

Concurrencia en C++11

CONTENIDO:

- Compilar y ejecutar nuestros programas en C++11
- Creación y ejecución de los hilos en C++11
- Exclusión Mutua y sincronización
- Bloqueo Recursivo
- Pausando hilos
- Llamar una vez
- Variables de Condición
- Tipos atómicos

BIBLIOGRAFÍA RECOMENDADA:

- Williams, Anthony. C++ Concurrency in Action: Practical Multithreading. 2012.
- Wicht, Baptiste. C++11 Concurrency Tutorial. 2012.

Compilar y ejecutar nuestros programas en C++11

- Ya instalado el compilador de C++11 en nuestro sistema...

- **Compilar:**

```
g++ nombreadarchivo.cpp -o archivosalida  
-pthread -std=c++11 -Wl,--no-as-needed
```

- **Ejecutar:**

```
./archivosalida
```

Creación y ejecución de los hilos

- En C++11, las distintas utilidades que nos permiten lanzar y manipular nuestros hilos se encuentran en la biblioteca **thread**. Por lo tanto, a la hora de desarrollar nuestro código, tendremos que incluir lo siguiente:

```
#include <thread>
```

- Esta biblioteca también nos aporta distintas utilidades para el control de la sincronización, variables atómicas,...

Creación y ejecución de los hilos

- Crear un hilo en C++11 es crear un objeto de la clase *std::thread*.
- A la hora de crear un objeto de esta clase, se le pasa como parámetro al constructor la tarea que tiene que realizar.

```
std::thread hilo(tarea);
```



- Una vez creado dicho objeto, **automáticamente** es lanzado.

Creación y ejecución de los hilos


- La tarea que le pasamos al constructor de nuestro hilo es una función. Esta función puede ser de dos formas:
 - 1. Función externa.
 - 2. Función Lambda (o función anónima).

Creación y ejecución de los hilos

■ Función externa:

```
#include <iostream>
#include <thread>

void hola() { std::cout << "Hola" << std::endl; }

int main() {
    std::thread hilo1(hola); 
    std::thread hilo2(hola);
    hilo1.join();
    hilo2.join();
    std::cout << "Soy el hilo padre" << std::endl;
    return 0;
}
```

Creación y ejecución de los hilos

■ Función lambda:

```
#include <iostream>
```

```
#include <thread>
```

```
int main() {
```

```
    std::thread hilo1([] () {std::cout << "Hola soy el  
hijo " <<
```



```
std::this_thread::get_id() << std::endl; } ) ;
```

```
std::thread hilo2([] () {std::cout << "Hola soy el  
hijo " <<
```

```
std::this_thread::get_id() << std::endl; } ) ;
```

```
hilo1.join();
```

```
hilo2.join();
```


```
std::cout << "Soy el hilo padre" << std::endl;
```

```
return 0;
```

```
}
```

Exclusión mutua y Sincronización

- Sea la siguiente clase:

```
#include <iostream>
#include <thread>
#include <vector>
struct Contador { 
    int valor = 0;
    void incremento() { valor++; }
};
```


Exclusión Mutua y Sincronización

```
int main() {  
    Contador cont;  
    std::vector<std::thread> hilos; //Creamos un vector de hilos  
    for(int i=0; i<10; ++i) {  
        hilos.push_back(std::thread([&cont]() { //Se le pasa la  
                                                //estructura como parámetro  
            for(int j=0; j<100; ++j) {cont.incremento(); }  
        }));  
    }  
    for(auto& thread : hilos) {  
        thread.join();  
    }  
    std::cout << cont.valor << std::endl;  
    return 0;  
}
```

Exclusión Mutua y Sincronización

- Se puede observar, tras varias ejecuciones del código anterior, que los resultados obtenidos no son el esperado (valor esperado: 1000).
- Esto se produce a causa del entrelazado.
- La biblioteca `thread` de C++11 nos aporta una herramienta para el control de la exclusión mutua y la sincronización: los **mutex**.

¿Qué son los mutex?

