RELAZIONE PROGETTO SISTEMI OPERATIVI

Giovanni Dini

Sessione Autunnale 2010/2011

SPECIFICHE DEL PROBLEMA

Vedasi il file Specifiche_progetto_OS_autunnale_2010-11.pdf.

ANALISI DEL PROBLEMA

Il progetto richiede lo sviluppo di un'applicazione *multithread* che simuli un aeroporto dotato di due piste e la gestione delle sue code mediante un gestore del traffico.

Dati in input:

- Numero di aerei

Dati in output:

- Tempo medio di attesa per ogni coda.

Gestione delle code da implementare:

Gli aerei si dividono in due categorie: privati e di linea. Possono inoltre richiedere due tipi di servizi: decollo e atterraggio. Ogni aereo ha inoltre un peso. La gestione delle code richiede che gli aerei in fase di atterraggio abbiano la precedenza su quelli in fase di decollo. Inoltre, gli aerei privati hanno la precedenza su quelli di linea. In aggiunta a questo, all'interno delle code stesse la precedenza viene attribuita all'aereo con il peso maggiore. Infine, a parità di queste condizioni, la precedenza viene data all'aereo che richiede prima il servizio secondo l'algoritmo FCFS (First Come - First Served).

Scelte progettuali:

Il linguaggio in cui è stata scritta l'applicazione è Java, su esplicita richiesta nelle specifiche del progetto. Java è un linguaggio orientato agli oggetti e che permette di implementare il multithreading in maniera piuttosto semplice. Sempre per esplicita richiesta nelle specifiche del progetto, non è possibile usare il costrutto synchronized che fornisce un valido aiuto automatico nella sincronizzazione. Di

contro, ogni forma di sincronizzazione deve essere realizzata utilizzando le primitive di sincronizzazione messe a disposizione dal linguaggio Java.

Per semplicità e velocità di realizzazione si è scelto di prendere i valori in input tramite tastiera.

Primitive di sincronizzazione utilizzate:

Le specifiche di progetto richiedono che l'aeroporto sia sviluppato come un oggetto condiviso tra aerei e gestore. Requisito fondamentale, quindi è la mutua esclusione. Le primitive di sincronizzazione utilizzate sono:

- Un ReentrantLock per permettere ad un solo thread alla volta di entrare nella rispettiva sezione critica.
- Sul ReentrantLock sono state generate variabili condizionali per l'utilizzo di metodi come await() e signal(). L'utilizzo di ReentrantLock ci è fondamentale anche nel fatto che, sospendendoci su di esso, tutti gli altri thread saranno obbligati a bloccarsi e aspettare che il lock venga rilasciato. Mediante la sospensione su una variabile condizionale, invece, gli altri thread mantengono la possibilità di entrare nella sezione critica o effettuare una sospensione a loro volta sulle variabili condizionali.

Progettazione delle code:

Le code progettate sono quattro. L'accesso ad ogni specifica coda viene garantito tramite la priorità di ogni thread. Di seguito la spiegazione:

- **Coda 1 (livello priorità 1):** E' la coda di livello più basso. Finiscono in questa gli aerei di linea che richiedono un decollo.
- **Coda 2 (livello priorità 2):** E' la coda in cui vengono posizionati gli aerei privati che richiedono un decollo.

- **Coda 3 (livello priorità 3):** In questa coda vengono inseriti gli aerei di linea che richiedono un atterraggio.
- **Coda 4 (livello priorità 4):** Questa è la coda con livello di priorità più alto. Al suo interno troviamo gli aerei privati che richiedono un atterraggio.

Si ricorda che l'ordinamento all'interno della coda stessa viene effettuato secondo l'attributo peso di ogni aereo. In caso di parità anche sul peso, le richieste sono evase secondo FCFS.

Un'altra cosa molto importante da far notare è che una volta richiesto un servizio, ogni aereo inoltra richiesta per il servizio complementare. Dato che i servizi determinano per la maggior parte la priorità di ogni aereo, prima della richiesta complementare l'aereo cambia priorità e sarà dunque inserito in una nuova coda.

Aggiunte varie:

