Programmazione di Sistema
Lab 5 - Thread

Obbiettivi

- Utilizzo dei thread
- Utilizzo dei socket
- Gestione delle risorse

Esercizio 1

Per iniziare a prendere confidenza con l'uso dei thread e dei pattern di sincronizzazione associati, consideriamo un caso tipico legato all'esecuzione concorrente: *il modello produttori e consumatori*. In questo modello, un certo numero di attività (i produttori) generano informazioni che devono essere ulteriormente elaborate da altre attività (i consumatori). Il numero di produttori è indipendente da quello dei consumatori (possono essere in rapporto 1:1, 1:N, N:1, N:M). Il sistema deve rispettare alcuni vincoli fondamentali: ogni produttore può generare un numero arbitrario (ma finito) di informazioni; le informazioni generate da un singolo produttore possono essere elaborate da qualsiasi consumatore; ogni informazione prodotta deve essere elaborata esattamente una volta sola (non deve capitare, cioè, che due consumatori elaborino la stessa informazione, né che un singolo consumatore elabori un'informazione due volte, né che un'informazione non sia elaborata da nessun consumatore); i produttori possono richiedere un tempo arbitrario per generare le proprie informazioni; quando tutti i produttori hanno terminato il proprio lavoro, i consumatori devono finire di elaborare gli elementi presenti nel sistema e poi terminare ordinatamente. Questo tipo di problema trova una naturale implementazione in un sistema dove produttori e consumatori sono costituiti da thread che si appoggiano ad una struttura condivisa, che deve essere thread-safe, per gestire il passaggio delle informazioni.

Definiamo quindi una classe `Jobs` che utilizzeremo per passare dei task tra dei producer e dei consumer.

```
template <class T>
class Jobs {
public:
      // inserisce un job in coda in attesa di essere processato, puo'
      // essere bloccante se la coda dei job e' piena
      void put(T job);
      // legge un job dalla coda e lo rimuove
      // si blocca se non ci sono valori in coda
      T get();
};
```

Quindi:

- Implementare, con un **singolo producer** e **multipli consumers** un sistema che ricerca una data stringa in tutti i file di testo contenuti in una data directory. Tale sistema è composto da:
  - Un'istanza della classe `Jobs` (detta `jobs`)
  - un **main** che riceve come parametri: la directory in cui sono contenuti i file (solo file con estensione .txt) e una regular expression che specifica il pattern da cercare.
  - un *producer* che scandisce la directory e legge in sequenza i file riga per riga, inserendole in jobs.
  - n *consumer* che leggono le righe da `jobs`, cercano se la regex è presente per ciascuna riga e ne stampano ogni occorrenza: nome del file, numero riga e match (**attenzione** alla sincronizzazione dell'output)
  - Sincronizzare *producer* e *consumer*, sospendendo i consumer se non vi sono dati disponibili, inserendo quindi nei metodi `put()` e `get()` le primitive di sincronizzazione necessarie.

    Riflessione: cosa capita quando i consumer sono troppo lenti rispetto alla capacità di produrre del consumer? Si modifichi quindi la `put()`, rendendola bloccante nel caso in cui la coda dei job superi una dimensione fissata.

    Quante condition variable servono in questo caso?

    Fare attenzione a gestire in modo corretto le condizioni di terminazione, garantendo che tutti i consumer terminino senza perdersi dei job quando il producer ha finito, modificando l'interfaccia del metodo `get()` e della classe Jobs se necessario; provare differenti strategie, discutendone pro e contro:
    - Inviare un valore sentinella che indica la terminazione
    - Una variabile che indica se ancora writer è attivo o meno; come può essere implementata questa variabile?
- Implementare multiple producer e multiple consumer
  - replicare tutti i punti del passo precedente, dividendo su più producer il ruolo di lettura dei file.
  - Il main thread legge i nomi del file e li inserisce in una coda di Jobs chiamata `fileJobs`, quindi m *consumer* leggono il file e producono ciascuno una serie di linee in una seconda coda di Jobs detta `lineJobs`, da cui altri *consumer*, che eseguono la ricerca, leggono le righe e cercano se vi sono match con la regex.
  - Anche in questo caso provare le differenti strategie di corretta terminazione.
- Implementare la coda dei job come un buffer circolare utilizzando un `std::vector` di dimensione fissa[1]
  - Quali vantaggi può avere rispetto ad una coda realizzata come una lista?

