Informática Industrial

Algunos elementos de concurrencia en C++v11

Del C al C++

- Desde C++ v11 se tiene soporte nativo, portable, dentro del lenguaje a la concurrencia:
  - Hilos o *threads* (que pueden ejecutar una función)
  - Los hilos pueden compartir datos (variables globales o variables locales de las que reciben una referencia)
  - Cuando hay condiciones de carrera ("*race conditions*") se pueden utilizar métodos para sincronizar acceso (*mutexes*)
  - Existen funciones libres de ayuda para evitar deadlocks (situaciones de bloqueo) o para saber el grado de concurrencia real (núcleos de CPU), entre otros.
  - Espera de eventos con variables de condición
  - Ejecución asíncrona de tareas. Promesas y futuros. Async.

Concurrencia. Hilos.

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

using namespace std;

void ImprimeVec(std::vector<int> & v)
{
    for (auto it=v.begin(); it !=v.end();++it )
        std::cout << *it << " ";
    std::cout << endl;
}

int main()
{
    std::vector<int> vec {1,5,7,8,9,10,11,45,-12};


    std::thread t(ImprimeVec,std::ref(vec));


    t.join();

     if (vec.empty())
        std::cout << "Ahora vector vacío" << std::endl;

    return 0;
}
```

La función **ImprimeVec** se ejecutará en hilo diferente. Recibe por referencia un vector.

Se crea hilo ***t*** con ***std::thread***. Se le pasa el nombre de la función a ejecutar (puede ser un functor) y el parámetro. Aquí es necesario ***std::ref*** para indicar explícitamente que se pasa por referencia.

Se espera a que el hilo t termine su ejecución.

Concurrencia. Hilos.

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

using namespace std;

void ImprimeVec(std::vector<int> && v)
{
    for (auto it=v.begin(); it !=v.end();++it )
        std::cout << *it << " ";
    std::cout << endl;
}

int main()
{
    std::vector<int> vec {1,5,7,8,9,10,11,45,-12};


    std::thread t(ImprimeVec,std::move(vec));


    t.join();

     if (vec.empty())
        std::cout << "Ahora vector vacío" << std::endl;

    return 0;
}
```

```
1 5 7 8 9 10 11 45 -12
Ahora vector vacio
```

La función **Imprime Vec** se ejecutará en hilo diferente. Recibe por movimiento (&&) un vector.

Se crea hilo ***t*** con ***std::thread***. Se le pasa el nombre de la función a ejecutar (puede ser un functor) y el parámetro. Aquí es necesario ***std::move*** para indicar explícitamente que se pasa por movimiento (muy eficiente).

Una vez "movido" el vector, el original queda vacío. Ver salida por consola.

Concurrencia. Hilos. Mutex

```cpp
#include <iostream>
#include <vector>
#include <thread>
#include <mutex>

const int NUM_HILOS {8};
const int NUM_ITER {10000};

struct Contador
{
    int valor;
    std::mutex mtx;                 //para garantizar EXCLUSION MUTUA
    Contador() : valor(0) {}

    void incrementa()
    {
        mtx.lock();         //cierra el paso a otro hilo
        ++valor;            //incrementa ahora sin problemas
        mtx.unlock();       //libera el acceso a valor
    }
};
int main()
{
    Contador cont;
    std::vector<thread> misHilos;

    for(auto i = 0; i < NUM_HILOS; ++i)
        misHilos.push_back(thread([&cont](){for(auto j = 0; j < 10000; j++)cont.incrementa();}));

    for(auto& hilo : misHilos) hilo.join();


    std::cout << "El valor del contador incrementado simultaneamente por los " << NUM_HILOS << " hilos es " <<cont.valor << std::endl;
    std::cout << "Debe ser igual a " << NUM_HILOS*NUM_ITER;

    return 0;
}
```

La estructura que define el tipo de la variable a compartir (campo valor) contiene un mutex (mtx) para coordinar el acceso a él.

Antes de incrementar el valor se debe adquirir el mutex (lock) y después de incrementar la variable compartida, liberar el mutex (unlock)

Se crean los hilos y se almacenan en un vector. La función se define como lambda. Nada impide haberlo hecho con función libre, o con functor. La variable cont se pasa por referencia.

Se espera por la terminación de todos los hilos

Mutex. Variables de condición

```
#include <iostream>
#include <thread>
#include <queue>
#include <condition_variable>
#include <mutex>
#include <chrono>
#include <algorithm>

struct comparte
{
    std::deque<int> v;
    std::mutex mx;
    std::condition_variable cv;
    void produce(int dato)
    {
        std::lock_guard<std::mutex> l(mx);
        v.push_back(dato);
        cv.notify_one();
    }
    int consume()
    {
        std::unique_lock<std::mutex> l(mx);
        cv.wait(l,[this]{return !this->v.empty();});
        int data = v.front();
        v.pop_front();
        return data;
    }

};
```

Define deque, mutex y variable de condición

Se adquiere *mutex* hasta el final del método. Se introduce dato en el *deque*. Se notifica a los *threads* que espera en la variable de condición cv.

Se adquiere *mutex* hasta el final del método pero puede ser desactivado antes.
cv.wait espera hasta que reciba una notifiación, entonces hace lock con el mutex y pregunta por la condición (lambda). Si es falso vuelve a esperar y hace unlock.

Mutex. Variables de condición

```
void produce_t(comparte& comp)
{
    std::vector<int> v(10);

    std::iota(v.begin(),v.end(),1);
    v.push_back(-1);
    for (auto x: v)
    {
        std::this_thread::sleep_for(std::chrono::seconds(1));
        comp.produce(x);
    }
}
void consume_t(comparte& comp)
{
    int data;
    do
    {
       data= comp.consume();
       std::cout << data << " ";
    } while (data >= 0);
}
int main()
{
    comparte comp;
    std::thread  produce(produce_t, std::ref(comp));
    std::thread  consume(consume_t, std::ref(comp));

    produce.join();
    consume.join();
}
```

Función para thread productora: rellena vector con elementos consecutivos , cada segundo envía elemento del vector a través del método produce del objeto compartido

Se obtiene el dato enviado para consumir utilizando la función del objeto compartido.

Se crean los dos hilos. Ambos comparten por referencia objeto de tipo comparte.

Async. Futures

```cpp
#include <iostream>
#include <thread>
#include <vector>
#include <future>
#include <algorithm>
#include <chrono>

int main()
{
    std::vector<std::future<int>> futures;
    for (int i = 0; i < 20; ++i)
    {
        std::future<int> fut = std::async(std::launch::async,[i]
        {

            std::this_thread::sleep_for(std::chrono::seconds(1));
            return i+10; //std::this_thread::get_id();
        });
        futures.push_back(std::move(fut));
    }
    for (auto &fut: futures)
    {
        fut.wait();
        std::cout << fut.get() << " ";

    }
    std::cout << std::endl;
}
```

Vector the futuros de tipo int.

Lanza asíncronamente 20 hilos que espera un tiempo y devuelven un entero.

Esperamos que estén listas las respuestas y obtenemos los enteros enviados.