PS 2: Looking for Group Synchronization

Inocencio, Rey Vincent

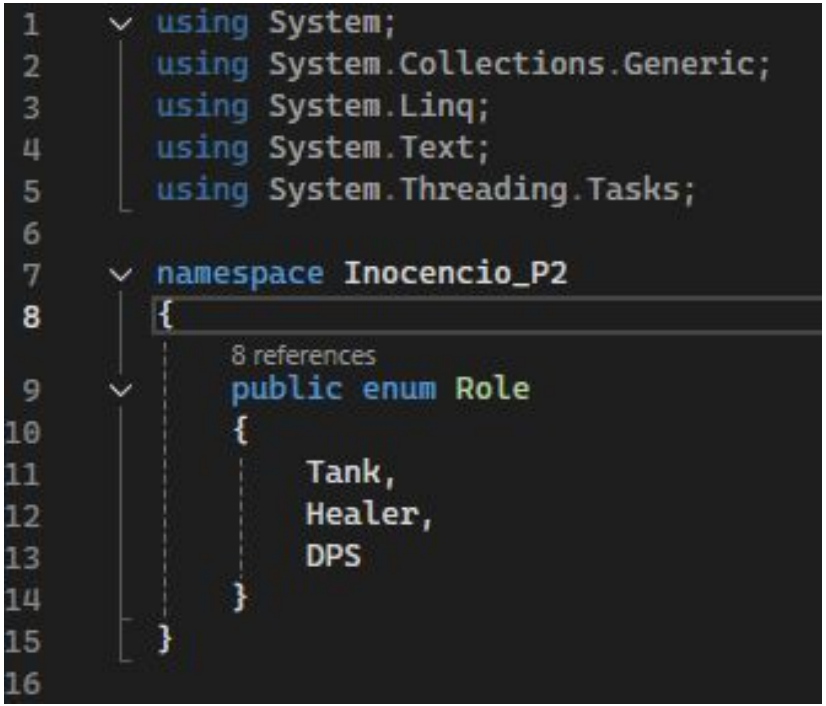
Specifications

The program is a simulation of the LFG (Looking for Group) Dungeon Queuing of an MMORPG. The program takes user input for the maximum number of concurrent instances, number of tank players, number of healer players, number of DPS players, minimum dungeon clear time, and maximum dungeon clear time.

Output:

- Current statuses of all instances
- Summary of instances' total execution time and total parties served.

Implementation - Role (Enum)



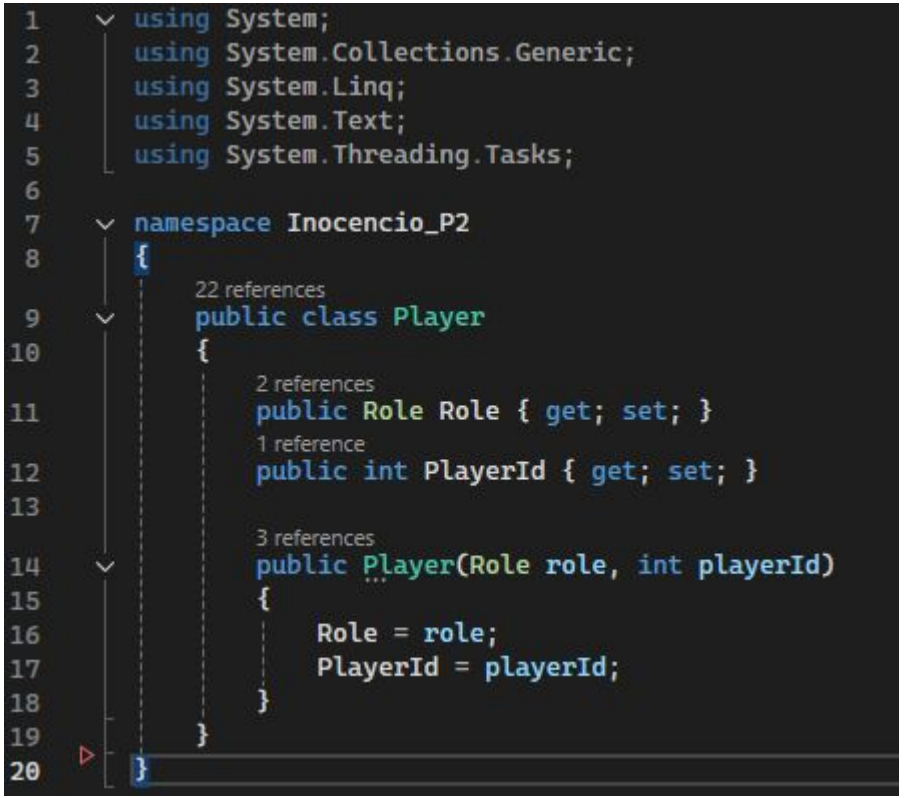
```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;

namespace Inocencio_P2
{
    8 references
    public enum Role
    {
        Tank,
        Healer,
        DPS
    }
}
```