- Son objetos.
- Son un caso especial de semáforo.
- Propiedad: Sólo un hilo puede obtener el cerrojo de un mutex al mismo tiempo.
- Poseen dos métodos:
 1. `lock()` : Permite al hilo actual obtener el cerrojo del mutex.
 2. `unlock()` : Libera el cerrojo.

¿Qué son los mutex?

- Si modificamos la estructura Contador y añadimos la cabecera mutex

```
struct Contador {  
    std::mutex mutex;  
    int valor = 0;  
    void incremento() {  
        mutex.lock();  
        valor++;  
        mutex.unlock();  
    }  
};
```



Se puede observar que, una vez añadido el mutex, el resultado obtenido en las ejecuciones es siempre el esperado.

Gestión Automática de Cerrojos

- Son objetos de la clase *std::lock_guard*.
- Tienen asociado un mutex que se les pasa como parámetro al constructor.

```
std::mutex cerrojo;  
void fun() {  
    std::lock_guard<std::mutex> guard(cerrojo);  
    //sección crítica  
}
```

- Cuando se crea un objeto *lock_guard*, intenta tomar posesión del mutex que se le da. Cuando se abandona el ámbito en el que se creó el *lock_guard*, se destruye y el mutex se libera.

Gestión automática de cerrojos

- Podemos editar la estructura Contador para que tenga gestión automática de cerrojos: 

```
struct Contador {  
    std::mutex mutex;  
    int valor = 0;  
    void incremento() {  
        std::lock_guard<std::mutex> guard(cerrojo);  
        valor++;  
    } //Se destruye el objeto guard y se libera  
    el cerrojo  
};
```




Bloqueo Recursivo

Diríjase a la carpeta de códigos y ejecute
`brmutex.cpp`

Bloqueo Recursivo

Como se puede apreciar tras la ejecución, el programa entra en interbloqueo y no termina.

Esto se debe a que en la función ambas, el hilo coge el cerrojo y llama a la función `mul()`. En esta función, el hilo intenta coger el cerrojo de nuevo, pero el cerrojo no ha sido liberado, produciendo el interbloqueo.

- Por defecto, un hilo no puede adquirir el mismo mutex dos veces.
- Se hace necesario el uso de un nuevo tipo de mutex: `std::recursive_mutex`. 
- Con este mutex se puede adquirir varias veces un cerrojo por un mismo hilo.



Bloqueo Recursivo

Vuelva a la carpeta de códigos y ejecute
`brrecursivemutex.cpp`

- Podrá observar que no se produce interbloqueo.

Pausando hilos

- La biblioteca de hilos de C++11 nos aporta un método para poder hacer esperar a nuestros hilos:

```
std::this_thread::sleep_for(tiempo);
```

- El tiempo que se le pasa como parámetro puede expresarse con distintas unidades temporales:

```
std::chrono::milliseconds | nanoseconds | seconds |  
hours | minutes | microseconds    (entero)
```

- Se necesita incluir: `#include <chrono>`

Pausando Hilos

- Veamos un ejemplo en el cual se duerme al hilo principal durante un corto periodo de tiempo:

```
#include <iostream>
```

```
#include <thread>
```

```
#include <chrono>
```

```
int main() {  
    std::cout << "Hola soy el hilo principal" << std::endl;  
    std::chrono::milliseconds duracion(2000);  
    std::this_thread::sleep_for(duracion);  
    std::cout << "He dormido 2000 ms" << std::endl;  
    return 0;  
}
```

Funciones `call_once` (llamar una vez)

- A veces queremos que una función sea llamada una sola vez sin importar el número de hilos que la utilicen.
- La biblioteca de C++11 nos ofrece una función para lograrlo:

```
std::call_once(bandera, función);
```

- La bandera que recibe como primer parámetro es de tipo `std::once_flag` y nos permite establecer un cierre que se ejecutará una vez.
- El segundo parámetro es la función que queremos que sea llamada una única vez.

