LABORATORIO DI RETI E SISTEMI DISTRIBUITI

HANDS-ON 8

Ricerca multithreading di numeri primi

Antonio Mastrolembo Ventura

543644

April 4, 2025

Indice

1 Introduzione

In questo hands-on andremo ad esplorare il funzionamento del **multithreading** attraverso un esercizio in C. Il multithreading permette l'esecuzione simultanea di più thread all'interno di un singolo processo, consentendo un miglior utilizzo delle risorse del sistema e una maggiore efficienza nei calcoli paralleli. Ogni thread rappresenta un flusso indipendente di esecuzione che condivide lo stesso spazio di memoria con gli altri thread del processo.

2 Definizione del problema

L'obiettivo di questo hands-on è quello di realizzare un programma multithread in linguaggio C che permetta di individuare tutti i numeri primi compresi tra 1 e $1.000.000$. Per fare ciò, utilizzeremo 4 thread, ciascuno incaricato di analizzare una porzione distinta del range complessivo dei numeri. Infine, andremo a modificare l'esercizio per associare ciascun thread a uno dei primi 4 core del processore. E' un processo chiamato *CPU Affinity* e consente infatti di legare l'esecuzione di un thread a uno specifico core.

3 Metodologia

Creiamo il programma `isPrimeThreads.c` e vediamo nel dettaglio la sua implementazione.

3.1 Libreria pthread

```
#include <pthread.h>
```

Includiamo la libreria per l'utilizzo dei thread POSIX.

3.2 Definizione delle costanti

```
#define MAX_NUMBER 1000000
#define MAX_THREAD 4

int rangePerThread = MAX_NUMBER / MAX_THREAD;
```

Definiamo due costanti: i numeri totali da analizzare e il numero di thread che useremo; successivamente calcoliamo il numero di valori che ciascun thread dovrà controllare dividendo il numero totale per il numero dei thread.

3.3 Funzione is_prime

```
bool is_prime(int n) {
    if (n <= 1) return false;
    if (n == 2) return true;
    if (n % 2 == 0) return false;

    for (int i = 3; i * i <= n; i += 2) {
        if (n % i == 0) return false;
    }

    return true;
}
```

La funzione restituisce `true` se il numero passato è primo, altrimenti `false`.

3.4 Funzione check_primes

```
void* check_primes(void* arg) {
    int start = *(int*)arg;
    int end = start + rangePerThread;

    // Controllo i numeri nel range
    for (int i = start; i < end; i++) {
        if (is_prime(i)) {
            printf("%d is prime\n", i);
        }
    }

    pthread_exit(NULL);
}
```

Questa è la funzione che verrà eseguita da ogni thread. Riceve in input l'inizio del range da analizzare e calcola il punto in cui si dovrà fermare. Successivamente controlla ogni numero nel suo range e se è primo lo stampa. Una volta esaminati i propri numeri, il thread termina con la chiamata `pthread_exit` e non ritorna nulla.

3.5 Funzione main

```
int main() {
    // Creo un array di thread
    pthread_t threads[MAX_THREAD];

    // Ciclo per creare un thread per ogni range
    for (int i = 0; i < MAX_THREAD; i++) {

        // Punto di partenza per il thread
        int start = i * rangePerThread;

        // Creo il thread
```

```
12          if (pthread_create(&threads[i], NULL, check_primes, &start)
    != 0) {
13              printf("Error creating thread %d\n", i);
14              return 1;
15          }
16      }
17
18      // Aspetto che tutti i thread finiscano
19      for (int i = 0; i < MAX_THREAD; i++) {
20          pthread_join(threads[i], NULL);
21      }
22  }
```

Nel main creiamo 4 thread attraverso la chiamata `pthread_create` a cui passiamo come argomenti: il puntatore alla variabile di tipo `pthread_t` dove verrà salvato l'identificatore del thread appena creato, eventuali attributi, la funzione che il thread dovrà eseguire e l'argomento che passiamo alla funzione.

Infine, tramite la chiamata `pthread_join`, il thread chiamante (nel nostro caso `main`) attende la terminazione di ciascuno dei thread creati. E' un'operazione fondamentale che garantisce che il programma principale aspetti la conclusione di tutti i thread prima di terminare.

3.6 Affinity

A questo punto possiamo modificare l'esercizio assegnando ad ogni thread uno specifico core del processore.

3.6.1 Libreria sched.h

```
1  #include <sched.h>
```

Includiamo la libreria che fornisce delle funzioni legate allo scheduling e, nel nostro caso, all'affinity.

3.6.2 Modifica dei parametri

```
typedef struct {
    int start;
    int core_id;
} thread_params;
```

Utilizziamo una struct perchè adesso dovremo passare più argomenti alla funzione eseguita dal thread.

