Concorrenza

- La concorrenza può essere ottenuta via
  - Multiprocessing: l'esecuzione di un programma è divisa tra più processi
    - Processo: programma in esecuzione
  - Multithreading: un singolo processo che gestisce più thread
    - Thread: flusso di esecuzione all'interno di un processo
- Task: blocco di codice eseguibile in concorrenza con altro codice
  - Funzione, function object, lambda
- IPC - Interprocess Communication
  - Non è standardizzata in C++, si può usare Boost.Interprocess

Thread

- Ogni programma C++ ha almeno un thread, quello che esegue main()
- Da C++11 la classe std::**thread** permette di gestire l'esecuzione un task
  - In alternativa è possibile usare boost::thread (linker: boost_system, boost_thread)
- Un thread figlio entra in esecuzione in seguito alla sua creazione
  - Termina quando raggiunge la fine del task: diventa *joinable*
- Il main thread dovrebbe attendere la terminazione del thread figlio via **join**()
  - In alternativa, con le dovute cautele, si può invocare detach()
- Una risorsa acceduta da più thread dovrebbe avere un accesso regolamentato
- Passaggio di argomenti a un task
  - Nel caso di funzioni si usa un costruttore variadico template
    - È richiesto l'uso di **ref**() / **cref**() per la corretta gestione delle reference

Sincronizzazione

- Se più thread accedono le stesse locazioni di memoria in lettura/scrittura
  - La scrittura da parte di un thread deve essere sincronizzata
  - Altrimenti il comportamento del programma è indefinito
- Nel caso più semplice anche un **join**() può bastare
  - Esempio, una variabile visibile da main thread e figlio
    - Il thread figlio modifica la variabile
    - Il main thread attende la fine dell'esecuzione del figlio
    - A questo punto il main thread può accedere in sicurezza la variabile
- Si parla di **race condition** quando più thread competono su un dato
  - Può essere benigna, ma spesso è fonte di problemi
- Soluzione via lock-free programming, vedi classe template std::**atomic**
  - Basata sull'uso di variabili accessibili in maniera atomica, inerentemente protette

Mutex

- Il mutex permette di regolamentare l'accesso a una variabile
  - I diversi thread l’accedono in modo mutualmente esclusivo
  - Il suo uso corretto è responsabilità del programmatore
    - Tutti gli accessi alla risorsa devono essere mediati dal mutex
    - I mutex devono essere acquisiti e rilasciati correttamente, per evitare il rischio di deadlock
- Si usa una istanza di **std::mutex**, che mette a disposizione i metodi:
  - **lock**() per acquisire la risorsa
  - **unlock**() per renderla di nuovo disponibile
- Sono usati direttamente solo in casi molto semplici
  - Quando è facile tenere sotto controllo il flusso di esecuzione
  - Esempio, simulazione di un metodo start per un thread

Lock

- Le classi lock permettono di seguire l'approccio RAII
  - Resource Acquisition Is Initialization
- std::**lock_guard** – uso comune, limitato a un blocco
- Funzione template variadica std::**lock**()
  - Semplifica la gestione nel caso siano richiesti più mutex
    - Nei casi più semplici basta mantenere un ordine di acquisizione definito
  - Si possono poi passare i lock a lock_guard, via std::**adopt_lock**
  - Da C++17 è disponibile std::scoped_lock, lock su più mutex
- std::**unique_lock** – maggior flessibilità
  - Può essere passato tra diversi scope, in quanto moveable (ma non copiabile)
  - Può essere associato a un mutex in modalità defer_lock
  - Può essere utilizzato per la gestione di condizioni

Condizione

- Comunicazione tra thread via std::condition_variable
- Richiede un unique_lock su di un mutex e una variabile che segnala lo stato corrente della comunicazione
- I metodi wait…() mettono il thread in attesa
- I metodi notify…() segnalano il cambiamento della condizione
- Esempio producer/consumer
  - Segnala da un thread all'altro in modo da alternare l'esecuzione