EXÁMENES 2024 - Práctica

QUINTANA

Para poder aplicar los conocimientos que vas atesorando en el Grado en Diseño y Desarrollo de Videojuegos de la URJC, decides poner en prueba tus habilidades programando un pequeño simulador basado en el estupendo Prison Architect de Chris Delay, con la temática de un antiguo juego de mesa denominado La fuga de Colditz, que está ambientado en un campo de prisioneros de la II guerra mundial. En dicho juego, el objetivo es fugarse del Castillo de Colditz, que los alemanes usan para retener a los prisioneros aliados.

En el juego, los jugadores pueden ser prisioneros o carceleros. Por su parte, el castillo está formado por distintas habitaciones (talleres, barracones, comedor, etc.), pudiendo haber varias de un mismo tipo.

Los prisioneros, para poder fugarse, se reúnen en las salas para planificar la fuga. Los guardias, por otro lado, pueden acceder a una sala, cuando ésta está vacía, para realizar un registro, o cuando hay más de 5 prisioneros, para disolver la reunión clandestina.

Las reglas que se deben cumplir son las siguientes:

- Cualquier número de presos puede estar en una habitación a la vez.
- Un carcelero sólo puede entrar a una habitación si no hay nadie (para realizar un registro) o si hay más de 5 prisioneros (para disolver la reunión clandestina).
- Cuando un carcelero entra en una sala, ningún preso puede entrar, pero los presos podrán salir.
- El carcelero no podrá abandonar la sala hasta que todos los presos hayan salido.

Se sobreentiende que el programa genera suficientes presos y carceleros. El tiempo que el carcelero pase registrando, o los presos confabulando en las habitaciones, es aleatorio, determinado por el desarrollador. Se pide:

PARTE 1 (1 punto) Describe en el cuadro de texto la estructura general del programa. Enumera los hilos, clases auxiliares y/o estructuras de datos que consideres necesarios. Enumera las relaciones de exclusión mutua y de sincronización condicional.

PARTE 2 (2 puntos) Implementar el ejercicio con sus hilos, así como los tipos y estructuras de datos necesarias para sincronizar todo y que permitan que el código no sea sensible a problemas de concurrencia. Se realizará el código suponiendo que sólo hay 1 carcelero y 1 habitación.

```
namespace Ejercicios_examenes
{
    public class MAYO2024_2
    {
        static Sala habitacion = new Sala();
        public static void Main()
        {
            GenerarPrisioneros();
            var carcelero = new Carcelero(habitacion);
            var thread = new Thread(carcelero.Accion);
            thread.Start();
        }

        static public void GenerarPrisioneros()
        {
            for (int i = 0; i < 10; i++)
            {
                var prisionero = new Prisionero($"S{i}",habitacion);
```


```
            var thread = new Thread(prisionero.Accion);
            thread.Start();
        }
    }
}

public class Prisionero
{
    public string name;
    private Sala habitacion;
    Random random = new Random();

    public Prisionero(string name, Sala habitacion)
    {
        this.name = name;
        this.habitacion = habitacion;
    }
    public void Accion()
    {
        while(true)
        {
            habitacion.añadirPrisionero(this);
        }
    }
}

public class Carcelero
{
    Random random = new Random();
    private Sala habitacion;

    public Carcelero(Sala habitacion)
    {
        this.habitacion=habitacion;
    }
    public void Accion()
    {
        while (true)
        {
            habitacion.añadirCarcelero();
            //Thread.Sleep(200);
        }
    }
}

public class Sala
{
    public int contador;
    public bool hayCarcelero;

    private Mutex EM_prisioneros=new Mutex();
    private Mutex EM_carcelero= new Mutex();
    private SemaphoreSlim barrera_noEntraCaco = new SemaphoreSlim(0);
    private SemaphoreSlim barrera_yaNoQuedanCacos = new SemaphoreSlim(0);
```

```
        public Sala()
        {
            contador = 0;
            hayCarcelero = false;
        }

        public void añadirPrisionero(Prisionero prisionero)
        {
            if (hayCarcelero == true)
            {
                barrera_noEntraCaco.Wait();
            }
            EM_prisioneros.WaitOne();
            contador++;
            Console.WriteLine($"Prisionero {prisionero.name} dentro de la sala. Hay
{contador} prisioneros");
            EM_prisioneros.ReleaseMutex();
            Console.WriteLine($"{prisionero.name} está confabulando ");
            Thread.Sleep(600);
            eliminarPrisionero(prisionero);
        }

        public void eliminarPrisionero(Prisionero prisionero)
        {
            EM_prisioneros.WaitOne();
            contador--;
            Console.WriteLine($"Prisionero {prisionero.name} fuera de la sala. Hay
{contador} prisioneros");
            if(hayCarcelero == true)
            {
                if (contador == 0)
                {
                    barrera_noEntraCaco.Release();
                    barrera_yaNoQuedanCacos.Release();
                }
                EM_prisioneros.ReleaseMutex();
            }
            else
            {
                EM_prisioneros.ReleaseMutex();
            }
        }

        public void añadirCarcelero()
        {
            if(contador > 5)
            {
                EM_carcelero.WaitOne();
                hayCarcelero = true;
                Console.WriteLine("Carcelero entra en la sala");
                Console.WriteLine($"Carcelero disolviendo reunion de {contador}
prisioneros");
                barrera_yaNoQuedanCacos.Wait();
                eliminarCarcelero();
            }
```

```
            else if(contador == 0)
            {
                EM_carcelero.WaitOne();
                hayCarcelero = true;
                Console.WriteLine("Carcelero entra en la sala");
                Console.WriteLine($"Carcelero realiza registro");
                Thread.Sleep(500);
                eliminarCarcelero();
            }
        }

        public void eliminarCarcelero()
        {
            hayCarcelero = false;
            Console.WriteLine("Carcelero sale de la sala");
            EM_carcelero.ReleaseMutex();
        }
    }
}
```

