

LABORATORIO DI RETI E SISTEMI DISTRIBUITI

HANDS-ON 10

Grafi di precedenza e parallelismo

Antonio Mastrolembro Ventura

543644

April 8, 2025

Indice

1	Introduzione	2
2	Definizione del problema	2
3	Metodologia	3
3.1	Struct "Datas"	3
3.2	Operazioni fondamentali	3
3.3	Funzione principale	4
3.3.1	Thread e Struct	4
3.3.2	Creazione e attesa dei thread	5
3.4	Considerazione	6
3.5	Makefile	6
3.6	Implementazione in Python	8
4	Presentazione dei risultati	10
5	Confronto con i processi	10
6	Conclusioni	11

1 Introduzione

Con questo hands-on affronteremo un esercizio pratico basato sulla programmazione concorrente ed, in particolare, sull'utilizzo dei thread. Andremo a vedere come attraverso il calcolo in parallelo possiamo rendere più efficiente lo svolgimento di una semplice espressione matematica suddividendola in più sotto-operazioni. Vedremo anche la creazione di un grafo di precedenza che rappresenta l'evoluzione di un processo, dalla sua "nascita", alla fine della sua esecuzione.

2 Definizione del problema

Il problema in analisi consiste nel calcolare il valore dell'espressione aritmetica:

$$(2 \times 6) + (1 + 4) \times (5 - 2)$$

utilizzando un approccio basato sulla programmazione concorrente con thread in linguaggio C. Per fare ciò andremo a scomporre l'espressione in tante sotto-espressioni assegneremo ad ogni thread la risoluzione di ognuna di queste operazioni elementari. Per facilitarne la pianificazione e la successiva implementazione, costruiamo un grafo di precedenza per l'espressione aritmetica.

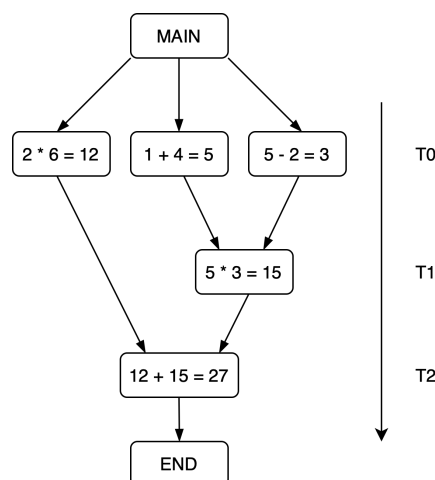


Figure 1: Grafo di precedenza

A ciascuna operazione è stato assegnato un istante temporale (T_n) corrispondente al momento in cui essa può essere eseguita, tenendo conto delle dipendenze precedenti.

3 Metodologia

Per sviluppare il problema utilizzeremo il linguaggio C attraverso la programmazione modulare, andando dunque a suddividere la logica del codice su più file rendendolo facilmente leggibile e gestibile. Vedremo successivamente la stessa logica implementata in Python.

3.1 Struct "Datas"

```
1 typedef struct {
2     int a, b;
3     int result;
4 } Datas;
```

Per prima cosa definisco un file `structDatas.h` che conterrà soltanto questa struttura dati in modo da poterla includere nei successivi due file che vedremo. Questa struct ha il compito di definire i dati che ogni thread dovrà gestire: `a`, `b` e `result`.

3.2 Operazioni fondamentali

```
1 #include <pthread.h>
2 #include "structDatas.h"
3
4 // Funzione per la moltiplicazione
5 void* multiply(void* arg){
6     Datas* data = (Datas*)arg;
7     data->result = data->a * data->b;
8
9     pthread_exit(NULL);
10 }
```

```

11
12 // Funzione per la somma
13 void* add(void* arg){
14     Datas* data = (Datas*)arg;
15     data->result = data->a + data->b;
16
17     pthread_exit(NULL);
18 }
19
20 // Funzione per la sottrazione
21 void* subtract(void* arg){
22     Datas* data = (Datas*)arg;
23     data->result = data->a - data->b;
24
25     pthread_exit(NULL);
26 }

```

Successivamente ho creato il file `computation.c` che conterrà tre funzioni differenti: ognuna svolge una determinata operazione matematica e, in questo caso, non ha valore di ritorno poichè la funzione `pthread_exit()` ha come argomento `NULL`. Nel caso in cui volessi ritornare un valore, dovrei invece passare come argomento il puntatore del valore che voglio restituire.

