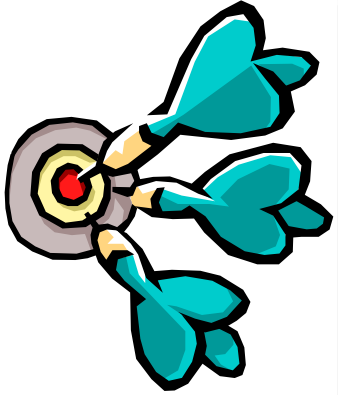
Programmazione concorrente in C++11 - Parte II

Programmazione di Sistema

A.A. 2017-18

C. Barberis, G. Malnati 2004-18

Argomenti



- Creazione di thread secondari
- Restituzione dei risultati
- Accedere al thread corrente

C. Barberis, G. Malnati 2004-18

Gestire la concorrenza a basso livello

- async() e future<T> permettono di creare facilmente composizioni di attività ad alto livello
  - A patto che i diversi compiti in cui l'algoritmo può essere suddiviso siano poco interconnessi
- In altri casi, occorre accedere alle funzionalità di basso livello
  - Gestendo esplicitamente la creazione dei thread, la sincronizzazione, l'accesso a zone di memoria condivise, l'uso delle risorse

La classe std::thread

- Modella un oggetto che rappresenta un thread del sistema operativo
  - Offrendo un'interfaccia omogenea e portabile per la sua creazione e gestione
- Se, come parametro del costruttore, riceve un oggetto di tipo Callable…
  - Un puntatore a funzione o un oggetto che implementa operator()
- … crea un nuovo thread all'interno del S.O. e ne inizia subito l'esecuzione

Esempio

```
#include <thread>
void f() {
  std::cout<<"Up &
Running!"<<std::endl;
}

int main() {
  std::thread t(f); //inizia ...

  //altre operazioni
  //nel thread principale

  t.join();  // Si blocca fino a che
             // il thread t termina
}
```

Specificare il compito da eseguire

- Il costruttore di thread riceve un oggetto chiamabile e, facoltativamente, un serie di parametri
  - Tali parametri saranno inoltrati all'oggetto chiamabile attraverso la funzione std::forward che mette a disposizione del destinatario (la funzione che deve essere invocata all'interno dell'oggetto chiamabile) un riferimento al dato originale o un riferimento RVALUE in funzione della natura del dato passato (RVALUE o LVALUE)
  - Per passare un dato come riferimento, occorre incapsularlo in un oggetto di tipo std::reference_wrapper<T> (tramite la funzioni std::ref(v) e std::cref(v) definite in <functional>)

Specificare il compito da eseguire

- I parametri passati per riferimento possono essere utilizzati per ospitare i valori di ritorno della funzione
  - In questo caso occorre garantire che il ciclo di vita di tali parametri sopravviva alla terminazione del thread
  - Ed attendere che il thread sia terminato prima di esaminarne il contenuto

Specificare il compito da

```
#include <thread>
void f(int& result) {
  result = … ;
}

int main() {
  int res1,res2;
  std::thread t1(f,std::ref(res1));
  std::thread t2(f,std::ref(res2));

  t1.join();
  t1.join();
  std::cout<<res1<<"  "<<res2<<"\n";
}
```

Specificare il compito attraverso un funtore

- Spesso è comodo utilizzare una funzione lambda come parametro
  - I parametri catturati come reference potranno essere modificati dal thread creato e resi visibili al codice del thread creante
- Questo può creare problemi di sincronizzazione e di accesso alla memoria
  - Se, nel frattempo, i parametri escono dal proprio scope

Differenze tra std::thread e std::async

- Non c'è scelta sulla politica di attivazione
  - Creando un oggetto di tipo thread con un parametro chiamabile, la libreria cerca di creare un thread nativo del sistema operativo
  - Se non ci sono le risorse necessarie, la creazione dell'oggetto std::thread fallisce lanciando un'eccezione di tipo std::system_error

Differenze tra std::thread e std::async

- Non c'è un meccanismo standard per accedere al risultato
  - L'unica informazione che viene associata al thread è un identificativo univoco, accessibile tramite il metodo get_id()
- Se durante la computazione del thread si verifica un'eccezione non ricuperata da un blocco catch, l'intero programma termina
  - Viene chiamato il metodo std::terminate()

Ciclo di vita di un thread

- Quando viene creato un oggetto std::thread occorre alternativamente
  - Attendere la terminazione della computazione parallela, invocando il metodo join()
  - Informare l'ambiente di esecuzione che non si è interessati all'esito della sua computazione, invocando il metodo detach()
  - Trasferire le informazioni contenute nell'oggetto thread in un altro oggetto, tramite movimento

Ciclo di vita di un thread

- Se nessuna di queste azioni avviene, e si distrugge l'oggetto thread, l'intero programma termina
  - Sempre invocando std::terminate()
- Se il thread principale di un programma termina, tutti i thread secondari ancora esistenti terminano improvvisamente
  - Senza possibilità di effettuare nessuna forma di salvataggio

