PRÁCTICA: "Paso de mensajes síncronos en C++"

Objetivos:

- Implementar una simulación del paso de mensajes síncrono entre dos hilos o más hilos utilizando una cola o canal bloqueante.

1. Acerca de Paso de mensajes síncronos

Es un modelo de comunicación en sistemas concurrentes y distribuidos donde **los procesos no comparten memoria**, sino que se comunican **enviando y recibiendo mensajes**.
Se puede implementar entre:

- Hilos en un mismo proceso
- Procesos en un mismo sistema operativo
- Procesos distribuidos en diferentes máquinas

2. Ventajas de Paso de mensaje síncronos

- Simplicidad de razonamiento: al bloquearse, los procesos se sincronizan naturalmente.
- No hay condiciones de carrera por acceso a memoria compartida.
- Muy útil para protocolos de comunicación controlados (RPC, cliente-servidor).
- Evita problemas como inconsistencia de datos compartidos.

3. Aplicaciones prácticas

- **Sistemas operativos distribuidos**:
  Para qué procesos se comuniquen entre sí sin compartir memoria, como en microkernels (ej. Minix).

- **Sistemas embebidos o de control industrial**:
  Para sincronizar sensores y actuadores que deben responder en tiempo real (ej. robot que espera piezas).

- **Aplicaciones cliente-servidor**:
  Para garantizar que el cliente reciba respuesta inmediata del servidor (ej. transacciones bancarias).

- **Sistemas de telecomunicaciones**:
  Para coordinar la transmisión y recepción de datos entre nodos o estaciones.

- **Programas educativos de concurrencia**:
  Para enseñar sincronización estricta entre procesos sin usar memoria compartida.

- **Modelado de procesos concurrentes (CSP)**:
  En teoría de la computación y lenguajes como Go o Erlang, donde la comunicación es vía paso de mensajes.

- **Juegos multijugador en red (por turnos)**:
  Donde un jugador no puede actuar hasta recibir el movimiento del otro.

4. Mutex en Paso de Mensajes síncronos

En muchos sistemas, el paso de mensajes se implementa mediante **buffers compartidos protegidos con mutex**. Para garantizar la **sincronización entre emisor y receptor**, se emplean **variables de condición** que representan:

- notEmpty: el receptor espera hasta que haya un mensaje disponible.
- notFull: el emisor espera hasta que haya espacio para enviar un mensaje.

5. Funcionalidades del Mutex en el Paso de Mensajes síncronos

a) **Exclusión Mutua**
La funcionalidad más básica y esencial del mutex es permitir la exclusión mutua, es decir, garantizar que solo un hilo o proceso pueda acceder a una sección crítica en un momento dado. En el contexto del paso de mensajes, esto significa que si un emisor está escribiendo un mensaje en un área compartida de memoria o en un buffer, ningún otro hilo, ni siquiera el receptor, puede acceder a ese mismo recurso hasta que el emisor haya terminado y libere el mutex. Esto previene que el receptor lea datos incompletos o corrompidos y asegura que las operaciones sobre los mensajes se realicen de manera atómica. La exclusión mutua, por tanto, es el primer paso para construir una comunicación segura entre hilos o procesos concurrentes que comparten memoria.

b) **Prevención de condiciones de carrera**
Otra funcionalidad importante del mutex es evitar condiciones de carrera, que ocurren cuando múltiples hilos acceden a un recurso compartido sin una adecuada sincronización y el resultado de la ejecución depende del orden en que se realicen los accesos. En un esquema de paso de mensajes, una condición de carrera podría presentarse si un hilo intenta leer un mensaje justo cuando otro hilo lo está escribiendo. El mutex previene este tipo de situaciones al imponer un control estricto sobre el acceso a los datos, permitiendo que solo un hilo interactúe con el recurso a la vez. Esto asegura que cada operación, ya sea lectura o escritura de un mensaje, se ejecute completamente antes de que otro hilo pueda intervenir, lo cual mantiene la coherencia y la integridad de los datos.

