High Performance Computing

Object Oriented Concurrency

Fernando R. Rannou
Departamento de Ingeniería Informática
Universidad de Santiago de Chile

September 22, 2023

Concurrencia y Orientación al Objeto

Concurrencia es una abstracción más antigua que **Orientación al Objeto** (OO)

- La abstracción de concurrencia fue definida en base al concepto de hebra de ejecución y sus primitivas de sincronización tales como mutex, semáforos, variables de condición y barreras
- Por otro lado, OO fue definido en términos de ocultamiento de información, herencia y asbtracción de datos
- Luego, concurrencia fue definida en términos de control de ejecución, y OO en términos de propiedades al nivel de lenguaje de programación
- Concurrencia es una abstracción de más bajo nivel que OO y por ende la unión de ambos en los lenguajes de programación no ha sido tan exitosa a nivel masivo

Lenguajes o librerías

La solución que muchos adoptan para incluir concurrencia en lenguajes OO es el uso de librerías, tales como Pthreads u OpenMP.

- Sin embargo, existe una tendencia a considerar esta solución como fallida, pues intenta unir dos abstracciones sin una análisis de la abstracción resultante. Es más, algunos autores piensan que la inclusión de hebras en cualquier programa es fuente de nodeterminismo, incluso cuando el resultado sea correcto
- En este sentido, el uso explícito de mecanismos de sincronización es una forma que tienen los programadores de eliminar dicho nodeterminismo

Luego, la inclusión de concurrencia mendiante el uso de hebras y/o librerias externas es no recomendable y la tendencia moderna es el uso de **lenguajes que incluyan la abstracción de concurrencia en su defínción**

Pthreads versus OpenMP

Recordar que:

Pthread

Estándar para la creación, control y sincronización de hebras de ejecución

OpenMP

Estándar para la programación paralela, multihebra, para sistemas de memoria compartida

Pthreads y C++ (sin concurrencia)

La siguiente es una solución típica para introducir multihebras Pthread en una clase C++

```
class Thread {
private:
     pthread_t tid;

protected:
     void *arg;
     static void *exec(void *thr);

public:
     Thread();
     ~Thread();
     void start(void *arg);
     void join();
     virtual void run() = 0;
};
```

El método `start()` es público pues es el método que los clientes invocan para ejecutar una hebra

Este método debe crear una hebra que ejecute el método `exec()`

La *signature* de `exec()` es `void * f(void *)`. Por qué?

`exec()` debe invocar a `run()` que es un método virtual puro el cual debe ser implementado por las clases que hereden de `Thread`

Ejemplo Pthread y C++

```
Thread::Thread() {
     cout << "Constructing Thread::Thread()" << endl;
}

Thread::~Thread() {
     cout << "Destroying Thread::~Thread()" << endl;
}

void Thread::start(void *arg) {
     int ret;
     this->arg = arg;
     if ((ret = pthread_create(&tid, NULL, &Thread::exec, this)) != 0) {
          cout << "could not create thread" << endl;
          throw "Error";
     }
}

void Thread::join() {
     pthread_join(tid, NULL);
}

void *Thread::exec(void *thr) {
     reinterpret_cast<Thread *> (thr)->run(); //El usuario hereda de Thread e
                                              //implementa run()
}
```

Usando la clase hebreada

Usemos la clase Thread para implementar un clase hebreada que calcule los números de Fibonacci.

$$f_n = f_{n-1} + f_{n-2}, \quad f_0 = 0, f_1 = 1$$

```
class MyThread : public Thread {
private:
     int fibonacci(int);
public:
     void run();
};

void MyThread::run() {
     int fib = fibonacci(*((int *) arg));
     cout << fib << endl;
}

int MyThread::fibonacci(int num) {
    switch(num)  {
       case 0: return 0;
       case 1: return 1;
       default:
         return fibonacci(num-2) +
                fibonacci(num-1);
    };
}
```

```
int main() {
     int x = 6;
     MyThread thr;
     thr.start(&x);
     thr.join();
}

$ g++ -o exe fibonacci-pthread.cc -lpthrea
$ ./ex2
Constructing Thread::Thread()
8
Destroying Thread::~Thread()
```

C++ y boost::thread

Boost es una librería para C++ que provee, entre muchas cosas, soporte para programación multihebra.

