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.2

1 Sincronizzazione

1.1 Mutex

```
#include <mutex>

class mutex;

void mutex::lock();
bool mutex::try_lock();
void mutex::unlock();
```

La modifica di dati condivisi tra due o più thread, per assicurare la correttezza del codice concorrente, deve avvenire all'interno di opportune *regioni critiche* di codice, che siano in grado di garantire l'accesso esclusivo ai dati protetti. Per la gestione delle regioni critiche lo standard C++11 definisce la classe *mutex* a supporto dell'omonimo meccanismo elementare di *mutua esclusione*, che espone i seguenti metodi:

- `mutex::lock()`: acquisisce il mutex, sospendendo il thread chiamante se il mutex è già posseduto da un altro thread;
- `mutex::try_lock()`: se nessun altro thread possiede il mutex, lo acquisisce e ritorna *true*, altrimenti ritorna *false* senza interrompere la trama di esecuzione;
- `mutex::unlock()`: rilascia il mutex, consentendo ad altri thread di prenderne possesso.

Nota: I metodi della classe mutex hanno importanti *precondizioni* che devono essere rispettate durante il loro utilizzo: i metodi `lock()` e `try_lock()` possono essere invocati solo se il thread corrente *non possiede* già il mutex ed il metodo `unlock()` può essere invocato solo quando il thread corrente *possiede* il mutex. Se queste precondizioni non vengono rispettate il comportamento del codice è *indefinito*.

Esercizio 1

Il programma C++ contenuto nel file `simple_wrong.cpp` crea M thread figli che in modo *concorrente* incrementano per N volte consecutive il valore di una variabile globale `count`. Il valore finale della variabile dovrebbe essere allora $N \times M$, ma se si esegue il programma il valore prodotto è in genere minore e spesso cambia. Perchè? Trovare e correggere l'errore, ricordando le ultime API che sono state introdotte.

Nota: Si dovrebbe riuscire a notare che provando ad eseguire il programma più volte, su macchine differenti e/o su differenti architetture si ottengono risultati *differenti*.

1.2 Locks

```
#include <mutex>

template<class mutex_type>
class unique_lock;

unique_lock::unique_lock(mutex_type& m);
unique_lock::~unique_lock();
unique_lock::lock();
unique_lock::unlock();
```

Per supportare un stile di codifica che renda più difficile commettere errori nell'utilizzo dei mutex, lo standard C++11 introduce alcune classi aggiuntive che permettono di manipolare in modo automatico le operazioni di lock/unlock sugli oggetti di tipo mutex, secondo un idioma di programmazione di tipo *RAII*[1].

La classe template `unique_lock`, ad esempio, è un *wrapper* generico che consente di rendere automatica l'esecuzione del metodo unlock() del mutex associato, invocandolo se necessario nel proprio distruttore:

- `unique_lock(mutex_type& m)`: *prende il possesso* del mutex ed esegue `m.lock()`;
- `~unique_lock()`: se *possiede* il mutex, esegue `m.unlock()`;

Un oggetto di tipo `unique_lock` possiede comunque metodi che permettono di manipolare in modo esplicito il mutex ad esso associato, ad esempio:

- `unique_lock::lock()`: *prende il possesso* del mutex ed esegue `m.lock()`;
- `unique_lock::unlock()`: *lascia il possesso* del mutex ed esegue `m.unlock()`.

Esempio:

Questi due codici sono equivalenti:

```
int a = 6, b = 9;
mutex m;

int foo()
{
m.lock();
int res = a * b;
m.unlock();
return res;
}
```

```
int a = 6, b = 9;
mutex m;

int foo()
{
unique_lock<mutex> l(m);
return a * b;
}
```

Esercizio 1bis

Convertire il codice dell'esercizio precedente affinché la regione critica sia *delimitata* mediante l'uso di un oggetto di tipo `unique_lock<mutex>`.

[1] **https://en.wikipedia.org/wiki/Resource_acquisition_is_initialization**

1.3 Condition variables

```
#include <condition_variable>

class condition_variable;

void condition_variable::notify_one();s
void condition_variable::notify_all();

void condition_variable::wait(unique_lock<mutex> & lock);
cv_status condition_variable::wait_for(unique_lock<mutex> & lock,
    chrono::duration<R,P> rel_time);
cv_status condition_variable::wait_until(unique_lock<mutex> & lock,
    chrono::time_point<R,P> abs_time);
```

Se il mutex consente di condividere un insieme di dati tra più thread, la variabile condizione permette di *condividere l'informazione che questi dati sono cambiati*: in molti casi infatti un thread che ha accesso ad un insieme di dati condivisi può effettuare elaborazioni su di essi soltanto dopo che un secondo thread ne ha cambiato lo stato, perciò è utile avere a disposizione un meccanismo esplicito che consenta al primo thread di sospendersi in attesa che il secondo gli *notifichi* la possibilità di proseguire l'elaborazione.

