PROGRAMMAZIONE CONCORRENTE

Il modello di esecuzione prevede:

- Istruzioni totalmente ordinate
- Un'unica memoria virtuale
- L'esecuzione di una singola istruzione alla volta seguendo l'ordine del programma

Quali sono i vantaggi della programmazione concorrente?

- Sfruttare le prestazioni di architetture multi-processore
- Migliorare la reattività delle interfacce grafiche
- Migliorare design e comprensione dei sistemi operativi

Come funzionano i Thread

Partiamo con alcuni esempi.

Esempio 1:

```
var x = 0;
Thread P1 { x = 500; }
Thread P2 { x = 0; }
Thread P3 { write (x); }
```

In questo programma non possiamo sapere con certezza l'output atteso in quanto dipende dall'ordine con cui andiamo ad eseguire i thread.
Se dovessimo scegliere come ordine: P1, P2, P3 allora l'ouput sarebbe x = 0 .
Se facessimo P2, P1 e P3 invece otteniamo x = 500.

Osservazioni: Questo esempio illustra il **non determinismo** della programmazione concorrente. Diversi ordini di esecuzione dei thread possono portare a risultati diversi.
Inoltre in questo caso abbiamo il fenomeno del race condition, ovvero due (o piu thread) accedono a una variabile condivisa simultaneamente e l'output dipende dall'ordine con cui i thread accedono alla variabile.

Esempio 2:

```
var x=0
Thread P1 {
  while (true)          x = 500;          endwhile
}
```

```
Thread P2 {
  while (true)          x = 0;          endwhile;

}

Thread P3 {
  while (true)        write(x);         endwhile;
}
```

In questo esempio abbiamo un numero infinito di output perche l'ordine e il tempo di esecuzione dei thread non sono deterministici.
Questo rende il programma imprevedibile.

Esempio 3

```
Var x = 100;

Thread P1 {
        x = x + 1;
}

Thread P2 {
        x = x – 1;
}
```

Se i thread **vengono eseguiti in sequenza**, l'output sarà deterministico e il valore finale di x sarà 100. Ad esempio:

Esecuzione di P1 seguita da P2:

P1: x = 100 + 1 => x = 101
P2: x = 101 - 1 => x = 100

Esecuzione di P2 seguita da P1:

P2: x = 100 - 1 => x = 99
P1: x = 99 + 1 => x = 100

Quando i thread vengono eseguiti in modo concorrente, le operazioni sui valori condivisi possono sovrapporsi, portando a risultati non deterministici a causa delle race condition. Ecco alcuni scenari possibili:

Scenario 1: P1 esegue completamente prima di P2:

P1: x = 100 + 1 => x = 101
P2: x = 101 - 1 => x = 100
Risultato: x = 100

Scenario 2: P2 esegue completamente prima di P1:

P2: x = 100 - 1 => x = 99
P1: x = 99 + 1 => x = 100
Risultato: x = 100

Scenario 3: Interruzione tra operazioni di lettura e scrittura:

P1 legge x (100), calcola x + 1 (101) ma non lo scrive ancora.
P2 legge x (100), calcola x - 1 (99) e scrive x = 99.
P1 scrive il risultato calcolato: x = 101.
Risultato: x = 101

Scenario 4: Interruzione simile a Scenario 3, ma con ordine inverso:

P2 legge x (100), calcola x - 1 (99) ma non lo scrive ancora.
P1 legge x (100), calcola x + 1 (101) e scrive x = 101.
P2 scrive il risultato calcolato: x = 99.
Risultato: x = 99

Thread in C++

Iniziamo a include la libreria thread.

Facciamo un esempio di una stampa :

1) Creiamo la nostra funzione:
```
void threadFunction(int id) {
        cout << “thread” << id << “is running”;
```

2) Ora creiamo i nostri thread dentro al main:
```
int main() {
    thread t1(threadFunction, 1);
    thread t2(threadFunction, 2);
```

3) Attendiamo la terminazione dei thread:
   t1.join();
   t2.join();

4) Facciamo la stampa
   cout << "Threads have finished execution\n";

Cosa succede se non uso I join()?

come output otterrei solo la stampa finale.
join() è fondamentale per assicurare che i thread secondari completino il loro lavoro prima che il thread principale continui o termini. Questo garantisce che tutte le operazioni nei thread secondari siano completate correttamente e che le risorse siano gestite appropriatamente.

Altro esempio con una stampa di stringa

```
#include <iostream>
#include <string>
#include <thread>
using namespace std;

void stampa (string messaggio) {
      cout << messaggio;
     }

int main() {
      thread t1(stampa, "hello world");

      t1.join();
}
```