Programación Concurrente con C++17

Seminario Número 1

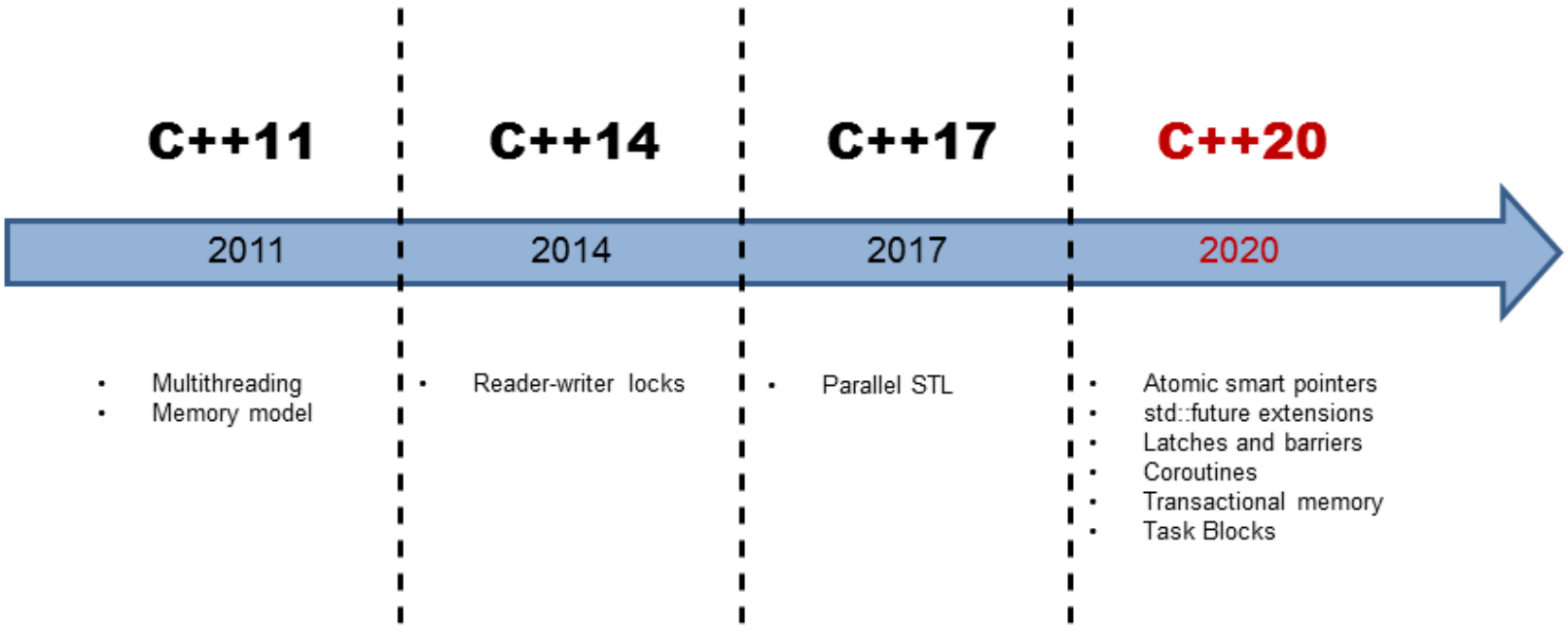
Antonio J. Tomeu[1]

[1]Departamento de Ingeniería Informática
Universidad de Cádiz

PCTR, 2019

Contenido

Evolución de las Capacidades Concurrentes en C++



La Biblioteca Thread

- C++ introduce a partir de la iteración de 2011 una nueva biblioteca para gestión de hebras concurrentes: <thread>
- Permite la creación y ejecución de hebras como objetos de clase thread
- También da soporte al programador para sincronizar hebras mediante cerrojos de clase `mutex`, y más complejos; también está disponible el uso de variables atómicas, y se ofrecen monitores con semántica de señalización SC y variables de condición

