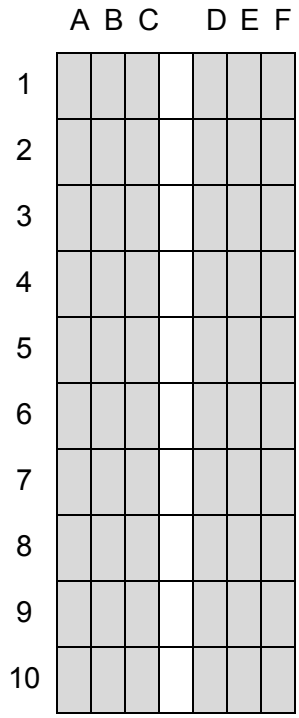
SISTEMI DI ELABORAZIONE
CORSO DI LAUREA MAGISTRALE IN INGEGNERIA ELETTRONICA

SPECIFICHE DI PROGETTO A.A. 2016/2017

Il progetto deve essere realizzato singolarmente (non è possibile realizzarlo in gruppo).

Il progetto consiste nello sviluppo di un'applicazione client/server. Client e server devono comunicare tramite socket TCP. Il server deve essere concorrente e la concorrenza deve essere implementata con i thread POSIX. Il thread main deve rimanere perennemente in attesa di nuove connessioni in ingresso. Ogni connessione accettata deve essere smistata verso un membro di un pool di thread preallocati, che hanno il compito di gestire le richieste.

Vogliamo realizzare un sistema per la prenotazione dei posti negli aerei. Gli aerei sono strutturati come indicato dalla seguente figura:



Ogni aereo è dotato di 10 file di posti. Le file sono identificate dai numeri che vanno da 1 a 10. Ogni fila ha tre posti su un lato dell'aereo e altri tre sull'altro lato (separati dal corridoio). I posti all'interno della singola fila sono identificati dalle lettere A, B, C e D, E, F. Ogni volo è identificato da una sigla alfanumerica univoca (per esempio AB1234). I client possono ottenere la lista dei voli e prenotare posti su un determinato volo.

LATO CLIENT

Il client viene avviato con la seguente sintassi:

```
./flight_client <host remoto> <porta>
```

dove

- `<host remoto>` è l'indirizzo dell'host su cui è in esecuzione il server;
- `<porta>` è la porta su cui il server è in ascolto.

I comandi disponibili per l'utente devono essere:

- `help`
- `list`
- `quit`
- `select` *`volo`*
- `show`
- `reserve` *`posto`*
- `deselect`

Significato dei comandi:

1. `help`: mostra l'elenco di comandi disponibili. Esempio di esecuzione:
   ```
   >help
   Comandi disponibili:
   help --> mostra l'elenco dei comandi disponibili
   list --> mostra la lista di voli disponibili
   quit --> disconnette il client dal server
   select volo --> seleziona il volo specificato
   show --> mostra lo stato dei posti relativamente al volo selezionato
   reserve posto --> prenota il posto specificato sul volo selezionato
   deselect --> deseleziona il volo precedentemente scelto
   ```

2. `list`: mostra la lista di voli e il numero di posti ancora disponibili. Esempio di esecuzione:
   ```
   >list
   Voli:
   AB1234 35
   KL5566 44
   QW9909 12
   ZA3256  2
   >
   ```

3. `quit`: disconnette il client dal server. In particolare il client chiude il socket con il server ed esce. Il server stampa un messaggio che documenta la disconnessione del client. Esempio di esecuzione:

```
>quit
Client disconnesso correttamente
```

4. `select` *`volo`*: seleziona il volo specificato. Il prompt dei comandi cambia andando ad includere l'identificatore del volo selezionato. Esempio di esecuzione:

```
>select AB1234
AB1234#
```

Gestire il caso in cui l'identificatore del volo sia non corretto. Esempio:

```
>select XY0123
Volo inesistente.
>
```

Ogni volo non può essere selezionato da più di 3 client contemporaneamente. Eventuali ulteriori client che tentano di selezionarlo rimangono bloccati nella `select` (in attesa che qualcuno esegua una `deselect`).

5. `show`: mostra lo stato dei posti di un aereo. Il comando può essere eseguito solo dopo che un volo è stato selezionato. Esempio di esecuzione:

```
AB1234#show
      ABC DEF
1     LOO OLO
2     LLL LLO
3     LLL LLL
4     OLO OLL
…
10    LOO OOO
AB1234#
```

Il carattere `O` è usato per indicare i posti occupati, il carattere `L` per indicare quelli liberi.