Gestire la terminazione

```
#include <thread>
class thread_guard {
  std::thread& t;
public:
  thread_guard(std::thread& t_): t(t_) {}

  ~thread_guard() {
    if(t.joinable()) t.join();
  }

  thread_guard(thread_guard const&)
                                    =delete;
  thread_guard& operator=(thread_guard
                          const&)=delete;
};
```

Restituire un risultato

- Se un thread calcola un risultato, come fa a metterlo a disposizione degli altri thread?
  - La soluzione richiede appoggiarsi ad una variabile condivisa
- Questo introduce un secondo problema:
  - Come fanno gli altri thread a sapere che il contenuto della variabile condivisa è stato aggiornato?
- La soluzione banale di introdurre un'ulteriore variabile (booleana) che indica la validità del dato non è una

Restituire un risultato

- Diventerebbe infatti un'ulteriore variabile condivisa che avrebbe bisogno di un indicatore di validità...
  - Problema del riordinamento
- Il modo più semplice di restituire un valore calcolato all'interno di un thread è basato sull'utilizzo di un oggetto di tipo std::promise<T>

std::promise<T>

- Rappresenta l'impegno, da parte del thread, a produrre, prima o poi, un oggetto di tipo T da mettere a disposizione di chi lo vorrà utilizzare...
- ...oppure di notificare un'eventuale eccezione che abbia impedito il calcolo dell'oggetto
- Dato un oggetto promise, si può conoscere quando la promessa si avvera, richiedendo l'oggetto std::future<T> corrispondente
  - Attraverso il metodo get_future()

Esempio

```
#include <future>

void f(std::promise<std::string> p) {
  try {
    //Calcolo il valore da
restituire...

    std::string result = …;
    p.set_value(std::move(result));
  } catch (...) {
    p.set_exception(
      std::current_exception());
  }
}
```

Esempio

```
int main() {
  std::promise<std::string> p;
  std::future<std::string>

f=p.get_future();
  //creo un thread, forzando p ad
essere
  //passata per movimento
  std::thread t(f,std::move(p));
  t.detach();

  // faccio altro...

  // accedo al risultato del thread
  std::string res=f.get();
}
```

Thread distaccati

- Se si crea un oggetto thread e si invoca il metodo detach(), l'oggetto thread si "stacca" dal flusso di elaborazione corrispondente...
- ... e può continuare la propria esecuzione senza, però, offrire più nessun meccanismo specifico per sapere quando termini
- Se il thread distaccato fa accesso a variabili globali o statiche, queste potrebbero essere distrutte mentre la computazione è in corso
  - Perché sta terminando il thread principale

Thread distaccati

- Se il thread principale termina normalmente (il main() ritorna)
  - L'intero processo viene terminato, con tutti i thread distaccati eventualmente presenti
- Se la terminazione del thread principale avviene per altre cause (si invoca l'API del S.O. che termina il thread corrente) questo può portare a errori sulla memoria
- std::quick_exit(...) permette di far terminare un programma senza invocare i distruttori della variabili globali e statiche
  - Che può essere un rimedio peggiore del male

Promesse e corse critiche

- Se si utilizza un oggetto di tipo promise per restituire un valore, si introduce un livello di sincronizzazione
- Il thread interessato ai risultati, invocando wait() o get() sul future corrispondente resta bloccato fino a che il thread detached non ha assegnato un valore
- Questo, però, non significa che il thread detached sia terminato
  - Potrebbe avere ancora del codice da eseguire (ad esempio, tutti i distruttori degli oggetti finora utilizzati)

Promesse e corse critiche

- La classe promise offre due metodi per evitare potenziali corse critiche
  - Tra la pubblicazione di un risultato nell'oggetto promise e la continuazione degli altri thread in attesa dello stesso
  - set_value_at_thread_exit(T val);
  - set_exception_at_thread_exit(std::exceptio n_ptr p);

