UNIVERSITÀ
DI PARMA

thread

Alberto Ferrari

definizione

- un ***thread*** è un flusso di istruzioni, all'interno di un processo, che lo scheduler può fare eseguire parallelamente e concorrentemente con il resto del processo
- un thread può essere pensato come una ***procedura*** che lavora in ***parallelo*** con altre procedure
- in un processo possono ***coesistere*** più thread
- i thread devono essere ***creati esplicitamente***
- quando il ***processo termina*** tutti i suoi ***thread terminano*** forzatamente
- durante la vita di un processo i thread vengono attivati e terminati dal programmatore

thread

- un ***processo*** ha al suo interno ***almeno un thread*** di esecuzione
- se il flusso esecutivo di un processo viene scomposto in ***più flussi concorrenti*** il processo ha al suo interno ***più thread***
- i thread di uno stesso processo ***condividono*** l'area ***dati*** e codice
  - è necessaria una ***sincronizzazione*** nell'accesso ai dati globali
- lo scambio di contesto (***contest switch***) fra thread è più ***veloce*** di quello tra processi

ciclo di vita

- il sistema operativo gestisce i thread applicando le politiche di ***scheduling*** dei processi
- un thread può essere in stato di
  - ***running***
    - in esecuzione
  - ***wait***
    - in attesa del verificarsi di una condizione
  - ***sleep***
    - in attesa dell'esecuzione
  - ***stopped***
    - ha concluso la sua esecuzione e confluisce con il thread che lo ha originato

visibilità e condivisione variabili

- le variabili allocate nello ***stack*** sono ***locali*** ai thread
  - i thread non condividono lo stack
  - le variabili ***locali*** ad un metodo sono locali ai thread
- le variabili allocate nello ***heap*** sono ***condivise*** dai thread di uno stesso processo
  - le variabili ***globali*** sono condivise da tutti i thread
  - ***attributi*** statici o di istanza della classe

multi-process vs multi-thread

vantaggi thread

- tutti i thread di un processo ***condividono*** lo stesso ***spazio di indirizzamento***
  - la ***comunicazione*** tra thread è ***più semplice*** della comunicazione tra processi
- velocità context switch
  - viene mantenuta buona parte dell'ambiente di lavoro

svantaggi thread

- ***concorrenza*** invece di parallelismo
  - necessario gestire la mutua esclusione delle risorse comuni

c++11 thread

- fino alla versione c++11 lo standard non prevedeva la gestione dei thread
  - varie implementazioni non standard e spesso non portabili
- c++11 propone la ***classe thread*** nell'header thread
  - standardizzazione rispetto alle varie soluzioni precedenti
  - http://en.cppreference.com/w/cpp/thread/thread
  - **#include <thread>**

creazione di un thread (1)

function pointer

- il thread ***t*** viene creato associando ad esso la funzione che deve eseguire
- il thread t ***esegue*** il codice associato alla funzione (*hello*)
- t.***join()***
  - il metodo join blocca il thread corrente fino a quando il thread t ha terminato la sua esecuzione

```
#include <iostream>    // std::cout
#include <thread>      // std::thread

void hello() {
  std::cout<<"Hello Concurrent World\n";
}

int main() {
  std::thread t(hello);
  t.join();
}
```

creazione di un thread (2)
function objects

- la classe Dummy ha ridefinito l'***operatore ()*** (*operatore di chiamata*)
- il thread ***t*** viene creato associando ad esso un ***oggetto*** della classe Dummy
- il thread t esegue il ***codice*** associato all'***operatore ()*** di Dummy
- viene passata una ***copia*** dell'oggetto

```
class Dummy {
  public :
    void operator()() {
      std::cout<<"Hello Concurrent World";
    }
};

int main() {
 Dummy w;
 std::thread t(w);
 t.join();
}
```

passaggio parametri (valore)

