Thread Pool C++

Maurizio Terreni

Università degli studi di Firenze
maurizio.terreni@stud.unifi.it

5 gennaio 2017

Sommario

Il progetto di mid-term svolge il compito di implementare una classe Thread Pool in C/C++ usando Pthreads e/o C++11. Il Thread Pool ha il compito di far eseguire tramite un numero limitato di thread una queue di task, al fine di ottimizzare il througthput.

I. Introduzione

Lo scopo di un Thread Pool è quello di sottomettere dei task (work) ad un numero limitato di thread. Sarà quindi compito del *Thread Pool* affidare i task ad un certo thread. I *Thread Pool* sono dotati di una coda interna di task in attesa e di un certo numero di thread con cui eseguirla. Nel progetto in questione viene eseguita la regola di tipo FIFO ovvero First Input First Output (Fig. 1) in quanto il task arrivato per primo sarà in testa di esecuzione e cosi via fino al termine della queue di work. Il concetto di ricorrere all'utilizzo della seguente strategia è quello di riutilizzo di un thread, infatti un thread viene riusato per far eseguire diversi task durante il suo ciclo di vita. Questo diminuisce l'overhead dovuto alla creazione dei thread e incrementa le presatazioni dei programmi che sfruttano questo tipo di concetto.

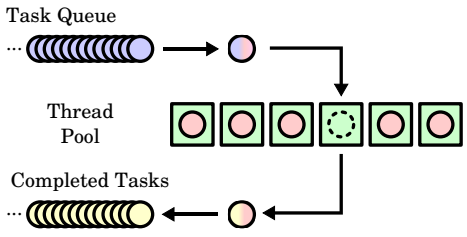

Figura 1: *Esempio grafico di ThreadPool*

II. Tempo di Overhead

Lo studio del tempo di Overhead è fondamentale per individuare le prestazioni di un sistema operativo. Esso rappresenta il tempo medio di CPU necessario per eseguire il moduli nel Kernel. Il calcolo ha la seguente espressione:

$$Overhead_{\%} = \frac{T_{CPU} perl'esecuzione del Kernel}{T_{CPU} totale di utilizzo} \quad (1)$$

Ovviamente più il tempo è basso, maggiore sarà la quantità di CPU che possiamo dedocare ai processi utente.

Nei linguaggi di programmazione ad alto livello, un esempio di overhead riguarda il tempo di esecuzione dei thread. Ogni thread infatti aggiunge il tempo necessario a gestire il meccanismo stesso di allocazione ed esecuzione del thread, questo tempo può essere ridotto utilizzando specifiche tecniche di ottimizzazione, ma non può essere eliminato del tutto. Una tecnicna utilizzata appunto è il thread pool che ci consente la generazione di un numero limitato di thread per l'esecuzione di una queue di N work in modo tale da evitare la creazione e l'allocazione di N thread.

III. Implementazione

Come possiamo vedere dal grafico UML (Fig. 2) notiamo che per realizzare il Thread Pool sono state implementate principalmente tre classi:

- Classe Queue
- Classe Work
- Classe ThreadPool

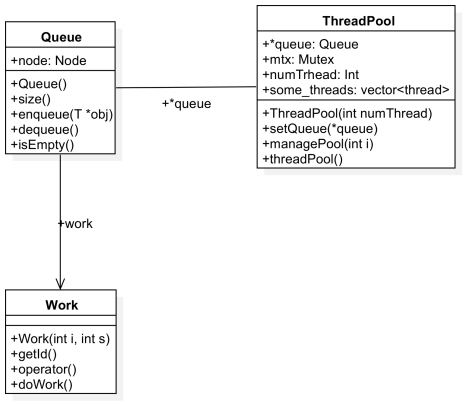

Figura 2: *Uml Progetto*

i. Classe Queue

La classe *Queue* ha il compito di gestire la coda di task da eseguire. L'implementazione è stata realizzata tramite l'utilizzo delle code con puntatore. Analizzando il codice possiamo osservare che l'oggetto *Queue* è caratterizzato da due puntatori, un puntatore alla testa della coda e uno al fondo della coda. Mentre la classe Node è caratterizzata da un campo data che contiene l'oggetto task e dal campo Next che è un puntatore al nodo successivo realizzando cosi la struttura concatenata (Fig. 3).

