P2 : LOOKING FOR

GROUP SYNCHRONIZATION

SIMONE FRANCESKA EMANUELLE M. CAPIO - S17 STDISCM

POSSIBLE DEADLOCK SCENARIO

- multiple threads waiting for shared resources (instances or players)
  - e.g. Thread A holds mutex as it waits for an assigned dungeon while Thread B also waits for the same mutex (no one releases it and they get stuck)

- Prevention
  - condition_variable is used (cv) to wait efficiently
  - once a thread is done, it calls cv.notify_all()

```
// regular or static version (all players arrive at the same time, no new joins)
void beginLFGQueue(int n, int t, int h, int d, int t1, int t2) {
    vector<Instance> instances(n);
    vector<thread> threads;

    int possible_parties = min({t, h, d / 3});
    cout << "Total parties that can form: " << possible_parties << endl;

    printInstanceStatus(instances);

    for (int i = 0; i < possible_parties; i++) {
        unique_lock<mutex> lock(mtx);
        cv.wait(lock, [&]() {
            return active_instances < n;
        });

        // find any empty instance slot
        int id = -1;    // placeholder
        for (int j = 0; j < n; j++) {
            if (!instances[j].active) {
                id = j;
                instances[j].active = true;
                break;
            }
        }

        if (id != -1) {
            active_instances++;
            cout << "Party entered instance " << id + 1 << endl;
            printInstanceStatus(instances);
            threads.emplace_back(dungeonRun, id, t1, t2, ref(instances));
        }
    }

    // wait for all dungeons to finish
    for (auto& t : threads) t.join();

    cout << "\nAll parties finished!\n\nFinal instance summary:\n";
    for (size_t i = 0; i < instances.size(); i++) {
        cout << "Instance " << i + 1
            << " | Parties served: " << instances[i].parties_served
            << " | Total time: " << instances[i].total_time << "s" << endl;
    }
};
```

POSSIBLE STARVATION SCENARIO

- threads keep getting delayed while others keep getting access
  - e.g. one instance is always reused while others never start
- Prevention
  - all waiting threads are awakened by cv.notify_all()
  - each party checks availability ***in order*** and proceeds only of if resources are free

```
void beginLFGQueueBonus(int n, int& t, int& h, int& d, int t1, int t2) {
    vector<Instance> instances(n);
    vector<thread> threads;

    printInstanceStatus(instances);

    // start background thread to add new players
    thread generator(generatePlayers, ref(t), ref(h), ref(d), ref(cv), ref(mtx));

    int parties_formed = 0;

    while (true) {
        unique_lock<mutex> lock(mtx);

        // wait until there's enough players to make a party and for a free instance or wake every 1 second to check if it can still make parties
        cv.wait_for(lock, chrono::seconds(1), [&]() {
            return ((t >= 1 && h >= 1 && d >= 3 && active_instances < n) || generator_done);
        });

        // if generator's done and no new part can form, break
        if (generator_done && (t < 1 || h < 1 || d < 3) && active_instances == 0)
            break;

        // double-check player counts before forming a party
        if (!(t >= 1 && h >= 1 && d >= 3 && active_instances < n)) {
            continue;
        }

        // form a party
        t--;
        h--;
        d -= 3;

        int id = -1;
        for (int j = 0; j < n; j++) {
            if (!instances[j].active) {
                id = j;
                instances[j].active = true;
                break;
            }
        }

        if (id != -1) {
            active_instances++;
            parties_formed++;
            cout << "Party #" << parties_formed << " entered instance " << id + 1
                 << " (T:" << t << ", H:" << h << ", D:" << d << ")\n";
            printInstanceStatus(instances);
            threads.emplace_back(dungeonRun, id, t1, t2, ref(instances));
        }

        lock.unlock();
    }

    // join all threads
    generator.join();
    for (auto& th : threads)
        if (th.joinable())
            th.join();

    cout << "\nAll parties finished!\n\nFinal instance summary:\n";
    for (size_t i = 0; i < instances.size(); i++) {
        cout << "Instance " << i + 1
             << " | Parties served: " << instances[i].parties_served
             << " | Total time: " << instances[i].total_time << "s" << endl;
    }
};
```

SYNCHRONIZATION MECHANISMS USED

```
// global synchronization variables
mutex mtx;
condition_variable cv;
int active_instances = 0;
bool generator_done = false;
```

```
lock_guard<mutex> lock(mtx);
int role = roleDist(gen);
```

- std::mutex
  - ensures only one thread can modify shared data at a time
- std::condition_variable
  - lets threads wait for a condition (like enough players or an empty instance)
  - prevents busy-waiting and saves CPU time
- 3. lock_guard and unique_lock
  - simplifies locking and ensures the mutex is always released, even if an error occurs
- threads
  - represent each party's dungeon run
  - run concurrently

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