- è possibile passare parametri alla funzione eseguita dal thread
- la lista dei parametri deve seguire il nome della funzione
- **`thread t(t_f,"hello"3)`**
  - il thread ***t*** viene creato ed eseguirà la funzione ***t_f*** con i ***valori "hello"*** e ***3*** passati come parametro

```
#include <iostream>
#include <thread>

void t_f(std::string s,int n) {
 for(int i=0;i<n;i++)
   std::cout<< s << " " << i << std::endl;
}

int main() {
  std::thread t(t_f,"hello",3);
  t.join();
  return 0;
}
```

passaggio parametri (riferimento)

- nel caso di passaggio per riferimento è necessario specificarlo al momento del passaggio del parametro mediante appropriata sintassi
- **`thread t(t_f,std::ref(a))`**
  - il thread ***t*** viene creato ed eseguirà la funzione ***t_f*** con il ***parametro attuale a*** passato per riferimento

```
#include <iostream>
#include <thread>

void t_f(int &f) {
  f++;
}

int main() {
  int a = 5;
  std::thread t(t_f,std::ref(a));
  t.join();
  std::cout << "a = " << a;
  return 0;
}
```

passaggio oggetto (riferimento)

- la sintassi std::ref permette anche il passaggio di oggetti per riferimento
- **`thread t(std::ref(w))`**
  - non viene passata una copia dell'oggetto w ma un riferimento a questo

```
class Dummy {
  public:
    Dummy(): val(0){}
    int getVal() { return val; }
    void operator()() {
        cout<<"thread: value is " << ++val;
    }
  private:
    int val;
};
int main() {
 Dummy w;
 std::thread t(std::ref(w));
 t.join();
 cout<<"process: value is "<<w.getVal();
}
```

thread id

- all'interno del thread si accede al suo id tramite
  `std::this_thread::get_id()`
- dal thread principale è possible ottenere l'id di un thread mediante il metodo get_id() richiamato sul thread
  `t.get_id()`

join

- il metodo ***join*** fa sì che il thread padre attenda la terminazione del thread figlio
- ***t.join()***
  - è un metodo che termina quando il thread t è ***terminato***
  - ***blocca l'esecuzione*** del metodo che ha chiamato t.join()

detach

- il metodo ***detach*** fa passare l'esecuzione del thread in ***background***
  - ***daemon***
- il processo ***non attende*** la terminazione del thread
- il thread ***non*** è più «***joinable***»
- al termine dell'esecuzione del processo il thread termina

detach

```
void proc_thread_1(){
        for(int n=0;n<10;n++){
                std::this_thread::sleep_for(std::chrono::milliseconds(300));
                cout <<  "thread 1: " << to_string(n) << endl;
        }
}
void proc_thread_2(){
        for(int n=0;n<20;n++){
                std::this_thread::sleep_for(std::chrono::milliseconds(1000));
                cout <<  "thread 2: " << to_string(n) << endl;
        }
}
int main()
{
        thread t1(proc_thread_1);
        thread t2(proc_thread_2);
        t1.join();
        t2.detach();
        char f; cin >> f;
}
```

thread id

```
void thread_function(int n){
  std::this_thread::sleep_for(std::chrono::milliseconds(1000*n));
  std::cout<<"Inside Thread :: ID  = "<<std::this_thread::get_id()<<std::endl;
}
int main() {
    std::thread threadObj1(thread_function,2);
    std::thread threadObj2(thread_function,5);
    if(threadObj1.get_id() != threadObj2.get_id())
        std::cout<<"Both Threads have different IDs \n";
    std::cout<<"From Main Thread :: ID of Thread 1 = "<<threadObj1.get_id()<<std::endl;
    std::cout<<"From Main Thread :: ID of Thread 2 = "<<threadObj2.get_id()<<std::endl;
    threadObj1.join();
    threadObj2.join();
    return 0;
}
```

```
Both Threads have different IDs
From Main Thread :: ID of Thread 1 = 2
From Main Thread :: ID of Thread 2 = 3
Inside Thread :: ID  = 2
Inside Thread :: ID  = 3
```

