Implementazione Thread Pool in C/C++ usando Pthreads

Problema

Parliamo un po' dei thread?

I thread nella programmazione forniscono un buon paradigma per creare applicazioni che eseguono più compiti nello stesso momento. Di solito se abbiamo bisogno di svolgere dei compiti creiamo un thread e glieli facciamo fare. Usare i thread può semplificare la logica dell'applicazione e nei casi di aver a disposizione processori multipli può portare ad avere dei vantaggi. Tuttavia il creare troppi thread può causare dei ritardi nelle performance dovute al contendersi delle risorse. L'applicazione potrebbe spendere molto del suo tempo nel dover aspettare l'accesso a delle risorse piuttosto che eseguire lavoro utile. Inoltre creare thread, anche se è più economico che creare dei processi, rimane piuttosto costoso. Creare molti thread per piccole quantità di lavoro è uno spreco.

Cosa è un thread Pool?

Il thread Pool lo possiamo considerare come il manager di un numero di thread che eseguono del lavoro su richiesta. Essi non terminano immediatamente. Quando un thread completa un compito, va nello stato "Idle", dove attende che gli sia assegnato un nuovo compito da eseguire. Quindi il consumo per creare e distruggere thread è limitato alla creazione e distruzione del numero dei thread attivi nel thread Pool. L'applicazione può avere più lavori da svolgere del numero dei thread presenti, anzi è questo il caso usuale. L'utilizzo del processore e il passaggio dei compiti sono migliorati dalla riduzione del tempo di accesso alle risorse contese. I task da fare sono processati in ordine e di solito più velocemente del caso in cui viene creato un thread per ogni task.

Obiettivo del progetto? In questo progetto voglio implementare il thread Pool in linguaggio C/C++ usando Pthreads creando i due file threadPool.h e threadPool.c. Cercherò anche di provare i vantaggi descritti precedentemente tramite due file di test.

Lavoro Svolto

Nel file threadpool.h:

- Ho dichiarato un nuovo tipo di dato che ho chiamato threadPool.

Codice sorgente:

```
/* Define new type: threadPool_Thread */
typedef struct threadPool threadPool_Thread;
```

- Ho dichiarato la funzione threadpool_init che è quella che ha il compito di creare il threadpool. Prende come parametri in ingresso il numero dei thread lavoratori e il numero della grandezza della coda e restituisce il puntatore al threadpool.

Codice sorgente:

```
threadPool_Thread *threadpool_init(unsigned int numThread, unsigned int
                                   queueSize);
```

- Ho dichiarato la funzione threadpool_add_task che è quella che aggiunge un lavoro nella coda dei lavori di un determinato threadpool. Prende come parametri di ingresso il puntatore al threadpool e la funzione con i suoi argomenti che rappresentano il lavoro da aggiungere. Restituisce il valore 0 nel caso tutto proceda senza problemi altrimenti un numero negativo.

Codice sorgente:

```
int threadpool_add_task(threadPool_Thread *pool, void (*function)(void
                        *), void *argument);
```

- In fine ho dichiarato la funzione threadpool_destroy che è quella che ha il compito di deallocare la memoria e cancellare lavori, lavoratori e threadpoll stesso. Come parametro prende solo il puntatore al threadpool da cancellare e restituisce un intero uguale a 0 se tutto è andato a buon fine o un numero negativo in caso di errore.

Codice sorgente:

- `int threadpool_destroy(threadPool_Thread *pool);`

Nel file threadpool.c:

- Ho implementato il tipo task ovvero il lavoro che dovrà essere svolto che risulta essere una funzione con possibili argomenti in ingresso e ho implementato la struttura threadpool.

- Ho deciso di creare questa struttura in modo tale che il numero di thread da gestire e la dimensione della grandezza della coda siano decisi a priori e passati come argomento alla funzione predisposta all'inizializzazione del threadpool.
  I lavori invece ho deciso di metterli in una coda dove mi memorizzo inizio, fine e numero di elementi e che poi gestisco in modo FIFO.
  Inoltre l'accesso alle risorse critiche le gestisco con i semafori.

Allego in seguito il codice sorgente relativo:

```
/* Threadpool */
struct threadPool {

    int threadNum;          /* Thread number */
    int queueSize;          /* Queue of task to do */
    task *queue;            /* Array containing the task queue */


    /* FIFO queued job */
    int head;               /* Index of the first element */
    int tail;               /* Index of the last element */
    int count;              /* Number of pending tasks */

    int started;            /* Number of started threads */

    int shutdown;           /* Flag indicating if the pool is …*/

    pthread_mutex_t lock;   /* Condition variable to lock worker …*/
    pthread_cond_t notify;  /* Condition variable to notify worker …*/
    pthread_t *threads;     /* Array containing worker threads ID */


};
```

La funzione threadpool_init ha il compito di:

- Allocare la memoria per la struttura del treadpool
- Inizializzare tutti gli elementi per gestire la coda
- Allocare memoria per i thread lavoratori
- Allocare memoria per i lavori nella coda
- Inizializzare le variabili condizione per l'accesso alle risorse critiche
- Creare i thread e iniziarli a fare lavorare

La funzione threadpool_thread è richiamata dentro la funzione threadpool_init. Nello specifico, nel momento in cui vengono creati i thread, essa viene passata come terzo argomento alla funzione pthread_create.

La funzione threadpool_thread ha il compito di:

- levare un lavoro dalla coda del threadpool
- aggiornare tutte le variabili per la gestione della coda.

La funzione threadpool_add_task ha il compito di:

- Aggiungere lavoro nella coda del threadpool. Lo fa attraverso un ciclo infinito che si ferma solo in caso di coda piena o di spegnimento del threadpool.

  Ad ogni iterazione:
- Controlla le variabili di accesso e le due condizioni precedentemente descritte
- Aggiunge la funzione in coda
- Aggiorna le variabili della coda

La funzione threadpool_destroy ha il compito di:

- Deallocare la memoria che ha usato per i thread
- Deallocare la memoria che ha usato per la coda
- Deallocare la memoria che ha usato per la struttura del threadpool

  Il tutto facendo i relativi controlli di esistenza del threadpool e dei semafori.

Utilizzo

Nel file testTP.c:

- Mi assicuro che il tutto funzioni attraverso degli asset
- Vado effettivamente a testare le funzionalità del threadpool

La funzione task ha il compito di:

- Presentare il thread che la esegue
- Aspettare del tempo (modificabile)
- Aggiornare la variabile condivisa done (rappresenta il numero dei lavori conclusi da parte dei thread)

La funzione main ha il compito di:

- Far partire il tempo di registrazione
- Creare il threadpool
- Riempire la coda del threadpool di lavori
- Aspettare che i thread concludano i lavori
- Fermare il tempo e calcolare quanto è stato impiegato

Nel file testST.c:

- Calcolo il tempo che impiego per creare un certo numero di lavori che saranno svolti ognuno da un singolo thread.

Tramite questi due test, cambiando le variabili, numero thread, numero dei lavori e/o complessità del lavoro è possibile avere una idea di quando conviene usare un metodo rispetto all'altro.