Funciones call_once (llamar una vez)

■ Sea el siguiente ejemplo:


```
#include <iostream>
```

```
#include <thread>
```

```
#include <mutex>
```

```
std::once_flag bandera; //Establecemos la bandera de tipo  
once_flag
```

```
void hacer_algo() {
```

```
    std::call_once(bandera, []() {std::cout<<"Llamado una   
vez"<<std::endl;}); //Este mensaje  
    //sólo se mostrará una vez.
```

```
    std::cout<<"Llamado cada vez"<<std::endl; //Se  
    //mostrará tantas veces como hilos tengamos. 
```

```
}
```

Funciones `call_once` (llamar una vez)

```
int main() {  
    std::thread t1(hacer_algo);  
    std::thread t2(hacer_algo);  
    std::thread t3(hacer_algo);  
    std::thread t4(hacer_algo);  
    t1.join(); t2.join();  
    t3.join(); t4.join();  
  
    return 0;  
}
```

Variables de Condición

- Una variable de condición **gestiona una lista de hilos a la espera de que otro hilo les notifique**. Cada hilo que quiere esperar sobre una variable de condición, tiene que adquirir el cerrojo primero. El cerrojo es liberado cuando el hilo comienza a esperar sobre la condición y adquirido cuando el hilo es despertado.
- En C++11, lo primero que debemos de incluir para poder emplear las variables de condición, es la cabecera en la que se encuentran:

```
#include <condition_variable>
```

- Luego, solo tendríamos que crear instancias de la clase `std::condition_variable`:

```
std::condition_variable vacio, lleno, ...;
```

Variables de Condición

- Poseen dos métodos:

1. `notify_one()` : Despierta a un hilo que esté esperando sobre la variable de condición que llamó al método.
2. `wait(cerrojo, predicado)` : Duerme al hilo actual sobre la variable de condición que llamó al método si el predicado devuelve *false*.

Variables de Condición

- El cerrojo que se le pasa como primer parámetro al método `wait` no es un mutex tal y como lo conocemos.
- Para que las variables de condición puedan usar los mutex, estos deben de ir envueltos por la clase *`std::unique_lock`*.
- Por lo tanto, habrá que crear primero el mutex y luego envolverlo con la clase anteriormente mencionada.

```
std::mutex cerrojo;  
  
std::unique_lock<std::mutex>  
cerrojo_vc(cerrojo);
```



Variables de Condición

Diríjase a la carpeta de códigos y ejecute
`BufferLimitado.cpp`

Tipos Atómicos

- Para poder usar tipos atómicos debemos de incluir la cabecera: `#include <atomic>`
- Una vez incluida, podremos crear variables atómicas como sigue:

```
std::atomic<tipo> variable1, variable2,....;
```

- Los tipos de las variables atómicas pueden ser los tipos primitivos o incluso, tipos que definamos.

Tipos Atómicos

- Podemos modificar la estructura *Contador*, vista al principio de estas transparencias, para que la memoria compartida sea una variable atómica entera:

```
#include <iostream>
#include <vector>
#include <thread>
#include <atomic>

struct ContadorAtomico {
    std::atomic<int> valor; //Creamos la variable
    atómica
    void incremento() { ++valor; }
    void decremento() { --valor; }
    int obtener() { return valor.load(); } //Obtenemos
    su valor
};
```

Tipos Atómicos

```
int main() {  
    ContadorAtomico contadoratomico; //Creamos una  
    instancia  
  
    contadoratomico.valor.store(0); //Establecemos el  
    valor de la variable a 0.  
  
    std::vector<std::thread> hilos;  
    for(int i = 0; i < 3; ++i){  
        hilos.push_back(std::thread([&contadoratomico]  
        ) {  
            for(int i = 0; i < 100; ++i){  
                contadoratomico.incremento();  
            }  
        }));  
    }  
}
```

Tipos Atómicos

```
for(int i = 0; i < 3; ++i){
    hilos.push_back(std::thread([&contadoratomico]
    () {
        for(int i = 0; i < 100; ++i){
            contadoratomico.decremento();
        }
    }));
}
for(auto& thread : hilos){ thread.join(); }
//Mostramos el valor
std::cout << contadoratomico.obtener() <<
std::endl;
return 0;
}
```