PRÁCTICA: "Laboratorio de uso de monitores en C++"

Objetivos:

- La presente guía pretende introducir conceptos prácticos de monitores en C++.

1. Acerca de monitores

Propuestos por Hoare en 1974, son un mecanismo abstracto de datos que encapsula un conjunto de recursos y un conjunto de operaciones sobre dichos recursos.
Un proceso solo puede acceder a las variables del monitor usando los procedimientos exportados por el monitor. La exclusión mutua en el acceso a los procedimientos del monitor, está garantizada.

2. Ventajas de los monitores

- **Simplicidad y claridad:** Los monitores proporcionan un mecanismo claro y estructurado para la sincronización, lo que facilita la comprensión y el mantenimiento del código.
- **Seguridad:** La exclusión mutua garantizada reduce la posibilidad de errores de concurrencia, como condiciones de carrera y deadlocks.
- **Modularidad:** Al encapsular recursos y operaciones, los monitores promueven un diseño modular, lo que facilita la reutilización de código y la separación de preocupaciones.

3. Aplicaciones prácticas de los monitores

Los monitores se utilizan en diversas aplicaciones donde es necesario gestionar la concurrencia y asegurar la exclusión mutua, tales como:

- **Sistemas operativos:** Para gestionar recursos compartidos como memoria y dispositivos de entrada/salida.
- **Programación de redes:** Para sincronizar el acceso a conexiones de red y datos compartidos.
- **Sistemas de bases de datos:** Para controlar el acceso concurrente a registros y tablas.

4. Mutex en Monitores

El término **mutex** proviene de *mutual exclusion* y representa un mecanismo que evita que múltiples procesos accedan simultáneamente a una misma sección crítica. En el contexto de los monitores, el mutex está **implícito**: cuando un proceso entra a un procedimiento del monitor, se le concede automáticamente la exclusividad de acceso, impidiendo que otros procesos ingresen al mismo tiempo.

Esta exclusión mutua implícita garantiza que no se produzcan interferencias durante la manipulación de los datos compartidos dentro del monitor, proporcionando una forma segura de sincronización sin que el programador tenga que gestionar directamente los bloqueos.

5. Funcionalidades del Mutex en Monitores

a) **Exclusión Mutua Automática**
El mutex integrado en el monitor asegura que solo un proceso puede estar ejecutando un procedimiento del monitor en un momento dado. Esto elimina la necesidad de usar primitivas externas como lock o semaphore para proteger secciones críticas.

b) **Sincronización Condicional**
Los monitores suelen incluir mecanismos de sincronización adicional, como variables de condición (condition variables), que permiten a los procesos suspender su ejecución y ceder el control del mutex cuando no se cumplen ciertas condiciones. Funciones como wait, signal o notify dependen de este mecanismo.

c) **Simplificación del Código**
Al encapsular la lógica de sincronización dentro del monitor, el uso de mutex se vuelve transparente para el programador, reduciendo la probabilidad de errores como olvidarse de liberar un bloqueo o crear interbloqueos (deadlocks).

d) **Bloqueo y Reanudación Coordinada**
El mutex no solo bloquea el acceso a secciones críticas, sino que también permite coordinar la reanudación de procesos de forma ordenada, lo que es clave para la correcta implementación de políticas de prioridad o condiciones de espera.

Clases y métodos para mutex	
lock_guard<mutex>	Proporciona un bloqueo simple
unique_lock<mutex>	Puede bloquear y desbloquear manualmente, necesario para condition_variable
scoped_lock o try_lock	Para múltiples mutex.
lock()	Bloquea el mutex.
unlock()	Libera el mutex.
Métodos para condition_variable	
wait(lock)	Espera a ser notificado, liberando el mutex durante la espera.
wait(lock, pred)	Añade un predicado que evita “despertares falsos”.
notify_one()	Despierta un hilo que esté esperando en esa condition_variable
notify_all()	Despierta todos los hilos que estén esperando.

6. Ejemplo de uso de monitores en C++

En C++, no hay un tipo de dato o palabra clave "monitor" como en otros lenguajes como Pascal o Java. Sin embargo, los monitores, son estructuras para sincronizar el acceso a recursos compartidos por múltiples hilos, se pueden implementar en C++ utilizando la combinación de mutex (exclusión mutua) y condición de variable.

