Process Synchronization

STDISCM S17

Atasha Dominique C. Pidlaoan

Problem Overview

WHAT WE WANT TO SOLVE

Objective: Manage dungeon queuing efficiently with process synchronization

Constraints:

- Max n concurrent dungeon instances
- Parties must contain 1 Tank, 1 Healer, 3 DPS
- Avoid deadlock & starvation
- Random dungeon duration t1 - t2 seconds

Solution: Implement thread synchronization for fairness & concurrency

Deadlock Explanation

Deadlock is a state where two or more processes are **permanently waiting** for each other to release a locked resource, causing the entire system to halt.

Potential Scenario:

- A dungeon instance successfully locks initial player roles (e.g., **Tank** and **DPS** players).
- It then waits indefinitely for a missing role (**Healer**), holding the locked Tank/DPS resources hostage.
- These locked resources are unavailable to any other waiting party, leading to **system starvation**.

Solution: Ensure resource availability before forming a party

Starvation

Starvation is when a ready player (process) is **repeatedly ignored** and cannot access needed resources (party slot), even if the resource is available.

Potential Scenario

- The system constantly forms new parties, but if the logic **always favors fresh arrivals** or certain role combinations, older players are **perpetually overlooked** in the queue.
- The ignored player is ready but **never selected** for a party, forcing them to wait indefinitely.

Strategy: Implement a **Fair Queuing Mechanism**.

Result: The assignment logic ensures **all waiting players are considered equally** and will eventually be assigned a party, eliminating the risk of indefinite waiting.

Synchronization Mechanisms Used

MUTEX

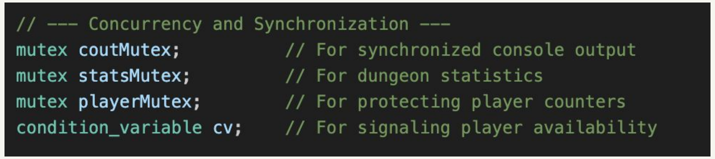
- Prevents **race conditions** on shared data.
- Ensures **only one thread** modifies critical data (like player counts) at a time.
- Guarantees **data integrity**.

THREADING

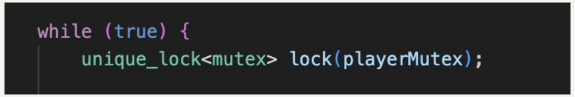
- Enables **concurrent handling** of all dungeon instances.
- Each dungeon runs on its own **separate thread**.
- Maximizes system **throughput**.

Code Implementation

Mutex Usage:

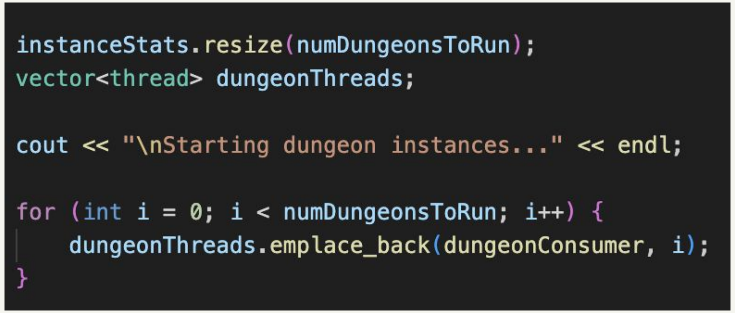


```
// --- Concurrency and Synchronization ---
mutex coutMutex;          // For synchronized console output
mutex statsMutex;         // For dungeon statistics
mutex playerMutex;        // For protecting player counters
condition_variable cv;    // For signaling player availability
```



```
  while (true) {
      unique_lock<mutex> lock(playerMutex);
```

Threading for Dungeon Instances:

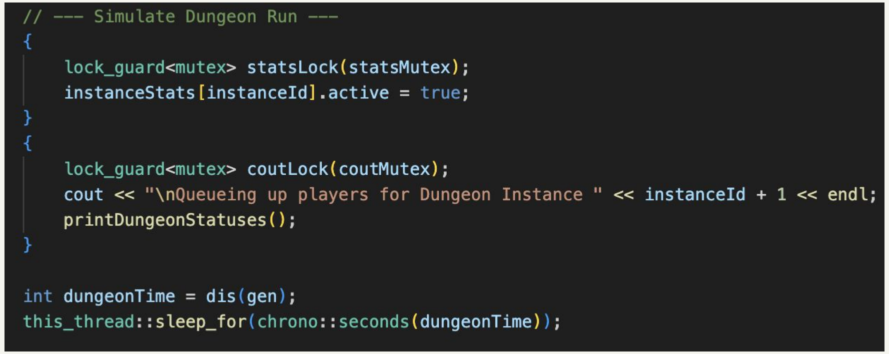


```
instanceStats.resize(numDungeonsToRun);
vector<thread> dungeonThreads;

cout << "\nStarting dungeon instances..." << endl;

for (int i = 0; i < numDungeonsToRun; i++) {
    dungeonThreads.emplace_back(dungeonConsumer, i);
}
```

Randomized Dungeon Time Simulation:



```
// --- Simulate Dungeon Run ---
{
    lock_guard<mutex> statsLock(statsMutex);
    instanceStats[instanceId].active = true;
}
{
    lock_guard<mutex> coutLock(coutMutex);
    cout << "\nQueueing up players for Dungeon Instance " << instanceId + 1 << endl;
    printDungeonStatuses();
}

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