3.3 Funzione principale

Andiamo adesso a vedere il main del nostro programma.

3.3.1 Thread e Struct

```

1     pthread_t thread1, thread2, thread3;
2
3     Datas data1 = {2, 6, 0};
4     Datas data2 = {1, 4, 0};
5     Datas data3 = {5, 2, 0};
6     Datas data4 = {0, 0, 0};
7     Datas data5 = {0, 0, 0};

```

In prima battuta definiamo i nostri tre thread che si occuperanno di eseguire le operazioni dell'espressione. Successivamente inizializziamo le struct con i dati che gestiranno i thread.

3.3.2 Creazione e attesa dei thread

```
1  pthread_create(&thread1, NULL, multiply, (void*)&data1);
2  pthread_create(&thread2, NULL, add, (void*)&data2);
3  pthread_create(&thread3, NULL, subtract, (void*)&data3);
4
5  pthread_join(thread1, NULL);
6  pthread_join(thread2, NULL);
7  pthread_join(thread3, NULL);
8
9  data4.a = data2.result;
10 data4.b = data3.result;
11
12 data5.a = data1.result;
```

Successivamente creiamo i primi tre thread che si occuperanno di calcolare rispettivamente un prodotto, una somma e una differenza (vedi espressione). Queste tre operazioni corrispondono alle foglie del grafo di precedenza e, non avendo dipendenze, possono essere eseguite contemporaneamente.

Dopo essere stato creato, ciascun thread viene sincronizzato tramite la funzione `pthread_join()`, che permette al thread principale di attendere il completamento di ogni operazione prima di procedere con le successive.

Infine assegniamo alle due struct i risultati ottenuti, in particolare i due operandi che dovranno essere computati successivamente alla struct `data4` e il risultato della prima computazione a `data5` (che sarà l'operazione finale).

A questo punto possiamo procedere con l'istante T_1 del nostro grafo di precedenza effettuando la quarta computazione dell'espressione:

```
1 pthread_create(&thread1, NULL, multiply, (void*)&data4);
2 pthread_join(thread1, NULL);
3
4 data5.b = data4.result;
```

Terminata la computazione all'istante T_1 , assegniamo il risultato ottenuto alla struct data5, in particolare al secondo operando dell'ultima operazione.

Per concludere effettuiamo l'ultima operazione seguendo la stessa logica applicata fino ad ora:

```
1 pthread_create(&thread2, NULL, add, (void*)&data5);
2 pthread_join(thread2, NULL);
```

In questo modo, data5 conterrà il risultato finale della nostra espressione.

3.4 Considerazione

Per lo svolgimento dell'esercizio ho deciso di implementare la logica senza sfruttare il valore di ritorno dei thread in quanto andavo a modificare direttamente la struct in memoria con il risultato appena computato. L'altra opzione sarebbe stata appunto quella di effettuare l'operazione, restituire il valore catturandolo con `pthread_join()` ed aggiornare nel main la struct.

3.5 Makefile

Per automatizzare il processo di compilazione ho creato un Makefile:

```
# Nome dell'eseguibile che vogliamo ottenere
TARGET = main

# Compilatore usato
CC = gcc

CFLAGS = -Wall -pthread
```

```
# File sorgente
SRCS = main.c computation.c

# File oggetto per il linking
OBJS = $(SRCS:.c=.o)

# Regola principale
$(TARGET): $(OBJS)
    $(CC) $(CFLAGS) -o $(TARGET) $(OBJS)

# Regola per compilare i file .c in .o
%.o: %.c
    $(CC) $(CFLAGS) -c $< -o $@

# Pulizia dei file compilati
clean:
    rm -f $(OBJS) $(TARGET)
```