Creación y Ejecución de Hebras

- Crear y ejecutar un hebras es relativamente simple. Basta con crear una instancia de la clase `std::thread`, utilizando como parámetro la función que contiene el código que deseamos que la hebra ejecute
- El uso del constructor de clase tiene un doble efecto; no sólo crea el nuevo objeto, sino que también lo arranca de forma **autómatica** (a diferencia de Java, para lo cuál se necesita el método `start`)
- La plantilla de uso general adopta entonces una estructura como la siguiente:

```
std::thread nombre(tarea)
```

- Aquí, `nombre` es la referencia al nuevo objeto de clase `thread` que estamos creando, y `tarea` es la función que contiene el código que la nueva hebra debe ejecutar

Técnicas de Creación de Hebras

- Existen dos técnicas ligeramente diferentes para especificar el código que una hebra debe ejecutar
- El método general es el ya explicando, dando como parámetro en el momento de crear la hebra una función que contiene el código que esta debe ejecutar (técnicamente, en realidad lo que se transfiere en un puntero a dicha función)
- También es posible utilizar una λ-función (también llamada función anónima), para el caso de que el código que la hebra a ejecutar sea muy pequeño, y no necesitemos escribir una función para contenerlo.

Hebras con Función Externa I

Código 1: code/extern.cpp

```
1  #include <thread>
2  #include <iostream>
3  #include <vector>
4
5  void hello(){
6      std::cout << "Hello from thread " <<
           std::this_thread::get_id() << std::endl;
7  }
8
9  int main(){
10     std::vector<std::thread> threads;
11
12     for(int i = 0; i < 5; ++i){
13         threads.push_back(std::thread(hello));
14     }
15
16     for(auto& thread : threads){
17         thread.join();
18     }
```

Hebras con Función Externa II

```
19
20        return 0;
21    }
```

Hebras con Función λ I

Código 2: code/lambda.cpp

```
1  #include <thread>
2  #include <iostream>
3  #include <vector>
4
5  int main(){
6      std::vector<std::thread> threads;
7      for(int i = 0; i < 5; ++i){
8          threads.push_back(std::thread([](){
9              std::cout << "Hello from thread " <<
                   std::this_thread::get_id() << std::endl;
10         }));
11     }
12     for(auto& thread : threads){
13         thread.join();
14     }
15     return 0;
16 }
```

Corrutina Múltiple con Array de Hebras I

Código 3: code/corutine.cpp

```
1  #include<iostream>
2  #include <thread>
3  using namespace std;
4
5  void hola(){
6    cout <<"Hola Mundo..." << this_thread::get_id()<< " ";
7  }
8  int main(){
9    int nHilos = 100;
10   thread hilos[nHilos];
11   for(int i=0; i<nHilos; i++)hilos[i]=thread (hola);
12   for(int i=0; i<nHilos; i++)hilos[i].join();
13
14   cout << "Hilo main tambien saluda...";
15   return(0);
16 }
```

Gestionando las Hebras I

1. En general, el programador puede necesitar realizar determinadas operaciones de gestión de sus hebras concurrentes, como son:
2. Hacer que una hebra espera a que termine otra (método `join`)
3. Hacer que una hebra espere un tiempo determinado «durmiendo»
4. Asignar prioridades a las hebras para alterar el orden en que se ejecutan
5. Interrumpir la ejecución de una hebra

«Durmiendo» Hebras: el método sleep_for I

- A través del método sleep_FOLR de la clase thread
- Tiene como parámetro el tiempo de sueño habitualmente expresado en milisegundos (hace falta #include <chrono>)
- Admite otras unidades de media temporal
- No es útil para planficar de manera adecuada

Uso del Método sleep I

Código 4: code/sleep.cpp

```
1  #include <iostream>
2  #include <thread>
3  #include <chrono>
4
5  int main(){
6    std::cout << "Hola soy el hilo principal" << std::endl;
7    std::chrono::milliseconds duracion(2000);
8    std::this_thread::sleep_for(duracion);
9    std::cout << "He dormido 2000 ms" << std::endl;
10 return 0;
11 }
```