Con Boost se puede crear:

1. una hebra que ejecute una función dada (muy parecido a una función C en Pthread)
2. una hebra que ejecute un **Functor**, es decir que ejecute un objeto como una función
3. una hebra que ejecute un método de un objeto
4. un objeto con su propia hebra de ejecución

Veamos cada uno de estos casos

Boost::thread. Función hebreada

En este método no existe orientación al objeto, luego, basta con implementar una función estilo C que calcule el número de Fibonacci

```
#include <boost/thread.hpp>

using namespace std;
int finalval;

int fibonacci(int num) {
    switch(num)  {
       case 0: return 0;
       case 1: return 1;
       default:
         return fibonacci(num-2) +
                fibonacci(num-1);
    };
}

void Fibonacci(int num) {
     cout << "hebra Fibonacci" << endl;
     finalval = fibonacci(num);
}
```

```
int main(int argc, char* argv) {
    cout << "hebra principal" << endl;

    boost::thread MyThread(Fibonacci, 6);
    MyThread.join();

    cout << "finalval = " << finalval << endl;
}
$ g++ -o exe fibonacci-boost1.cc -lboost_thread
$ ./exe
hebra principal
inicio hebra Fibonacci
finalval = 8
```

Note que la función donde comienza la ejecución de la hebra debe ser `void`.

Boost::thread. Hebreando un functor

Ahora, tenemos una clase que la invocamos como función

```
class Fibonacci {
public:
  Fibonacci(int n, int *finalval) :
                   n(n),
                   finalval(finalval) {}

  void operator()() {
      cout << "hebra Fibonacci" << endl;
      *finalval = doFibonacci(n);
  }

  int doFibonacci(int num) {
     switch(num) {
       case 0: return 0;
       case 1: return 1;
       default:
         return doFibonacci(num-2) +
                doFibonacci(num-1);
    };
  }

private:
     int n;
     int *finalval;
};
```

```
int main(int argc, char* argv) {

     int finalval;
     cout << "hebra principal" << endl;

     Fibonacci Fib(6, &finalval);
     boost::thread MyThread(Fib);
     MyThread.join();

     cout << "finalval = " << finalval << endl;
}

$ ./exe
hebra principal
hebra Fibonacci
finalval = 8
```

Boost::thread. Hebreado un método

```
class Fibonacci {

public:
  void doFibonacci(int num, int *finalval)
  {
       cout << "hebra Fibonacci" << endl;
       *finalval = getfibonacci(num);
  }

  int getfibonacci(int num) {
     switch(num) {
       case 0: return 0;
       case 1: return 1;
       default:
         return getfibonacci(num-2) +
                getfibonacci(num-1);
    };
  }

private:
     int n;
     int *finalval;
};
```

```
int main(int argc, char* argv) {
     int finalval;
     cout << "hebra principal" << endl;

     Fibonacci Fib;
     boost::thread MyThread(
                    &Fibonacci::doFibonacci,
                    &Fib, 6, &finalval);

     MyThread.join();
     cout << "finalval = " << finalval << endl;
}

$ ./exe
hebra principal
hebra Fibonacci
finalval = 8
```

Boost::thread. Objeto con su propia hebra

En este método, la hebra es un miembro privado del objeto. El `main()` no manipula la hebra.

```
class Worker {
private:
    boost::thread MyThread;
    int finalval;
public:
    Worker() { }

    void run(int M) {
       cout << "hebra Fibonacci" << endl;
       MyThread = boost::thread(&Worker::getfibonacci, this, M);
    }

    int getfibonacci(int num) {
        finalval = runfibonacci(num);
    }

    int runfibonacci(int num) {
     switch(num) {
       case 0: return 0;
       case 1: return 1;
       default:
         return runfibonacci(num-2) +
                runfibonacci(num-1);
     };
    }
```

```
int main(int argc, char* argv) {
    cout << "hebra principal" << endl;

    Worker w;
    w.run(6);

    w.join();
    cout << "finalval = " << w.getval() << endl;
}
```

Pero...

