POLITECNICO DI TORINO

Programmazione di sistema

Esame del 19/6/2021 (API programming - Malnati)

	RICCARDO TEDESCO 269094
Iniziato	sabato, 19 giugno 2021, 11:04
Terminato	sabato, 19 giugno 2021, 12:34
Tempo impiegato	1 ora 30 min.
Punteggio	13,0/13,0
Valutazione	**15,0** su un massimo di 15,0 (**100**%)

Domanda 1

Completo

Punteggio ottenuto 3,0 su 3,0

Si descriva il concetto di risultato differito, come esposto dalla classe C++11 std::future<T>, indicando tre modi alternativi offerti dalla libreria standard per creare un oggetto di tale tipo e illustrando, per ciascuno di essi, un possibile caso d'uso che ne evidenzi l'utilità.

La classe std::future<T> è una classe astratta (non copiabile ma solo movibile) usata da alcune classi come, ad esempio, std::async, std::promise e std::package_tack.

Le tre classi suddette permettono di fare sincronizzazione ad alto livello, cioè consentono di eseguire una computazione in parallelo senza doversi preoccuparsi esplicitamente di gestire le cose a basso livello (ad esempio di dover gestire manualmente mutex e condition variable).

Queste tre classi possono essere usate quando si ha garanzia che non ci sia accesso condiviso a dei dati.

Esse offrono la garanzia che il risultato di una computazione sarà, prima o poi, restituito (questo è il concetto di risultato differito). Esse ottengono questo risultato sfruttando appunto un oggetto di tipo future (restituiscono un future<T>).

Un oggetto di tipo future permette a chi ne invoca il metodo get() di prelevare il risultato della computazione. Tale metodo restituisce il dato se questo è presente, altrimenti blocca il thread corrente finché il dato non è pronto. E' possibile anche usare il metodo wait(), wait_until() e wait_for() per controllare se il risultato è presente senza prelevarlo. In particolare wait_for e wait_until permettono di non bloccarsi oltre un certo tempo/data se il risultato ancora non è pronto.

Il risultato da un oggetto thread può essere prelevato solo una volta, quindi se si invoca get() due volte, alla seconda viene lanciata un'eccezione. E' possibile risolvere questo problema usando un oggetto di tipo shared_future<T>, che permette appunto di chiamare più volte get() (oltre ad

essere anche copiabile e non solo movibile).

Un oggetto future sfrutta il paradigma RAII, e quindi il suo distruttore, se nessuno l'ha invocata, esegue automaticamente la get(). Come conseguenza di ciò, se un oggetto future viene distrutto il thread resterà comunque bloccato finché la computazione non è terminata.

ESEMPIO con std::async

```
void f(int i)
{
  return std::string(i); //Esempio stupido che genera una stringa partire da un intero.
}
int main()
{
std::future<string> myFuture = std::async(f, 5); //Faccio calcolare f(5) a un altro thread (potenzialmente).
std::string s = f(6); //Il thread corrente calcola f(6)
std::string result = s + myFuture.get(); //Invocando get, se async non ha potuto creare un nuov thread calcolerò io il risultato.
return 0;
}
```

async in particolare può ricevere un parametro opzionale che può assumere:

- std::async --> crea un nuovo thread a cui demandaere l'esecuzione. Se ciò non è possibile, viene lanciata un'eccezione
- std::defer --> chi invocherà get avrà il compito di svolgere la computazione
- se nessuno dei due è indicato, vien prima provata async, altrimenti diventa defer.

std::package_task è molto spesso usata per la realizzazione di un thread pool per rappresentare i task da inserire nella coda di esecuzione condivisa. Essa fornisce il metodo get_future() per restituire un oggetto future.Il thread che vuole passare un package_task al thread pool, quindi, dovrebbe prima invocare il metodo get_future() per farsi restituire una future dalla quale, prima o poi, potrà recuperare il risultato.

La classe promise, infine, pure garantisce che prima o poi il risultato di una computazione sarà restituito. Garantisce inoltre che, se la computazione lancia una eccezione, questa venga rilanciata quando si invoca la get della future. Una promise fornisce, infatti, i metodi set_value e set_exception

```
void f1(std::promise p<int>)
{
  //Esegue una qualche computazione.
```

```
}
int main()
{
  std::promise<int> prom;
  std::future<int> futureDumb = prom.get_future();
  std::thread t(f1, std::move(prom));
  t.detach();  //Qui posso fare tranquillamente detach grazie alla future.
  //Eseguo qualche altra cosa in parallelo.
  int result = futureDumb.get();
}
```

Nel caso, della promise bisogna fare attenzione al fatto che il thread principale potrebbe terminare quando la promise ancora non è stata completamente finalizzata (è vero che la computazione è terminata quando si estre il risultato da get(), ma la promise potrebeb ancora dover eseguire il distruttore). Per tale ragione, una promise fornisce anche i metodi set_value_at_exit e set_exception_at_exit.

