Multihilos en C++

Elaborado por: Ukranio Coronilla

En general una aplicación distribuida necesita realizar diversas acciones de manera casi simultánea, por ejemplo estar recibiendo mensajes al tiempo que procesa información recibida con anterioridad para ser enviada de nuevo. Por otra parte y con intención de utilizar los recursos de manera óptima, se necesita hacer uso de toda la capacidad de cómputo mediante la utilización de todos los procesadores que puedan estar disponibles en el sistema de cómputo. Esto implica el desarrollo de sistemas altamente concurrentes en cada nodo de cualquier sistema distribuido.

Afortunadamente el nuevo estándar de C++ conocido como C++0x ha incluido las bibliotecas necesarias para el manejo de hilos, lo cual hace el código portable al poderse compilar de manera independiente al sistema operativo.

Todos los métodos y clases para el manejo de hilos, se encuentran declaradas en la librería <thread> y mantienen el paradigma orientada a objetos.

A continuación se muestra un programa con un hilo principal, el cual crea dos hilos. El proceso principal espera a que los dos hilos terminen y después finaliza. Para compilar los programas que utilizan <thread> se necesita la opción -pthread, y la opción -std=c++11 para indicar el uso del nuevo estándar de C++, aunque puede que su compilador no lo requiera. Por ejemplo, si se guarda el programa como `hilos1.cpp` se debe compilar así:

```
g++ -std=c++11  hilos1.cpp -o hilos1 -lpthread
```

```
#include <iostream>
#include <unistd.h>
#include <thread>

using namespace std;

void funcion(int valor)
{
3       cout<<"Hilo " <<  this_thread::get_id() << " Valor recibido: " << valor << endl;
        sleep(2);
}

int main()
{
1       thread th1(funcion, 5), th2(funcion, 10);

        cout  << "Proceso principal espera que los hilos terminen\n";
2       th1.join();
        th2.join();
        cout << "El hilo principal termina\n";
```

```
        exit(0);
}
```

Al ejecutar este programa con el comando time nos podremos dar cuenta que los hilos se ejecutan concurrentemente, pues el programa demora dos segundos.

```
$ time ./hilos1
Proceso principal espera que los hilos terminen
Hilo 140431445059328 Valor recibido: 5
Hilo 140431436666624 Valor recibido: 10
El hilo principal termina

real    0m2.004s
user    0m0.000s
sys     0m0.000s
```

Asimismo si lo ejecutamos varias veces podemos ver que las impresiones pueden traslaparse debido a que `cout` manda el flujo de salida hacia un buffer, en donde las impresiones concurrentes podrían llegar en desorden. Para evitar estos inconvenientes, en lo subsiguiente al utilizar hilos solo utilizaremos printf para imprimir a pantalla.

El código tiene tres líneas importantes que se explican a continuación:

Línea 1 Se instancian dos objetos tipo `thread` denominados th1 y th2, los cuales van a ejecutar la función llamada `funcion`.

Línea 2 El hilo principal espera a que haya terminado de ejecutarse el hilo th1 antes de terminar.

Línea 3 Obtiene el identificador de hilo que le corresponde.

Ejercicio 1 Modifique el programa para utilizar la función `printf` en lugar de `cout` y lograr que las impresiones siempre sean correctas (use `%ld` para imprimir el identificador del hilo).

Ejercicio 2 Cree una variable entera global (fuera de `main`) con valor inicial de cero. Haga una función para que uno de los hilos incremente la variable una unidad y después se duerma un segundo y otra función para que otro hilo la decremente en una unidad y después se duerma un segundo. Al final, cuando terminen de ejecutarse los dos hilos, imprima el valor de dicha variable global en el hilo principal. ¿Los hilos comparten la variable?

Ejercicio 3 Modifique el ejercicio anterior para que uno de los hilos decremente n veces la variable global y en el otro hilo se incremente n veces la variable global. Cuando ambos hilos terminen imprima el valor de la variable en el programa principal que debería ser cero. Incremente el valor de n hasta que se produzcan inconsistencias debidas a las condiciones de competencia.

