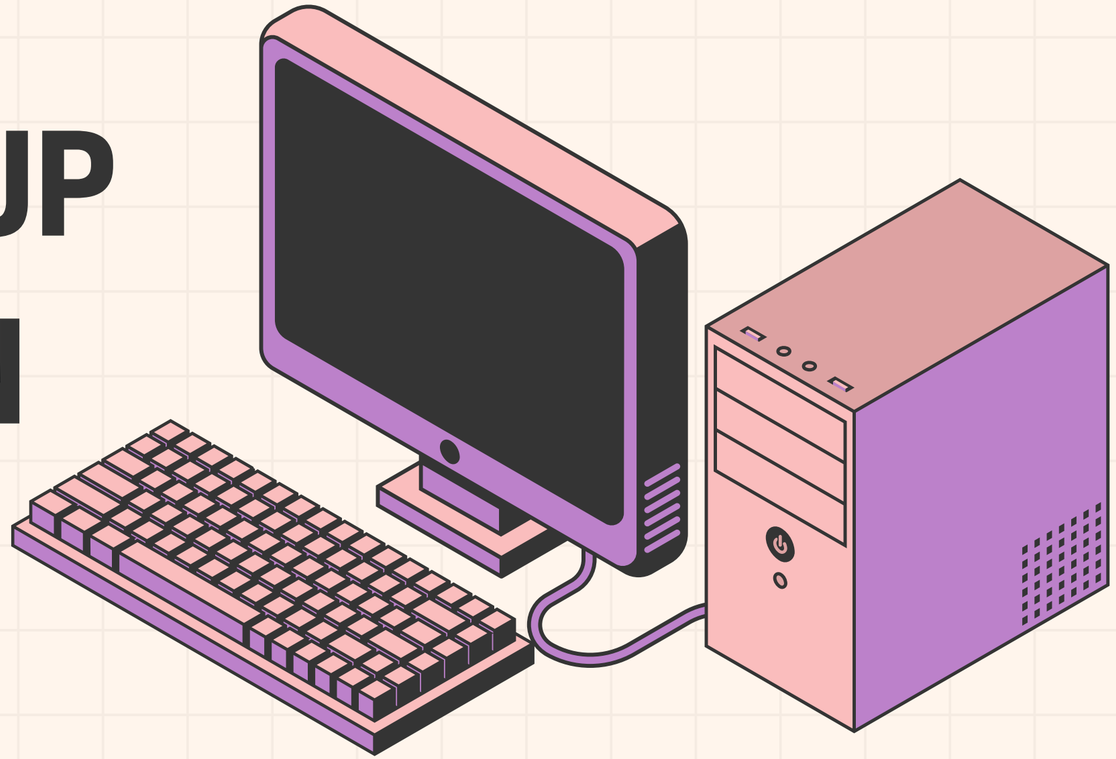
LOOKING FOR GROUP SYNCHRONIZATION

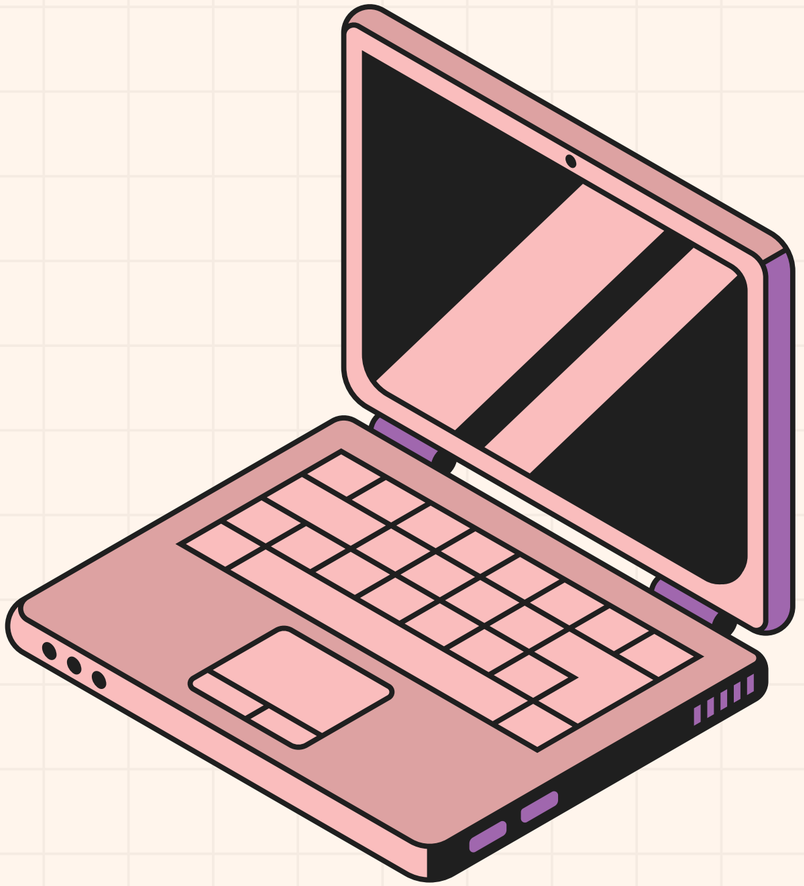


ALFARO, NATHANIEL LUIS V.

STDISCM - S18

SYNCHRONIZATION PROBLEM OVERVIEW

- **Resource:** Limited Dungeon Instances
- **Party Composition Required:** 1T/1H/3D
- **Challenge:** Manage concurrent access to instances
- **Goal:** No deadlock, no starvation



SYNCHRONIZATION MECHANISMS USED

COUNTING SEMAPHORE
(G_INSTANCE_SLOTS)

- Controls access to limited instance slots
- Blocks when no slots available
- Wakes waiting threads when slots freed

DATA MUTEX (G_DATA_MUTEX)

- Protects shared resources:
  - Instance status array
  - Player counts (T/H/D)
  - Party formation checks
- Prevents race conditions

CONDITION VARIABLE
(IN SEMAPHORE)

- Efficient thread waiting/waking
- No busy waiting
- FIFO queue for fairness

WHY DEADLOCK CANNOT OCCUR?

NO CIRCULAR DEPENDENCIES

- Resources acquired in one step
- No nested resource holdings

SINGLE RESOURCE TYPE

- Parties only need instance slots
- No multiple resource types → no circular wait

RESOURCE RELEASE GUARANTEED

- Fixed duration dungeons
- Resources always freed after use

NO HOLD AND WAIT

```
// Atomic resource acquisition
g_instance_slots->acquire();
// Use instance
// Always release when done
g_instance_slots->release();
```

HOW WE PREVENT STARVATION?

NO PRIORITY SYSTEM

- All parties treated equally
- No thread can monopolize instances

FAIR QUEUING

```
// In CountingSemaphore::acquire()
cv.wait(lock, [&]() { return count > 0; });
// First thread waiting is first to wake
```

RESOURCE AVAILABILITY

- Instances always released
- Equal chance for waiting parties

BOUNDED WAIT TIME

- Fixed dungeon duration (t1 to t2)
- Guaranteed resource release
- New parties can always form eventually

CONCLUSION

NO DEADLOCKS
OBSERVED IN ANY TEST
CASE

FAIR RESOURCE
DISTRIBUTION PROVEN

EFFICIENT
SYNCHRONIZATION

ACHIEVED
SCALABLE FROM SMALL
TO LARGE LOADS