Replanificación Voluntaria: El Método `yield` I

- Una hebras puede «insinuar» al planificador que puede ser replanificada, permitiendo que otras hebras se ejecuten
- El comportamiento exacto del método `yield` depende de la implementación y, en particular, del funcionamiento del planificador del sistema operativo
- No es una técnica de planificación adecuada
- Es poco utilizado en la práctica

Uso del Método yield I

Código 5: code/yield.cpp

```
1  #include <iostream>
2  #include <chrono>
3  #include <thread>
4
5
6  void little_sleep(std::chrono::microseconds us)
7  {
8      auto start = std::chrono::high_resolution_clock::now();
9      auto end = start + us;
10     do {
11         std::this_thread::yield();
12     } while (std::chrono::high_resolution_clock::now() < end);
13 }
14
15 int main()
16 {
17     auto start = std::chrono::high_resolution_clock::now();
18
19     little_sleep(std::chrono::microseconds(100));
```

Uso del Método yield II

```
20
21      auto elapsed = std::chrono::high_resolution_clock::now() -
            start;
22      std::cout << "waited for "
23                <<
                      std::chrono::duration_cast<std::chrono::microseconds
24                << " microseconds\n";
25  }
```

Ejemplo de Condición de Carrera I

Código 6: code/condCarrera.cpp

```
1  #include<iostream>
2  #include <thread>
3
4  int x = 0; //variable compartida. Puede ser sobreescrita.
5  int nVueltas;
6
7  void hola(){
8    std::cout <<"Hola Mundo...";
9    for(int i=0; i<nVueltas; i++)x++; //x++ es una seccion
         critica.
10 }
11
12 int main(){
13   std::cout << "Introducir numero de iteracione por hilo: ";
14   std::cin >> nVueltas;
15   std::thread h(hola);
16   std::thread i(hola);
17   h.join();
18   i.join();
```

Ejemplo de Condición de Carrera II

```
19      std::cout << x; //En general, el valor de x sera distinto al
            esperado
20  }
```

¿Y si se comparte una estructura? I

Código 7: code/condCarreraconStruct.cpp

```
1  #include <iostream>
2  #include <thread>
3  #include <vector>
4  using namespace std;
5
6
7  struct Cuenta{
8    int val=0;
9    void inc(){val++;}
10 };
11
12
13 int main()
14 {
15   vector<thread> hilos;
16   int nHilos = 100;
17   Cuenta contador;
18
19   for(int i=0; i<nHilos; ++i)
```

¿Y si se comparte una estructura? II

```
20      {hilos.push_back(thread([&contador](){for(int
            i=0;i<1000;i++){contador.inc();}}));}
21
22    for(auto& thread : hilos){thread.join();}
23
24    cout << contador.val;
25    return(0);
26  }
```

Técnicas de Control de Exclusión Mutua y Sincronización en C++ I

- C++ ofrece una amplia variedad de técnicas de control de la exclusión mutua (y de sincronización) que incluyen:
- Cerrojos
- Cerrojos reentrantes
- Objetos Atómicos
- Monitores
- Variables de condición (dentro de un monitor)

Control de Exclusión Mutua con Cerrojos: mutex I

- C++ proporciona cerrojos mediante la clase `mutex`
- La sección crítica se encierra entre los métodos `lock()` y `unlock()`
- No provee cerrojos reentrantes, lo cuál los hace inútil en programas recursivos.

