STDISCM

Problem Set 2 - Looking for Group Synchronization

Hans Martin F. Rejano S14

Possible Deadlock and Starvation Explanation

- Deadlock
  - **Instance Threads Waiting Indefinitely**:
    - If no parties are available in the queue, instance threads handled by the **instanceHandler** will remain blocked in **cv.wait(lock, [id] { return !partyQueue.empty() || isDone; })**.
    - However, this does not result in a deadlock because the dispatcher ensures that parties are assigned as long as they exist, and **isDone** is set when termination occurs.
  - **Dispatcher Not Notifying Instances**:
    - The dispatcher must notify available instances when parties are in the queue **(cv.notify_one())**
    - If **cv.notify_one()** is missing or misplaced, instances might wait indefinitely. However, the program ensures that notifications occur when an instance becomes available.

```
// handler to manage dungeon execution
void instanceHandler(int id) {
    while (true) {
        unique_lock<mutex> lock(mtx); // lock mutex to access shared re
        cv.wait(lock, [id] { return !partyQueue.empty() || isDone; });
        if (isDone && partyQueue.empty()) break;
```

```
// dispatcher to manage party assignments
void dispatcher(int n) {
    while (true) {
        unique_lock<mutex> lock(mtx);
        if (isDone && partyQueue.empty()) break;

        for (int i = 0; i < n; i++) {
            if (!partyQueue.empty() && isInstanceAvailable[i]) {
                cv.notify_one(); // notify an instance to process
            }
        }
    }
}
```

- The use of condition variables (**cv.wait() and cv.notify_one()**) prevents deadlock by allowing threads to wait for a signal before proceeding.

Possible Deadlock and Starvation Explanation

- Starvation
  - **Some Parties Never Get Assigned**:
    - If the **queue** processing favors certain parties over others, some could remain in the queue indefinitely.
    - However, since parties are processed FIFO (first-in, first-out), all will eventually get served.
  - **Some Instances Remain Idle While Others Overwork**:
    - If only some instance threads are being notified, others may not get assigned work.
    - The dispatcher checks all available instances (for (int i = 0; i < n; i++)), ensuring even work distribution.
  - Since the dispatcher iterates to all instances, it ensures that all instances get served, avoiding starvation.

```cpp
// populate the party queue with generated parties
for (int i = 0; i < max_parties; i++) {
    partyQueue.push({ i + 1, dis(gen) });
}
```

```cpp
// dispatcher to manage party assignments
void dispatcher(int n) {
    while (true) {
        unique_lock<mutex> lock(mtx);
        if (isDone && partyQueue.empty()) break;

        for (int i = 0; i < n; i++) {
            if (!partyQueue.empty() && isInstanceAvailable[i]) {
                cv.notify_one(); // notify an instance to process
            }
        }
    }
}
```

Synchronization mechanisms used to solve the problem

- **Mutex**
  - Provides exclusive access to shared resources – **partyQueue, isInstanceAvailable, and instance_status**
  - Before modifying or accessing shared data, a lock is acquired using **unique_lock<mutex> lock(mtx)**
  - **Prevents race conditions** where multiple threads could modify the same data at the same time.
- **Condition Variables**
  - **Used to synchronize instance threads** so they only proceed when there is a party available
  - **cv.wait(lock, [id] { return !partyQueue.empty() || isDone; })** ensures that instance threads wait until they are needed
  - **cv.notify_one()** in the dispatcher wakes up waiting threads to process a party.
- **Shared Data Structures**
  - The use of a **queue** for the parties ensures that all are processed in FIFO order.
  - The use of a **boolean vector** tracks available instances to prevent multiple threads from taking the same party.

GitHub Repository

- https://github.com/xLelouch03/STDISCM-ProblemSet2