MÓSTOLES

Los años 80 marcaron un antes y un después en la historia de los juegos de rol. Si bien Dungeons & Dragons había sentado las bases del género en la década anterior, la llegada de títulos como Dungeon Master supuso un salto cualitativo en términos de dinamismo e inmersión.

Dungeon Master, publicado en 1987 por FTL Games, revolucionó el género al perfeccionar dos mecánicas específicas. Por un lado, el sistema de movimiento basado en cuadrículas, permitiendo a los jugadores desplazarse por el escenario con mayor precisión y libertad. Y por otro, la implementación del combate en tiempo real, transformando la experiencia de juego, alejándola de la abstracción por turnos (tan habitual en aquella época).

Aquellos cambios no solo afectaron a la jugabilidad, sino que también repercutieron en la narrativa. La posibilidad de reaccionar de forma instantánea a los eventos del juego y de ejecutar acciones de forma simultánea fomentó la interacción y la improvisación, dando lugar a historias más dinámicas y envolventes.

Su influencia en la industria fue innegable, inspirando a toda una generación de desarrolladores y popularizando estas mecánicas en el género de juegos de los juegos de rol. Llegó a ser el juego más vendido de toda la plataforma Atari ST, y uno de los más exitosos en Amiga.

Para este examen, se aplicarán los principios de programación concurrente en C# para simular situaciones de movimiento y combate en el mundo de Dungeon Master.

SISTEMA DE MOVIMIENTO

En esta parte, se debe crear un sistema para simular el movimiento de múltiples criaturas dentro del laberinto. Se requiere lo siguiente:

- Implementar una clase Laberinto que represente el mundo de Dungeon Master como una matriz de tamaño nxn. Esta clase debe proporcionar métodos para agregar criaturas al laberinto, moverlas dentro del laberinto y verificar la ocupación de cada casilla del laberinto.
- Crear una clase Criatura que represente a cualquier entidad viva en el laberinto, ya sea un héroe o un enemigo.
- Implementar un sistema de movimiento aleatorio para las criaturas dentro del laberinto. Las criaturas deben moverse de manera aleatoria dentro del laberinto, evitando colisiones con otras

criaturas (en cada casilla solamente podrá existir una criatura) y las paredes del laberinto (es decir, no podrán salirse de los límites del laberinto).