- `notify_one()`: pone in esecuzione (al più) uno dei threads che sono in attesa sulla variabile condizione.
- `notify_all()`: pone in esecuzione tutti i threads che sono in attesa sulla variabile condizione.
- `wait()`: rilascia il lock ed *atomicamente* pone il thread in attesa di notifica; quando il thread viene notificato, il metodo esce dopo avere acquisito di nuovo il lock;
- `wait_for()`: equivale a `wait()`, però il parametro `rel_time` consente di specificare un *intervallo di tempo* oltre il quale la funzione esce comunque, con valore di ritorno `cv_status::timeout`, anche se la variabile condizione non è stata notificata;
- `wait_until()`: equivale a `wait()`, però il parametro `abs_time` consente di specificare un *istante temporale* (nel futuro) al quale la funzione esce comunque, con valore di ritorno `cv_status::timeout`, anche se la variabile condizione non è stata notificata.

Esempio:

Thread A:

```
{
    unique_lock<mutex> lock(mtx);

    while ( ! <condition> )
        cond.wait(lock);

    /* elaborazione... */
}
```

Thread B:

```
{
    unique_lock<mutex> lock(mtx);

    /* elaborazione... */

    if ( <condition> )
      cond.notify_one();
}
```

Esercizio 2

I filosofi a cena:

> Cinque filosofi sono seduti attorno a un tavolo circolare, ogni filosofo ha un piatto di spaghetti tanto scivolosi che necessitano di due forchette per poter essere mangiati, sul tavolo vi sono in totale cinque forchette. La vita di ogni filosofo è scandita da due momenti distinti, un momento in cui esso pensa, ed un momento in cui esso mangia. Quando un filosofo ha fame, prende la forchetta che sta a sinistra e quella che sta a destra del suo piatto, mangia per un po' e poi mette sul tavolo le due forchette e ricomincia a pensare...

La directory `filosofi` contiene alcuni file che rappresentano la soluzione al problema dei cinque filosofi realizzata in linguaggio C++. I cinque thread (i filosofi) mangiano e pensano in concorrenza tra loro, quindi occorre realizzare un monitor per gestire in modo corretto la loro sincronizzazione nell'uso delle risorse condivise (le forchette). Il file `monitor.h` definisce l'interfaccia del monitor proposto, che dovrà essere implementata facendo uso di mutex e variabili condizione nel file `monitor.cpp`.

Suggerimento: È possibile risolvere il problema in molti modi diversi, ad esempio:

A) Ogni filosofo, quando vuole iniziare a mangiare (*pickup*) attende (in una coda *comune*) che le sue forchette siano libere, cioè che i filosofi alla sua destra e sinistra non stiano mangiando a loro volta. Quando ripone le forchette (*putdown*) segnala *a tutta la coda* che ci sono nuove fochette disponibili.

B) Ogni filosofo, quando vuole iniziare a mangiare (*pickup*) attende (in una coda *personale*) che le sue forchette siano libere. Quando ripone le forchette (*putdown*) controlla lo stato dei filosofi che gli stanno a fianco ed eventualmente (se possono farlo) segnala *loro* che possono iniziare a mangiare.

Esercizio 3

Monitor lettori/scrittori:

La directory `lettori-scrittori` contiene alcuni file che rappresentano una applicazione di esempio in cui più thread accedono in modo concorrente ad una *ipotetica* risorsa condivisa, alcuni esclusivamente per leggere ed alcuni esclusivamente per scrivere in essa. Le fasi di lettura e di scrittura in realtà sono simulate nel codice eseguito dai thread con un'attesa di qualche secondo.

Occorre realizzare i metodi della classe definita in `monitor.h` in modo che consentano ai thread di operare in modo corretto sulla risorsa.

Si ricorda che il problema classico dei lettori/scrittori può essere risolto secondo quattro differenti politiche:

- **strong writer preference**: un lettore, quando arriva, attende se la regione critica è occupata da uno scrittore o se c'è almeno uno scrittore in coda. Quando la regione critica viene liberata (da uno scrittore o dall'ultimo lettore), se ci sono scrittori in coda, viene risvegliato per primo uno di essi.
- **weak writer preference**: un lettore, quando arriva, attende se la regione critica è occupata da uno scrittore o se c'è almeno uno scrittore in coda. Quando la regione critica viene liberata non è specificato chi viene risvegliato per primo.
- **strong reader preference**: un lettore, quando arriva, attende se la regione critica è occupata da uno scrittore ma non attende se ci sono scrittori in coda. Quando la regione critica viene liberata (da uno scrittore), se ci sono lettori in coda, vengono risvegliati per primi.
- **weak reader preference**: un lettore, quando arriva, attende se la regione critica è occupata da uno scrittore ma non attende se ci sono scrittori in coda. Quando la regione critica viene liberata non è specificato chi viene risvegliato per primo.

Scegliete la soluzione che preferite per svolgere l'esercizio, oppure provate le diverse modalità e verificatene le differenze nel comportamento.

Suggerimento: La *preference* a lettori o scrittori viene decisa realizzando in modo appropriato le condizioni di attesa e risveglio all'interno del monitor. La costruzione di un monitor *strong* richiede l'uso di due code separate per lettori e scrittori mentre per un monitor *weak* è sufficiente una sola coda.