UNIVERSITÀ DEGLI STUDI DI PARMA
Dipartimento di Ingegneria e Architettura
Corso di Laurea Magistrale in Ingegneria Informatica

CORSO DI SISTEMI OPERATIVI E IN TEMPO REALE
Esercitazione *Real-Time* n.1

1 C++ Threads

```
#include <thread>

template< class Function, class... Args >
thread::thread( Function&& f, Args&&... args );

void thread::join();
void thread::detach();
```

Lo standard del linguaggio C++, a partire dalla revisione *ISO/IEC 14882:2011*, definisce un set di API native per avviare più thread *concorrenti* all'interno di un stesso processo. Le funzionalità di linguaggio e di libreria definite dallo standard *C++11* consentono di gestire la creazione, la cancellazione e la sincronizzazione dei singoli thread che vengono utilizzati:

- `class thread`: questa classe rappresenta e controlla una singola trama di esecuzione;
- `thread::thread(...)`: crea un nuovo thread, che viene posto in esecuzione concorrente con il thread chiamante;
- `thread::join()`: sospende l'esecuzione del thread chiamante finchè il thread specificato termina l'esecuzione;
- `thread::detach()`: consente al thread di proseguire l'esecuzione in modo indipendente.

Suggerimento: per avere informazioni sulle funzionalità di libreria del C++ è sufficiente esegure una ricerca sul portale `https://en.cppreference.com/w/` (in alto a dx).

Utilizzo

```
/* funzione eseguita dal nuovo thread */
void my_thread_function(int value)
{
    /* "elaborazione"... (puo' utilizzare "value") */
}

int main()
{
    std::thread my_thread(my_thread_function, 42);
    /* elaborazione... */
    my_thread.join();
    return 0;
}
```

Esercizio 1

Si vuole far diventare multithread questo semplice programma che stampa la stringa *Hello World* dopo una pausa di 1 secondo:

```
void print_hello()
{
    std::this_thread::sleep_for(std::chrono::seconds(1));
    std::cout << "Hello World!" << std::endl;
}

int main()
{
    print_hello();
    return 0;
}
```

Per fare questo, all'interno della funzione `main()`, occorre creare un nuovo thread che dovrà eseguire al proprio interno il codice della funzione `print_hello()`. Dopo che il thread figlio è stato posto in esecuzione, il padre dovrà sospendersi affichè il primo possa concludere la propria elaborazione.

Suggerimento: in ambiente GNU/Linux si può fare uso del compilatore *GCC* per compilare codice C++11, specificando la revisione dello standard con il parametro *-std=c++11*; inoltre, se il codice necessita di manipolare più trame di esecuzione, occorre abilitare le funzionalità multithread specificando l'opzione *-pthread*.

```
g++ -Wall -pthread -std=c++11 -o hello hello.cpp
```

Esercizio 2

Date due matrici quadrate $A = \{a_{i,j}\}$ e $B = \{b_{i,j}\}$ di dimensione $N \times N$, si definisce la matrice prodotto $C = \{c_{i,j}\}$ come segue:

$$c_{i,j} = \sum_{k=1}^{n} a_{i,k} \cdot b_{k,j}$$

Il file `matrix_serial.c` esegue il calcolo del prodotto tra due matrici mediante un unico thread, che calcola iterativamente tutti i singoli valori della matrice C. Partendo dal codice *seriale* si deve realizzare una versione multithread del programma in cui N thread separati calcolano in modo concorrente le singole righe della matrice C.

Suggerimento: il costruttore della classe thread consente di specificare la funzione che verrà invocata dal nuovo thread creato e passare ad essa i valori attuali dei parametri previsti nella sua dichiarazione; il passaggio di parametri per riferimento deve essere reso esplicito all'interno del costruttore utilizzando le funzioni `std::ref()` o `std::cref()`, ad esempio:

```
void thread_function(int a, int b, int & c)
{
    c = a * b;
}

int main()
{
    int res = 0;
    std::thread th(thread_function, 6, 9, std::ref(res));

    std::cout << "What do you get when you multiply six by nine?" << std::endl;
    th.join();
    std::cout << res << std::endl;

    return 0;
}
```

Misurare il tempo

Lo standard C++11 definisce un set di API, incluse nell'header `<chrono>`, che consentono di effettuare *misure di tempo*. Il supporto messo a disposizione dipende dall'hardware e dal sistema operativo, ma può raggiungere una precisione molto elevata, spesso paragonabile a quella ottenibile utilizzando strumenti di basso livello forniti direttamente dall'HW di sistema.