El mundo se representa como una matriz cuadrada de tamaño nxn, donde cada casilla puede contener a una criatura a la vez. Las criaturas no se podrán mover en diagonal.

SISTEMA DE COMBATE

En esta parte mejorarán el sistema creado en la Parte 1 para simular un combate entre el héroe y varios enemigos. Se requiere lo siguiente:

- Extender la clase Criatura para incluir métodos de combate, como Atacar() y Defenderse(). Estos métodos deben ser ejecutados de manera concurrente con el movimiento de las criaturas.
- Implementar un sistema de combate en el que el héroe y los enemigos puedan atacarse entre sí cuando estén en ubicaciones próximas dentro del laberinto. Los ataques deben ser gestionados de manera concurrente para garantizar un combate fluido. Si una criatura se mueve al ser atacada, el ataque se falla siempre.
- Diseñar una estrategia de defensa para el héroe. Este debe ser capaz de esquivar los ataques enemigos (es decir, interrumpirlos) y responder con ataques propios de manera concurrente.

Se pide lo siguiente:

PARTE 1 (1 punto)

Plantea el ejercicio. ¿qué clases necesitas? ¿qué tipos de hilos hay que definir? ¿qué relaciones de exclusión mutua y sincronización condicional necesitas? Define también, si fuera necesario, otros elementos necesarios como estructuras de datos, interrupciones, eventos, etc.

Será necesario para la PARTE 2

- Una clase Laberinto con los métodos: agregarCriatura(), moverCriatura(), casillaLibre(), liberarCasilla() y un int de tamaño, una matriz con las casillas (0=libre, 1=ocupada) y un contador de criaturas en el tablero.
    - Será necesario tener una exclusión mutua para comprobar de forma protegida si la casilla está libre para evitar que dos criaturas entren a la misma casilla al mismo tiempo.
    - Y será necesario otra exclusión mutua para aumentar o disminuir de manera controlada la variable contador del número de criaturas en el tablero.
- Una clase Criatura que tenga una referencia a la clase Laberinto y un método Exec() desde donde se llaman a los métodos para moverlas (while(true))y agregarlas.
- Se creará un hilo por cada Criatura en el tablero que ejecutará el método Exec().

Para la PARTE 3:

- Será necesario una exclusión mutua que bloquee a la criatura si va a atacar a otra o a un héroe que ya estaba previamente atacando. Sólo los héroes no serán afectados por esta exclusión y podrán realizar su método de defensa como respuesta.

PARTE 2 (2 puntos)

Implementa el sistema de movimiento, teniendo en cuenta que entre las ciraturas solamente puede existir un héroe, y para un mundo de nxn casillas (con n>3), se deberán crear n-1 criaturas. Contempla la situación en la que una criatura no tenga a dónde moverse en el mapa. Muestra una traza por pantalla cada vez que una criatura se mueva, indicando su nueva posición en el mapa (p.ej. CRIATURA02 avanza a [3,4]).