3.6.3 Assegnazione dei core

Fissiamo ad ogni thread un core e passiamo gli argomenti alla funzione che eseguirà il thread:

```
for (int i = 0; i < MAX_THREAD; i++) {
        thread_data[i].start = i * rangePerThread;
        thread_data[i].core_id = i;

        if (pthread_create(&threads[i], NULL, check_primes, &
    thread_data[i]) != 0) {
            perror("Errore creazione thread");
            return 1;
        }
    }
```

3.6.4 Configurazione dell'affinity

```
    cpu_set_t cpuset;
    CPU_ZERO(&cpuset);
    CPU_SET(core, &cpuset);

    if (pthread_setaffinity_np(pthread_self(), sizeof(cpu_set_t), &
    cpuset) != 0) {
        perror("Errore nell'impostare l'affinita'");
    }
```

Dopo aver stabilito il core che ogni thread deve avere, inizializziamo un `CPU set` con la funzione `CPU_ZERO` e, successivamente, tramite `CPU_SET`, aggiungiamo il core desiderato al set.
Infine, chiamiamo la funzione `pthread_setaffinity_np()` che assegna effettivamente ciascun thread a un core specifico.

3.6.5 Verifica

```
1    int actual_core = sched_getcpu();
2    printf("Thread per range [%d, %d) in esecuzione sul core %d\n",
     start, end, actual_core);
```

Possiamo infine utilizzare la funzione `sched_getcpu()` per capire su che core sta eseguendo un thread.

4 Presentazione dei risultati

4.1 Affinity

Per verificare che il tutto funzioni correttamente diamo una prima occhiata all'affinity impostata. Possiamo utilizzare il comando:

```
$ watch -n 0.5 "ps -eLo pid,tid,psr,cmd | grep 'isPrimeThreads'"
```

Questo comando monitora ogni 0.5 secondi i processi attivi e i relativi thread, core e comando eseguito. Nel mio caso l'output ottenuto è il seguente:

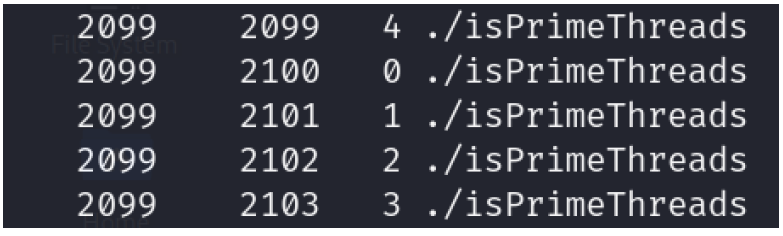

Figure 1: Affinity

Osserviamo che il processo principale ha PID 2099 e gestisce più thread identificati dai rispettivi TID (Thread ID) da 2100 a 2103. La colonna PSR (Proces-

sor/Core ID) indica il core della CPU su cui ciascun thread sta effettivamente girando.
Notiamo che ogni thread è stato assegnato a un core differente:

- Il thread con TID 2100 è eseguito sul core 0;
- Il thread con TID 2101 è eseguito sul core 1;
- Il thread con TID 2102 è eseguito sul core 2;
- Il thread con TID 2103 è eseguito sul core 3.

Questo conferma che l'affinity è stata impostata correttamente, rispettando la distribuzione dei thread sui core disponibili. Notiamo infine che il processo principale (PID 2099) sembra essere in esecuzione sul core 4, dunque il sistema operativo ha assegnato il processo principale a un core libero che non abbiamo specificato.

4.2 Threads

Un'altra cosa importante da notare è come i thread si alternino nell'esecuzione sui core assegnati.

```
745273 is prime (thread su core 2)
745301 is prime (thread su core 2)
959279 is prime (thread su core 3)
745307 is prime (thread su core 2)
959323 is prime (thread su core 3)
959333 is prime (thread su core 3)
959339 is prime (thread su core 3)
959351 is prime (thread su core 3)
959363 is prime (thread su core 3)
```

Figure 2: Parallelismo

Dall'output possiamo vedere come i thread si alternino e lavorino in modo parallelo. Questa alternanza è una caratteristica naturale del multithreading,

poiché ogni thread lavora indipendentemente sugli intervalli assegnati e produce output nel momento in cui completa un'elaborazione.

5 Conclusioni

In questo hands-on abbiamo realizzato un programma in C che sfrutta il multithreading per cercare numeri primi in parallelo. Dopo una prima implementazione base, abbiamo esteso il programma introducendo l'affinity, assegnando ciascun thread a un core specifico. Secondo i risultati ottenuti abbiamo dimostrato come i thread vengano effettivamente eseguiti in parallelo comprendendo il comportamento dei thread su una macchina multicore.