thread id

- ***main_id*** è globale ed è l'identificatore del thread principale
- ***this_thread::get_id()*** restituisce l'identificatore del thread in esecuzione
- la funzione ***is_main_thread()*** è chiamata sia dal thread principale sia dal thread figlio (th)

```
thread::id main_id = this_thread::get_id();

void is_main_thread() {
  if ( main_id == this_thread::get_id() )
    cout << "This is the main thread" << endl;
  else
    cout << "This is not the main thread"<<endl;
}

int main() {
  is_main_thread();
  thread th (is_main_thread);
  th.join();
  return 0;
}
```

array di thread

- ***thread t[num_threads]*** è la dichiarazione di un array di thread
- ***t[i] = thread(call_from_thread)*** tutti i thread sono associati alla stessa funzione
- ***t[i].join()*** il processo attende la terminazione di tutti i thread

```
Launched by thread Launched by thread Launched from the main
Launched by thread 5
2
4
Launched by thread 9Launched by thread 10
Launched by thread 7

Launched by thread 11
Launched by thread 3
Launched by thread 8
Launched by thread 6
```

```
static const int num_threads = 10;
void call_from_thread() {
  cout << "Launched by thread ";
  cout << std::this_thread::get_id() << endl;
}
int main() {
  thread t[num_threads];
  //Launch a group of threads
  for (int i = 0; i < num_threads; ++i) {
     t[i] = thread(call_from_thread);
  }
  cout << "Launched from the main" << endl;
  //Join the threads with the main thread
  for (int i = 0; i < num_threads; ++i) {
     t[i].join();
  }
  return 0;
}
```

thread

condivisione di risorse e mutua esclusione

accesso a risorsa condivisa

- esempio ricerca sequenziale in un array
  - un primo thread ricerca nella prima parte dell'array mentre un altro cerca nella seconda parte
  - ricerca del valore x nell'array v di n elementi:
    - ***thread t1(cerca, v, 0, n/2, x);***
    - ***thread t2(cerca, v, n/2, n, x);***
  - un thread interrompe la ricerca quando ha trovato x (l'altro prosegue)
  - è possibile fare in modo che anche l'altro thread termini
    - variabile comune (trovato)
    - ***thread t1(cerca, v, 0, n/2, x, ref(trovato));***
    - ***thread t2(cerca, v, n/2, n, x, ref(trovato));***

esempio

- ***ricerca*** el in v da start a end
- ***trovato*** può essere modificato anche da ***altri thread***

```
void cerca(int v[], int start, int end,
           int el, bool &trovato){
  string sout;
  int i=start;
  while (i<end && !trovato) {
   sout = «contr." + to_string(i) + "\n";
   cout << sout;
   if (v[i]==el) {
     sout="trov. pos "+to_string(i)+"\n";
     cout << sout;
     trovato = true;
   }
   i++;
  }
}
```

accesso concorrente

- l'***accesso concorrente*** ai dati (in generale alle risorse) può provocare situazioni di
  - ***incoerenza***
    - il risultato finale può dipendere dalla sequenza con cui i processi vengono eseguiti
  - ***starvation***
    - attesa indefinita (*inedia*) impossibilità perpetua, da parte di un processo pronto per l'esecuzione, di ottenere le risorse di cui necessita per essere eseguito
  - ***deadlock***
    - due o più processi si bloccano a vicenda aspettando che uno esegua una certa azione che serve all'altro e viceversa
- la parte del processo in cui si opera con risorse condivise è definita ***sezione critica***

sezione critica

- è importante ***individuare*** e ***minimizzare*** la sezione critica
- gestione della sezione critica:
  - ***mutua esclusione***
    - un solo processo deve trovarsi nella sezione critica
  - ***attesa indefinita***
    - evitare che un processo rimanga perennemente in attesa di accedere a una sezione critica
  - ***controllo***
    - un processo che si trova fuori dalla sezione critica non deve interferire con l'accesso a questa da parte di altri processi

corsa critica

- ***race condition*** (corsa critica)
- in un sistema basato su ***processi paralleli***, il ***risultato*** finale dell'esecuzione ***dipende*** dalla ***sequenza*** con cui i processi vengono eseguiti
- soluzione alla corsa critica:
  - algoritmi che prevedono la ***mutua esclusione***
    - se la risorsa condivisa è occupata da un processo nessun altro processo potrà accedervi
- se i processi condividono la risorsa unicamente in modalità di lettura non c'è problema di race condition
  - i processi non possono modificare lo stato della risorsa