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


```
using static System.Net.Mime.MediaTypeNames;

namespace Examen
{
    internal class DungeonMaster
    {
        static int NUM_CRIATURAS = 5;
        static int TAMAÑO_JUEGO = 5;
        static void Main(string[] args)
        {
            Laberinto laberinto = new Laberinto(TAMAÑO_JUEGO);
            for(int i=0; i < NUM_CRIATURAS - 1; i++)
            {
                var criatura = new Criatura(i,laberinto);
                var thread = new Thread(criatura.Exec);
                thread.Start();
            }
        }
    }

    public class Laberinto
    {
        public int tamaño;
        public int[,] tablero;
        public int num_criaturas;
        private Random rand = new Random();

        private SemaphoreSlim EM_casillas = new SemaphoreSlim(1);
        private SemaphoreSlim EM_criaturas = new SemaphoreSlim(1);
        private SemaphoreSlim EM_coordenadas = new SemaphoreSlim(1);

        public Laberinto( int t)
        {
            tamaño = t;
            tablero = new int[tamaño,tamaño];

            EM_casillas.Wait();
            for(int i=0; i<tamaño-1; i++)
            {
                for(int j=0; j<tamaño-1; j++)
                {
                    tablero[i, j] = 0; //Todas las casillas libres
                }
            }
            EM_casillas.Release();
            num_criaturas = 0;
        }

        public void agregarCriatura(Criatura c)
        {
            Random rand = new Random();
            int x, y;
            EM_criaturas.Wait();

            if(num_criaturas== (tamaño * tamaño))
```

```
            {
                Console.WriteLine($"Criatura {c.id} no se puede añadir (falta espacio)");
                EM_criaturas.Release();
            }
            else
            {
                num_criaturas++;
                EM_criaturas.Release();

                do
                {
                    x = rand.Next(0, tamaño-1);
                    y = rand.Next(0, tamaño-1);

                } while (casillaLibre(x, y) == false);

                Console.WriteLine($"Criatura {c.id} se coloca en {x},{y}");
            }
        }

        public void moverCriatura(Criatura c)
        {
            int dir = rand.Next(0, 4);
            int nextX=0;
            int nextY=0;
            int actX = c.posx;
            int actY = c.posy;

            switch (dir)
            {
                case 0: //hacia arriba
                    nextX = actX;
                    if(actY == 0)
                    {
                        nextY = 0;
                    }
                    else
                    {
                        nextY = actY - 1;
                    }
                    Console.WriteLine($"Criatura {c.id} se mueve hacia arriba");
                    break;
                case 1: //hacia abajo
                    nextX = actX;
                    if (actY == tamaño-1)
                    {
                        nextY = tamaño-1;
                    }
                    else
                    {
                        nextY = actY + 1;
                    }
                    Console.WriteLine($"Criatura {c.id} se mueve hacia abajo");
                    break;
                case 2: //derecha
```

```
                if (actX == tamaño - 1)
                {
                    nextX = tamaño - 1;
                }
                else
                {
                    nextX = actX + 1;
                }
                nextY = actY;
                Console.WriteLine($"Criatura {c.id} se mueve hacia derecha");
                break;
            case 3: //izquierda
                if (actX == 0)
                {
                    nextX = 0;
                }
                else
                {
                    nextX = actX - 1;
                }
                nextY = actY;
                Console.WriteLine($"Criatura {c.id} se mueve hacia izquierda");
                Thread.Sleep(800);
                break;
        }

        if (casillaLibre(nextX, nextY) == true)
        {
            c.posx = nextX;
            c.posy = nextY;
            liberarCasilla(actX, actY);
            Console.WriteLine($"Criatura {c.id} en la posición {nextX},{nextY}");
        }
        else
        {
            Console.WriteLine($"Criatura {c.id} no se puede mover");
        }
    }

    public bool casillaLibre(int x, int y)
    {
        bool libre = true;
        EM_casillas.Wait();

        if (tablero[x, y] == 0)
        {
            libre = true;
            tablero[x, y] = 1;
        }
        else
        {
            libre = false;
        }

        EM_casillas.Release();
```


```
            return libre;

        }

        public void liberarCasilla(int x, int y)
        {
            EM_casillas.Wait();
            tablero[x, y] = 0;
            EM_casillas.Release();
        }
    }

    public class Criatura
    {
        Laberinto lab;
        public int id;
        public int posx;
        public int posy;
        public Criatura(int i, Laberinto l)
        {
            id = i;
            lab = l;
        }

        public void Exec()
        {
            lab.agregarCriatura(this);
            while (true)
            {
                lab.moverCriatura(this);
            }
        }
    }
}
```

PARTE 3 (2 puntos)
Implementa el sistema de combate. Cada vez que una acción de ataque sea exitoso (no sea esquivado o fallido), se deberá imprimir una traza con el id del atacante (p.ej. HEROE, CRIATURA01, etc), su posición en el mapa, id del defensor, su posición en el mapa, y un texto explicando el tipo de ataque (p.ej. MURCIELAGO DRENA VIDA, MORDEDURA VENENOSA DE ARAÑA, MIRADA PETRIFICANTE DE GÁRGOLA, o entreo otros, para el héroe, ATAQUE CON ESPADA, HACKA, MAZA etc., METEORO, BOLA DE FUEGO, RELÁMPAGO, LLAMA SAGRADA, ESPADA ESPIRITUAL...). Asimismo, cuando el héroe interrumpa un ataque, también mostrará una traza parecida, pero en el texto explicará que ha defendido. Por último, si un ataque es fallido o esquivado, se indicará también por traza.
El alumno debe adjuntar todos los archivos .cs necesarios para ejecutar correctamente el ejercicio. Se deben incluir archivos distintos para la PARTE 2 y PARTE 3. Los archivos se deben adjuntar directamente, sin incluir en archivos comprimidos de ningún tipo.