- Defines the different player roles available in the application.
- Categorizes players and ensure that parties are formed with the correct composition.

Implementation - Player (Class)

- Represents individual players in the program.
- Contains the players' role, and unique identifier.
- Created based on user input and then enqueued into their designated role's queue in the DungeonQueue class.



```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;

namespace Inocencio_P2
{
    22 references
    public class Player
    {
        2 references
        public Role Role { get; set; }
        1 reference
        public int PlayerId { get; set; }

        3 references
        public Player(Role role, int playerId)
        {
            Role = role;
            PlayerId = playerId;
        }
    }
}
```

Implementation - Party (Class)

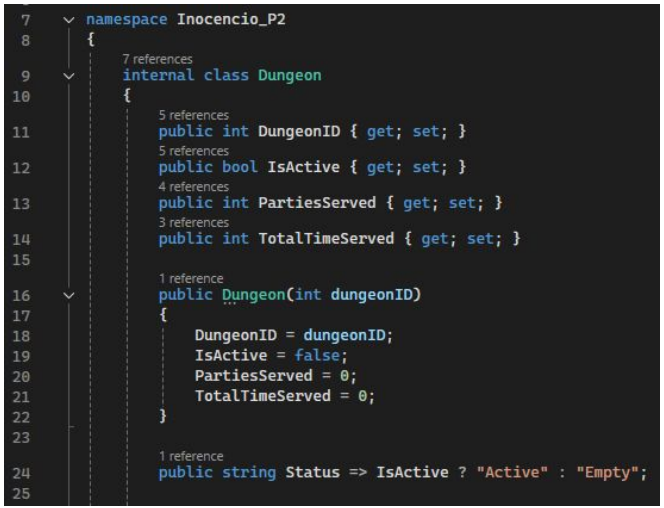
```
namespace Inocencio_P2
{
    6 references
    class Party
    {
        1 reference
        public Player Tank { get; set; }
        1 reference
        public Player Healer { get; set; }
        1 reference
        public List<Player> DPS { get; set; }

        1 reference
        public Party(Player tank, Player healer, List<Player> dpsPlayers)
        {
            if (dpsPlayers.Count != 3)
            {
                throw new ArgumentException("A party must have exactly 3 DPS players.");
            }
            Tank = tank;
            Healer = healer;
            DPS = new List<Player>(dpsPlayers);

        }
    }
}
```

- Serves as the model for a complete party of players.
- Is formed by the DungeonQueue when there are sufficient people.