corsa critica: esempio

- per testare la corsa critica inserire un eventuale ritardo casuale prima del decremento

```
int dato;       // global
void modifica(){
  if (dato>0) {
    dato--;
  }
}
int main() {
        dato = 1;
        thread t1(modifica);
        thread t2(modifica);
        t1.join();
        t2.join();
        cout << "dato: " << dato << endl;
        return 0;
};
```

race condition - esempio

- due thread devono decrementare il valore della variabile globale ***dato*** solo se questa ha valore positivo
- entrambe i thread eseguono il codice ***if(dato>0) dato--;***
- in base alla sequenza di esecuzione dei due thread, la variabile dato può assumere ***valori diversi***

	thread 1	thread 2
dato = 1		
dato = 1	if(dato>0)	
dato = 0	dato--;	
dato = 0		if(dato>0)
dato = 0		

	thread 1	thread 2
dato = 1		
dato = 1	if(dato>0)	
dato = 1		if(dato>0)
dato = 0	dato--;	
dato = -1		dato--;
dato = -1		

race condition su risorsa


```
void stampa(string testo){
 for(int a=0;a<testo.length();a++){
   cout << testo[a]; flush(cout);
   … ritardo …
 }
}

class Repl {
public :
 string testo;
 Replicante(string t){ testo=t; }
 void operator()(){ stampa(testo); }
};
```

```
int main()
{
 thread t1(Repl ("Io ne ho viste cose"));
 thread t2(Repl ("I've seen things"));
 t1.join();
 t2.join();
 return 0;
}
```

mutex

- ***mutex*** (***mut***ual ***ex***clusion)
  - procedimento di sincronizzazione fra processi o ***thread concorrenti***
  - ***impedisce*** che più task paralleli accedano ***contemporaneamente*** a ***dati*** o a ***risorse*** soggette a ***race condition*** (corsa critica)
- una ***variabile mutex*** serve per la ***protezione*** delle ***sezioni critiche***:
  - variabili condivise modificate da più thread
- solo un thread alla volta può accedere ad una risorsa protetta da un mutex
- tipi di mutex
  - ***semplice*** (**std::mutex**) è un ***semaforo binario***
  - semplice con ***timeout*** (**std::timed_mutex**) definisce un ***tempo massimo di attesa***
  - ***ricorsivo*** (**std::recursive_mutex**) permette ***più accessi*** prima di essere occupato
  - ricorsivo con timeout (**std::recursive_timed_mutex**)

mutex: utilizzo

- il primo thread che ha accesso alla coda dei lavori blocca l'accesso agli altri thread
- ***std::mutex mtx;*** // mutex for critical section
- ***mtx.lock();*** // lock mutex
- quando ha portato a termine il suo compito sblocca l'accesso
- ***mtx.unlock();*** // unlock mutex

corsa critica: mutex

- la funzione opera sul ***mutex m*** passato per reference
- ***m.lock()*** impedisce agli altri thread di operare su m
- ***m.unlock()*** sblocca il mutex
- in questo caso il risultato è ***sempre*** 0

```
int dato;       // global
void modifica(mutex &m){
  m.lock();
  if (dato>0) {
    dato--;
  }
  m.unlock();
}
int main() {
        dato = 1;mutex mtx;
        thread t1(modifica, ref(mtx));
        thread t2(modifica, ref(mtx));
        t1.join();
        t2.join();
        cout << "dato: " << dato << endl;
};
```