6. `reserve` *`posto`*: prenota un posto. Il comando può essere eseguito solo dopo che un volo è stato selezionato. Il comando prevede le varianti illustrate dai seguenti esempi:

   `reserve 3B`: prenota il posto 3B se disponibile. Se il posto è già occupato mostra un opportuno messaggio. Esempio di esecuzione:

   ```
   AB1234#reserve 1F
   Il posto 1F è stato prenotato.
   AB1234#reserve 8C
   Il posto 8C è occupato.
   AB1234#
   ```

   `reserve *`: prenota un posto qualunque, lasciando scegliere il server. Esempio di esecuzione:

   ```
   AB1234#reserve *
   Il posto 5A è stato prenotato.
   ```

```
AB1234#reserve *
Non ci sono posti disponibili.
AB1234#
```

`reserve W` e `reserve I`: prenota un posto qualunque vicino al finestrino (`W`) o vicino al corridoio (`I`). Esempio di esecuzione:

```
AB1234#reserve W
Il posto 6F è stato prenotato.
AB1234#reserve I
Non ci sono posti disponibili vicino al corridoio.
AB1234#
```

7. `deselect`: deseleziona il volo precedentemente scelto. Il comando può essere eseguito solo se un volo è attualmente selezionato. Esempio di esecuzione:

```
AB1234#deselect
>deselect
Nessun volo selezionato.
```

LATO SERVER

Il server `flight_server` è concorrente ed utilizza thread ausiliari per gestire le richieste che provengono dai client. Il thread main, prima di mettersi in attesa di connessioni, prealloca un pool di thread ausiliari. Il thread main assegna ad un thread ausiliario (libero) del pool ogni nuova connessione che riceve. Il numero di thread del pool è specificato a tempo di compilazione (è una costante e non varia durante l'esecuzione del programma).

La sintassi del comando è la seguente:

```
./flight_server <host> <porta> <file>
```

dove:

- `<host>` è l'indirizzo su cui il server viene eseguito;
- `<porta>` è la porta su cui il server è in ascolto;
- `<file>` è un file di testo che contiene la lista dei codici dei voli che il server deve gestire; inizialmente tutti i posti di tutti i voli sono liberi.

Possiamo schematizzare lo schema di funzionamento del server nel seguente modo:

1. il thread main crea NUM_THREAD thread gestori;
2. il thread main si mette in attesa delle connessioni in ingresso;
3. quando riceve una connessione in ingresso, il thread main:

    a. controlla il numero di thread occupati (quelli che stanno attualmente eseguendo una richiesta);
    b. se tutti i thread del pool sono occupati si blocca in attesa che uno diventi libero;
    c. se c'è almeno un thread libero ne sceglie uno (senza nessuna politica particolare) e lo attiva;
4. il thread gestore (all'infinito):
    a. si blocca finché non gli viene assegnata una richiesta;
    b. riceve il comando da un client;
    c. esegue la richiesta del client;
    d. se il thread ha ricevuto il comando `quit`, il thread torna libero e a disposizione per servire un altro client. Altrimenti, il thread si blocca in attesa di un nuovo comando dallo stesso client.

Ne consegue che ogni thread gestore è associato ad uno e ad un unico client per tutto il tempo che quest'ultimo è connesso al server.
Una volta mandato in esecuzione `flight_server` deve stampare a video delle informazioni descrittive (caricamento della lista di voli, creazione del socket di ascolto, creazione dei thread, connessioni accettate, operazioni richieste dai client, ecc.).

Un esempio di esecuzione del server è il seguente:

```
$ ./flight_server 127.0.0.1 1235 lista.txt
MAIN: Ho caricato la lista di voli contenuta nel file lista.txt
Indirizzo: 127.0.0.1 (Porta: 1235)
THREAD 0: pronto
THREAD 1: pronto
THREAD 2: pronto
THREAD 3: pronto
MAIN: connessione stabilita con il client 127.0.0.1:1235
MAIN: E' stato selezionato il thread 0
THREAD 0: ricezione comando list
MAIN: connessione stabilita con il client 127.0.0.1:1235
MAIN: E' stato selezionato il thread 1
THREAD 0: ricezione comando select
THREAD 1: ricezione comando list
THREAD 1: ricezione richiesta select
...
```

AVVERTENZE E SUGGERIMENTI

Test.
Si testino le seguenti configurazioni:

- un client viene avviato quando alcuni thread gestori sono già occupati (ma ce n'è almeno uno libero);
- un client viene avviato quando non ci sono più thread gestori liberi.

Modalità di trasferimento dati tra client e server (e viceversa).
Client e server si scambiano dati tramite socket TCP. Prima che inizi ogni scambio è necessario che il ricevente sappia quanti byte deve leggere dal socket.

Ogni risorsa condivisa deve essere protetta da opportuni meccanismi semaforici.

Non utilizzare meccanismi di attesa attiva in nessuna parte del codice.
Esempio: Quando un client è in attesa che l'utente inserisca un comando, il thread corrispondente nel server si deve bloccare (l'operazione *recv()* è bloccante).

VALUTAZIONE DEL PROGETTO

Il progetto viene valutato durante lo svolgimento dell'esame. Il codice sarà compilato ed eseguito su una distribuzione Ubuntu Linux. Si consiglia di testare il programma su Ubuntu prima dell'esame. La valutazione prevede le seguenti fasi.

1. **Compilazione dei sorgenti**. Il client e il server devono essere compilati dallo studente in sede di esame utilizzando il comando gcc (non sono accettati Makefile) e attivando l'opzione -Wall che abilita la segnalazione di tutti i warning. Si consiglia di usare tale opzione anche durante lo sviluppo del progetto, interpretando i messaggi forniti dal compilatore e provvedendo ad eliminarli.
2. **Esecuzione dell'applicazione**. Il client e il server vengono eseguiti simulando una tipica sessione di utilizzo. In questa fase si verifica il corretto funzionamento dell'applicazione e il rispetto delle specifiche fornite.
3. **Esame del codice sorgente**. Il codice sorgente di client e server viene esaminato per controllarne l'implementazione.