Programación concurrente en C++11

Arquitectura de Computadores

J. Daniel García Sánchez (coordinador)
David Expósito Singh
Javier García Blas
Óscar Pérez Alonso
J. Manuel Pérez Lobato

Grupo ARCOS
Departamento de Informática
Universidad Carlos III de Madrid

Motivación

- C++11 (ISO/IEC 14882:2011) ofrece un modelo de concurrencia propio.
  - Revisiones menores en C++14.
  - Más revisiones esperadas para C++17.
- Cualquier implementación que cumple el estándar debe proporcionarlo.
  - Resuelve problemas inherentes a **PThreads**.
  - Portabilidad de código concurrente: **Windows**, **POSIX**, ...
- **Implicaciones**:
  - Modificaciones en el **lenguaje**.
  - Modificaciones en la **biblioteca estándar**.
- Influencia sobre C11 (ISO/IEC 9899:2011).
- **Importante**: Concurrencia y paralelismo son conceptos **relacionados** pero **distintos**.

Estructura

- El lenguaje **ofrece**:
  - Un nuevo **modelo de memoria**.
  - Variables **thread_local**.
- La **biblioteca estándar ofrece**:
  - **Tipos atómicos**.
    - Útiles para programación **libre de cerrojos** de forma portable.
  - **Abstracciones portables** para la concurrencia.
    - **thread**.
    - **mutex**.
    - **lock**.
    - **packaged_task**.
    - **future**.

2 Visión general de la biblioteca

- Hilos
- Acceso a datos compartidos
- Esperas
- Tareas asíncronas

Lanzamiento de hilos

- Un **hilo** representado por la clase **std::thread**.
  - Normalmente representa un hilo del SO.

Lanzando un hilo a través de una función

```
void f1();
void f2();

void g() {
  thread t1{f1};  // Lanza un hilo que ejecuta f1()
  thread t2{f2};  // Lanza un hilo que ejecuta f2()

  t1.join();  // Espera a que t1 termine.
  t2.join();  // Espera a que t2 termine.
}
```

Objetos compartidos

- Dos hilos pueden acceder a un **objeto compartido**.
- Posibilidad de **carreras de datos**.

Acceso a variables compartidas

```
int x = 42;

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

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

- **Paso de argumentos simplificado** sin necesidad de **casts**.

Paso de argumentos

```
void f1(int  x);
void f2(double x, double y);

void g()  {
  thread t1{f1,  10};  // Ejecuta f1(10)
  thread t2{f1};  // Error
  thread t3{f2,  1.0}  // Error
  thread t4{f2,  1.0,  1.0};  // Ejecuta f2(1.0,1.0)
  // ...
  // Joins de hilos
```

Hilos y objetos función

- **Objeto función**: Objeto invocable como función.
- Sobrecarga/redefinición del **operador ()**

Objeto función en un hilo

```
struct mifunc {
  mifunc(int val) : x{val} {}
  void operator()() { haz_algo(x); }
  int x;
};

void g() {
  mifunc f1{10}; // Construye objeto f1
  f1 (); // Invoca operador de llamada f1.operator()
  thread t1{f1}; // Ejecuta f1 () en un hilo
  thread t2{mifunc{20}}; // Construye temporal e invoca
  // ...
  // Joins de hilos
```

2 Visión general de la biblioteca

- Hilos
- Acceso a datos compartidos
- Esperas
- Tareas asíncronas

Exclusión mutua

- **mutex** permite controlar el acceso con **exclusión mutua** a un recurso.
  - **lock()**: **Adquiere** el cerrojo asociado.
  - **unlock()**: **Libera** el cerrojo asociado.

Uso de mutex

```
mutex m;
int  x = 0;

void f()  {
   m.lock();
   ++x;
   m.unlock();
}
```

Lanzamiento de hilos

```
void g()  {
   thread t1(f);
   thread t2(f);
   t1.join();
   t2.join();
   cout << x << endl;
}
```

Problemas con `lock()`/`unlock()`

- **Posibles problemas**:
  - Olvido de **liberar cerrojo**.
  - **Excepciones**.
- Solución: **unique_lock**.
  - **Patrón**: **RAII** (Resource Acquisition Is Initialization).

Cerrojo automático