Implementation - Dungeon (Class)



```
namespace Inocencio_P2
{
    7 references
    internal class Dungeon
    {
        5 references
        public int DungeonID { get; set; }
        5 references
        public bool IsActive { get; set; }
        4 references
        public int PartiesServed { get; set; }
        3 references
        public int TotalTimeServed { get; set; }

        1 reference
        public Dungeon(int dungeonID)
        {
            DungeonID = dungeonID;
            IsActive = false;
            PartiesServed = 0;
            TotalTimeServed = 0;
        }

        1 reference
        public string Status => IsActive ? "Active" : "Empty";
```

```
        1 reference
        public void RunInstance(Party party, int currentPartyNumber, int minTime, int max
        {
            IsActive = true;
            PartiesServed++;
            int partyNumber = PartiesServed;

            Random rand = new Random();
            int duration = rand.Next(minTime, maxTime + 1);

            Console.WriteLine($"Instance {DungeonID} is now serving party {currentPartyNu

            Thread.Sleep(duration * 1000);

            TotalTimeServed += duration;

            IsActive = false;
            Console.WriteLine($"Instance {DungeonID}: Party {currentPartyNumber} has comp
        }
    }
}
```

- Represents individual dungeon instances
- Serves parties and logs the instance’s status.

Implementation - DungeonQueue (Class)

```
3 references
internal class DungeonQueue
{
    SemaphoreSlim s;
    private readonly object queueLock = new object();
    private int minTime;
    private int maxTime;
    private int lastUsedInstanceIndex = 0;

    private Queue<Player> tankQueue;
    private Queue<Player> healerQueue;
    private Queue<Player> dpsQueue;

    private List<Dungeon> dungeonInstances;
    1 reference
    public IReadOnlyList<Dungeon> Dungeons => dungeonInstances.AsReadOnly();

    1 reference
    public DungeonQueue(int maxInstances, int minTime, int maxTime)
    {
        s = new SemaphoreSlim(maxInstances);
        tankQueue = new Queue<Player>();
        healerQueue = new Queue<Player>();
        dpsQueue = new Queue<Player>();
        dungeonInstances = new List<Dungeon>();
        this.minTime = minTime;
        this.maxTime = maxTime;

        for (int i = 1; i <= maxInstances; i++)
        {
            dungeonInstances.Add(new Dungeon(i));
        }
    }
```

```
3 references
public void EnqueuePlayer(Player player)
{
    lock (queueLock)
    {
        switch (player.Role)
        {
            case Role.Tank:
                tankQueue.Enqueue(player);
                break;
            case Role.Healer:
                healerQueue.Enqueue(player);
                break;
            case Role.DPS:
                dpsQueue.Enqueue(player);
                break;
        }
    }
}

1 reference
public Task Start(int numParties)
{
    return Task.Run(() => ProcessQueues(numParties));
}

1 reference
private void ProcessQueues(int numParties)
{
    int formedParties = 0;
    List<Task> tasks = new List<Task>();

    while (formedParties < numParties)
    {
        Party party = TryFormParty();
        if (party != null)
        {
            formedParties++;
```

Implementation - DungeonQueue (Class)

```
150                 {
151                     Console.WriteLine(" ");
152                     Console.WriteLine($"Instance Statuses at {DateTime.Now:yyyy-MM-dd HH:mm:ss}");
153                     foreach (var dungeon in dungeonInstances)
154                     {
155                         Console.WriteLine($"Instance {dungeon.DungeonID}: {dungeon.Status}");
156                     }
157                     Console.WriteLine(" ");
158                 }
159             }
160
                1 reference
161             public async Task MonitorDungeonStatuses(CancellationToken token)
162             {
163                 try
164                 {
165                     while (!token.IsCancellationRequested)
166                     {
167                         PrintStatuses();
168                         await Task.Delay(5000, token);
169                     }
170                 }
171                 catch (TaskCanceledException)
172                 {
173
174                 }
175                 finally
176                 {
177                     PrintStatuses();
178                 }
179             }
180
```

```
                1 reference
181             public int RemainingTanks
182             {
183                 get { lock (queueLock) { return tankQueue.Count; } }
184             }
185
                1 reference
186             public int RemainingHealers
187             {
188                 get { lock (queueLock) { return healerQueue.Count; } }
189             }
190
                1 reference
191             public int RemainingDPS
192             {
193                 get { lock (queueLock) { return dpsQueue.Count; } }
194             }
195         }
196     }
197
```

Implementation - DungeonQueue (Class)

- Central system of the application.
- Manages the entire process of queuing players, forming parties, and assigning them to dungeon instances.
- Contains: List of dungeon objects, FIFO queues for each role
- Periodically displays the status of all dungeons, and prints the final statistics.