EXÁMENES 2023 - Práctica

Pregunta 8.- En 1989, Peter Molyneux revolucionó el género de los videojuegos de estrategia introduciendo el concepto de simulación de dios con POPULOUS. En dicho juego, el jugador asumía el papel de un poderoso ser divino con el control absoluto sobre el destino de una civilización. El objetivo consistía en ayudar a su pueblo a prosperar y expandirse, pudiendo asimismo utilizar el poder divino para influir en la geografía y en los elementos naturales del mundo, como crear montañas, bajar valles o invocar desastres naturales. Esta innovadora mecánica de manipulación del terreno combinada con el manejo estratégico de los recursos y la gestión de la población hicieron de POPULOUS una experiencia única y adictiva.

Para el examen se solicita replicar una parte del comportamiento del juego en C#. Para ello, se van a enfrentar dos bandos, que constan inicialmente de 5 soldados cada uno. Cada soldado dispone de un valor aleatorio de vida y de ataque. Los soldados, de manera independiente, pueden realizar tres tareas **diferentes: atacar a un soldado rival, fusionarse con un soldado aliado, o explorar el mundo**. **Cuando atacan a un soldado rival, el soldado atacado resta a su vida el valor de ataque del soldado atacant**e. Al llegar la vida de un soldado a 0, este se **elimina automáticamente** de la partida. Cuando se **fusionan, uno de los soldados desaparece, y el otro soldado pasa a tener como vida y ataque la suma de sus valores con los del soldado eliminado**. Por último, **al explorar, el soldado no hace ninguna acción relevante. Cada vez que realiza una tarea, esperará un tiempo hasta poder realizar otra, que será decidida siempre de manera aleatoria.**

Además, los bandos van generando nuevos soldados cada cierto tiempo. Dicho tiempo depende del número de soldados que disponga dicho bando, provocando, que cuantos más soldados tenga un bando, más rápido genere nuevos soldados, y viceversa.

Por último, para evitar que se alargue mucho el juego (por ejemplo, 20 segundos), se activará el modo guerra santa, en el que los bandos ya no generan nuevos soldados y en el que estos ya solo disponen de las acciones de atacar o fusionarse. De esta forma, los soldados se enfrentan hasta que uno de los bandos se queda sin soldados, momento en el que el juego se detiene, mostrando por consola el bando vencedor. Esta situación puede ocurrir de manera previa al modo guerra santa.

Parte 1 (1 punto) Plantea el ejercicio. ¿qué clases necesitas? ¿qué tipos de hilos hay que definir? ¿qué relaciones de exclusión mutua y sincronización condicional necesitas? Define también, si fuera necesario, otros elementos necesarios como estructuras de datos, interrupciones, eventos, etc.

- Clase Soldado
  - Variables: int vida, int daño, string nombre;
  - Con los métodos Atacar(), Fusionar(), Explorar() y EstaMuerto();
- Clase Bando:
  - Variables: List<Soldado> soldados, string nombre
  - EM: Mutex EM_lista pra proteger la lista de soldados al añadir y eliminar
  - Evento Action<Soldado> SoldadoEliminado cuando uno de ellos muere o se fusiona
  - Métodos: AgregarSoldado(Soldado s), EliminarSoldado(Soldado s), RealizarAccion(Soldado s, Bando e), GenerarSoldado()
- Clase Juego:
  - Variables: Bando bandoA, Bando bandoB, bool modoGuerraSanta, int durGuerraSanta
  - Métodos: Iniciar(), RealizarAccionesBando(Bando b, Bando e), OnSoldadoEliminado(Soldado eliminado), ComprobarGanador(),ModoGuerraSanta()
  - Hilos necesarios:
    - Dos hilos para la generación de los soldados de cada bando
    - Dos hilos para la realización de acciones de cada bando

Parte 2 (3 puntos) En esta parte, los alumnos deben resolver las funcionalidades definidas en el enunciado, implementando el comportamiento de los soldados, la generación de nuevos soldados, las tareas de ataque, fusión y exploración, así como el modo guerra santa y la detección del bando vencedor.