c) **Sincronización de acceso entre hilos**
Además de proteger los datos, el mutex cumple una función de sincronización al forzar que los hilos se coordinen entre sí antes de acceder a recursos compartidos. En el paso de mensajes, esto significa que el emisor y el receptor deben turnarse para usar el buffer o espacio de comunicación, y que uno debe esperar al otro cuando ya hay un mensaje no procesado o cuando el canal está vacío. Esta sincronización implícita permite que se respete el orden lógico de la comunicación, especialmente en modelos donde se quiere simular un paso de mensajes de tipo síncrono, en el que el emisor no puede continuar hasta que el receptor haya recibido el mensaje. El mutex, al actuar como un candado, impone esta disciplina temporal entre los hilos involucrados.

d) **Protección del buffer compartido en sistemas de comunicación**
Cuando se utiliza una estructura de datos compartida, como una cola o buffer circular, para implementar el paso de mensajes, el mutex garantiza que las operaciones sobre el buffer —como insertar o extraer un mensaje— se realicen sin interferencias. Sin mutex, dos hilos podrían modificar índices o datos del buffer al mismo tiempo, produciendo errores difíciles de detectar. La funcionalidad del mutex aquí es crucial, ya que asegura que las operaciones sean seguras, ordenadas y consistentes. Además, esta protección permite implementar buffers de capacidad limitada que imitan tanto comunicación síncrona (capacidad 1) como asíncrona (capacidad >1), sin comprometer la validez de los mensajes.

Librerías y métodos necesarios en C++:

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. Paso de mensajes síncronos en C++

En C++, el paso de mensajes síncronos se puede implementar usando varios mecanismos de sincronización de las bibliotecas estándar: <*mutex*> y <*condition_variable*>.

Ejemplo 1:

Si tenemos una variable compartida “mensaje”, diseñe un programa que gestione el paso de dicha variable de forma síncrona entre un proceso emisor y receptor, asegure una correcta comunicación (debe existir un mensaje para que el receptor confirme recepción).

```
#include <thread>
#include <mutex>
#include <condition_variable>
#include <queue>

using namespace std;

mutex mtx;
condition_variable cv;
string mensaje;
bool listo = false;

void receptor() {
    unique_lock<mutex> lock(mtx);
    cout<<"----------No hay mensaje, el receptor espera..."<<endl;
    cv.wait(lock, [] { return listo; });
    cout << "Receptor recibio el mensaje: " << mensaje << endl;
    cout<<"----------El receptor proceso el mensaje... "<<endl;
    listo = false;
    lock.unlock();
    cv.notify_one();
}
void emisor() {
    unique_lock<mutex> lock(mtx);
    mensaje = "Hola soy el emisor!";
    cout<<"Emisor envia mensaje: "<<mensaje<<endl;
    listo = true;
    lock.unlock();
    cv.notify_one();
    unique_lock<mutex> wait_lock(mtx);
    cout<<"----------El emisor espera a que el receptor procese el mensaje..."<<endl;
    cv.wait(wait_lock, [] { return !listo; });
    cout << "Emisor continua su proceso." << endl;
}

int main() {
    thread t1(receptor);
    thread t2(emisor);
    t1.join();
    t2.join();
    return 0;
}
```

Salida:

```
----------No hay mensaje, el receptor espera...
Emisor envia mensaje: Hola soy el emisor!
----------El emisor espera a que el receptor procese el mensaje...
Receptor recibio el mensaje: Hola soy el emisor!
----------El receptor proceso el mensaje...
Emisor continua su proceso.
```

Al ejecutar el programa, podrá verificar que la variable compartida “mensaje”, se transmite de forma síncrona entre el proceso emisor y receptor, utilizando funciones de <*mutex*> y <*condition_variable*> con objetivo de bloquear (o poner en espera) y desbloquear los procesos, asegurando la sincronización entre la emisión y recepción de mensajes.