3.6 Implementazione in Python

Possiamo adesso vedere la stessa logica applicata in C, per Python.

```
1 import threading
2
3 # Funzione per la moltiplicazione
4 def multiply(data):
5     data["result"] = data["a"] * data["b"]
6
7 # Funzione per la somma
8 def add(data):
9     data["result"] = data["a"] + data["b"]
10
11 # Funzione per la sottrazione
12 def subtract(data):
13     data["result"] = data["a"] - data["b"]
14
15
16 # Main
17 def main():
18     # Inizializzazione dei dati
19     data1 = {"a": 2, "b": 6, "result": 0}
20     data2 = {"a": 1, "b": 4, "result": 0}
21     data3 = {"a": 5, "b": 2, "result": 0}
22     data4 = {"a": 0, "b": 0, "result": 0}
23     data5 = {"a": 0, "b": 0, "result": 0}
24
25     # Creazione e avvio dei primi tre thread
26     thread1 = threading.Thread(target=multiply, args=(data1,))
27     thread2 = threading.Thread(target=add, args=(data2,))
28     thread3 = threading.Thread(target=subtract, args=(data3,))
29
30     thread1.start()
31     thread2.start()
32     thread3.start()
33
```

```

34     # Attesa del completamento dei primi tre thread
35     thread1.join()
36     thread2.join()
37     thread3.join()
38
39     data4["a"] = data2["result"]
40     data4["b"] = data3["result"]
41     data5["a"] = data1["result"]
42
43     # Creazione e avvio del quarto thread
44     thread4 = threading.Thread(target=multiply, args=(data4,))
45     thread4.start()
46
47     # Attesa del completamento del quarto thread
48     thread4.join()
49
50     data5["b"] = data4["result"]
51
52     # Creazione e avvio del quinto thread
53     thread5 = threading.Thread(target=add, args=(data5,))
54     thread5.start()
55
56     # Attesa del completamento del quinto thread
57     thread5.join()
58
59     print("Risultato finale:", data5["result"])
60
61 if __name__ == "__main__":
62     main()

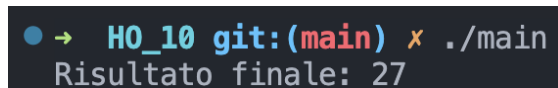
```

4 Presentazione dei risultati

Per compilare il programma è sufficiente aprire un terminale e digitare nella directory contenente il Makefile

```
$ make
```

Al termine del processo avremo un file eseguibile chiamato `main`. Lanciandolo il nostro output sarà:



```
● → HO_10 git:(main) x ./main  
Risultato finale: 27
```

Figure 2: C - Output

Che è il risultato corretto della nostra espressione.

Analogamente, lanciando il programma scritto in Python, otterremo lo stesso risultato:



```
● → HO_10 git:(main) x python3 main.py  
Risultato finale: 27
```

Figure 3: Python - Output

5 Confronto con i processi

Per questa implementazione abbiamo utilizzato i thread, ma è possibile fare la stessa cosa utilizzando i processi? La risposta è sì, è effettivamente possibile fare lo stesso con i processi ma ci sono delle differenze rispetto ai thread.

I thread condividono lo stesso spazio di memoria del processo principale quindi vedono e modificano le stesse variabili globali, le stesse strutture allocate dinamicamente e ogni area dati allocata dal processo. I processi, al contrario, hanno uno spazio di memoria separato: quando un processo genera un nuovo processo figlio (con la funzione `fork()` in C), viene creata una copia indipendente dello spazio di memoria del processo padre. Di conseguenza, qualsiasi

modifica fatta dal processo figlio non sarà visibile al padre. Esistono alcuni meccanismi per condividere dati tra processi come ad esempio le pipe con l'uso della chiamata di sistema `pipe()` oppure altri sistemi di memoria condivisa e socket.

In Python, ad esempio, esiste il modulo `multiprocessing` che consente di creare processi separati in modo simile a quanto avviene con i thread nel modulo `threading`.

6 Conclusioni

In questo hands-on abbiamo visto un grafo di precedenza e come poterlo utilizzare per rappresentare le dipendenze tra le operazioni di un'espressione aritmetica. Questo tipo di rappresentazione ci ha permesso di capire quali operazioni potessero essere svolte in parallelo.

Abbiamo implementato la stessa logica in due linguaggi di programmazione: il linguaggio C ed il linguaggio Python attraverso, rispettivamente, le librerie `pthread.h` e `threading` ed infine abbiamo tirato fuori il risultato corretto da entrambe le implementazioni.

Considerando come possibilità quella di utilizzare i processi, possiamo concludere che questo approccio, seppur effettivamente possibile, richiede una gestione più complessa della comunicazione fra più unità di esecuzione. Al contrario, l'utilizzo dei thread si è rivelato più semplice ed efficace in questo contesto, grazie alla possibilità di accedere direttamente alla stessa memoria condivisa.