En Boost, nada impide que una clase tenga más de una hebra. Por ejemplo:

```
class Worker {
private:
    boost::thread MyThread1;
    boost::thread MyThread2;
public:
    Worker() { }

    void run() {
       MyThread1 = boost::thread(&Worker::dorealwork, this);
       MyThread2 = boost::thread(&Worker::dorealwork, this);
    }
    void join()   {
       MyThread1.join();
       MyThread2.join();
    }

private:
    void dorealwork() { // Thread-safe?
       cout << boost::this_thread::get_id() <<
       " working real hard here..." << endl;
    }
};
```

```
int main(int argc, char* argv) {
    cout << "hebra principal" << endl;

    Worker w; // Constructor sin hebra
    w.run(); // Cuántas hebras?
    w.join();
    cout << "fin hebra principal " << endl;
}
```

```
$ ./exe
hebra principal
0x1833070 working real hard here...
0x1833370 working real hard here...
fin hebra principal
```

Además..

No existe restricción a la invocación del objeto, es decir, lo siguiente es válido:

```
int main(int argc, char* argv) {
    cout << "hebra principal" << endl;
    Worker w;
    w.run();
    w.run();
    w.join();
    cout << "fin hebra principal " << endl;
}

$ ./exe
hebra principal
0x11e0370 working real hard here...
0x11e0070 working real hard here...
0x11e0970 working real hard here...
0x11e0670 working real hard here...
fin hebra principal
```

Más aún..

Y si el método que realmente realiza trabajo es público?:

```
class Worker {
private:
    boost::thread MyThread1;
public:
    Worker() { }
    void run() {
       MyThread1 = boost::thread(&Worker::dorealwork, this);
       MyThread2 = boost::thread(&Worker::dorealwork, this);
    }
    void dorealwork() { // Thread-safe?
       cout << boost::this_thread::get_id() <<
           " working real hard here..." << endl;
    }
...
};
int main(int argc, char* argv) {
    cout << "hebra principal" << endl;
    Worker w;
    w.dorealwork(); // Who executes this method?
    w.run();
    w.join();
}

$ ./exe
hebra principal 0x102b090
0x102b090 working real hard here...
0x102b540 working real hard here...
0x102b240 working real hard here...
```

μC++

μ C++ es un dialecto de C++ que introduce mecanismos de concurrencia en C++.

uC++ no es un compilador nuevo. Define extensiones sintácticas a C++ y las implementa a través de un traductor.

μ C++ un proyecto liderado por Peter A. Buhr, de la Universidad de Waterloo, Canada.

Nota: Todo el material que sigue ha sido preparado en base a los papers y apuntes de clase de Peter. A. Buhr.
`http://plg.uwaterloo.ca/~pabuhr/`

Propiedades de ejecución

El diseño de μC++ se basa en el estudio de las siguientes propiedades elementales de ejecución para clases (y objetos)

- Hebra
- Estado de ejecución
- Exclusión mútua

Hebra

La ejecución de código que ocurre independientemente y posiblemente concurrentemente con otras ejecuciones

- La labor de una hebra es avanzar la ejecución de un programa, actualizando el estado de ejecución
- La ejecución de una hebra es secuencial
- Concurrencia es cuando muchas hebras se ejecutan alternadamente en un procesador
- Paralelismo es cuando muchas hebras se ejecutan al mismo tiempo en procesadores distintos
- Un hebra puede estar en tres estados: corriendo, bloqueada, o lista

Estado de ejecución

La información de estado necesaria para permitir ejecución independiente

- Entre la información necesaria encontramos, los contenidos de los registros del procesador, incluyendo al program counter, los datos locales del objeto y los datos locales al bloque actual de ejecución
- Generalmente los estados de ejecución se almacenan en stacks
- Cada lenguaje de programación define lo que constituye un estado de ejecución, pero esta información debe permitir interrumpir una hebra y reanudar su ejecución en un tiempo futuro
- Cambio de contexto es cuando el control se transfiere de un estado de ejecución a otro, no cuando se transfiere el control de una a hebra a otra!!!!