Ejemplo 1:

Si tenemos una variable compartida “i”, diseñe un programa que gestione su incremento e impresión de su valor.

```
#include <iostream>
#include <thread>
#include <mutex>
#include <chrono>

using namespace std;
int i = 0;              // Variable compartida
mutex mtx;              // Mutex para controlar el acceso

void inc() {
    lock_guard<mutex> lock(mtx);
    i += 2;
}
void valor() {
    lock_guard<mutex> lock(mtx);
    cout << i << endl;
}
void proceso_incrementar() {
    while (true) {
        inc();
        this_thread::sleep_for(chrono::milliseconds(500));
    }
}
```

```
void proceso_imprimir() {
    while (true) {
        valor();
        this_thread::sleep_for(chrono::milliseconds(500));
    }
}
int main() {
    thread t1(proceso_incrementar);
    thread t2(proceso_imprimir);

    t1.join();
    t2.join();

    return 0;
}
```

Al ejecutar el programa, podrá verificar que la variable compartida “i”, se va incrementando e imprimiendo. Con *lock_guard<mutex>...* bloquea la sección de código para que tenga acceso exclusivo, asegurando la exclusión mutua, al salir de su ámbito (por ejemplo la funcionalidad donde se creó) el mutex se desbloquea automáticamente, sin embargo, notará que en algunos casos se imprime el mismo valor, o se incrementa más de una vez, esto es debido a que si bien solo un hilo ingresa a la sección crítica (incrementado o imprimiendo), existe la posibilidad que el mismo hilo o uno diferente ingrese al mismo proceso.

Para evitar dicho escenario se deberá añadir condición de variable.

Ejemplo 2:

Si tenemos una variable compartida “i”, diseñe un programa que gestione su incremento e impresión de su valor, para este caso el hilo sólo imprimirá después de que la variable compartida haya sido incrementada, asegurando sincronización.

```cpp
#include <iostream>
#include <thread>
#include <mutex>
#include <condition_variable>
#include <chrono>

using namespace std;

int i = 0;                          // variable compartida
bool listo_para_imprimir = false; // bandera de condición

mutex mtx;
condition_variable cv;

void incrementar() {
    while (true) {
        unique_lock<mutex> lock(mtx);
        i += 2;
        cout << "Incrementado a: " << i << endl;

        listo_para_imprimir = true;      // se activa la condición
        cv.notify_one();                 // despierta al hilo imprimir
        lock.unlock();

        this_thread::sleep_for(chrono::milliseconds(500));
    }
}
 void imprimir() {
     while (true) {
         unique_lock<mutex> lock(mtx);

         // Espera hasta que listo_para_imprimir sea true
         cv.wait(lock, [] { return listo_para_imprimir; });

         cout << "Imprimiendo: " << i << endl;

         listo_para_imprimir = false;     // se desactiva la condición
         lock.unlock();

         this_thread::sleep_for(chrono::milliseconds(500));
     }
 }
```

```
int main() {
    thread t1(incrementar);
    thread t2(imprimir);

    t1.join();
    t2.join();

    return 0;
}
```

Al ejecutar el programa, podrá verificar la implementación de *mutex* para asegurar la exclusión mutua y *condition_variable* para la sincronización entre hilos y procesos. Tenga en cuenta que, *unique_lock…* permite bloquear y desbloquear de forma manual, *cv.wait(lock. [] {return listo para imprimir; });* es una expresión lambda que retorna un booleano y en este caso, el hilo solo continuará cuando *listo_para_imprimir == true.*

7. Ejercicios propuestos

1. Respecto al Ejemplo 2, notará que se ejecuta de forma indefinida, agregue una constante que límite dichas iteraciones a 10, tenga en cuenta que los iteradores deberán estar dentro de la funciones incrementar e imprimir, asegure la exclusión mutua y sincronización de procesos.

2. Cree un programa con 3 procesos: *incrementar_1*, *incrementar_2* e *imprimir*, el programa deberá incrementar en 1 y 2 a una variable compartida, cada vez que se incremente deberá ejecutar el proceso *imprimir* (puede utilizar *notify_all*), agregar un límite de 10 iteraciones, asegure la exclusión mutua y sincronización de procesos.

```
Se incrementa 1.
Imprimiendo: 1
Se incrementa 2.
Imprimiendo: 3
Se incrementa 1.
Imprimiendo: 4
Se incrementa 2.
Imprimiendo: 6
Se incrementa 1.
Imprimiendo: 7
Se incrementa 2.
Imprimiendo: 9
.
.
.
```