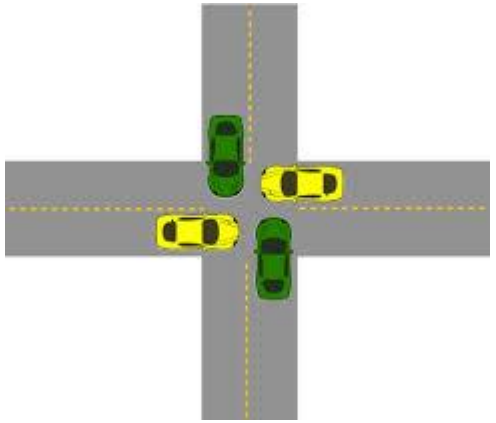
mutex: eccezioni

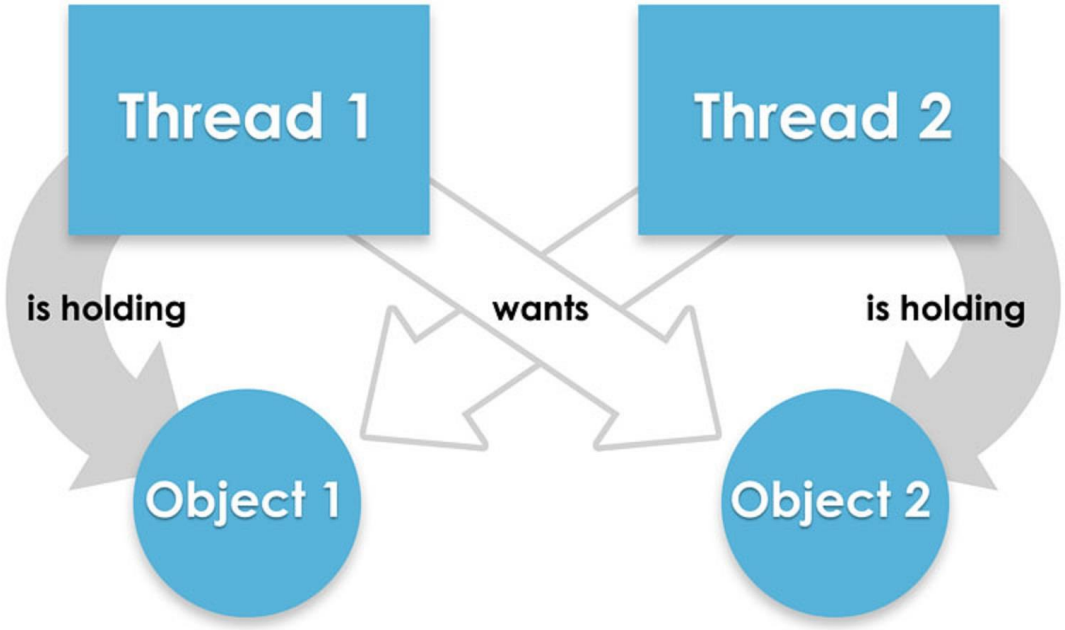
- il mutex ***non gestisce eccezioni***
- se il thread genera un errore all'interno della sezione critica il thread termina ma il ***mutex non viene sbloccato***
- una soluzione è quella di racchiudere la sezione critica in un costrutto ***try catch*** ed effettuare l'operazione di unlock nella sezione catch
- una soluzione migliore è quella di utilizzare un ***lock_guard*** che automaticamente sblocca il mutex al termine della funzione

```
void gestioneThread(mutex &m){
   lock_guard<mutex> lock(m);
   …
}
```

deadlock

- lo stallo (***deadlock***) si verifica quando due o più processi ***si bloccano a vicenda*** aspettando che uno esegua una certa azione che serve all'altro e viceversa





deadlock

```
void f_th1(mutex &a,mutex &b) {
        a.lock();
        b.lock();
        b.unlock();
        a.unlock();
}
void f_th2(mutex &a,mutex &b) {
        b.lock();
        a.lock();
        a.unlock();
        b.unlock();
}
```

```
int main() {
 mutex mtx1,mtx2;
 thread t1(f_th1,ref(mtx1),ref(mtx2));
 thread t2(f_th2,ref(mtx1),ref(mtx2))
 t1.join();
 t2.join();
 cout << "end program\n";
 return 0;
}
```

timed_mutex

- il timed_mutex permette di evitare l'attesa indefinita su un mutex
- si definisce il tempo massimo di attesa try_lock_for(time)
- poi si può verificare se il mutex è accessibile per il thread o se risulta ancora locked
- ☹ problemi derivanti dal fatto di essere supportato o meno dai vari compilatori
- per testare l'esempio seguente utilizzare il compilatore/esecutore online http://en.cppreference.com/w/cpp/thread/timed_mutex/try_lock_for