Ejemplo 2:

Para el segundo caso, se creará una cola, el emisor irá emitiendo mensajes y el receptor recibe y confirma recepción.

```
#include <thread>
#include <mutex>
#include <condition_variable>
#include <queue>

using namespace std;

mutex mtx;
condition_variable cv;
queue<int> mensajes;
bool listo = false;
const int N = 3;

void receptor() {
    for (int i = 0; i < N; ++i) {
        unique_lock<mutex> lock(mtx);
        cout<<"----------No hay mensaje el receptor espera..."<<endl;
        cv.wait(lock, [] { return listo; });
        int msg = mensajes.back();
        cout << "Receptor recibio el mensaje: " << msg << endl;
        cout<<"----------El receptor proceso el mensaje..."<<endl;
        listo = false;
        lock.unlock();
        cv.notify_one();
    }
}
```

```
void emisor() {
    for (int i = 0; i < N; ++i) {
        unique_lock<mutex> lock(mtx);
        mensajes.push(i);
        cout<<"Emisor envia mensaje: "<<i<<endl;
        listo = true;
        lock.unlock();
        cv.notify_one();

        unique_lock<mutex> wait_lock(mtx);
        cout<<"----------El emisor espera a que el receptor procese el mensaje..."<<endl;
        cv.wait(wait_lock, [] { return !listo; });
        cout << "Emisor puede continuar con el siguiente mensaje o finalizar. "<<endl;
    }
}
int main() {
    thread hilo1(receptor);
    thread hilo2(emisor);
    hilo1.join();
    hilo2.join();
    return 0;
}
```

Salida:

```
----------No hay mensaje el receptor espera...
Emisor envia mensaje: 0
----------El emisor espera a que el receptor procese el mensaje...
Emisor envia mensaje: 0
----------El emisor espera a que el receptor procese el mensaje...
----------El emisor espera a que el receptor procese el mensaje...
Receptor recibio el mensaje: 0
----------El receptor proceso el mensaje...
----------No hay mensaje el receptor espera...
Emisor puede continuar con el siguiente mensaje o finalizar.
Emisor envia mensaje: 1
----------El emisor espera a que el receptor procese el mensaje...
Receptor recibio el mensaje: 1
----------El receptor proceso el mensaje...
----------No hay mensaje el receptor espera...
Emisor puede continuar con el siguiente mensaje o finalizar.
Emisor envia mensaje: 2
----------El emisor espera a que el receptor procese el mensaje...
Receptor recibio el mensaje: 2
----------El receptor proceso el mensaje...
Emisor puede continuar con el siguiente mensaje o finalizar.
```

Al ejecutar el programa, podrá verificar que se añadió una cola de mensajes, el receptor procesa el último mensaje de la cola (después de cada emisión), deberá tener en cuenta que, una vez cada proceso finalice su objetivo (emitir y receptar), notifica o desbloquea al siguiente proceso (incluso si el bucle for no ha culminado su ciclo en el proceso inicial).

7. Ejercicio propuesto.

Desarrollar un programa en C++ que simule el paso de mensajes mediante comunicación síncrona entre tres procesos: Emisor, Traductor y Receptor.

- El Emisor generará N índices enteros consecutivos, comenzando desde 1.
- El Traductor recibirá cada índice, lo multiplicará por 10 y lo reenviará.
- El Receptor obtendrá el índice traducido, lo mostrará en pantalla y notificará al emisor.

Se debe asegurar que la comunicación entre los procesos siga un flujo estrictamente sincronizado, de modo que:

1. El Emisor no emita un nuevo índice hasta recibir confirmación de que el Receptor procesó el anterior.
2. El Traductor no procese ningún valor hasta que el Emisor lo haya generado.
3. El Receptor no actúe hasta que el Traductor haya traducido el mensaje.

Salida:

```
Emisor emitió: 1
Traductor proceso: 1 a 10
Receptor recibió: 10
Emisor es notificado con recepción, puede continuar o finalizar.
Emisor emitió: 2
Traductor proceso: 2 a 20
Receptor recibió: 20
Emisor es notificado con recepción, puede continuar o finalizar.
Emisor emitió: 3
Traductor proceso: 3 a 30
Receptor recibió: 30
Emisor es notificado con recepción, puede continuar o finalizar.
Emisor emitió: 4
Traductor proceso: 4 a 40
Receptor recibió: 40
Emisor es notificado con recepción, puede continuar o finalizar.
Emisor emitió: 5
Traductor proceso: 5 a 50
Receptor recibió: 50
Emisor es notificado con recepción, puede continuar o finalizar.
```