```
mutex m;
int  x = 0;

void f()  {
   // Adquiere el cerrojo
   unique_lock<mutex> l{m};
   ++x;
}  // Libera el cerrojo
```

Lanzamiento de hilos

```
void g()  {
   thread t1(f);
   thread t2(f);
   t1.join();
   t2.join();

   cout << x << endl;
}
```

Adquisición de múltiples `mutex`

- La función **lock()** permite adquirir a la vez varios **mutex**.
  - Adquiere todos o ninguno.
  - Si alguno está bloqueado espera dejando libres todos.

Adquisición múltiple

```
mutex m1, m2, m3;

void f ()  {
  lock(m1, m2, m3);

  // Acceso a datos compartidos

  // Cuidado: No se liberan los cerrojos
  m1.unlock();
  m2.unlock();
  m3.unlock()
}
```

Adquisición de múltiples `mutex`

- Especialmente útil en cooperación con **unique_lock**

Adquisición múltiple automática

```
void f() {
  unique_lock l1{m1, defer_lock};
  unique_lock l2{m2, defer_lock};
  unique_lock l3{m3, defer_lock};

  lock(l1, l2, l3);
  // Acceso a datos compartidos

} // Liberación automática
```

2 Visión general de la biblioteca

- Hilos
- Acceso a datos compartidos
- Esperas
- Tareas asíncronas

Esperas de tiempo

- Acceso al reloj.

```
using namespace std::chrono;
auto t1 = high_resolution_clock::now();
```

- Diferencia de tiempos.

```
auto dif = duration_cast<nanoseconds>(t2-t1);
cout << dif.count() << endl;
```

- Especificación de una espera.

```
this_thread::sleep_for(microseconds{500});
```

Variables 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**:
  1. Libera el cerrojo y espera una notificación.
  2. 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()**:
  1. Despierta a uno de los hilos esperando en la condición.

2 Visión general de la biblioteca

- Hilos
- Acceso a datos compartidos
- Esperas
- Tareas asíncronas

Tareas asíncronas y futuros

- 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.

La clase thread

- La abstracción de hilo representada mediante clase **thread**.
- Correspondencia uno-a-uno con los hilos del sistema operativo.
- Todos los hilos de una aplicación se ejecutan en el mismo espacio de direcciones.
- Cada hilo tiene su propia pila.
- Peligros:
  - Pasar un puntero o una referencia no constante a otro hilo.
  - Paso de referenica a través de captura en expresiones lambda.
- **thread** representa un enlace a un hilo del sistema.
  - No se pueden copiar.
  - Si se pueden mover.

Construcción de hilos

- Un hilo se construye con una función y los argumentos que se debe pasar a la función.
  - Plantilla con número variable de argumentos.
  - Seguro en tipos.

Ejemplo

```
void f();
void g(int, double);

thread t1{f}; // OK
thread t2{f, 1}; // Error: demasiados argumentos.

thread t3{g, 1, 0.5}; // OK
thread t4{g}; // Error: faltan argumentos.
thread t5{g, 1, "Hola"}; // Error: tipos incorrectos
```

Construcción y referencias

- El constructor de **thread** es una plantilla con argumentos variables.

```
template <class F, class ...Args>
explicit  thread(F&& f, Args&&... args);
```

- El paso de parámetros a un hilo es por valor.
- Para forzar el paso de parámetros por referencia:
  - Usar una función de ayuda para **reference_wrapper**.
  - Usar lambdas y capturas por referencia.

```
void f(registro  & r);

void g(registro  & s) {
  thread t1{f,s};  // Copia de s
  thread t2{f,  ref(s)};  // Referencia a s
  thread t3{[&]  {  f(s);  }};  // Referencia a s
}
```

Construcción en dos etapas

- La construcción incluye el inicio de la ejecución del hilo.
  - No hay operación separada para *iniciar* la ejecución.

Productor/Consumidor

```
struct productor {
  productor(cola<peticiones> & c);
  void operator()();
  //  ...
};

struct consumidor {
  consumidor(cola<peticiones> & c);
  void operator()();
  //  ...
};
```

Etapas

```
void f () {
  // Etapa 1: Construcción
  cola<peticiones> c;
  productor prod{c};
  consumidor cons{c};

  // Etapa 2: Arranque
  thread tp{prod};
  thread tc{cons};

  // ...
```

Hilo vacío

- El constructor por defecto crea un hilo sin tarea de ejecución asociada.