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

namespace Ejercicios_examenes
{
    public class _2023_POPULOUS
    {
        static void Main()
        {
            Juego juego = new Juego();
            juego.Iniciar();
        }
    }

    public class Soldado
    {
        public int vida;
        public string nombre;
        public int daño;
        Random rand = new Random();

        public Soldado(string n)
        {
            nombre = n;
            vida = rand.Next(1,10);
            daño = rand.Next(1,10);
        }

        public void Atacar(Soldado s)
        {
            Console.WriteLine($"{nombre} ataca a {s.nombre} causando {daño} de daño.");
            s.Daño(daño);
        }

        public void Daño(int d)
        {
            vida -= d;
            if(vida <= 0)
            {
                Console.WriteLine($"{nombre} se elimina ");
                //Thread.Sleep(100000);
            }
        }
        public void Fusionar(Soldado s)
        {
            vida += s.vida;
            daño += s.daño;
```

```
            Console.WriteLine($"{nombre} se fusiona con {s.nombre}");
        }

        public void Explorar()
        {
            Console.WriteLine($"{nombre} esta explorando");
            Thread.Sleep(1000);
        }

        public bool EstaMuerto()
        {
            if(vida == 0)
            {
                return true;
            }
            else
            {
                return false;
            }
        }

    }

    public class Bando
    {
        private List<Soldado> soldados;
        private string name;

        private Mutex EM_lista = new Mutex();

        private Random rand = new Random();

        public  Bando(string n)
        {
            name = n;
            soldados= new List<Soldado>();
        }

        public void AgregarSoldado(Soldado s)
        {
            EM_lista.WaitOne();
            soldados.Add(s);

            Console.WriteLine($"{s.nombre} añadido al bando {name}");
            EM_lista.ReleaseMutex();
        }

        public void EliminarSoldado(Soldado s)
        {
            EM_lista.WaitOne();
            if (soldados.Remove(s))
            {
                Console.WriteLine($"{s.nombre} eliminado del bando {name} quedan
{soldados.Count}");
```


```
        }
        EM_lista.ReleaseMutex( );
    }

    public void GenerarSoldado()
    {
        while (true)
        {
            Thread.Sleep(10000/(soldados.Count+1));
            var nuevoSoldado = new Soldado($"S{soldados.Count + 1}");
            AgregarSoldado(nuevoSoldado);
        }
    }

    public void RealizarAccion(Soldado s, Bando e)
    {
        while (s.EstaMuerto() == false)
        {
            Thread.Sleep (300);

            int accion = rand.Next(0, 3);

            switch(accion)
            {
                case 0: //atacar
                    var s_enemigo = e.ObtenerSoldadoAleatorio();
                    if(s_enemigo != null)
                    {
                        s.Atacar(s_enemigo);
                        if(s_enemigo.EstaMuerto()==true)
                        {
                            e.EliminarSoldado(s_enemigo);
                        }
                    }
                    break;
                case 1: //fusionar
                    var aliado = ObtenerSoldadoAleatorio();
                    if(aliado != null)
                    {
                        s.Fusionar(aliado);
                        EliminarSoldado(aliado);
                    }
                    break;
                case 2: //Explorar
                    s.Explorar();
                    break;
            }
        }
    }

    public Soldado ObtenerSoldadoAleatorio()
    {
        if(soldados.Count == 0)
        {
            return null;
```


```
            }
            return soldados[rand.Next(0, soldados.Count)];
        }

        public int ObtenerCantidadSoldados()
        {
            EM_lista.WaitOne();
            int num = soldados.Count;
            EM_lista.ReleaseMutex();
            return num;
        }
    }

    public class Juego
    {
        private Bando bandoA;
        private Bando bandoB;
        private bool modoGuerraSanta;
        private int durGuerraSanta;

