La concorrenza con C++11 async e la libreria Threading

La libreria Threading e i principali costrutti per sfruttare il parallelismo, dalla semplicità di 'async' e del tipo 'future' alla gestione dei data race con i tipi 'atomic'

La **libreria per il threading** è una delle caratteristiche più importanti introdotte con **C++11**. Si tratta di una libreria piuttosto vasta i cui capisaldi sono:

- **sdt:thread**, una classe che rappresenta un thread in esecuzione
- costrutti per la sincronizzazione
- la funzione template **async** per l'avvio di task simultanei
- il tipo di archiviazione **thread_local** per la dichiarazione di dati unici per-thread

Possiamo considerare un **thread** una parte del programma in esecuzione. In C++11, come nella maggior parte dei linguaggi moderni, un thread può condividere uno spazio di indirizzamento con altri thread. In questo a differenza di un processo, che generalmente non condivide i dati in maniera diretta con altri processi.

async e future

async è una funzione che fornisce agli sviluppatori una semplificazione per gestire i casi più semplici di concorrenza, ovvero quelli in cui non abbiamo:

- *dipendenza dai dati*, non ci sono thread che devono utilizzare i dati che stiamo elaborando
- *dipendenza di controllo*, la nostra operazione non blocca altre parti del programma

Per capire meglio di cosa si tratta vediamo subito un esempio in cui viene eseguita una somma parallela di tutti gli elementi di un vettore utilizzando tale costrutto:

```
#include <iostream>
#include <vector>
#include <algorithm>
#include <numeric>
#include <future>
// Funzione che produce più task se l'intervallo
// end-beg è molto ampio
template<typename RAIter>
int parallel_sum(RAIter begin, RAIter end)
{
  // Verifica che la lunghezza sia inferiore a 1000
  // in tal caso non utilizza la async
```

```
  typename RAIter::difference_type length = end-begin;
  if(length < 1000)
    return std::accumulate(begin, end, 0);
  // Creazione di un nuovo thread con std::async
  RAIter mid = begin + length/2;
  // nuovo thread
  auto handle = std::async(std::launch::async, parallel_sum<RAIter>, mid, end);
  int sum = parallel_sum(begin, mid);
  // Somma gli elementi dei singoli task
  return sum + handle.get();
}
int main()
{
    std::vector<int> v(10000, 1);
    std::cout << "The sum is " << parallel_sum(v.begin(), v.end()) << 'n';
}
```

Come si può notare si tratta di un uso molto semplicistico della concorrenza, che non necessita dell'utilizzo esplicito di thread, lock e buffer. Il tipo della variabile `handle` è determinato dal tipo di ritorno della funzione `std::async` della libreria standard che è un `std::future`.
La funzione `get()` di un oggetto di tipo `future` resta in attesa finché il thread non termina la propria esecuzione. Pertanto `async` ha il compito di distribuire i thread in maniera opportuna mentre `future` quello di effettuare l'operazione di join dei thread.
Data la semplicità del costrutto non è possibile utilizzare **async** per avviare task che si occupano di I/O, manipolano i mutex, o interagiscono con altri task. Per tali scopi è più appropriato utilizzare altre funzionalità offerte dalla libreria di threading di cui parliamo nel paragrafo che segue.

std::thread e std::join

Un thread è lanciato costruendo un std::thread con una funzione o con un oggetto funzione. Ad esempio il seguente codice lancia due funzioni in due thread separati.

```
#include<thread>
#include <iostream>
void f()
{
  std::cout << "f()" << std::endl;
}
struct F {
  void operator()()
  {
    std::cout << "F()" << std::endl;
  };
};
int main()
{
  std::thread t1{f};    // f() viene eseguita in un thread separato
  std::thread t2{F()};  // F()() viene eseguita in un thread separato
  return 0;
}
```

Chi ha già lavorato con i thread in passato si sarà già accorto che, indipendentemente dal corpo delle funzioni `f()` ed `F()`, tale codice non è in grado di produrre risultati utili. Il problema è che il

programma potrebbe terminare prima o dopo che `t1` esegua `f()` e prima o dopo che `t()` esegua `F()`.
Pertanto è necessario accertarsi che i due task siano terminati prima di proseguire con l'elaborazione. Per far ciò è necessario richiamare la funzione **join** che assicura che il programma non sarà terminato fino a che i due thread non saranno completati.
In tale contesto join va inteso come "aspettare che il thread termini". Di seguito è riportato il main dell'esempio precedente con l'introduzione delle due join.

```
int main()
{
  std::thread t1{f};    // f() viene eseguita in un thread separato
  std::thread t2{F()};  // F()() viene eseguita in un thread separato
  t1.join();
  t2.join();
  return 0;
}
```

Recuperare il valore di ritorno da una funzione eseguita in un thread

Nell'esempio che abbiamo visto non è possibile avere parametri e un risultato di ritorno dalle funzioni eseguite nei due thread. Per questo la libreria standard mette a disposizione **bind** che propone delle facilities adatte allo scopo.
Con task semplici, non vi è alcuna nozione di un valore di ritorno e, pertanto, è sufficiente l'utilizzo di `std::future` come valore di ritorno. In alternativa, si può passare un argomento ad un task passandogli un parametro in cui andare ad inserire il risultato. Di seguito è riportato lo *pseudo-codice* di un esempio che utilizza `std::bind` per passare argomenti ad una funzione:

```
#include <vector>
#include <iostream>
#include <thread>
#include <functional>
// inserisce il risultato in res
void f(std::vector<int>&v, double* res)
{
  (*res) = 0;
  for(unsigned int i = 0; i < v.size(); i++)
    (*res)+=v.at(i);
}
int main()
{
  double res1;
  std::vector<int> myVec(10000, 1);
  // f(some_vec, &res1) viene eseguita in un thread differente
  std::thread t1{std::bind(f,myVec,&res1)};
  t1.join();
  std::cout << res1 << 'n';
  return 0;
}
```

Gestire errori nei thread

Se un thread genera un'eccezione e non la gestisce da sé viene chiamata la funzione std::terminate(); tipicamente ciò implica la terminazione del programma. Un `std::future` è in grado di trasmettere una eccezione al thread genitore/chiamante oppure può trasmettere un codice di errore.