```
thread() noexcept;
```

- Útil en combinación con el constructor de movimiento.

```
thread(thread &&) noexcept;
```

- Se puede mover una tarea de ejecución de un hilo a otro.
  - El hilo original se queda sin tarea de ejecución asociada.

```
thread crea_tarea();
thread t1 = crea_tarea();
thread t2 = move(t1); // t1 está vacío ahora
```

Identidad de un hilo

- Cada hilo tiene un identificador único.
  - Tipo **thread::id**.
  - Si el **thread** no está asociado con un hilo **get_id()** devuelve **id{}**.
  - El identificador del hilo actual se obtiene con **this_thread::get_id()**.
- **t.get_id()** devuelve **id{}** si:
  - No se le ha asignado una tarea de ejecución.
  - Ha finalizado.
  - La tarea se ha movido a otro **thread**.
  - Se desasociado (**detach()**).

Operaciones sobre `thread::id`

- Es un tipo dependiente de la implementación, pero debe permitir:
  - Copia.
  - Operadores de comparación (**==**, **<**, ...).
  - Salida mediante el operador **«**.
  - Transformación *hash* mediante la especialización **hash<thread::id>**.

Ejemplo

```
void imprime_id(thread & t) {
  if (t.get_id() == id{})
    cout << "Hilo no válido" << endl;
  else {
    cout << "Hilo actual: " << this_thread::get_id() << endl;
    cout << "Hilo recibido: " << t.get_id() << endl;
  }
}
```

Unión de tareas

- Cuando un hilo desea esperar a que otro hilo finalice puede usar la operación **join()**.
  - **t.join()** → espera a que t haya finalizado.

Ejemplo

```
void f () {
  vector<thread> vt;
  for (int i=0; i< 8; ++i) {
    vt.push_back(thread(f,i));
  }

  for (auto & t : vt) { // Espera a que todos los hilos terminen
    t.join ();
  }
}
```

Tareas periódicas

Idea inicial

```
void actualiza_barra() {
  while (!tarea_terminada()) {
    this_thread::sleep_for(chrono::second(1))
    update_progress();
  }
}

void f() {
  thread t{actualiza_barra};
  t.join();
}
```

- ¿Problemas?

¿Qué pasa si se olvida join?

- Si se sale del alcance donde se define el hilo, se invoca al destructor.
- Problema: Se podría perder el vínculo con un hilo del sistema que se seguiría ejecutando y al que no se podría acceder.
  - Si no se ha hecho **join()** el destructor llama a **terminate()**.

Ejemplo

```
void actualiza () {
  for (;;) {
    muestra_reloj(stead_clock::now());
    this_thread::sleep_for(second{1});
  }
}

void f () {
  thread t{actualiza};
}
```

- Se ejecuta **terminate()** al abandonar **f()**.

Destrucción

- Objetivo: Evitar que un hilo sobreviva a su objeto **thread**.
- Solución: Si un **thread** es *unible* su destructor invoca **terminate()**.
  - Un **thread** es unible si está asociado a un hilo del sistema.

Ejemplo

```
void comprueba() {
  for (;;) {
    comprueba_estado();
    this_thread::sleep_for(second{10});
  }
}

void f() {
  thread t{comprueba};
} // Destrucción sin join() -> Invoca a terminate()
```

Problemas con la destrucción

Ejemplo

```
void f();
void g();

void ejemplo() {
  thread t1{f};  // Hilo que ejecuta la tarea f
  thread t2;  // Hilo vacío.

  if (modo == modo1) {
    thread tg {g};
    // ...
    t2 = move(tg); // tg vacía, t2 asociada a g()
  }

  vector<int> v{10000}; // Podría lanzar excepciones
  t1.join();
  t2.join();
}
```

- ¿Qué ocurre si el constructor de **v** lanza una excepción.
- ¿Qué ocurre si se llega al final con **modo==modo1**?

Hilo automático

- Se puede usar el patrón RAII.
  - Resource Acquisition Is Initialization.

Un hilo que hace join