soluzione (parziale) al deadlock


```
void f_th1(timed_mutex &a,timed_mutex &b){
  if (a.try_lock_for(_time)) {
    cout << "f_th1: a locked\n";
    this_thread::sleep_for(_time);
    if (b.try_lock_for(_time)) {
      cout << "f_th1: b locked\n";
      this_thread::sleep_for(_time));
      b.unlock();
      a.unlock();
      cout << "f_th1: terminated\n";
    } else {
      a.unlock();
      cout << "f_th1: b timout\n";
    }
  } else {
    cout << "f_th1: a timout\n";
  }
}
```

```
// f_th2 analogo con a e b invertiti
int main() {
 timed_mutex mtx1,mtx2;
 thread t1(f_th1,ref(mtx1),ref(mtx2))
 thread t2(f_th2,ref(mtx1),ref(mtx2))
 t1.join();
 t2.join();
 cout << "end program\n";
 return 0;
}
/* possible output:
 * f_th1: a locked
 * f_th2: b locked
 * f_th1: b timout
 * f_th2: a locked
 * f_th2: terminated
 * end program
*/
```

prenotazione multipla

- una soluzione al problema di deadlock precedente è quella di utilizzare unique_lock<mutex> che permette di prenotare il lock di un mutex senza bloccarlo
- nell'esempio vengono prenotati il mutex a e b e solo nel momento in cui entrambi sono disponibili vengono bloccati

```
void f_th1(mutex &a,mutex &b) {
        unique_lock<mutex> lock_a(a, defer_lock);
        unique_lock<mutex> lock_b(b, defer_lock);
        lock(lock_a,lock_b);
        cout << "f_th1: terminated\n";
}
```

soluzione al deadlock: prenotazione

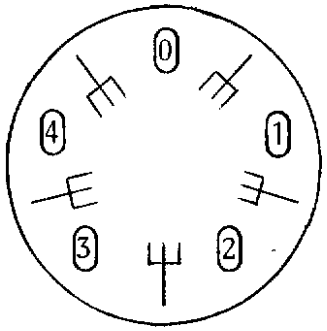

```
void f_th1(mutex &a,mutex &b) {
 unique_lock<mutex> lock_a(a, defer_lock);
 unique_lock<mutex> lock_b(b, defer_lock);
 lock(lock_a,lock_b);
 cout << "f_th1: terminated\n";
}

void f_th2(mutex &a,mutex &b) {
 unique_lock<mutex> lock_b(b, defer_lock);
 unique_lock<mutex> lock_a(a, defer_lock);
 lock(lock_a,lock_b);
 cout << "f_th2: terminated\n";
}
```

```
int main() {
 mutex mtx1,mtx2;
 thread t1(f_th1,ref(mtx1),ref(mtx2));
 thread t2(f_th2,ref(mtx1),ref(mtx2));
 t1.join();
 t2.join();
 cout << "end program\n";
 return 0;
}
```

```
f_th2: terminated
f_th1: terminated
end program
```

il problema dei 5 filosofi

- Dijkstra 1965: ***cinque filosofi*** siedono ad una tavola rotonda con un piatto di *spaghetti* davanti, una *forchetta* a *destra* e una *forchetta* a *sinistra*
  - ci sono 5 filosofi, 5 piatti di spaghetti e 5 forchette
- la vita di un filosofo consiste di periodi alterni di ***mangiare*** e ***pensare***
- ciascun filosofo ha bisogno di ***due forchette*** per mangiare ma ne prende una per volta
- dopo essere riuscito a prendere due forchette il filosofo mangia per un po', poi lascia le forchette e ricomincia a pensare
- si vuole sviluppare di un algoritmo che ***impedisca***
  - lo stallo (***deadlock***)
    - ciascuno dei filosofi tiene in mano una forchetta senza mai riuscire a prendere l'altra
  - la morte d'inedia (***starvation***)
    - uno dei filosofi non riesce mai a prendere entrambe le forchette



