LFG Dungeon Queue System - Synchronization & Deadlock Prevention

Naman S-12

What Is Dreadlock

Occurs when two or more processes wait indefinitely for resources held by each other.
Prevents further execution as no process can proceed.

Deadlock Characteristics

Circular wait condition

Resource holding

No progress possible

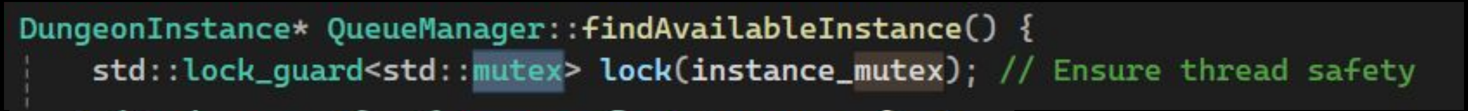
Example in Our System

All dungeon instances are full, but no party can complete their dungeon.

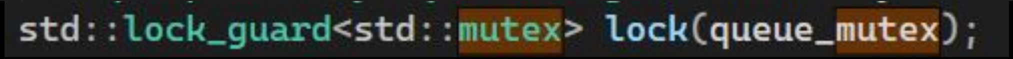
New parties cannot enter, leading to an indefinite wait.

Preventing Dreadlock

Mutex Locks (std::mutex) – Prevents multiple threads from modifying shared resources simultaneously.

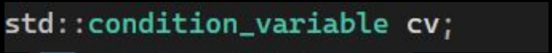


```
DungeonInstance* QueueManager::findAvailableInstance() {
    std::lock_guard<std::mutex> lock(instance_mutex); // Ensure thread safety
```

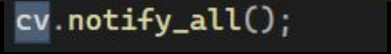


```
std::lock_guard<std::mutex> lock(queue_mutex);
```

Condition Variables (std::condition_variable) – Ensures waiting threads wake up properly when an instance becomes free.

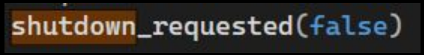


```
std::condition_variable cv;
```

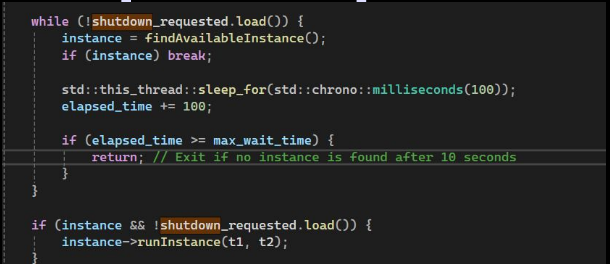


```
cv.notify_all();
```

Proper Resource Release (shutdown_requested) – Ensures that dungeon instances become available after a party finishes.
Its further execution as no process can proceed.



```
shutdown_requested(false)
```

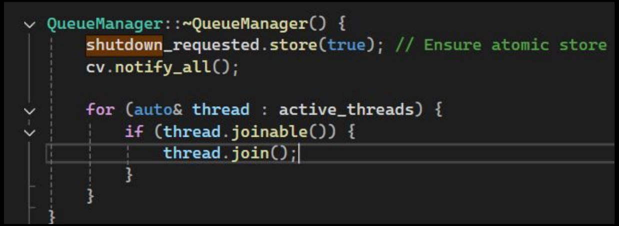


```
while (!shutdown_requested.load()) {
    instance = findAvailableInstance();
    if (instance) break;

    std::this_thread::sleep_for(std::chrono::milliseconds(100));
    elapsed_time += 100;

    if (elapsed_time >= max_wait_time) {
        return; // Exit if no instance is found after 10 seconds
    }
}

if (instance && !shutdown_requested.load()) {
    instance->runInstance(t1, t2);
}
```



```
QueueManager::~QueueManager() {
    shutdown_requested.store(true); // Ensure atomic store
    cv.notify_all();

    for (auto& thread : active_threads) {
        if (thread.joinable()) {
            thread.join();
        }
    }
}
```

Understanding Starvation

What is Starvation?

When a process never gets access to necessary resources due to other higher-priority processes continuously using them.

Some players or dungeon instances may never get used.

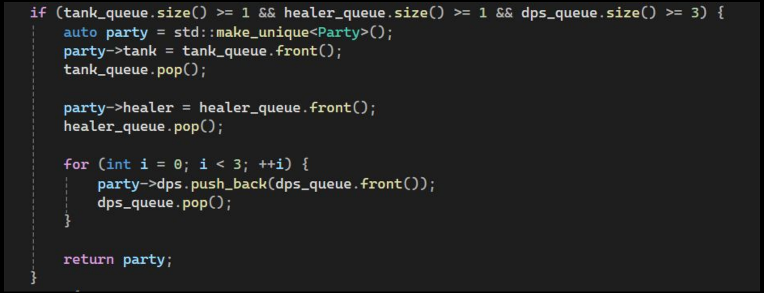
Example in Our System:

If instances always favor the first queue, some instances may never run. If DPS players run out, Tanks and Healers may be stuck indefinitely.

Preventing Starvation

Techniques Used:

First-Come, First-Served (FCFS) Queueing-
Ensures fair allocation of Tanks, Healers, and DPS.



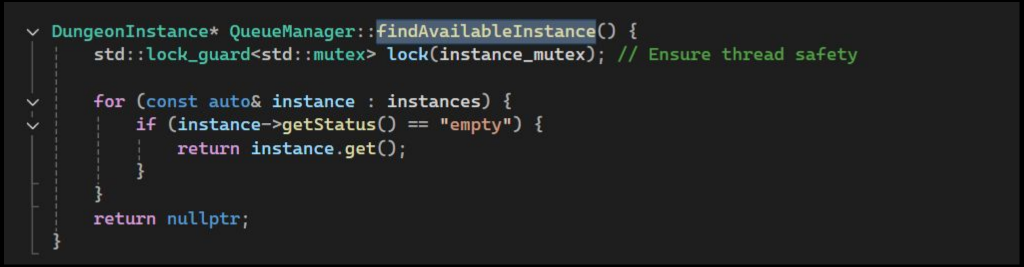
```
if (tank_queue.size() >= 1 && healer_queue.size() >= 1 && dps_queue.size() >= 3) {
    auto party = std::make_unique<Party>();
    party->tank = tank_queue.front();
    tank_queue.pop();

    party->healer = healer_queue.front();
    healer_queue.pop();

    for (int i = 0; i < 3; ++i) {
        party->dps.push_back(dps_queue.front());
        dps_queue.pop();
    }

    return party;
}
```

Balanced Dungeon Assignments-
findAvailableInstance() selects any free instance, preventing some from being idle.



```
DungeonInstance* QueueManager::findAvailableInstance() {
    std::lock_guard<std::mutex> lock(instance_mutex); // Ensure thread safety

    for (const auto& instance : instances) {
        if (instance->getStatus() == "empty") {
            return instance.get();
        }
    }
    return nullptr;
}
```

Mechanism	Purpose
std::mutex	Prevents multiple threads from modifying shared resources at the same time
std::lock_guard	Ensures automatic lock release when a function exits
std::atomic<bool>	Prevents race conditions when checking shutdown status
std::condition_variable	Avoids busy waiting when checking for available instances
std::Queue	Ensures fair First-Come, First-Served processing of players

Thank you