```
struct hilo_automatico : thread {
  using thread::thread;  // Todos los constructores de thread
  ~hilo_automatico() {
    if (joinable()) join();
  }
};
```

- El constructor adquire el recurso.
- El destructor libera el recurso.
- Evita el goteo de recursos.

Simplificación con RAII

- Simplificación de código y seguridad.

Ejemplo

```
void ejemplo() {
  hilo_automatico t1{f}; // Hilo que ejecuta la tarea f
  hilo_automatico t2; // Hilo vacío.

  if (modo == modo1) {
    hilo_automatico tg {g};
    // ...
    t2 = move(tg); // tg vacía, t2 asociada a g()
  }

  vector<int> v{10000}; // Podría lanzar excepciones
}
```

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.

Ejemplo

```
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

- Incovenientes:
  - 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.
- Recomendaciones:
  - Evitar el uso de hilos no asociados.
  - Mover los hilos a otro alcance (via valor de retorno).
  - Mover los hilos a un contenedor en un alcance mayor.

Un bug difícil de cazar

- Problema: Cuidado con el acceso a variables locales desde un hilo no asociado despues de su destrucción.

Ejemplo

```
void g()  {
   double x = 0;
   thread t{[&x]{  f1 () ;  x  = f2 ()  ;}};   // Si g ha terminado -> Problema
   t.detach();
}
```

Operaciones sobre el hilo actual

- Operaciones sobre el hilo actual como funciones globales en sub-espacio de nombres **this_thread**.
  - **get_id()**: Obtiene el identificador del hilo actual.
  - **yield()**: Permite que potencialmente se seleccione otro hilo para ejecución.
  - **sleep_until(t)**: Espera hasta un determinado instante de tiempo.
  - **sleep_for(d)**: Espera durante una duración determinada de tiempo.
- Esperas con tiempo:
  - Si se modifica el reloj, **wait_until()** se ve afectado.
  - Si se modifica el reloj, **wait_for()** **no** se ve afectado.

Variables locales al hilos

- Alternativa a **static** como especificador de almacenamiento: **thread_local**.
  - Una variable **static** tiene una única copia compartida por todos los hilos.
  - Una variable **thread_local** tiene una copia por cada hilo.
- Tiempo de vida: duración de almacenamiento de hilo (*thread storage duration*).
  - Se inicia antes de su primer uso en el hilo.
  - Se destruye en la salida del hilo.
- Razones para usar almacenamiento local al hilo:
  - Transformar datos de alamacenamiento estático a almacenamiento local al hilo.
  - Mantener cachés de datos locales a cada hilo (acceso exclusivo).
    - Importante en máquinas con caché separada y protocolos de coherencia.

Una función con caché de cálculos

```
thread_local map<int, int> cache;

int calcula_clave(int x) {
  auto i = cache.find(x);
  if (i != cache.end()) return i->second;
  return cache[arg] = algoritmo_complejo_y_lento(arg);
}

vector<int> genera_lista(vector<int> v) {
  vector<int> r;
  for (auto x : v) {
    r.push_back(calcula_clave(x));
  }
}
```

- Se evita sincronización.
- Puede que se repita algún cálculo.

Clasificación de *mutex*

- Representan el acceso exclusivo a un recurso.
  - **mutex**: *Mutex* básico no recursivo.
  - **recursive_mutex**: Un *mutex* que puede ser adquirido más de una vez por un mismo hilo.
  - **timed_mutex**: *Mutex* no recursivo con operaciones con tiempo límite.
  - **recursive_timed_mutex**: *Mutex* recursivo con operaciones con tiempo límite.
- Solamente un hilo puede poseer un *mutex* en un momento dado.
  - Adquirir un *mutex* $\rightarrow$ obtener acceso exclusivo al objeto.
    - Operación bloqueante.
  - Liberar un *mutex* $\rightarrow$ Liberar el acceso exclusivo al objeto.
    - Permite que otro hilo obtenga el acceso.

Operaciones

- Construcción y destrucción:
  - Se puede construir por defecto.
  - No se pueden copiar ni mover.
  - El destructor puede dar comportamiento no definido si el *mutex* no está libre.
- Adquisición y liberación:
  - **m.lock()**: Adquiere el *mutex* de forma bloqueante.
  - **m.unlock()**: Libera el *mutex*.
  - **r = m.try_lock()**: Intenta adquirir el *mutex*, devolviendo indicación de éxito.
- Otras:
  - **h = m.native_handle()**: Devuelve el identificador dependiente de la plataforma de tipo **native_handle_type**.