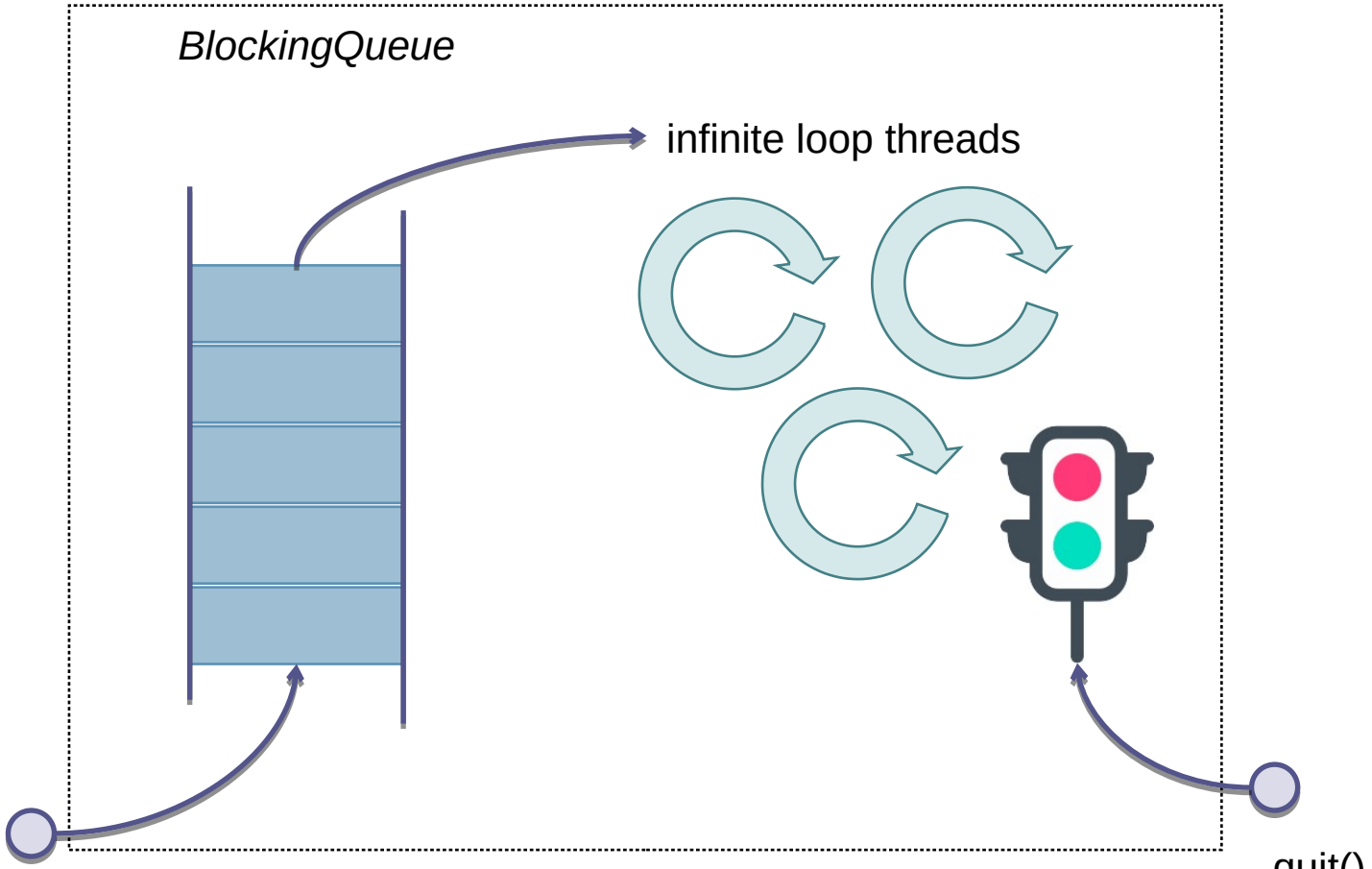
std::packaged_task<T(Args...)>

- Astrazione di un'attività in grado di produrre, prima o poi, un oggetto di tipo T
  - Come conseguenza dell'esecuzione di una funzione (o di un funzionale) che viene incapsulata nell'oggetto stesso, insieme ai suoi argomenti
- Quando un packaged_task viene eseguito, la funzione incaspulata viene invocata
  - Il risultato prodotto (o l'eventuale eccezione generata) viene utilizzato per valorizzare l'oggetto std::future associato al task

Thread pool

- In molte situazioni, è conveniente creare un numero limitato di thread (legato al numero di core disponibili) cui demandare l'esecuzione di compiti puntuali
  - Tali compiti non sono noti a priori e possono essere eseguiti da uno qualsiasi dei thread appositamente creati
  - La classe packaged_task si presta particolarmente per la realizzazione di una coda in cui ospitare le attività richieste e da cui i diversi thread del pool possono attingere le attività da svolgere
- Il numero di core disponibili può essere stimato attraverso la funzione
  - std::thread::hardware_concurrency();

Thread pool


BlockingQueue
infinite loop threads
quit()
submit(std::packaged_task<T(Args)> t)

Conoscere la propria identità

- Lo spazio dei nomi std::this_thread offre un insieme di funzioni che permettono di interagire con il thread corrente
- La funzione get_id() restituisce l'identificativo del thread corrente

Sospendere l’esecuzione

- La funzione sleep_for(duration) sospende l'esecuzione del thread corrente per almeno il tempo indicato come parametro
- La funzione sleep_until(time_point) interrompe l'esecuzione almeno fino al momento indicato
- La funzione yield() offre al sistema operativo la possibilità di rischedulare l'esecuzione del thread

Spunti di riflessione

- Si realizzi un programma che ricerchi una stringa in un elenco di file di testo
  - La ricerca in ciascun file è affidata a un thread separato, creato servendosi della classe std::thread
  - Il thread principale raccoglie e stampa i risultati