Trabajo Práctico 2
Paradigmas de la Programación

I102 - Grupo 1

Abril Diaco
Martina Gurevich

Profesor: Mariano Scaramal

Universidad de San Andrés
Junio de 2025

Introduccion

A lo largo del siguiente trabajo se buscó resolver las consignas de la manera mas clara, eficiente y fiel a lo pedido posible. Todos los ejercicios fueron compilados con los flags requeridos, para los cuales no hubo ningun *warning* en la ejecucion de los mismos.

1. Pokedex

El propósito de este ejercicio es la creación de una `Pokedex` la cual almacena pokemones con su respectiva información. Para este ejercicio, tomamos la decisión de hacer el punto extra(d), tal que en todas las clases se implementaron métodos de serialización.

Instanciación de Clases

Clase `Pokemon`

Se creó la clase `Pokemon`, siguiendo lo pedido en la consigna. Se definieron los métodos necesarios para el correcto funcionamiento. Además, se sobrecargó el operador `==`, el cual compara los nombres de los Pokemones. Esto fue necesario ya que, en la instancia de la clase `Pokedex`, se hace uso del `Pokemon` como clave de un `std::unordered_map`, y esta sobrecarga permite buscar dentro del contenedor objetos `Pokemon`.

Clase `PokemonInfo`

Se creó la clase `PokemonInfo`, siguiendo lo pedido en la consigna. Para esta clase, se optó por utilizar distintos contenedores en los atributos.

- **AtaquesDisponiblesPorNivel**: se utilizó un `std::unordered_map` por el motivo de que es conveniente el uso de un diccionario, pero no es condición necesaria que este mismo esté ordenado. De esta manera, el acceso por claves al contenedor es de $\mathcal{O}(1)$ gracias al sistema de *Hashing*.
- **ExperienciaProximoNivel**: se utilizó un `std::vector<int>` en el cual cada posición representa el nivel al cual el pokémon puede tener acceso. A su vez, el acceso a cada nivel tiene orden $\mathcal{O}(1)$ indexando en la posición correspondiente.

Cabe destacar que, a la hora de guardar la `PokemonInfo` como valor en el `unordered_map` de la `Pokédex`, el compilador genera un operador de asignación implícito. Como no hay manejo de memoria dinámica, no es necesario definir un constructor de copia.

Clase `Pokedex`

La clase `Pokedex` fue diseñada conforme a las especificaciones de la consigna. Para almacenar la información de los Pokemons, se utilizó un contenedor `std::unordered_map`, en el cual las claves son objetos de la clase `Pokemon` y los valores son instancias de la clase `PokemonInfo`. Dado que `unordered_map` requiere una función de hash para operar con claves personalizadas, se definió un *functor* denominado `PokemonHash`, el cual aplica una función de hash sobre el atributo `nombre` del objeto `Pokemon`. Esto garantiza que las instancias puedan ser correctamente indexadas y recuperadas mediante hashing.

Adicionalmente, se implementó un constructor sobrecargado que permite inicializar una `Pokedex` con un nombre de archivo. Si se proporciona dicho nombre, el constructor invoca automáticamente un método de deserialización que carga los datos previamente guardados en el archivo binario correspondiente. Por otro lado, cada vez que se agrega o elimina un Pokemon mediante los métodos `agregarPokemon` o `eliminarPokemon`, se actualiza el archivo serializando el estado actual del contenedor.

En el archivo de ejemplo provisto, se demuestra este comportamiento: primero se crea una instancia de `Pokedex` con un nombre de archivo, se agregan y eliminan varios Pokemons, y posteriormente se instancia una nueva `Pokedex` a partir del mismo archivo, la cual restaura correctamente los datos serializados previamente.

2. Control de Aeronave en Hangar Automatizado

En este ejercicio se implementó una simulación utilizando `std::mutex` y `std::jthread` para modelar el despegue coordinado de cinco drones desde una plataforma compartida. El objetivo fue garantizar que cada drone pueda despegar únicamente cuando sus zonas adyacentes estén libres, evitando condiciones de carrera y *deadlocks* mediante mecanismos apropiados de sincronización.

Clase `Despegue`

Atributos

Se definieron los atributos `zonas` y `m2`, con el objetivo de controlar las zonas habilitadas para el despegue de cada drone y la buena impresión por pantalla de los eventos.

- **Zonas**: Es un arreglo de `std::mutex` de tamaño ZONAS (constante que representa la cantidad total de zonas de despegue). Cada mutex representa una zona que debe ser adquirida por un drone antes de iniciar su ascenso. La zona i es compartida entre el drone i y el *i-1*
- **m2**: Es un `std::mutex` utilizado exclusivamente para proteger las impresiones por consola. Dado que múltiples hilos pueden intentar escribir simultáneamente, esta protección evita que los mensajes se superpongan.

Metodos

Se definió un único método void `Volar(int i)` el cual simula el proceso de despegue de un drone, haciendo las verificaciones y procesos necesarios para bloquear y acceder a las zonas disponibles, tal que los drones en el resto de `std::jthread` no generen deadlocks al querer acceder a zonas ocupadas.