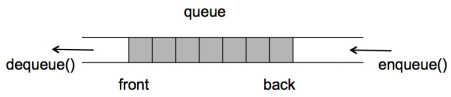

Figura 3: *Esempio coda con puntatori*

```
void enqueue(T *obj){
        Node<T>* tmp = new Node<T>();
        tmp->data = obj;
        tmp->next = NULL;
        if ( isEmpty() ) {
            front = rear = tmp;
        }else {
            rear->next = tmp;
            rear = tmp;
        }
        count++;
}
```

Listing 1: *Esempio di Enqueue()*

Il codice in figura mostra come avviene l'operazione di *enqueue*. Come parametro viene fornito un oggetto generico, infatti non conosciamo a priori quale tipo di task che aggiungiamo, ma viene semplicemente aggiunto alla queue, sarà compito del *ThreadPool* eseguirlo in modo corretto. Una volta ottenuto il riferimento all'oggetto fornito viene eseguito un controllo sulla queue se vuota l'oggetto in questione viene aggiunto sia alla testa sia alla coda altrimenti viene concatenato.

```
T * dequeue(){
        if ( isEmpty() )
            return NULL;
        T *risu = front->data;
        Node<T>* tmp = front;
        front = front->next;
        count--;
        delete tmp;
        return risu;
}
```

Listing 2: *Esempio di Dequeue()*

Mentre come possiamo notare nella *dequeue()* eseguiamo un controllo a priori per verificare che la coda non sia vuota, nel caso in cui siano presenti elementi viene estratto il nodo in testa e restituito al chiamante.

ii. Classe Work

```
class Work {
  private:
    int id;
    int s;
  public:
    Work(int i,int s);
    int getId();
    void doWork();
};
```

Listing 3: *Classe Work*

La classe *Work* non è altro che il task da eseguire, al momento della creazione riceve come parametri due interi, uno indica l'id del task e l'altro il tempo di sleep. Il metodo *doWork()* esegue uno sleep di *s* secondi con *s* generato

casualmente nel *main* in modo da fornire tempimpistiche di terminazione diverse per ogni task.

iii. Classe ThreadPool

La classe *ThreadPool* ha lo scopo di eseguire tutti i task attraverso un numero limitato di thread. Come parametri viene fornita la queue popolata con i work e il numero di task da eseguire contemporaneamente.

```
1 void ThreadPool::threadPool() {
2     int i = 0;
3     printf("Totale Work nella queue = %d\n" , queue->size());
4     for (int i = 0; i < this->numThread; ++i)
5         some_threads.push_back(std::thread(&ThreadPool::managePool,
              this, i));
6     for (auto& t: some_threads) t.join();
7     printf("Totale Work nella queue %d\n" , queue->size());
8 }
```

Listing 4: *Metodo ThreadPool()*

Come possiamo vedere in figura al momento in cui viene richiamato il metodo *threadPool()* vengono generati e successivamente lanciati gli *N* thread. Terminato il primo ciclo *for* viene richiamato il *join()* su ogni thread in modo da poter attendere la terminazione dei task nella queue prima di terminare il metodo.

```
1 void ThreadPool::managePool(int i) {
2     while(!queue->isEmpty()) {
3         mtx.lock();
4         Work w = *(queue->dequeue());
5         printf("Thread %d run work %d\n",i,w.getId());
6         mtx.unlock();
7         w.doWork();
8     }
9 }
```