Il nucleo della libreria consiste in alcune classi di tipo *clock*:

- `steady_clock`: rappresenta un clock monotono (*che non può mai decrementare*) con ticks costanti, perciò è generalmente adatto per confrontare istanti temporali nel breve periodo.
- `system_clock`: rappresenta il clock di sistema (*wall clock*), perciò è il corretto riferimento temporale sul lungo periodo ma nel breve potrebbe non essere strettamente monotono, in quanto può subire sporadiche correzioni per effetto di passaggi all'ora legale/solare, eventi di sincronizzazione NTP o impostazioni manuali.
- `high_resolution_clock`: rappresenta per definizione il clock con la granularità più piccola disponibile nel sistema (*clock tick*); può essere un clock indipendente, ma puó anche essere un alias di *steady_clock* o di *system_clock*, in base all'hardware.

La libreria `<chrono>` consente di manipolare dati di tipo `duration`, che rappresentano intervalli di tempo, e dati di tipo `time_point`, che rappresentano invece istanti temporali. Nota: un dato di tipo *time_point* equivale ad un valore di *duration* a partire dal punto di riferimento del rispettivo clock (*clock epoch*).

Pseudo-codice per misurare il tempo di esecuzione di una porzione di codice:

```
auto start = std::chrono::high_resolution_clock::now();
/* elaborazione... */
auto stop = std::chrono::high_resolution_clock::now();

std::chrono::duration<double, std::milli> elapsed(stop - start);
std::cout << "Elapsed [ms]: " << elapsed.count() << std::endl;
```

Esercizio 2bis

Realizzare un algoritmo facendo uso di più thread concorrenti può avere un effetto significativo sul tempo totale che viene richiesto per il suo completamento. Aggiungere delle funzioni che misurino il tempo speso per l'esecuzione del programma precedente che calcola il prodotto tra due matrici (seriale e multithread) e valutare i risultati, possibilmente eseguendo il codice su macchine differenti e/o al variare della dimensione dei dati.

Suggerimento: In ambiente UNIX esistono molti modi per misurare il tempo di esecuzione di un programma: si può esegure il programma preceduto dal comando `time` sulla shell (esempio: `time ls`), oppure si possono utilizzare alcune funzioni di libreria come `clock()` o `gettimeofday()`. Spesso l'hardware fornisce strumenti propri che possono essere utilizzati, direttamente o indirettamente, per eseguire misure di tempo; ad esempio i processori Intel (e compatibili) possiedono istruzioni macchina che permettono di accedere ad alcuni contatori interni ad alta definizione (*RDTSC*): l'uso di questi contatori consente di ottenere misure temporali particolarmente precise, ma pone diversi problemi di compatibilità[1].

In generale, quando i requisiti dell'applicazione e del suo contesto di utilizzo lo consentono é sicuramente preferibile utilizzare strumenti standard messi a disposizione dal linguaggio C++ per realizzare le funzionalità necessarie, perció si suggerisce di utilizzare le API fornite dalla libreria `<chrono>`.

Esercizio 2ter

Modificare il programma precedente, nella versione multithread, in modo tale che ad ogni thread venga assegnata non più una sola riga da elaborare ma un blocco di righe consecutive della matrice risultante e valutare l'impatto sui tempi di esecuzione.

Suggerimento: il metodo statico `hardware_concurrency()` della classe `thread` ritorna il numero di thread concorrenti che sono supportati dall'hardware, perciò può essere utilizzato per decidere il numero di thread da impiegare per ottenere il massimo *speed-up* nell'esecuzione di un algoritmo di calcolo parallelo.

[1] **http://en.wikipedia.org/wiki/Time_Stamp_Counter**