Gli oggetti di sincronizzazione ad alto livello, come per i thread, sono utili se il numero di operazioni da eseguire è elevato, altrimenti sarebbe più costoso creare un thread che non svolgere la computazione da sé.

Facilitono la restituzione di un risultato rispetto ai thread normali.

Commento:
ok

Domanda 2

Completo

Punteggio ottenuto 3,0 su 3,0

In un sistema operativo conforme alle specifiche Posix, cosa succede se in un processo in cui sono in esecuzione più thread, viene invocata la chiamata di sistema fork()? Al suo ritorno, quanti thread saranno presenti nel processo figlio e quanti nel processo padre? Che tipo di controllo può esercitare il programmatore per gestire questo tipo di situazione?

Nel caso di un programma concorrente (avente quindi più di un thread) l'esecuzione della fork creerebbe problemi: il thread figlio, infatti, avrebbe solo un thread mentre tutte le primitive di sincronizzazione usate dal padre potrebbero così trovarsi in stati incongruenti.

Per gestire questo tipo di situazioni, il programmatore può usare la funzione da invocare prima di eseguire una fork (se invocata solo una volta, tutte le fork saranno sempre soggette ad essa).

void pthread_atfork(void (*prepare)(void), void (*parent)(void), void(*child)(void));

La funzione riceve tre puntatori a funzione:

- il puntatore alla funzione prepare, ha il compito di eseguire delle operazioni preliminari prima di invocare la fork (ad esempio fermare i threads (non è detto che sia semplice da fare)).
- il puntatore alla funzione parent eseguirà del codice nel processo padre una volta che la fork sarà ritornata. Essa dovrebbe ad esempio fare ripartire i threads;
- il puntatore alla funzione child eseguirà del codice nel processo figlio una volta che la fork sarà ritornata.

Sabbene venga fornita la funzione pthread_atfork la realizzazione delle tre funzioni che essa riceve sono spesso complicate. Per tale ragione. sarebbe opportuno non effettuare alcuna fork all'interno di un processo concorrente. O comunque, se è possibile, fare la fork prima che il processo abbia generato ulteriori threads oltre a suo thread principale.

Commento:
ok

Domanda 3

Completo

Punteggio ottenuto 2,0 su 2,0

Si spieghino le differenze tra costruzione di copia e costruzione per movimento nel linguaggio C++11 in termini di comportamento e di prestazioni, e si descriva un caso d'uso nel quale il movimento risulta vantaggioso rispetto alla copia.

Il costruttore di copia permette di inizializzare un oggetto a partire da una istanza di un oggetto (della stessa classe) già esistente.

La sintassi è:

MyClass(const MyClass& other) { ... }

in cui possiamo osservare che bisogna passare come parametro sempre un riferimento costante a un oggetto.

Dal punto di vista della semantica, esso dovrebbe copiare ricorsivamente il contenuto dell'oggetto other tale e quale è all'interno dell'oggetto che si sta inizializzando.

La versione di default fornita dal compilatore copia tutto così com'è, e nel caso dei puntatori duplica l'indirizzo e non l'area di memoria. Ciò può essere problematico perché due oggetti punteranno alla stessa area di memoria credendo però di essere gli unici possessori di quel puntatore.

Un puntatore quindi richiede l'attenzione esplicita del programmatore, ipotizziamo ad esempio che MyClass abbia un char* ptr e un campo length, si dovrebeb scrivere

```
MyClass(const MyClass& other) : length(other.length)
{
    this->ptr = new char[this->length];
    memcpy(this->ptr, other.ptr, length);
    //L'implementazione fornita dal compilatore invece farebbe this->ptr = other.ptr
}
```

L'operazione di copia può essere molto onerosa in termini di tempo (oltre che rischiosa se non si fanno le cose per bene e richiede sempre una forte attenzione in caso di manutenzione della classe). Inoltre, moltri oggetti non sono copiabili (e.g. thread, mutex, future) perché la loro copia sarebbe pericolosa (è possibile impedire la copia contrassegnando il costruttore di copia sia l'operatore di assegnazione per copia come delete).

Per tutte queste ragioni suddette, spesso è conveniente usare il costruttore per movimento. Esso in pratica "cede" il proprio contenuto a un altro oggetto, in quanto sta per essere distrutto. Sono automaticamente candidati al movimento i valori di ritorno di una funzione, le variabili anonimi e tutte quegli oggetti che stanno per terminare il loro ciclo di vita).

La sintassi del costruttore di movimento è

MyClass(MyClass && other) { .... }

osserviamo la presenza di && che si legge "r-value reference" (attenzione: NON è un riferimento a un riferimento). Inoltre other non è contrassegnato come const, proprio perché l'oggetto sarà modificato.

Semanticamente, il costruttore di copia deve fare in modo che i dati che veranno distrutti non causino problemi in seguito:

```
MyClass(MyClass && other) : length(other.length)
{
```

this->ptr = other.ptr;

other.ptr = nullptr; //Istruzione fondamentale per evitare danni in seguito.

