Threads en C++11

Antonio Espín Herranz

Contenidos

- Clase thread
- Paso de parámetros a los hilos.
- Regiones críticas, interbloqueos, condiciones de carrera.
- Mecanismos de sincronización en hilos:
  - Mutex
- Variables de condición.
- Esquema productor / consumidor.
- Futures y tareas asíncronas.

threads

- Soporte en C++11
- Para trabajar con hilos, incluir el fichero .H
  - **#include <thread>**

- Para compilar con g++:
  - **g++ –std=c++11 fichero.cpp –o fichero –lpthread**

- Para compilar con make:
  - set(CMAKE_CXX_FLAGS "${CMAKE_CXX_FLAGS} -lpthread")
  - set (CMAKE_CXX_STANDARD 11)
  - set (CMAKE_CXX_STANDARD_REQUIRED ON)

Lanzamiento de Hilos

- En C++11 un hilo se puede lanzar de 3 formas distintas:
  - Con una función.
    - La función puede tener parámetros o no.
  - Con un objeto de una clase que implemente el operador ()
    - También puede ser una estructura con la implementación de dicho operador.
  - Con una función lambda.

Con una función

- Primero se define una función:

```
void funcion_hello(){
        int i;
        for (i = 0 ; i < 10 ; i++)
                std::cout << "Hello " << i << std::endl;
}

std::thread h1(funcion_hello);
h1.join();
```

- También se puede inicializar el hilo con las {}

  std::thread h2 {función_hello}

Con una clase + operador ()

```
class MiFuncion {
        public:
                void operator()(){
                        for (int i = 0 ; i < 10 ; i++)
                                std::cout << "Operador () " << i << std::endl;
                }
};

std::thread h2( (MiFuncion()) );
h2.join();

// Ojo, se instancia la clase MiFuncion() se necesitan los paréntesis extras.
```

Con una función lambda

```
std::thread h3([]{
        for (int i = 0 ; i < 10 ; i++)
                std::cout << "Lambda " << i << std::endl;
        });

h3.join();
```

Condiciones de carrera

```
int x = 42;
void f () { ++x; }
void g() { x=0; }
void h() { cout << "Hola" << endl; }
void i () { cout << "Adios" << endl; }

// La variable x las comparten dos hilos sin ningún tipo de protección.
void carrera() {
        thread t1{ f };
        thread t2{g};
        t1.join () ;
        t2.join () ;
        thread t3{h};
        thread t4{ i };
        t3.join ();
        t4.join () ;
}
```

Paso de argumentos a un hilo

- A un hilo se le pueden pasar un número indeterminado de argumentos.
- La función que ejecute el hilo tiene que tener todos esos argumentos.
- Al instanciar el hilo se le manda como primer parámetro la función que tiene que ejecutar.

```
void funcion(int x, std::string s){
	std::cout << "Parametro int: " << x << std::endl;
	std::cout << "Parametro string: " << s << std::endl;
}

// suele hacer un casting automático de const char * a std::string
std::thread hilo(funcion, 1, std::string("hola"));
hilo.join()
```

Paso de argumentos a un hilo

- La definición de la clase thread:
- El constructor recibe una función y un número indeterminado de argumentos, que pueden ser 0 o n
- **thread thread(Function&& *f*, Args&&... *args*);**

- Un hilo termina cuando finaliza la rutina que ejecuta (por ejemplo, realiza un proceso n veces y termina) y llama a la instrucción **return**.

Paso de parámetros por referencia

- Cuando queremos pasar un parámetro a un hilo por referencia se tiene que indicar en la construcción del hilo.
- Para ello se dispone de la función std::ref(param)

- #include <functional>
- #include <thread>

- **void** f (registro & r) ;

- **void** g(registro & s) {
  - thread t1{ f ,s}; // Copia de s
  - thread t2{ f , std::ref (s) }; // Referencia a s
  - Thread t3 {[&] { f (s) ; }}; // Referencia a s, con la lambda también se puede indicar.

Esperar a que termine un hilo: **join**()

- Siempre se lanza un hilo principal (desde main) y a partir de este se van creando el resto de hilos.
- Para esperar a que un hilo termine se dispone del método **join**().
- Sólo se puede llamar una vez al método **join**.
- Se dispone de la función en thread: **joinable**() se aplica sobre un objeto thread y devuelve true / false para indicar si se puede hacer join a un hilo o no.