Ejemplo

Acceso exclusivo

```
mutex mutex_salida;

void imprime(int x) {
  mutex_salida.lock();
  cout << x << endl;
  mutex_salida.unlock();
}

void imprime(double x) {
  mutex_salida.lock();
  cout << x << endl;
  mutex_salida.unlock();
}
```

Lanzamiento de hilos

```
void f() {
  thread t1{imprime, 10};
  thread t2(imprime, 5.5};
  thread t3(imprime, 3);

  t1.join();
  t2.join();
  t3.join();
}
```

Errores en exclusión mutua

- En caso de error se lanza la excepción **system_error**.
- Códigos de error:
  - **resource_deadlock_would_occur**.
  - **resource_unavailabe_try_again**.
  - **operation_not_permitted**.
  - **device_or_resource_busy**.
  - **invalid_argument**.

```
mutex m;
try  {
   m.lock();
   //
   m.lock();
}
catch (system_error & e) {
   cerr  << e.what() << endl;
   cerr  << e.code() << endl;
}
```

Tiempo límite

- Operaciones soportadas por **timed_mutex** y **recursive_timed_mutex**.

- Añaden operaciones de adquisición con especificación de tiempo límite.
  - **r = m.try_lock_for(d)**: Intenta adquirir el *mutex* durante una duración **d**, devolviendo indicación de éxito.
  - **r = m.try_lock_until(t)**: Intenta adquirir el *mutex* hasta un momento en el tiempo, devolviendo indicación de éxito.

Variables condición

- Sincronización de operaciones entre hilos.
- Optimizada para la clase **mutex** (alternativa **condition_variable_any**)).
- Construcción y destrucción:
  - **condition_variable c{}**: Crea una variable condición.
    - Puede lanzar **system_error**.
  - Destructor: Destruye la variable condición.
    - Requiere que no haya ningún hilo esperando en condición.
  - No se pueden copiar ni mover.
  - Antes de destruirla se debe despertar a todos los hilos bloqueados en la variable.
    - O se podrían quedar bloquedos para siempre.

Operaciones de notificación/espera

- Notificación:
  - **c.notify_one()**: Despierta a uno de los hilos en espera.
  - **c.notify_all()**: Despierta a todos los hilso en espera.
- Espera incondicional (**l** de tipo **unique_lock<mutex>**):
  - **c.wait(l)**: Se bloquea hasta que consigue adquirir el cerrojo **l**.
  - **c.wait_until(l,t)**: Se bloquea hasta que consigue adquirir el cerrojo **l** o se llega al tiempo **t**.
  - **c.wait_for(l,t)**: Se bloquea hasta que consigue adquirir el cerrojo **l** o pasa la duración **d**.
- Espera con predicados.
  - Admiten como argumento adicional un predicado que debe satisfacerse.

Revisando el producto consumidor

Inyectando el predicado en wait

```
void consumidor() {
  for (;;) {
    unique_lock<mutex> l{m};

    cv.wait(l, [this]{return !cola.empty();});

    auto p = cola.front();
    cola.pop();
    l.unlock();

    procesa(p);
  };
}
```

Resumen

- C++ ofrece un modelo de concurrencia mediante una combinación de lenguaje y biblioteca.
- La clase **thread** abstra un hilo de SO.
- La sincronización se consigue combinando **mutex** y **condition_variable**.
- **std::async** ofrece un mecanismo de alto nivel para ejecución de tareas.
- **std::future** permite transportar resultados y excepciones entre hilos.
- **thread_local** ofrece un soporte portable para almacenamiento local al hilo.

Referencias

- *C++ Concurrency in Action. Practical multithreading.*
  Anthony Williams.
  Capítulos 2, 3 y 4.

Programación concurrente en C++11

Arquitectura de Computadores

J. Daniel García Sánchez (coordinador)
David Expósito Singh
Javier García Blas
Óscar Pérez Alonso
J. Manuel Pérez Lobato

Grupo ARCOS
Departamento de Informática
Universidad Carlos III de Madrid