Looking for Group Synchronization

Problem Set 2

ABENOJA, Amelia Joyce L.
STDISCM - S14

```cpp
void DungeonManager::processParties() {
    std::vector<std::thread> instanceThreads;

    // Create threads for each instance
    for (auto& instance : instances) {
        instanceThreads.emplace_back([this, instance]() {
            while (true) {
                auto [partyID, duration] = fetchNextPartyFromQueue();
                if (partyID == -1) return;                    // Sto

                instanceSemaphore.acquire();                  // Acq
                instance->executeParty(partyID, duration);
                instanceSemaphore.release();                  // Rel
            }
        });
    }

    // Sleep for 1 second before stopping processing
    std::this_thread::sleep_for(std::chrono::seconds(1));
    {
        std::lock_guard<std::mutex> lock(queueMutex);
        stopProcessing = true;
    }
    instanceNotifier.notify_all(); // Wake up all waiting threads

    // Join all threads before printing summary
    for (auto& thread : instanceThreads) {
        if (thread.joinable()) {
            thread.join();
        }
    }
}
```

Possible Deadlock

Occurs if all dungeon instances are waiting indefinitely for parties to execute, but the necessary resources are not being released, preventing the execution of the program as a whole.

In the **processParties()** function of the **DungeonManager** class, each instance thread acquires resources before executing a party. If a thread acquires a resource but fails to release it after usage, other threads waiting for the same resource will be blocked indefinitely, leading to deadlock.

```cpp
std::pair<int, int> DungeonManager::fetchNextPartyFromQueue()
{
    std::unique_lock<std::mutex> lock(queueMutex);
    instanceNotifier.wait(lock, [this] { return stopProcessing || !partyQueue.empty(); });

    if (stopProcessing && partyQueue.empty()) return { -1, -1 }; // Return invalid pair to

    auto [partyID, duration] = partyQueue.top();
    partyQueue.pop();
    return { partyID, duration };
}
```

Possible Starvation

Occurs when a party is indefinitely delayed because other parties are continuously prioritized and scheduled before it, preventing it from being executed.

In the **fetchNextPartyFromQueue()** function, the party at the top of the queue is immediately selected for execution. If higher-priority parties (e.g., shorter-duration ones in a **Shortest Job First** approach) keep arriving, lower-priority parties may experience prolonged delays or may never be executed.

1. **Semaphores Mutual Exclusion (Mutex)**
   - Used to ensure that dungeon instances (threads) do not hold resources longer than necessary.
   - Prevents multiple threads from accessing shared resources simultaneously, allowing fair resource distribution.
   - Ensures that once a thread completes its task, it releases resources for other threads to use.

Synchronization Mechanisms

```
void DungeonManager::processParties() {
    std::vector<std::thread> instanceThreads;

    // Create threads for each instance
    for (auto& instance : instances) {
        instanceThreads.emplace_back([this, instance]() {
            while (true) {
                auto [partyID, duration] = fetchNextPartyFromQueue();
                if (partyID == -1) return;                          // Sto

                instanceSemaphore.acquire();                        // Acq
                instance->executeParty(partyID, duration);
                instanceSemaphore.release();                        // Rel
            }
        });
    }

    // Sleep for 1 second before stopping processing
    std::this_thread::sleep_for(std::chrono::seconds(1));
    {
        std::lock_guard<std::mutex> lock(queueMutex);
        stopProcessing = true;
    }
    instanceNotifier.notify_all(); // Wake up all waiting threads

    // Join all threads before printing summary
    for (auto& thread : instanceThreads) {
        if (thread.joinable()) {
            thread.join();
        }
    }
}
```

2. **Monitors**

- A **manual implementation of monitors** using a conditional variable and a mutex.
- Notifies waiting threads when an instance becomes **available (empty)**, allowing parties to execute as soon as possible.
- Avoids **starvation** by ensuring no thread waits indefinitely for resources.

Synchronization Mechanisms

```
void DungeonManager::processParties() {
    std::vector<std::thread> instanceThreads;

    // Create threads for each instance
    for (auto& instance : instances) {
        instanceThreads.emplace_back([this, instance]() {
            while (true) {
                auto [partyID, duration] = fetchNextPartyFromQueue();
                if (partyID == -1) return;                          // Sto

                instanceSemaphore.acquire();                        // Acq
                instance->executeParty(partyID, duration);
                instanceSemaphore.release();                        // Rel
            }
        });
    }

    // Sleep for 1 second before stopping processing
    std::this_thread::sleep_for(std::chrono::seconds(1));
    {
        std::lock_guard<std::mutex> lock(queueMutex);
        stopProcessing = true;
    }
    instanceNotifier.notify_all(); // Wake up all waiting threads
```

```
void DungeonManager::addPartyToQueue(int partyID, int duration) {
    std::lock_guard<std::mutex> lock(queueMutex);
    partyQueue.emplace(partyID, duration);
    instanceNotifier.notify_one(); // Notify waiting threads to st
}
```

```cpp
struct CompareParty {
    bool operator()(const std::pair<int, int>& a, const std::pair<int, int>& b) {
        return a.second > b.second;  // Min-heap: shortest duration first
    }
};
std::priority_queue<std::pair<int, int>, std::vector<std::pair<int, int>>, CompareParty> partyQueue;
```

3. **Shortest Job First (SJF) Scheduling**
   - Prioritizes parties with shorter durations to minimize total waiting time.
   - Since all parties are assumed to arrive at the same time, this prevents excessively long wait times for parties with shorter execution times.
   - It optimally minimizes overall waiting time and improves resource utilization.

Synchronization Mechanisms

Problem Set 2

End of Slides