[1] `https://en.wikipedia.org/wiki/Circular_buffer`

Esercizio 2

Riprendere l'esercizio dell'esercitazione precedente mediante un pool di thread che esegue il paradigma *map-reduce* implementato con i processi.

- Il main thread legge le linee del file di log e le inserisce in una coda sincronizzata `lineJobs`
- I *mapper* consumano le linee da `resultJobs` e scrive i risultati nella struttura di accumulazione.
- Gestire in modo corretto le condizioni di terminazione: il programma deve stampare tutti i risultati quando l'ultimo risultato è stato inserito nella struttura di accumulazione.
- Provare a pensare ad una soluzione con più reducer in parallelo. È possibile? Se sì, quali accorgimenti occorre prendere? Provare ad implementare una soluzione con più *producer*.

Esercizio 3

Si realizzi un sistema di chat di gruppo composto da un server e un client. **Server**

- Il server gestisce una unica chat di gruppo comune a tutti gli utenti.
- Rimane in attesa di connessioni su una porta scelta all'avvio
- Accetta senza autenticazione la connessione di ogni client tramite un socket, che si presenta inviando il proprio nickname su un linea di testo
- Mantiene una lista dei nickname di tutti i client connessi e degli ultimi N messaggi
- Quando un client si connette, gli manda la lista di tutti gli altri nickname connessi
- Quando il sistema riceve un messaggio da un client, lo invia a tutti gli altri client connessi aggiungendo il nickname di chi ha inviato
- Permette l'invio di messaggi privati utilizzando il comando "/private" nel messaggio inviato da un client: `/private nickname testo_messaggio`
- Dopo 60 secondi di inattività disconnette un client
- Valutare le possibili strategie di gestione delle connessioni dei client, tenendo conto che potenzialmente si possono avere anche molte connessioni aperte contemporaneamente e implementarne una

Client

- I client hanno un'interfaccia basata su linea di comando, stampano i messaggi ricevuti sullo standard output e leggono dallo standard input
- All'avvio il client riceve come parametri:
  - Indirizzo IP del server
  - Porta su cui è in ascolto
  - Nickname dell'utente
- Fino a quando non è connesso e ha registrato con successo il nickname, il client visualizza la scritta "connecting to nome server..."
- Terminato l'handshake iniziale, il client visualizza:

  - la lista degli utenti connessi (su una singola linea, separati da spazio)
  - gli ultimi n messaggi, uno per riga

- Un programma interattivo deve contemporaneamente leggere da console i messaggi da inviare e stampare quelli ricevuti, deve inoltre evitare di mischiare input e output; ad esempio non deve stampare i messaggi ricevuti mentre l'utente sta scrivendo. Una tecnica per gestire queste situazioni è non appoggiarsi direttamente alla console, che ha solo un flusso lineare in "append", ma memorizzare tutti gli eventi di input/output su dei buffer interni e poi fare il refresh della console, cancellando e ristampando tutto, ogni qualvolta cambia qualcosa. Su linux è possibile utilizzare **ncurses**[2]. Con queste librerie è possibile un accesso avanzato alla console, in particolare ci interessano:
  - una funzione per leggere un carattere per volta senza attendere un newline e senza stamparlo (`getch()funziona con ncurses, occorre anche chiamare \lstinline{noecho()` all'inizio del programma oltre a `initsrc()`)
  - una funzione per cancellare lo screen (ncurses: `clear()`)
  - una funzione per scrivere una linea sullo schermo (ncurses: `addstr(char *)`)
- Occorrono quindi due thread:
  - Uno per leggere dallo standard input ogni carattere scritto dall'utente e memorizzarlo in un buffer interno
  - Uno per leggere i messaggi in arrivo dal server e memorizzarli in una apposita struttura

  Inoltre ad ogni evento di IO occorre:
  - Cancellare lo schermo
  - Stampare tutti i messaggi ricevuti e il messaggio che l'utente sta editando, così da rendere l'esperienza utente accettabile

[2] `https://tldp.org/HOWTO/NCURSES-Programming-HOWTO/`