la classe Filosofo


```
class Filosofo {
private:
  int id;       //identificatore
  mutex *destra;       //forchetta destra
  mutex *sinistra;     //forchetta sinistra
  string pensiero; //pensiero del filosofo
public:
Filosofo(int n,mutex *d,mutex *s,string p)
: id(n),destra(d),sinistra(s),pensiero(p)
{}
void operator()() {
  for(int a=0; a<3; a++) {
    pensa();
    mangia();
  }
}
```

```
 void pensa() {
   cout << pensiero;
   <sleep>
}

void mangia() {
 unique_lock<std::mutex> lock_a(*destra,
                                std::defer_lock);
 unique_lock<std::mutex> lock_b(*sinistra,
                                std::defer_lock);
 lock(lock_a,lock_b);
 cout << "messaggio";
 <sleep>
 cout << "ho finito di mangiare";
}
```

soluzione al problema dei 5 filosofi


```
int main() {
 string pensiero[5]= { "alla liberta
 … …};
 Filosofo* filosofi[5];
 mutex forchette[5];
 thread* threads[5];
 for(int f=0; f<5; f++) {
   filosofi[f]=new Filosofo(f,
        &forchette[f],
         &forchette[(f+1)%5],pensiero[f]);
   threads[f]= new thread(*filosofi[f]);
 }
 for(int f=0; f<5; f++) {
   threads[f]->join();
 }
}
```

- al ***filosofo f*** vengono assegnate le ***forchette f*** e ***f+1*** che sono rappresentate da ***mutex***
- ogni ***filosofo*** è rappresentato da un ***thread*** che opera in concorrenza sugli altri mediante i mutex
- il processo ***termina*** quando sono terminati tutti i thread (*ogni filosofo ha pensato e mangiato per 3 volte*)

code di attesa

- la gestione delle ***code di attesa*** permette di ***sospendere*** uno o più ***thread*** fino al verificarsi di un certo ***evento*** (*o lo scadere di un timeout*)
- per gestire le code di attesa la STL mette a disposizione le ***condition_variable***
- gli oggetti condition_variable mettono a disposizione tre metodi fondamentali:
  - ***wait()*** il thread si pone in ***attesa*** accodandosi alla condition
  - ***notify_one()*** viene «risvegliato» un thread in coda alla condition
  - ***notify_all()*** vengono risvegliati tutti i thread in attesa sulla coda

condition e mutex

- a una condition è sempre associato un ***mutex*** per evitare race condition
  - esempio: thread che esegue una wait mentre un altro thread esegue un notify
  - notify potrebbe avvenire contemporaneamente alla wait e venire quindi persa creando un deadlock
- qualche ***problema*** di implementazione:
  - non è sempre garantito che l'***ordine*** in cui i thread vengono riattivati sia uguale a quello in cui si sono inseriti in coda (≅ *FIFO*)
  - può succedere che ***più di un*** thread venga risvegliato
    - opportuno verificare nuovamente che la condizione per cui sono in attesa sia soddisfatta

esempio

```
class Esecutore {
public :
 int num_th; // numero del thread
 condition_variable *coda; //coda attesa
 mutex *m;  // mutex per la coda
 Esecutore(int n,
    condition_variable *cond, mutex *mtx)
 { coda=cond; num_th=n; m=mtx; }
 void operator()(){
   <in attesa>
   unique_lock<std::mutex> lk(*m);
   coda->wait(lk);
   <risvegliato>
 }
};
```

```
int main() {
 condition_variable *coda;
 coda = new condition_variable();
 mutex *m= new mutex();
 …
 thread* threads[n_th];
 for(t=0; t<n_th; t++){
  threads[t]= new
            thread(Esecutore(t,coda,m));
 }
 for(t=0; t<n_th-3; t++){
   coda->notify_one();
 }
 coda->notify_all();
 for(t=0; t<n_th; t++) threads[t]->join();
}
```

c++11 thread riferimenti

- Williams, Anthony, C++ concurrency in action : practical multithreading, Shelter Island, NY : Manning Publ., 2012. - 528 p.
- http://www.cplusplus.com/reference/thread/thread/