Non c'è nessun modo per **richiedere ad un thread di terminare** (ad esempio inviandogli una richiesta per farlo uscire quanto prima possibile) o per forzare la terminazione di un thread (ad esempio tramite una kill). Tuttavia la libreria standard C++11 lascia la possibilità allo sviluppatore di:

- progettare il proprio meccanismo di interruzione dei thread utilizzando dati condivisi che un thread chiamante può impostare sul thread chiamato per controllarne l'esecuzione e che lo fa uscire velocemente appena viene impostato
- utilizzare i costrutti nativi messi a disposizione dalla nozione di thread fornita dal sistema operativo tramite **thread::native_handle**
- effettuare una kill del processo tramite **std:quick_exit**
- effettuare una kill del programma tramite **std::terminate**

Queste quattro opzioni rappresentano tutto ciò che il comitato di standardizzazione ha potuto concordare nelle varie fasi di standardizzazione. In particolare, i rappresentanti dello standard POSIX erano contrari ad ogni forma di cancellazione dei thread nonostante il modello di risorse di C++ si basi sui distruttori.

Gestire i data race

Il problema di base con i thread sono i **data race**, che si verificano quando due thread in esecuzione provano ad accedere ad un singolo indirizzo in maniera indipendente causando risultati indefiniti. Se, ad esempio, uno scrive sull'oggetto e l'altro legge l'oggetto nello stesso istante si ha un data race in base a quale delle due operazioni viene eseguita prima.

I risultati non solo sono indefiniti, ma molto spesso sono anche completamente imprevedibili. Di conseguenza, C++11 definisce alcune regole per evitare che il programmatore commetta uno di questi errori sulla concorrenza del dati:

- una funzione della libreria standard C++ non dovrebbe accedere direttamente o indirettamente agli oggetti accessibili dai thread diversi dal thread corrente a meno che non si acceda direttamente o indirettamente tramite gli argomenti della funzione
- Una funzione della libreria C++ standard non dovrebbe modificare direttamente o indirettamente oggetti accessibili dai thread diversi dal thread corrente a meno che gli oggetti siano acceduti direttamente o indirettamente attraverso argomenti non const della funzione
- Alle implementazioni della libreria standard C++ è richiesto di evitare data race quando si modificano elementi della stessa sequenza in maniera concorrente

L'accesso simultaneo ad un oggetto stream, ad un oggetto buffer stream, o a una libreria C di stream da multipli stream potrebbe risultare in un data race a meno che non sia specificato diversamente. Pertanto è opportuno stare attenti a non condividere un output stream tra due thread a meno che non si possa controllare l'accesso ad esso.

Abbiamo alcune possibilità:

- attendere un thread per un periodo di tempo specificato
- controllare l'accesso ad alcuni dati utilizzando dei meccanismi di mutual exclusion
- controllare l'accesso ad alcuni dati utilizzando i lock
- attendere per un'azione di un altro task utilizzando una variabile condizionale
- restituire un valore da un thread attraverso un `std::future`

Un altro modo per evitare problemi di data race è l'utilizzo degli Atomic.

Gli Atomic

Gli atomic sono un insieme di tipi che, per definizione, non possono incorrere in errori di tipo data race ovvero in errori dovuti ad operazioni di lettura/scrittura concorrente.

In sostanza se un thread scrive su un oggetto di tipo atomic mentre un altro thread lo sta leggendo il comportamento dell'applicazione è ben definito. In aggiunta accessi ad oggetti atomici possono essere utilizzati per sincronizzare più thread o per ordinare accessi in memoria non atomica utilizzando **std::memory_order**.
Per utilizzare le operazioni atomiche la nuova libreria **C++11** mette a disposizione il template **std::atomic<>** che consente di creare un equivalente atomico di un tipo definito dall'utente.

La nuova libreria standard offre anche delle specializzazioni del template `std::atomic` per i tipi interi e per i puntatori. Ecco un primo semplice esempio:

```
#include <atomic>
#include <stdio.h>
int main()
{
  std::atomic_int a(5);
  ++a;
  printf("%dn", (int)a);
  return 0;
}
```

Vediamo invece ora un esempio più significativo in cui mostriamo la reale utilità degli atomic nell'accesso/scrittura concorrente:

```
#include <thread>
#include <iostream>
#include <atomic>
std::atomic_int a(5);
void f()
{
        a++;
        std::cout << "f()" << std::endl;
}
struct F {
     void operator()()
     {
        a++;
        std::cout << "F()" << std::endl;
     };
};
int main()
{
       std::thread t1{f};      // f() viene eseguita in un thread separato
       std::thread t2{F()};    // F()() viene eseguita in un thread separato
       t1.join();
       t2.join();
       std::cout << "a: " << a << std::endl;
       return 0;
}
```

Nell'esempio i thread t1 e t2 incrementano entrambi la variabile intera a, che è dichiarata come `atomic_int`. In tal caso il costrutto atomic evita un errore di tipo data race, che potrebbe verificarsi nel caso in cui la variabile avesse un tipo di dato int.

Link utili

- **std::async** – cppreference