Listing 5: *Metodo ManagePool()*

Ogni Thread quindi ha il compito di prelevare dalla coda attraverso un'operazione atomica l'elemento in testa, attendere che il task sia completato e procedere con l'elemento successivo, tutto questo finchè sono presenti elementi nella queue.

iv. Classe Main

Il main come possiamo osservare nel Sequence Diagram (Fig. 4) ha il compito di generare un numero di *Work* associandogli un id e un tempo di terminazione casuale. Successivamente aggiunge il task alla coda popolandola cosi con i vari work da far eseguire. Una volta popolata la queue viene passata come parametro all'oggetto threadPool che avrà come compito quello di estrarre un task dalla coda ed eseguirlo.

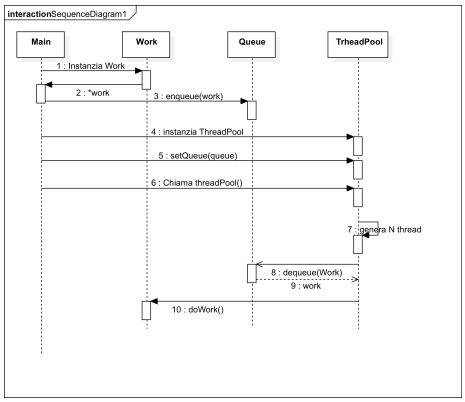

Figura 4: *Sequence Diagram*

IV. Esecuzione

Come possiamo vedere dall'output (Fig. 5) generato al momento dell'esecuzione abbiamo un totale di 15 task e 4 thread abilitati ad eseguirli. Al momento della generazione i 4 thread prendono i primi 4 task da eseguire stampando affianco il relativo id. Al momento in cui il work viene richiamato stampa affianco al proprio identificativo la durata in secondi dello sleep. Al termine il rispettivo thread viene liberato ed è quindi in grado di poter estrarre dalla coda il task in testa ed eseguirlo.

```
Tread Pool Terreni Maurizio Parallel Computing 2016-2017
Numero di work eseguiti contemporaneamente 4
Numero di work nella queue 15


Totale Work nella queue = 15
Thread 1 run work 1
Thread 0 run work 2
Thread 2 run work 3
Thread 3 run work 4
######  work 1 start with duration 11
######  work 2 start with duration 15
######  work 3 start with duration 13
######  work 4 start with duration 11
######  work 1 end
######  work 4 end
Thread 1 run work 5
Thread 3 run work 6
######  work 5 start with duration 15
######  work 6 start with duration 2
```

Figura 5: *Output generato al momento dell'esecuzione*

V. Conclusioni

Come descritto in precedenza il vantaggio nell'utilizzo del Threadpool è proprio nel fatto che andiamo a riutilizzare per l'esecuzione di alcuni task, thread già creati in precedenza, quindi significa risparmiare il tempo necessario ad allocare in memoria un thread.

# Task	ThreadPool	Thread
5	0.149 ms	0.259 ms
15	0.173 ms	1.016 ms
25	0.184 ms	1.890 ms
50	0.183 ms	3.441 ms
100	0.157 ms	3.501 ms

Tabella 1: *Tempi necessari a generare n Task*

Nella tabella (Tab. 1) è possibile notare il tempo (espresso in millisecondi) impegato per generare i thread necessari ad eseguire i vari task. Come possiamo notare *ThreadPool* impiega un tempo pressochè costante in quanto ogni volta vengono generati soltanto 4 thread, mentre a fianco possiamo notare come il tempo cresce in maniera esponenziale se dovessimo generare un numero di thread pari al numero di task.

Riferimenti bibliografici

[C++ step by step] R.Lago, R.Manca, A.Monti (2004). Elemond scuola & azienda.

[Patterns for Parallel Programming] T.Mattson, B.Sanders, B.Massingill Addison-Wesley Professional

[Git - Repository] https://github.com/maurizioterreni/ThreadPoolC-