Exclusión Mutua con mutex: Ejemplo I

Código 8: code/condCarreraMutex.cpp

```
1  #include<iostream>
2  #include <thread>
3  #include <mutex>
4
5  int x = 0; //variable compartida.
6  int nVueltas;
7  std::mutex mimutex;
8
9  void hola(){
10   std::cout <<"Hola Mundo...";
11   for(int i=0; i<nVueltas; i++){
12       mimutex.lock();
13       x++; //x++ es una seccion critica controlada.
14       mimutex.unlock();
15   }
16 }
17 int main(){
18   std::cout << "Introducir numero de iteracione por hilo: ";
19   std::cin >> nVueltas;
```

Exclusión Mutua con mutex: Ejemplo II

```
20      std::thread h(hola);
21      std::thread i(hola);
22      h.join();
23      i.join();
24      std::cout << x; //Ahora, x siempre vale lo mismo...
25  }
```

Exclusión Mutua de una Estructura con mutex: Ejemplo I

Código 9: code/condCarreraconStructMutex.cpp

```
1  #include <iostream>
2  #include <thread>
3  #include <vector>
4  #include <mutex>
5  using namespace std;
6
7
8  struct Cuenta{
9    int val=0;
10   mutex cerrojo;
11   void inc()
12   {
13     cerrojo.lock();
14     val++;
15     cerrojo.unlock();
16   }
17 };
18
19
```

Exclusión Mutua de una Estructura con mutex: Ejemplo II

```
20  int main()
21  {
22    vector<thread> hilos;
23    int nHilos = 100;
24    Cuenta contador;
25
26    for(int i=0; i<nHilos; ++i)
27      {hilos.push_back(thread([&contador](){for(int
            i=0;i<1000;i++){contador.inc();}}));}
28
29    for(auto& thread : hilos){thread.join();}
30
31    cout << contador.val;
32    return(0);
33  }
```

Recordemos: los objetos de clase mutex carecen de reentrancia I

Código 10: code/noreentrantmutex.cpp

```
1  #include <iostream>
2  #include <thread>
3  #include <mutex>
4
5  struct Compleja {
6      std::mutex mutex;
7      int i;
8
9      Compleja() : i(0) {}
10
11     void mul(int x){
12         std::lock_guard<std::mutex> cerrojo(mutex);
13         i *= x;
14     }
15
16     void div(int x){
17         std::lock_guard<std::mutex> cerrojo(mutex);
18         i /= x;
```

Recordemos: los objetos de clase mutex carecen de reentrancia II

```
19          }
20
21          void ambas(int x, int y){
22              std::lock_guard<std::mutex> cerrojo(mutex);
23              mul(x);
24              div(y);
25          }
26      };
27
28
29      int main(){
30          Compleja compleja;
31          compleja.ambas(32, 23);
32
33          return 0;
34      }
```

Cerrojos Reentrantes: `recursive_mutex` I

- C++ proporciona cerrojos recursivos mediante la clase `recursive_mutex`
- La sección crítica se encierra entre los métodos `lock()` y `unlock()`
- Ahora la posesión del bloqueo es por hebra, lo cuál permite llamadas recursivas o llamadas entre funciones diferentes que tiene todo (o parte) de su código bajo exclusión mutua controlada por el mismo cerrojo recursivo
- Dicho de otra forma, el cerrojo puede ser adquirido varias veces por la misma hebra

Ejemplo de Uso de Cerrojo con Reentrancia I

Código 11: code/recursivemutex.cpp

```
1  #include <iostream>
2  #include <thread>
3  #include <mutex>
4
5  struct Compleja {
6      std::recursive_mutex mutex;
7      int i;
8
9      Compleja() : i(0) {}
10
11     void mul(int x){
12         std::lock_guard<std::recursive_mutex> cerrojo(mutex);
13         i *= x;
14     }
15
16     void div(int x){
17         std::lock_guard<std::recursive_mutex> corrojo(mutex);
18         i /= x;
19     }
```

Ejemplo de Uso de Cerrojo con Reentrancia II

```
20
21      void ambas(int x, int y){
22          std::lock_guard<std::recursive_mutex> cerrojo(mutex);
23          mul(x);
24          div(y);
25      }
26  };
27
28
29  int main(){
30      Compleja compleja;
31      compleja.ambas(32, 23);
32
33      return 0;
34  }
```