Exclusión mútua

Es el requerimiento de acceso exclusivo a recursos compartidos

- Cuando dos o más hebras acceden a un recurso compartido y al menos una de ellas es un writer sobre dicho recurso, se debe garantizar que las acciones concurrentes produzcan resultados consistente
- EM puede o no implementarse con atomicidad

Objetos en uC++

Las tres propiedades elementales de ejecución son propiedades de objetos. Por ejemplo, un objeto puede o no tener una hebra, puede o no tener un estado de ejecución, y puede o no poseer exclusión mútua

Combinaciones de estas propiedades dan origen a objetos con comportamientos distintos

En uC++ se utilizan definiciones del lenguaje para crear objetos con estas propiedades, no a través de mecanismo auxiliares, como los vistos en Pthread y Boost

Construcciones de alto nivel en uC++

Propiedades de objetos		Propiedades de métodos de objetos	
hebra	estado de ejecución	sin EM	con EM
no	no	objeto clase	monitor
no	si	corutina	corutina monitor
si	no	rechazado	rechazado
si	si	rechazado	tarea

Propiedades de objetos		Declaración	
hebra	estado de ejecución	sin EM	con EM
no	no	**class**	**_Monitor**
no	si	**_Coroutine**	**_Mutex _Coroutine**
si	si	rechazado	**_Task**

Características básicas

- **Objeto clase**: sin hebra, sin estado de ejecución, sin EM:
  Este es el clásico objeto de la mayoría de los lenguajes OO.

  La hebra que invoca al objeto clase y el estado de ejecución que trae dicha hebra son necesarios para permitir ejecución del objeto clase

  Ya que este tipo de objeto no provee EM, normalmente debería accesarse por una sola hebra
- **monitor**: sin hebra, sin estado, con EM:
  Es un objeto clase que además garantiza EM en la ejecución de sus miembros

  La ejecución de sus miebros es realizada no por el monitor mismo, sino por hebras que lo invocan

Características básicas

- **corutina**: sin hebra, con estado de ejecución, sin EM:
  Una corutina posee su propio estado de ejecución, pero no hebra.

  Una corutina permite la suspensión de la ejecución de su código en un punto dado y la reanudación a partir de dicho punto en el futuro

  Al igual que los caso anteriores, este tipo de objeto necesita la hebra llamadora para avanzar y cambiar su estado

  La hebra realiza cambio de contexto cuando ejecuta la corutina y cuando retorna de ella
- **corutina monitor**: sin hebra, con estado de ejecución, con EM:
  Agrega a la construcción anterior la capacidad de EM en la ejecución de sus miembros

Características básicas

- **tarea**: con hebra, con estado de ejecución, con EM:
  La tarea es el objeto más complejo de uC++, pues es el único (en uC++) que permite incorporar concurrencia y paralelismo

  Algunos utilizan el término *objeto activo* para este tipo de objeto, pero no siempre los OA proveen EM implícitamente como en uC++
- **rechazados**
  - Si una hebra no posee estado de ejecución, no permitiría concurrencia segura. Si se utilizara el estado de ejecución de otra hebra llamadora, una de las dos tendría que suspender su ejecución y por lo tanto tendriamos dos hebras ejecutando secuencialmente una misma porción de código
  - Una hebra con estado de ejecución puede ejecutarse por si sola. Pero si no provee EM, otra hebra podría modificar sus miembros, lo cual puede resultar en datos inconsistentes.
  - Si en este último caso se utilizan mecanismo de locking explícitos se violaría los requerimientos de diseño del lenguaje