process synchronization **without deadlock** or **starvation**
- Randomized dungeon run time between **t1** and **t2 seconds**

potential deadlock issues and solution

- **Scenario:** A party begins forming but lacks a required role—for example, if a thread locks resources while waiting for a healer, while other threads also hold locks on the available tanks or DPS players. This can lead to a situation where no thread can complete its party because the required resources are held by different threads, effectively causing the system to halt.

- **Solution:** To avoid this, the design ensures that the entire check for resource availability (1 tank, 1 healer, and 3 DPS) is done atomically under the protection of a single mutex (playerMutex). This way, a thread will only proceed to lock and deduct player counts if all necessary roles are available, preventing partial allocations that could lead to deadlock.

potential starvation issues and solution

- **Scenario:** Certain threads or dungeon instances continually get access to the required players while others are left waiting indefinitely. For instance, if new players continually arrive or if resource allocation is biased toward certain instances, some parties might never get formed.

- **Solution:** Implement a fair queuing mechanism where the resource check and deduction are handled uniformly across all threads. By using the mutex to protect the player count check, every thread is guaranteed an equal opportunity to form a party as long as the required players are present. This fairness in accessing shared resources prevents any one thread from monopolizing the available players, thereby avoiding starvation.

synchronization mechanisms employed

- Mutexes:
  - **coutMutex:** Protects console output to prevent garbled prints.
  - **playerMutex:** Ensures atomic deduction of player counts when forming parties.
  - **statsMutex:** Manages access to and updates of the dungeon instance statistics.

```
mutex coutMutex;
mutex playerMutex;
mutex statsMutex;
```

synchronization mechanisms employed

- Threading Model:
  - Each dungeon instance runs on a dedicated thread.
  - Threads simulate work by sleeping for a randomized period (within t1 to t2 seconds).

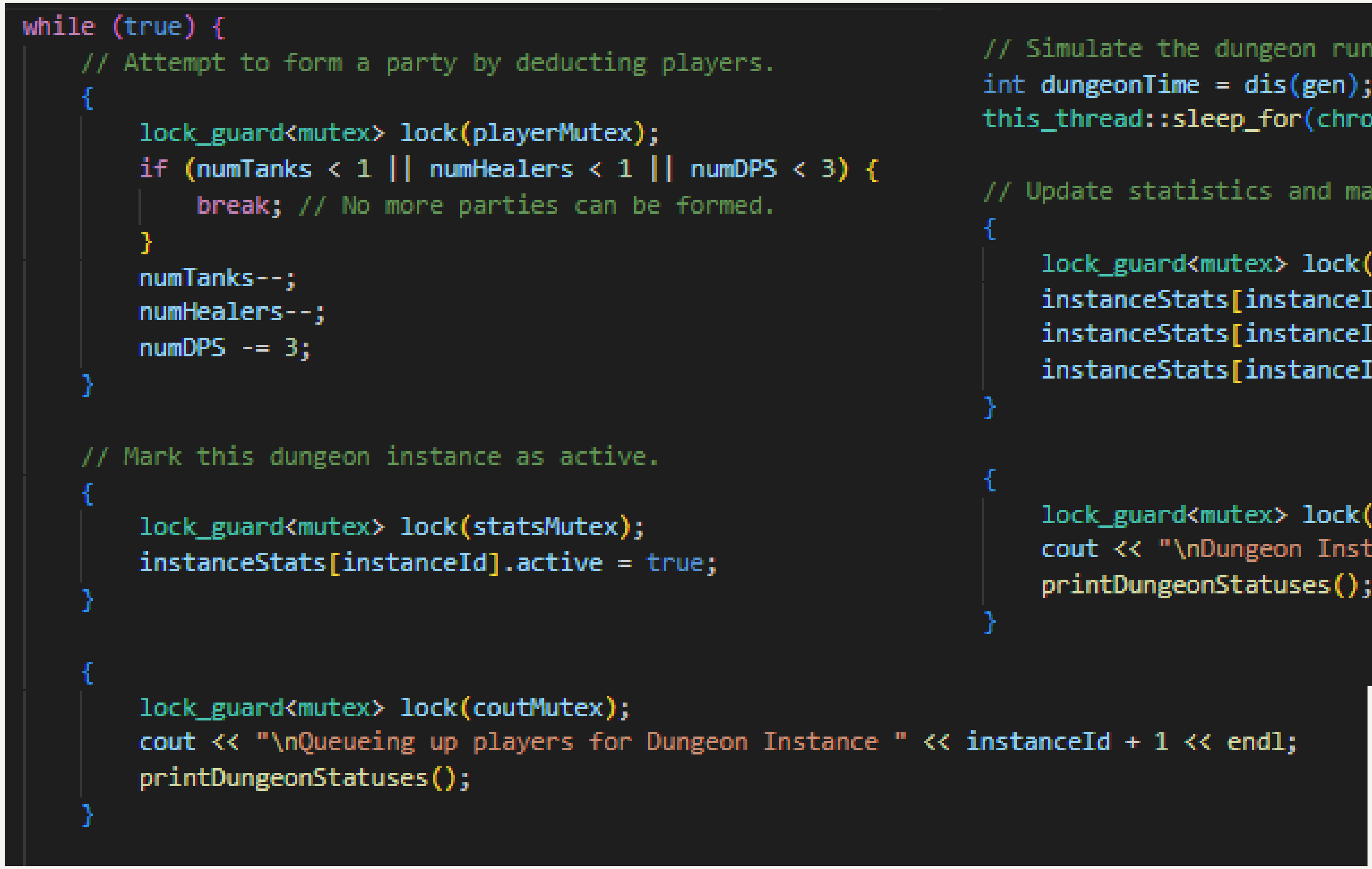
```
vector<thread> dungeonThreads;
for (int i = 0; i < numDungeons; i++) {
    dungeonThreads.emplace_back(queueParty, i);
}
```

```
int dungeonTime = dis(gen);
this_thread::sleep_for(chrono::seconds(dungeonTime));
```

synchronization mechanisms employed

- Design Outcome:
  - **Deadlock Avoidance:** By avoiding circular wait through careful ordering of locks.
  - **Fair Resource Distribution:** Prevents starvation by ensuring all threads eventually access shared resources.