Limitando el Accesso a una sola vez: call_once I

- En ocasiones, interesa limitar a una única vez el número de veces que una función puede ser llamda, con indepedencia del número de hebras que la utilicen
- Para ello, se puede utilizar la función call_once, que tiene el siguiente prototipo:

```
std:call_once(flag, funcion);
```

- Aquí, flag es un objeto de clase std::once_flag, que nos proporciona la semántica de «llamada una sola vez» sobre el código que se encuentra en el parámetro funcion

Ejemplo de Uso de call_once I

Código 12: code/callonce.cpp

```
1  #include <iostream>
2  #include <thread>
3  #include <mutex>
4
5  //Establecemos la bandera de tipo once_flag
6  std::once_flag bandera;
7
8  void hacer_algo() {
9    //el mensaje siguiente solo se mostrar una vez.
10     std::call_once(bandera, [](){std::cout<<"Lamado una
           vez"<<std::endl;});
11     //Se mostrara tantas veces como hilos tengamos
12     std::cout<<"Llamado cada vez"<<std::endl;
13 }
14
15 int main(){
16   std::thread t1(hacer_algo);
17   std::thread t2(hacer_algo);
18   std::thread t3(hacer_algo);
```

Ejemplo de Uso de call_once II

```
19      std::thread t4(hacer_algo);
20      t1.join(); t2.join();
21      t3.join(); t4.join();
22      return 0;
23  }
```

Tipos Atómicos I

- La biblioteca <atomic> proporciona componentes para efectuar operaciones seguras sobre secciones crítica de grano fino (muy pequeñas)
- Cada operación atómica es individible frente o otras operaciones atómicas sobre el mismo dato
- Los datos procesados de esta forma están libre de condiciones de carrera
- Es posible aplicar atomicidad sobre información compartida tanto sobre datos de tipo primitivo, como sobre tipos de datos definidos por el programador

Atomicidad de Tipos Primitivos I

Código 13: code/atomicInt.cpp

```
1  #include<iostream>
2  #include <thread>
3  #include <atomic>
4
5  //se define el objeto atomico
6  std::atomic_int x(0);
7  int nVueltas;
8
9  void hola(){
10   std::cout <<"Hola Mundo...";
11   for(int i=0; i<nVueltas; i++)x++;
12 }
13
14 int main(){
15   std::cout << "Introducir numero de iteracione por hilo: ";
16   std::cin >> nVueltas;
17   std::thread h(hola);
18   std::thread i(hola);
19   h.join();
```

Atomicidad de Tipos Primitivos II

```
20      i.join();
21      std::cout << x;
22  }
```

Atomicidad de Objetos Complejos I

Código 14: code/atomic.cpp

```
1  #include <iostream>
2  #include <vector>
3  #include <thread>
4  #include <atomic>
5
6  struct ContadorAtomico {
7      std::atomic<int> valor;
8      void incremento(){ ++valor; }
9      void decremento(){ --valor; }
10     int obtener(){ return valor.load(); }
11 };
12
13 int main(){
14     ContadorAtomico contadoratomico;
15     contadoratomico.valor.store(0);
16     std::vector<std::thread> hilos;
17     for(int i = 0; i < 3; ++i){
18         hilos.push_back(std::thread([&contadoratomico](){
19             for(int i = 0; i < 100; ++i){
```

Atomicidad de Objetos Complejos II

```
20                  contadoratomico.incremento();
21              }
22          }));
23      }
24      for(int i = 0; i < 3; ++i){
25          hilos.push_back(std::thread([&contadoratomico](){
26              for(int i = 0; i < 100; ++i){
27                  contadoratomico.decremento();
28              }
29          }));
30      }
31      for(auto& thread : hilos){
32          thread.join();
33      }
34      std::cout << contadoratomico.obtener() << std::endl;
35      return 0;
36  }
```