}

Se nell'esempio sopra non si facesse other.ptr = nullptr; il distruttore di other dealocherebeb la memoria, lasciando un puntatore corrotto in this->ptr.

Il compilatore non fornisce alcuna implementazione del costruttore di movimento, fa quindi sempre implementato a mano se si vuole usare.

Un oggetto può essere esplicitamente convertito in un r-value reference usando la funzione std::move(oggetto), naturalmente poi non bisognerà più accedere a quell'oggetto.

Il costruttore di movimento, oltre ad essere in genere necessario per gli oggetti non copiabili, è molto più vantaggioso del costruttore di copia per oggetti di grosse dimensioni.

Negli esempi fatti sopra, infatti, ipotizzando di avere un campo length molto grande, al costruttore di movimento basterebbe solo muovere un puntatore, mentre il costruttore di copia deve copiare tutto il contenuto dell'area di memoria che sta in ptr.

Queste operazioni sono fonte di molti problemi, specie in caso di presenza di puntatori (soprattutto se si andasse a considerare gli operatori di assegnazione che sono ancora più problematici). Per tale ragione, il C++11 fornisce gli smartpointers per evitare di dovere "combattere" troppo spesso con questi problemi.

Commento:
ok

Domanda 4

Completo

Punteggio ottenuto 5,0 su 5,0

La classe SingleThreadExecutor implementa il concetto di ThreadPool basato su un singolo thread incapsulato all'interno di ciascuna sua istanza. In assenza di richieste di lavoro, tale thread resta fermo senza consumare cicli macchina.

Attraverso il metodo submit(...) è possibile affidare ad un'istanza di SingleThreadExecutor un compito da eseguire. Tale compito viene inizialmente accodato e, non appena il thread incapsulato è libero, viene eseguito.

Attraverso il metodo close() è possibile impedire l'ulteriore accodamento di compiti (eventuali tentativi di invocare submit(...) dopo la chiamata a close() origineranno un'eccezione).

Attraverso il metodo join() è possibile attendere che tutti i compiti ancora da svolgere siano svolti e il thread incapsulato nell'istanza termini.

Si implementi tale classe usando le funzionalità offerte dalla libreria C++11, definendo tutte le parti eventualmente mancanti nella definizione della classe.

Si faccia attenzione al fatto che il codice che può essere sottomesso all'esecutore è arbitrario e può contenere richieste di sottomissione di ulteriori compiti allo stesso esecutore.

```
class SingleThreadExecutor {
  public:
    void submit(std::packaged_task<void()> t); //invia un compito da eseguire o lancia un'eccezione se l'
istanza è chiusa
    void close(); //impedisce la sottomissione di compiti ulteriori, permettendo la terminazione del thread
incapsulato
    void join(); //attende la terminazione del thread incapsulato
};
```

```
class SingleThreadExecutor
{
public:
  SingleThreadExecutor() : bOpen(true)
  {
    std::thread t{[this]() {
      this->handle(); //Chiama la funzione privata.
    });
    this->myThread = std::move(t);
  }

  ~SingleThreadExecutor()
  {
    this->close();
  }

  void submit(std::package_task<void> t)
  {
     const std::lock_guard<std::mutex> lg{this->m};
     if (this->bOpen == false)
       throw std::logic_error("errore");
     //Inserisco un nuovo task.
     this->tasks.push_back(std::move(t));
     this->cv.notify_one();
  }

  void close()
  {
    const std::lock_guard<std::mutex> lg{this->m};
    this->bOpen = false;
    this->cv.notify_one();
  }
```

```
  void join()
  {
    if (this->myThread.joinable() == true)
    {
      //Chiamo la close e aspetto che tutti i task siano stati eseguiti.
      this->close();
      this->myThread.join();
    }
  }
private:
  std::mutex m;
  std::condition_variable cv;
  bool bOpen;
  std::thread myThread;
  std::list<std::package_task<void>> tasks;

  void Handle(void)
  {
    std::unique_lock<std::mutex> ul{this->m};
    while (this->bOpen == true || this->tasks.empty() == false)
    {
      this->cv.wait(ul, [this]() {
        //Non aspetto se è stata invoca close() oppure se o dei task da eseguire.
        return (this->bOpen == false || this->tasks.empty() == false);
      });

      //Verifico che sono stato svegliato ed ho un task da eseguire (altrimenti significa che devo
interrompere)
      if (this->tasks.empty() == false)
      {
        //Estraggo il prossimo task da eseguire e poi lo rimuovo dalla coda.
        std::package_task<void> currentTask = this->tasks.front();
        this->tasks.pop_front();
        //Libero il lock prima di svolgere il task, svolgo il task e poi riprendo il lock.
        ul.unlock();
        currentTask();
        ul.lock();
      }
    }
  }
};
```

Commento:
Occorre che il distruttore attenda anche la terminazione del thread invocando il metodo join().