        Thread generacionBandoA;
        Thread generacionBandoB;
        public Juego()
        {
            bandoA = new Bando("bandoA");
            bandoB = new Bando("bandoB");
            durGuerraSanta = 20000;
        }

        public void Iniciar()
        {
            for(int i = 1; i <= 5; i++)
            {
                bandoA.AgregarSoldado(new Soldado($"A{i}"));
                bandoB.AgregarSoldado(new Soldado($"B{i}"));
            }
            generacionBandoA = new Thread(bandoA.GenerarSoldado);
            generacionBandoB = new Thread(bandoB.GenerarSoldado);
            generacionBandoA.Start();
            generacionBandoB.Start();

            Thread accionesBandoA= new Thread(()=>RealizarAccionesBando(bandoA, bandoB));
            Thread accionesBandoB= new Thread(()=>RealizarAccionesBando(bandoB, bandoA));
            accionesBandoA.Start();
            accionesBandoB.Start();

            while (true)
            {
                if (ComprobarGanador() == true)
                {
                    Thread.CurrentThread.Abort();
                }
            }
        }
```

```
        private void RealizarAccionesBando(Bando b, Bando e)
        {
            while (b.ObtenerCantidadSoldados() > 0)
            {
                Soldado soldado = b.ObtenerSoldadoAleatorio();
                if (soldado != null)
                {
                    b.RealizarAccion(soldado, e);
                    Thread.Sleep(100);
                }
            }
        }

        public bool ComprobarGanador()
        {
            if (bandoA.ObtenerCantidadSoldados() == 0)
            {
                Console.WriteLine("Bando B es el ganador!");
                return true;
            }
            else if (bandoB.ObtenerCantidadSoldados() == 0)
            {
                Console.WriteLine("Bando A es el ganador!");
                return true;
            }
            return false;
        }


    }
}
```

EXAMEN MAYO 2024 - Teoría

(0.25 ptos) ¿Cuál es una ventaja de usar modelos puros de paso de mensajes?

a. Mayor facilidad de depuración
b. Mayor escalabilidad
c. Menor uso de memoria
d. Mejor rendimiento en sistemas de un solo procesador

(0.25 ptos) ¿Qué técnica se utiliza para evitar condiciones de carrera al compartir estructuras de datos entre varios hilos?

a. Exclusión mutua
b. Estructuras de datos no concurrentes
c. Algoritmos recursivos
d. Sincronización de tiempo

(0.25 ptos) En la programación concurrente, ¿qué se entiende por "indeterminismo"?

a. Los resultados de una ejecución dependen de un proceso causal lineal
b. Las ejecuciones del programa pueden producir resultados diferentes dependiendo de la secuencia de instrucciones.
c. Los procesos se ejecutan en un solo procesador.
d. Todas las ejecuciones del programa producen el mismo resultado.

(0.25 ptos) ¿Cuál es el propósito principal de las instrucciones atómicas en programación concurrente?

a. Mejorar la velocidad de ejecución de los procesos.
b. Permitir la comunicación entre procesos.
c. Asegurar que ciertas operaciones se realicen de manera indivisible y sin interferencias.
d. Reducir el uso de memoria compartida.

(1 pto) Escoge una propiedad de corrección de seguridad, explícala e ilústrala con un ejemplo en C#.

La ausencia de interbloqueos pasivos o deadlocks: es un bloqueo que ocurre cuando uno o varios procesos queda esperando indefinidamente de forma que no se puede avanzar en el programa.