Implementation - Program (Class)

```
internal class Program
{
    0 references
    static void Main(string[] args)
    {
        try
        {
            Console.WriteLine("Enter input values separated by space in the following order:");
            Console.WriteLine("n t h d t1 t2");

            string input = Console.ReadLine();
            string[] tokens = input.Split(' ');

            if (tokens.Length != 6)
            {
                Console.WriteLine("Please provide exactly 6 numbers separated by spaces.");
                return;
            }

            int maxInstances = int.Parse(tokens[0]);
            int numTanks = int.Parse(tokens[1]);
            int numHealers = int.Parse(tokens[2]);
            int numDPS = int.Parse(tokens[3]);
            int minTime = int.Parse(tokens[4]);
            int maxTime = int.Parse(tokens[5]);

            if (maxInstances <= 0 || numTanks <= 0 || numHealers <= 0 || numDPS <= 0 || minTime <= 0 || maxTime <= 0)
            {
                Console.WriteLine("All input values must be greater than zero.");
                return;
            }

            // Validate that t2 is not less than t1.
            if (maxTime < minTime)
            {
                Console.WriteLine("The maximum clear time (t2) cannot be less than the minimum clear time (t1).");
                return;
            }

            int totalParties = Math.Min(numTanks, Math.Min(numHealers, numDPS / 3));
            Console.WriteLine($"Total parties that can be formed: {totalParties}");

            DungeonQueue dungeonQueue = new DungeonQueue(maxInstances, minTime, maxTime);

            // Enqueue players for each role.
            int playerId = 1;  // Unique identifier counter.
            for (int i = 0; i < numTanks; i++)
                dungeonQueue.EnqueuePlayer(new Player(Role.Tank, playerId++));
            }
            for (int i = 0; i < numHealers; i++)
            {
                dungeonQueue.EnqueuePlayer(new Player(Role.Healer, playerId++));
            }
            for (int i = 0; i < numDPS; i++)
            {
                dungeonQueue.EnqueuePlayer(new Player(Role.DPS, playerId++));
            }

            CancellationTokenSource cts = new CancellationTokenSource();

            Task monitorTask = dungeonQueue.MonitorDungeonStatuses(cts.Token);

            Task processingTask = dungeonQueue.Start(totalParties);
            processingTask.Wait();

            cts.Cancel();
            monitorTask.Wait();

            Console.WriteLine("All processing complete.");
            Console.WriteLine(" ");

            Console.WriteLine("Final Dungeon Statistics:");
            foreach (var dungeon in dungeonQueue.Dungeons)
            {
                Console.WriteLine($"Instance {dungeon.DungeonID} served for a total of {dungeon.TotalTim
            }
            Console.WriteLine(" ");
            Console.WriteLine("Remaining players in queues:");
            Console.WriteLine($"Tanks: {dungeonQueue.RemainingTanks}");
            Console.WriteLine($"Healers: {dungeonQueue.RemainingHealers}");
            Console.WriteLine($"DPS: {dungeonQueue.RemainingDPS}");

            Console.ReadKey();
        }
        catch (FormatException fe)
        {
            Console.WriteLine("Input was not in the correct format. Please enter valid numbers.");
            Console.WriteLine(fe.Message);
        }
        catch (Exception ex)
        {
            Console.WriteLine("An error occurred while processing inputs.");
            Console.WriteLine(ex.Message);
        }
```

Implementation - Program (Class)

- Application’s main entry point
- Handles User input
- Enqueues players based on provided input
- Coordinates the startup and termination of the system, printing final dungeon statistics at the end.

Possible Deadlock

Deadlock happens when two or more processes/threads are locked in waiting for resources from each other. In this program, the player queue is a shared process across dungeon instances.

The program prevents deadlocks through the use of locks and semaphores (SemaphoreSlim.)