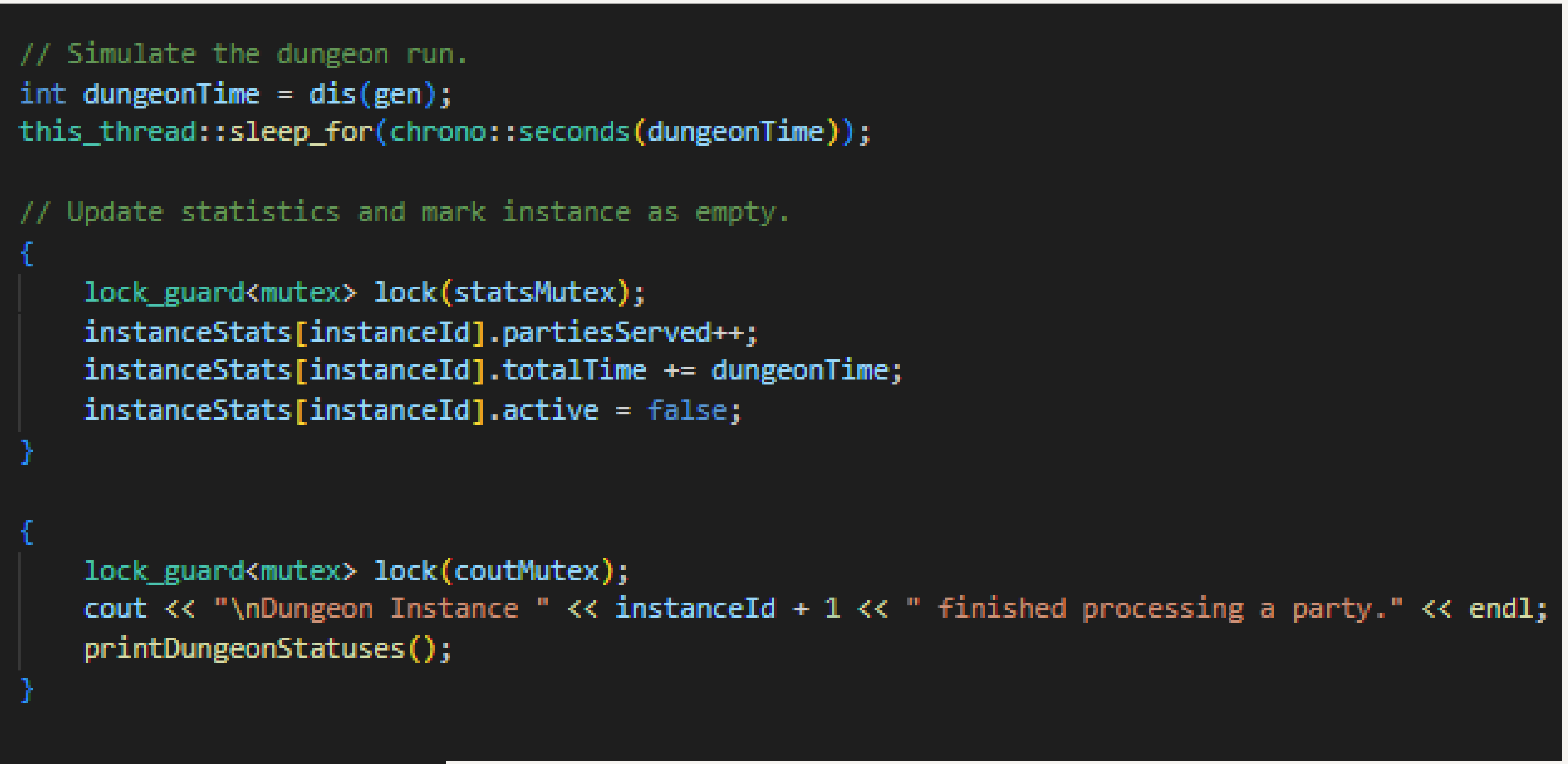
synchronization mechanisms employed



```cpp
while (true) {
    // Attempt to form a party by deducting players.
    {
        lock_guard<mutex> lock(playerMutex);
        if (numTanks < 1 || numHealers < 1 || numDPS < 3) {
            break; // No more parties can be formed.
        }
        numTanks--;
        numHealers--;
        numDPS -= 3;
    }

    // Mark this dungeon instance as active.
    {
        lock_guard<mutex> lock(statsMutex);
        instanceStats[instanceId].active = true;
    }

    {
        lock_guard<mutex> lock(coutMutex);
        cout << "\nQueueing up players for Dungeon Instance " << instanceId + 1 << endl;
        printDungeonStatuses();
    }
```



```cpp
// Simulate the dungeon run.
int dungeonTime = dis(gen);
this_thread::sleep_for(chrono::seconds(dungeonTime));

// Update statistics and mark instance as empty.
{
    lock_guard<mutex> lock(statsMutex);
    instanceStats[instanceId].partiesServed++;
    instanceStats[instanceId].totalTime += dungeonTime;
    instanceStats[instanceId].active = false;
}

{
    lock_guard<mutex> lock(coutMutex);
    cout << "\nDungeon Instance " << instanceId + 1 << " finished processing a party." << endl;
    printDungeonStatuses();
}
```