PROBLEM SET
NO. 2

CAASI S24

STDISCM AY 2024-2025 | DLSU

requirements

  - Only ***n*** dungeon instances active concurrently.
  - Standard party composition: **1 tank**, **1 healer**, **3 DPS**
  - Handle process synchronization **without deadlock** or **starvation**
  - Randomized dungeon run time between **t1** and **t2 seconds**

- **Outputs:**
  - Current status of each dungeon instance (active/empty).
  - Summary of parties served and total time per dungeon.
  - Leftover players printed.

potential deadlock issues and solution

- **Scenario:**
    - Imagine multiple dungeon threads attempting to form a party concurrently.
    - If one thread locks a resource (e.g., reserves a tank) and then waits for a healer while another thread reserves the healer and waits for a tank, both threads become blocked, waiting for the other to release the resource.
    - This circular wait can lead to deadlock, where no thread can proceed because each holds a piece of the necessary resources.

potential deadlock issues and solution

- **Solution:**
  - To avoid this, our design performs the entire resource check (1 tank, 1 healer, and 3 DPS) atomically within a single mutex lock (**playerMutex**).
  - By ensuring that the check and deduction of all required player counts occur together, we prevent partial allocation and eliminate circular wait conditions.
  - **Outcome:** Threads only reserve resources if all required roles are available, thereby avoiding deadlock.

potential starvation issues and solution

- **Scenario:**
  - Consider if certain dungeon threads repeatedly gain access to available players due to thread scheduling policies or resource contention.

  - This could leave other threads waiting indefinitely for the resources, effectively “starving’ them of the chance to form a party.

potential starvation issues and solution

- **Solution:**
  - We address starvation by using mutexes (especially playerMutex) to provide a fair, first-come-first-served mechanism for accessing and updating player counts.

  - All threads have an equal chance to enter the critical section, ensuring that no single thread monopolizes the resources.

  - **Outcome:** Uniform resource allocation guarantees that every dungeon thread eventually forms a party if resources permit, preventing starvation.

synchronization mechanisms employed

- Mutexes:
  - **coutMutex:** Protects console output to prevent garbled prints.
  - **playerMutex:** Ensures atomic deduction of player counts when forming parties.
  - **statsMutex:** Manages access to and updates of the dungeon instance statistics.

```
mutex coutMutex;
mutex playerMutex;
mutex statsMutex;
```

why so many?

Each mutex isolates a specific critical section—ensuring safe player updates, accurate stats, and clean output—so we avoid deadlock and simplify our design

synchronization mechanisms employed

- Dynamic Thread Adjustment
  - Calculate maximum full parties available.
  - Launch threads equal to the minimum of (configured n or available parties).
  - Unused dungeon instances will have their statuses printed as "empty" with zero parties served and zero time served.

synchronization mechanisms employed

- Threading Model:
  - Each dungeon instance runs on its own thread, which continuously processes parties until no complete party can be formed.

  - Threads are reused; they loop to form new parties rather than terminating immediately, leading to efficient resource utilization and meeting project specifications.