Possible Deadlock Solution

```
1 reference
private void ProcessParty(Party party, int currentPartyNumber)
{
    s.Wait();
    try
    {
        Dungeon instance = GetAvailableDungeonInstance();
        if (instance != null)
        {
            instance.RunInstance(party, currentPartyNumber, minTime, maxTime);
        }
    }
    finally
    {
        s.Release();
    }
}
```

- Semaphore (s.wait()) is called outside of any locks, ensuring that there is no risk that a thread is holding a lock while waiting.

Possible Deadlock Solution

```
1 reference
private Party FormParty()
{
    lock (queueLock)
    {
        if (tankQueue.Count > 0 && healerQueue.Count > 0 && dpsQueue.Count >= 3)
        {
            Player tank = tankQueue.Dequeue();
            Player healer = healerQueue.Dequeue();
            List<Player> dpsPlayers = new List<Player>();
            for (int i = 0; i < 3; i++)
            {
                dpsPlayers.Add(dpsQueue.Dequeue());
            }
            return new Party(tank, healer, dpsPlayers);
        }
        else
        {
            return null;
        }
    }
}
```

- Locks are acquired ONLY for the brief period needed to safely access shared data.

Possible Deadlock Solution

```
1 reference
private void ProcessParty(Party party, int currentPartyNumber)
{
    s.Wait();
    try
    {
        Dungeon instance = GetAvailableDungeonInstance();
        if (instance != null)
        {
            instance.RunInstance(party, currentPartyNumber, minTime, maxTime);
        }
    }
    finally
    {
        s.Release();
    }
}
```

- Semaphores are always released in the finally block, ensuring that if an exception occurs during the processes, the semaphore is still released and threads are not blocked as a result.

Possible Starvation

Starvation happens when threads never access a resource because other threads are prioritized, leaving the starved thread unable to function. In the program, starvation can happen if players or parties are continuously skipped, or if a thread is unable to access shared resources because other threads are always acquiring the semaphore.

The program handles this by using FIFO (first in, first out) queues to ensure all players are served in the order they arrive, using slim semaphores to limit the number of concurrently active dungeons, and applying Round-Robin in selecting dungeons.

Possible Starvation Solution

```
private Party FormParty()
{
    lock (queueLock)
    {
        if (tankQueue.Count > 0 && healerQueue.Count > 0 && dpsQueue.Count >= 3)
        {
            Player tank = tankQueue.Dequeue();
            Player healer = healerQueue.Dequeue();
            List<Player> dpsPlayers = new List<Player>();
            for (int i = 0; i < 3; i++)
            {
                dpsPlayers.Add(dpsQueue.Dequeue());
            }
            return new Party(tank, healer, dpsPlayers);
        }
        else
        {
            return null;
        }
    }
}
```

- In FormParty(), players are dequeued in FIFO order, guaranteeing that once players are in the queue, they will eventually be included in a party if there are sufficient players.

Possible Starvation Solution

```
private void ProcessParty(Party party, int currentPartyNumber)
{
    s.Wait();
    try
    {
        Dungeon instance = GetAvailableDungeonInstance();
        if (instance != null)
        {
            instance.RunInstance(party, currentPartyNumber, minTime, maxTime);
        }
    }
    finally
    {
        s.Release();
    }
}
```

- Particularly in .NET, semaphores are designed to work in a fair manner -> Semaphores are granted access in order of arrival (FIFO), ensuring that once a thread is waiting to acquire an instance, it will eventually acquire it when it becomes available

Possible Starvation Solution

```
private Dungeon GetAvailableDungeonInstance()
{
    lock (dungeonInstances)
    {
        int instanceCount = dungeonInstances.Count;
        for (int i = 0; i < instanceCount; i++)
        {
            int index = (lastUsedInstanceIndex + i + 1) % instanceCount;
            if (!dungeonInstances[index].IsActive)
            {
                lastUsedInstanceIndex = index;
                return dungeonInstances[index];
            }
        }
    }
    return null;
}
```

- A Round-robin mechanism is applied in selecting dungeon instances to ensure that parties are distributed evenly across dungeon instances, reducing the change that one given dungeon is perpetually idle.