```
volatile int compartida=0;
SemaphoreSlim sem;
public void proceso1{
      Console.WriteLine("soy el proceso 1");
      sem.Wait();
      compartida++;
      sem.Release();
}
public void proceso2{
      Console.WriteLine("soy el proceso 2");
      sem.Wait();
      compartida++;
      sem.Release();
}
public void main(){
```


```
        sem= new SemaphoreSlim(0);
        new Thread(proceso1).Start();
        new Thread(proceso2).Start();
    }
```

En este código el semáforo se inicializa con cero permisos y por eso ambos procesos quedarán siempre esperando produciéndose un deadlock.

(1.5 ptos) Discute los desafíos y soluciones al recorrer una estructura de datos concurrente mientras se modifica.

Cuando se hacen accesos a una estructura de datos que se modifica se pueden dar dirtyreads. Estas dirtyreads ocurren cuando al acceder a los datos de estas estructuras se pueden obtener resultados tanto antiguos como nuevos debido a los cambios concurrentes en estas. Para solucionar este problema se utilizan las snapshots que son instantáneas de una estructura de datos para poder realizar el acceso a ella con la seguridad de que los datos no van a cambiar mientras estamos leyendo.

Podemos encontrar estructuras de datos ya preparadas para los accesos concurrentes y que tienen en cuenta la posibilidad de dirtyreads. Estas estructuras suelen utilizar bloqueos ligeros para evitarlas y las operaciones de escritura suelen ser bloqueantes para evitar que se lea una estructura mientras está siendo modificada.

(1.5 ptos) En la ecuación del principio de información enunciada por Singhal y Zhida en 1998, uno de los parámetros directamente proporcionales al cálculo de los recursos de un videojuego es la puntualidad con la que debe entregarse cada paquete de datos. Este principio destaca la necesidad de encontrar un equilibrio entre la responsividad y la consistencia en los sistemas de videojuegos.
Se pide lo siguiente:

1. **Define qué se entiende por responsividad y consistencia en el contexto de los videojuegos.**
2. **Proporciona ejemplos de situaciones en videojuegos donde se deba priorizar la responsividad sobre la consistencia y viceversa.**
3. **Comenta las posibles estrategias que los desarrolladores pueden emplear para lograr un equilibrio adecuado entre estos dos aspectos.**

La **responsividad** consiste en que la información llegue de forma rápida a los nodos.
La **consistencia** consiste en que todos los nodos del juego tengan la misma información y depende mucho de la arquitectura (centralizada, distribuida, etc)
En un juego donde compites contra otros jugadores como en un FPS será importante tener una alta responsividad ya que en este tipo de juegos priman los reflejos de los jugadores.
Por otro lado en juegos por ejemplo por turnos la responsividad seguirá siendo importante aunque no esencial. Será más importante la consistencia y que todos los jugadores vean la misma información.
Para ello los desarrolladores utilizan las propiedades de compensación.

- **Optimización del protocolo**: ya que enviar mensajes tiene un coste y por ello se reduce el tamaño de los mensajes (con técnicas como la comprensión y la cantidad de mensajes enviados.
    - Al reducir el tamaño de los mensajes aumenta el coste computacional de procesado ya que hay que descomprimirlo. Para comprimir se utilizan técnicas

como la comprensión delta (evitar info redundante), reducir el nivel de detalle de la información enviada o filtrar la información no importante.
    - Para reducir el número de mensajes se utilizan técnicas como el time-based approach o el quorum-based approach que envían la información tras x tiempo o tras llegar a una cantidad determinada
- **Interpolación en el cliente**: se utilizan los dos últimos ticks para crear frames intermedios, de esta forma el cliente se actualiza con un framerate superior a los ticks del servidor permitiendo un juego más fluido
- **Extrapolación en el cliente**: consiste en hacer una predicción del estado de algunos objetos y luego realizar la convergencia con la realidad que viene del servidor. Sirve para simular un comportamiento continuo y fluido de los objetos.
- **Predicción en el cliente**: se trata de realizar acciones en el cliente nada más recibir el input ya que estas acciones serán iguales a las recibidas por el servidor siempre y cuando no haya ninguna interacción externa. En caso de haberla sería necesario realizar la convergencia. Esta propiedad busca aumentar la responsividad.
- **Interest Manager**: se da cuando el cliente no necesita conocer toda la información del juego y el servidor se encarga de pasar únicamente la información de utilidad a cada jugador.
- **Simulación sincronizada**: en esta propiedad todos los clientes tienen una copia total del juego y se realizan simulaciones deterministas. Solo se pasa aquella información que cambia (es decir, que no es determinista).