Monitores I

- En C++, un monitor es un objeto donde todos métodos (excepto el constructor) se ejecutan en exclusión mutua bajo el control de un mismo cerrojo
- Si es necesario, es posible disponer de variables de condición, para hacer esperar a hebras por condiciones concretas. En tal caso, el cerrojo a utilizar deberá ser de clase `unique_lock`, lo cuál permite instanciar variables de condición
- En caso de utilizar variables de condición, la semántica de señalización del monitor es SC, lo que obliga al uso de condiciones de guarda

Ejemplo de Uso de Variables de Condición I

Código 15: code/conditionexample.cpp

```
1  #include <iostream>
2  #include <thread>
3  #include <mutex>
4  #include <condition_variable>
5
6  std::mutex mtx;
7  std::condition_variable cv;
8  bool ready = false;
9
10 void print_id (int id) {
11   std::unique_lock<std::mutex> lck(mtx);
12   while (!ready) cv.wait(lck);
13   // ...
14   std::cout << "thread " << id << '\n';
15 }
16
17 void go() {
18   std::unique_lock<std::mutex> lck(mtx);
19   ready = true;
```

Ejemplo de Uso de Variables de Condición II

```
20    cv.notify_all();
21  }
22
23  int main ()
24  {
25    std::thread threads[10];
26    // spawn 10 threads:
27    for (int i=0; i<10; ++i)
28      threads[i] = std::thread(print_id,i);
29
30    std::cout << "10 threads ready to race...\n";
31    go();                       // go!
32
33    for (auto& th : threads) th.join();
34
35    return 0;
36  }
```

Ejemplo de Monitor: productor/consumidor I

Código 16: code/prodcon.cpp

```
1  #include <iostream>
2  #include <thread>
3  #include <mutex>
4  #include <condition_variable>
5  using namespace std;
6
7  struct Buffer
8  {
9    int* buffer;
10   int  tam;
11   int In_Ptr;
12   int Out_Ptr;
13   int cont;
14
15   mutex em;
16   condition_variable not_full;
17   condition_variable not_empty;
18
19   Buffer(int capacidad)
```

Ejemplo de Monitor: productor/consumidor II

```
20      {
21        tam=capacidad;
22        In_Ptr=0; Out_Ptr=0;
23        cont=0;
24        buffer = new int[tam];
25      }
26
27      ~Buffer(){delete[] buffer;}
28
29      void insertar(int dato)
30      {
31        unique_lock<mutex> l(em);
32        while(cont==tam){not_full.wait(l);}
33        buffer[In_Ptr]=dato;
34        In_Ptr=(In_Ptr+1)%tam;
35        ++cont;
36        not_empty.notify_one();
37        l.unlock();
38      }
39
40      int extraer()
41      {
```

Ejemplo de Monitor: productor/consumidor III

```
42      unique_lock<mutex> l(em);
43      while(cont==0){not_empty.wait(l);}
44      int result = buffer[Out_Ptr];
45      Out_Ptr=(Out_Ptr+1)%tam;
46      --cont;
47      not_full.notify_one();
48      return(result);
49      l.unlock();
50    }
51
52  };
53
54   void productor(int id, Buffer& buffer)
55    {
56      for(int i=0; i<100; ++i)
57      {
58        buffer.insertar(i);
59        cout<<"Productor "<<id<<" inserto "<<i<<endl;
60      }
61    }
62
63    void consumidor(int id, Buffer& buffer)
```

Ejemplo de Monitor: productor/consumidor IV

```
64      {
65       for(int i=0; i<50; ++i)
66       {
67        int valor = buffer.extraer();
68        cout<<"Consumidor "<<id<<" extrajo "<<valor<<endl;
69       }
70      }
71
72  int main()
73  {
74    Buffer buffer(200);
75    thread c1(consumidor, 0, ref(buffer));
76    thread c2(consumidor, 1, ref(buffer));
77    thread p1(productor, 0, ref(buffer));
78    c1.join();
79    c2.join();
80    p1.join();
81    return(0);
82  }
```

Bibliografía

C++ Reference

https://en.cppreference.com/w/

2019

Williams, A.

C++ Concurrency in Action

2012