Vectores de hilos

- Los hilos se pueden combinar con la clase **vector** para tener varios hilos.

```
#include <vector>
#include <thread>


class Hilo {
	public:
	void operator()(){		// Muestra el identificador del hilo
		std::cout << "Dentro del hilo: " << std::this_thread::get_id() << " esta ejecutando" << std::endl;
	}
};

std::vector<std::thread> hilos;
// Creamos 10 hilos y se añaden al vector:
for (int i = 0 ; i < 10 ; i++)
	hilos.push_back(std::thread( (Hilo()) ));

// Ahora esperamos a que acaben todos los hilos:
std::cout << std::endl << "Esperamos por todos los hilos" << std::endl;
for (auto &h : hilos)
	h.join();
```

mutex

- Al igual que en POSIX los **mutex** (cerrojo) nos sirven para sincronizar el acceso de varios hilos a un recurso compartido para evitar condiciones de carrera y que se corrompa la memoria.

  - La 1ª forma: más propensa a errores se puede olvidar el desbloqueo del mutex:

```
#include <mutex>
miMutex.lock();       // Adquiere el cerrojo
// Actualizar el recurso;
miMutex.unlock();   // Libera el cerrojo
```

  - La 2ª forma: es más segura, se evita el posible error de la primera forma. El mutex se libera automáticamente.

```
std::lock_guard<std::mutex> guard(miMutex);
// Actualizar el recurso y después se libera automáticamente.
```

  - La 3ª forma: es equivalente a lock_guard → unique_lock
  - **unique_lock**<mutex> milock {miMutex};
  - // Actualizar el recurso y después se libera automáticamente.

lock_guard vs unique_lock

- **lock_guard** y **unique_lock** son más o menos lo mismo; lock_guard es una versión restringida con una interfaz limitada.

- **lock_guard** siempre tiene un candado desde su construcción hasta su destrucción.

- **unique_lock** puede crearse sin bloqueo inmediato, puede desbloquearse en cualquier momento de su existencia y puede transferir la propiedad del bloqueo de una instancia a otra.

- Por lo tanto, siempre utilizaremos lock_guard, a menos que se necesiten las capacidades de unique_lock.
- Una variable condition_variable necesita a unique_lock.

detach: Hilos no asociados

- Se puede indicar que un hilo sigue ejecutando después de que el destructor se ejecute con **detach()**.

- Útil para tareas que se ejecutan como demonios.

```
void actualiza () {
    for (;;) {
    muestra_reloj(stead_clock::now());
    this_thread :: sleep_for(second{1});
    }
}
void f () {
    thread t { actualiza };
    t .detach();
}
```

Problemas con hilos no asociados

- Inconvenientes:
  - Se pierde el control de qué hilos están activos.
  - No se sabe si se puede usar el resultado generado por un hilo.
  - No se sabe si un hilo ha liberado sus recursos.
  - Se podría acabar accediendo a objetos que han sido destruidos.

Variables de condición

- Mecanismo para sincronizar hilos en acceso a recursos compartidos:
  - wait(): Espera en un mutex.
  - notify_one(): Despierta a un hilo en espera.
  - notify_all(): Despierta a todos los hilos en espera.

- Productor / Consumidor
  - **class** peticion ;
  - queue<peticion> cola; // Cola de peticiones
  - condition_variable cv;
  - mutex m;
  - **void** productor();
  - **void** consumidor();

Consumidor

```
void consumidor() {
    for (;;) {
        unique_lock<mutex> l{m};
        while (cv.wait( l ) ) ;
        auto p = cola. front () ;
        cola.pop();
        l .unlock() ;
        procesa(p);
    };
}
```

- Efecto de **wait**
  - Libera el cerrojo y espera una notificación.
  - Adquiere el cerrojo al despertarse.

Productor

- **void** productor() {
  - **for** (;;) {
    - peticion p = genera();
    - unique_lock<mutex> l{m};
    - cola.push(p);
    - cv.notify_one() ;
  - }
- }

- Efecto de notify_one()
  - Despierta a uno de los hilos que están esperando en la condición.

Tareas asíncronas y future

- Una tarea **asíncrona** permite el lanzamiento simple de la ejecución de una tarea:
  - **En otro hilo** de ejecución.
  - Como una **tarea diferida**.

- Un **futuro** es un objeto que permite que un hilo pueda devolver un valor a la sección de código que lo invocó

Invocación de tareas asíncronas

```
#include <future>
#include <iostream>

int main() {
    std :: future<int> r = std :: async(tarea, 1, 10);
    otra_tarea() ;
    std :: cout << "Resultado= " << r.get() << std :: endl;
    return 0;
}
```

Uso de futuros

- **Idea general**:
  - Cuando un hilo necesita pasar un valor a otro hilo pone el valor en una **promesa**.
  - La implementación hace que el valor esté disponible en el correspondiente **futuro**.

- Acceso al **futuro** mediante **f.get()**:
  - Si se ha asignado un valor → obtiene el valor.
  - En otro caso → el hilo llamante se bloquea hasta que esté disponible.
  - Permite la transferencia transparente de excepciones entre hilos.