El funcionamiento del método es el siguiente:

- **Identificación de zonas:** Cada drone requiere acceso exclusivo a dos zonas: su zona izquierda `(zonas[i])` y su zona derecha `(zonas[(i+1) mod ZONAS])`. Este esquema garantiza que cada zona esté asociada exactamente a dos drones, evitando accesos de memoria inválidos
- **Bloqueo múltiple de zonas**: Para evitar *deadlocks*, se utiliza la función `std::lock(zonaIzq, zonaDer)`. Esta función intenta adquirir ambos mutex en un orden seguro, evitando condiciones en las que dos hilos se bloqueen esperando indefinidamente por los mutex del otro. En el caso que una de las zonas esté ocupada, el hilo se quedará detenido en esa línea hasta que ambas zonas estén liberadas, evitando así el despegue indebido de un drone.
- **Ascenso**: Una vez adquiridos ambos mutex, el drone simula el ascenso durante 5 segundos mediante `std::this_thread::sleep_for(std::chrono::seconds(5))`. Durante este tiempo, como el drone no ha alcanzado los 10m, las zonas se mantienen bloqueadas.
- **Impresión**: Toda salida por consola relacionada con el evento de despegue se encuentra protegida por m2 mediante un `std::lock()`, garantizando que la salida sea legible y no interrumpida por otros hilos.
- **Desbloqueo de Zonas**: una vez alcanzados los 10m, las zonas son liberadas con el método `std::unlock()`.

Finalmente, en el archivo de ejemplo, se implementó un ciclo for que lanza cinco instancias de `std::jthread`, cada una ejecutando el método volar, simulando así el proceso de despegue concurrente de los cinco drones.

3. Sistema de Monitoreo y Procesamiento de Robots Autónomos

En esta última sección del trabajo práctico , se utilizaron mecanismos de multithreading. Para las tareas de inspección periódicas y reportes generados por distintos sensores. Se tuvieron en cuenta `std::mutex`, `std::unique_ lock`, `std::lock_guard`, `std::threads` y `std::conditional_variables`.

Variables globales

Para el correcto funcionamiento del programa se tomó la decisión de hacer variables globales:

- `int sensores_terminados`: lleva cuenta de cuántos sensores ya finalizaron.
- `NUM_SENSORES`, `NUM_ROBOTS`: cantidades constantes de robots y sensores.
- `bool terminado`: booleano que indica true si las tareas de todos los sensores han terminado.
- `atomic_int contadorTareas` : variable atómica para utilizar como IdTarea.
- `queue<Tarea>cola`: almacenar las tareas generadas por los sensores.
- `mutex mtx`: mutex para controlar la produccion y procesamiento de tareas de manera segura.
- `std::condition_variable condicionEsperada`: retornará true si la cola de tareas no está vacía o si todos los sensores terminaron de producir Tareas. Es utilizada para avisarle a los Robots cuando accionar.

A su vez se creó un `Struct Tarea` y dos funciones `Void Sensor(int id Sensor)` y `void Robot(int idRobot)`.

Struct Tarea

Incluye un `idSensor`, que indica número de Sensor. Un `idTarea` que indica a qué número de tarea corresponde y una descripción de la tarea.

Función `void Sensor()`

Simula un sensor que produce 3 Tareas, mediante un bucle `for`. Cada tarea se creó dentro de un bloque protegido por un `lock_guard<mutex>` en un scope aparte, tal que cuando la ejecución termine, el mutex se desbloquea automáticamente y se notifica que el sensor terminó de agregar tareas a la cola. También, se utilizó `std::sleep_for` para simular tiempo de creación.

Al finalizar sus tareas, el sensor incrementa el contador global `sensores_terminados`. Cuando todos los sensores completan sus tareas, se cambia el estado de la variable booleana y se notifica a los Robots la finalización del trabajo.

Función `void Robot()`

Esta función se encarga de ejecutar las tareas siempre y cuando haya sensores manejando la creación de las mismas. Al robot al que se le asigne una tarea, debe esperar mediante `condicionEsperada.wait(condicion)` que se agregue productos a la cola para poder llevar a cabo la tarea.

Para evitar condiciones de carrera, esta parte del código se encuentra dentro de un `unique_lock<mutex>lockUnique(mtx);` para un manejo más controlado de la ejecución del robot. Si hay tareas, las toma de la cola, desbloquea el mutex, y las ejecuta (simulado con `sleep_for`). Si esto no sucede, los sensores finalizaron y se imprime un mensaje para comentar la condición.

Función `main()`

En el `main` se crearon dos vectores de `std::jthreads`, es decir que los hilos se unen automáticamente al finalizar, lo que evita usar `join()`:

- `vector<jthread>robots`
- `vector<jthread>sensores`

Cada `std::jthread` utiliza cada una de sus respectivas funciones y opera sobre ellas.

Finalmente, mediante este código se evidencia que el uso de `lock_guard` en los sensores garantiza eficiencia y seguridad al liberar automáticamente el mutex. La elección de `unique_lock` en los robots permite utilizar `condition_variable` correctamente. A su vez, la variable atómica previene condiciones de carrera al generar IDs únicos. Por ultimo, se considera que el diseño creado asegura que no se pierdan tareas y que todos los hilos terminen correctamente.

Comandos de compilación

Para la compilacion se utilizó CMake, por lo que se deberán seguir los siguienets pasos para compilar cada ejercicio.

1. Debe posicionarse en la carpeta del ejercicio correpondiente.
2. En la terminal ejecutar:
   a) `cd build`
   b) `make`
   c) `./bin/programa`

Conclusión final

A lo largo de este trabajo se exploraron tanto el uso y manejo de mecanismos de sincronización como la administración de contenedores de la STL. Estas herramientas fueron implementadas de manera tal que permitieron aumentar la eficiencia y robustez del código desarrollado.

Al concluir este Trabajo Práctico, se adquirieron conocimientos fundamentales sobre el manejo adecuado del ciclo de vida de los hilos *(thread lifetime)*, el uso de mecanismos de sincronización y técnicas de concurrencia orientadas a evitar condiciones de carrera *(race conditions)* y bloqueos mutuos *(deadlocks)*. Asimismo, se profundizó en el uso eficiente de contenedores, seleccionando aquellos que mejor se adaptaban a las necesidades específicas de la estructura del programa, priorizando la eficiencia algorítmica y el uso adecuado de memoria.