P2 - Looking for Group Synchronization

S12 - Kaitlyn Tighe

Compilation Setup

- Compile using g++
  - g++ main.cpp Matchmaker.cpp -o myProgram -std=c++11 -pthread
    - main.cpp Matchmaker.cpp: all source files
    - -o myProgram: output file name
    - -std=c++11: enables modern C++ features
    - -pthread: enables threading support
- Run
  - ./myProgram

NOTE: this project was created in MacOS. The project may not run as intended on Windows/LinuxOS

Structure

- **Folder Structure**
  - **Main.cpp:** program entry point
  - **Matchmaker.cpp**
    - Input validation
    - Player queue management
    - Threaded dungeon simulation
    - Party creation and logging
  - **Matchmaker.h**
    - Function and variable declaration
- **Used C++ as Language**
  - For familiarity as used in OPESY Class
  - For the multithreaded support such as thread/mutex
  - Control over threads for race conditions and preventing deadlocks

Input Validation

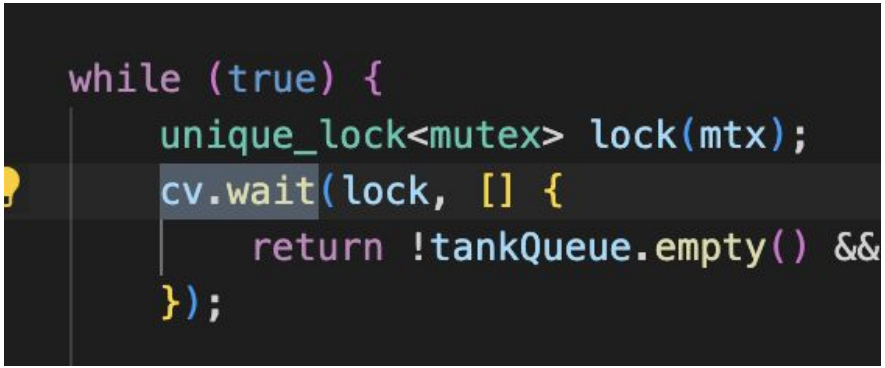
- Max instances must be > 0
- At least 1 tank, 1 healer, and 3 DPS
- Total players must be enough to form 1 party
- Min time > 0
- Max time > 0 and must be ≥ min time
- Shows warning if:
  - Max instances > 100
  - DPS is much higher than tanks + healers

Possible deadlock and starvation

- **Deadlock Avoidance**
  - All required players for a party are acquired together while holding the lock to ensure safe and consistent access
  - No circular waiting or holding partial resources; prevents classic deadlock scenario
- **Starvation Prevention**
  - All players enter a queue, follows FIFO
  - No preference is given to certain player types or threads
  - Condition variables ensure fair wake-ups when enough players are available

Synchronization mechanisms used to solve the problem

- cv.wait
  - To suspend threads until enough players are available and instances are free
  - Prevents busy-waiting, improves efficiency



Synchronization mechanisms used to solve the problem

- std::thread:
  - Spawns a new thread for each party run, simulating real-time instance execution

```
std::thread([party] {
    runDungeon(party.instanceID, party.id);
}).detach();
```

Synchronization mechanisms used to solve the problem

- lock_guard & unique_lock:
  - lock_guard is a simpler way to lock a section of code so only one thread can access shared data, and it automatically unlocks when that section finishes.
  - unique_lock makes sure that only one part of the program can add players to the queue at a time, so nothing gets messed up when multiple threads are running.

```
lock_guard<mutex> lock(mtx);
instanceStatus[instanceID] = "active";
cout << "\n[" << getCurrentTime() << "] [Instance " << instanceID + 1
     << "] Serving Party " << partyID << "   → ACTIVE" << endl;
```

```
void addPlayersToQueue() {
    for (int i = 0; i < numTanks; i++) {
        unique_lock<mutex> lock(mtx);
        tankQueue.push(i + 1);
        cv.notify_all();
    }
    for (int i = 0; i < numHealers; i++) {
        unique_lock<mutex> lock(mtx);
        healerQueue.push(i + 1);
        cv.notify_all();
    }
    for (int i = 0; i < numDPS; i++) {
        unique_lock<mutex> lock(mtx);
        dpsQueue.push(i + 1);
        cv.notify_all();
    }
}
```

Synchronization mechanisms used to solve the problem

- mutex:
  - Protects access to shared queues and instance states.
  - Used in with unique_lock and lock_guard
  - Used in:
    - addPlayersToQueue() – when adding players to queues
    - partyManager() – when checking and forming parties
    - runDungeon() – when updating instance status and stats
    - showSummary() – when showing final results