Ejercicio 4 Para resolver el problema de condiciones de competencia existe en C++ la clase template `atomic`. Los objetos instanciados con esta clase no provocan condiciones de competencia al ser accedidos por distintos hilos. Pruebe que resuelve el problema del ejercicio 3 declarando un objeto con el mismo nombre de la variable global como sigue:

```
atomic<int> global(0);
```
Es necesario incluir el header:
```
#include <atomic>
```
Y para imprimir su valor es necesario usar `cout`

Si no se desea usar variables globales, es posible declarar la variable dentro de la función principal y enviarla a los hilos como parámetro. En este caso y para evitar condiciones de competencia se declara como:
```
atomic<int> var_local(0);
```
Pasándose por referencia al hilo con ayuda de la función `ref` como sigue:
```
thread th1(funcion1, ref(var_local))
```
El parámetro se recibiría en la función `funcion1` como:
```
void funcion1(atomic<int>& variable)
```

Ejercicio 5 Pruebe el paso por referencia de una variable local atómica hacia los dos hilos y observe que no provoca condiciones de competencia.

En diversas situaciones queremos que una región crítica solo sea ejecutada por un solo hilo de ejecución y lograr así la exclusión mutua. Para ello utilizamos los semáforos binarios mejor conocidos como mutex. Un mutex se declara como sigue:

```
#include <mutex>
mutex m;
```

y tiene los siguientes métodos:
```
m.lock();  //Para bloquear la ejecución
//Aquí va el código de la región crítica
m.unlock();     //Para desbloquear la ejecución
```

Ejercicio 6 Modifique el ejercicio 3 para usar mutex y resolver la inconsistencia.

En diversas situaciones es necesaria la sincronización de hilos, para ello se utilizan semáforos, sin embargo las librerías `thread` de C++ no tienen implementados los semáforos☹.

A continuación se muestra una implementación de una clase semáforo con ayuda de un mutex y una variable de condición, la cual se basó en código tomado de:

https://stackoverflow.com/questions/4792449/c0x-has-no-semaphores-how-to-synchronize-threads

y al cual se le han realizado algunas modificaciones para su fácil uso.

Interfaz de `Semaforo.h`

```
#ifndef SEMAFORO_H_
#define SEMAFORO_H_
```

```
#include <mutex>
#include <condition_variable>

class Semaforo
{
private:
    std::mutex mutex_;
    std::condition_variable condition_;
    unsigned long count_;

public:
    Semaforo(unsigned long = 0);
    void post();
    void wait();
    void init(unsigned long);
};

#endif
```

Implementación de `Semaforo.cpp`

```
#include <mutex>
#include <condition_variable>
#include "Semaforo.h"

using namespace std;

Semaforo::Semaforo(unsigned long c){
    count_ = c;
}

void Semaforo::post(){
    unique_lock<decltype(mutex_)> lock(mutex_);
    ++count_;
    condition_.notify_one();
}

void Semaforo::wait(){
    unique_lock<decltype(mutex_)> lock(mutex_);
    while(!count_)
        condition_.wait(lock);
    --count_;
}

void Semaforo::init(unsigned long valor){
    count_ = valor;
}
```

Ejercicio 7 Pruebe esta clase Semaforo con el siguiente código principal que sincroniza impresiones a pantalla entre dos hilos concurrentes.

```
#include <iostream>
#include <unistd.h>
#include <thread>
#include "Semaforo.h"
#define n 10

using namespace std;
```

```
int global = 10;
Semaforo sem1, sem2;

void funcion1()
{
	while(global > 0){
		sem1.wait();
		printf("Soy el hilo 1, y esta es la impresion %d\n", global-- );
		sem2.post();
	}
}

void funcion2()
{
	while(global > 0){
		sem2.wait();
		printf("Soy el hilo 2, y esta es la impresion %d\n", global-- );
		sem1.post();
	}
}

int main()
{
	//Inicializa los semaforos
	sem1.init(1);
	sem2.init(0);

	thread th1(funcion1), th2(funcion2);

	th1.join();
	th2.join();

	exit(0);
}
```

Después de probarlo, modifique el programa para que se sincronicen las impresiones consecutivas de tres hilos en vez de que sean dos hilos.