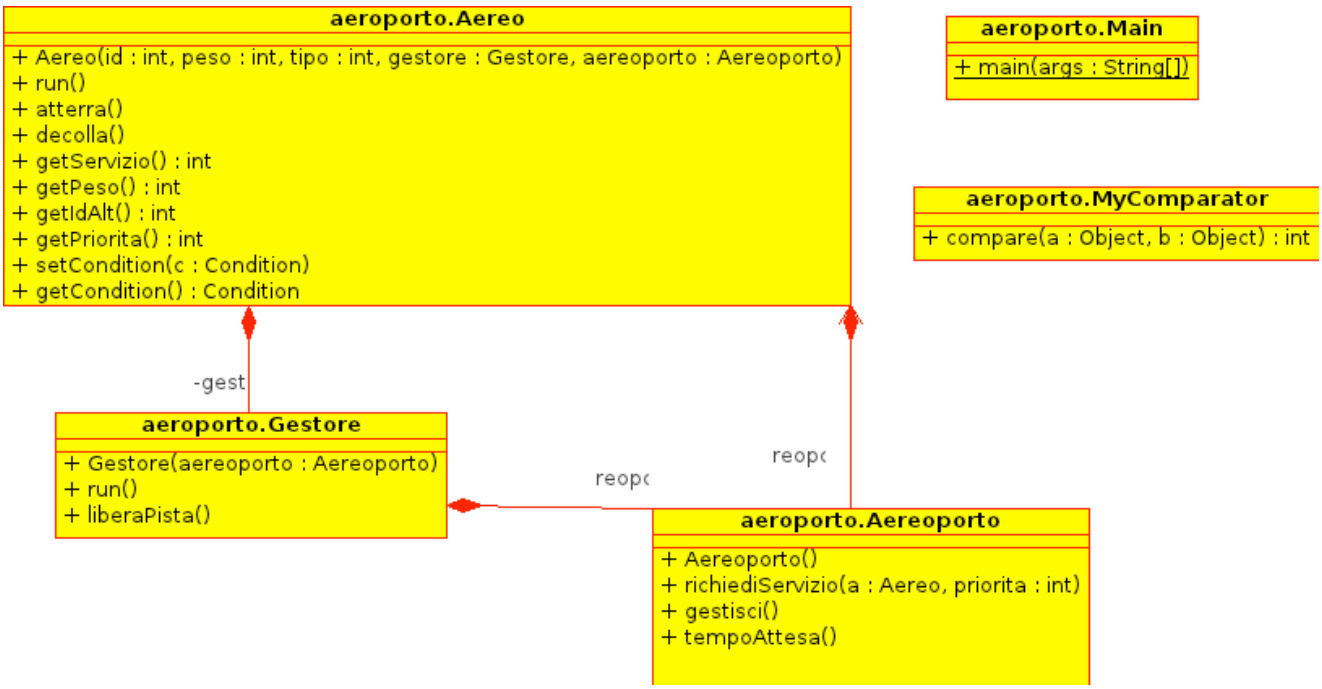
Come richiesto, se non ci sono richieste in attesa, il gestore del traffico si sospende senza occupare inutilmente cicli di CPU.

Allo stesso modo, una volta inoltrata la richiesta, gli aerei si sospendono in attesa di essere svegliati dal gestore del traffico, una volta venuto il loro turno.

Il tipo di aereo (privato/di linea), il servizio richiesto (decollo/atterraggio) e il peso degli aerei vengono generati pseudo-casualmente ogni volta che un aereo nuovo viene creato. Dopo la generazione, inoltre, viene modificata la priorità dell'aereo secondo gli attributi che sono stati generati. Una volta ottenuto il primo servizio, l'aereo modifica il suo attributo di servizio (decollo -> atterraggio o atterraggio -> decollo) e, di conseguenza, la sua priorità.

Per facilitare la gestione delle code, i metodi di richiesta del servizio verranno invocati dagli aerei direttamente all'aeroporto e non al gestore.

IMPLEMENTAZIONE
DIAGRAMMA DELLE CLASSI



CLASSE MAIN:

La classe main contiene il punto di lancio del programma. Istanzia un numero a scelta dell'utente di aerei e un gestore, dopodiché manda tutti i thread in esecuzione. In ultimo, stampa a video i tempi di attesa medi.

CLASSE AEREO:

La classe aereo contiene un metodo run() che definisce il comportamento dell'oggetto. Specifica un servizio che richiederà all'aeroporto (come già detto, il servizio viene richiesto all'aeroporto e non al gestore per facilitare inserimento e rimozione dalle code dell'aereo). Quando l'aeroporto concede il turno all'aereo vengono simulati i tempi di decollo e atterraggio.

CLASSE GESTORE:

La classe del gestore contiene il metodo run() che definisce il comportamento dell'oggetto. L'oggetto accede all'aeroporto per gestire le richieste degli aerei. Il processo viene terminato da un thread interrupt da parte della main.

CLASSE AEROPORTO:

La classe aeroporto si occupa della gestione delle code mediante il gestore, fornisce inoltre il metodo per il calcolo dei tempi medi d'attesa per ogni coda.

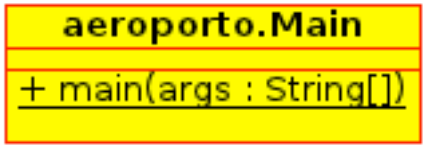
CLASSE MYCOMPARATOR:

Questa classe non è null'altro che una riscrittura del criterio di comparazione su cui il metodo sort applica un mergesort. Viene utilizzato dal nostro programma nel momento in cui gli aerei vengono aggiunti alla coda e ordinati in base al peso.

Il rapporto tra l'aeroporto e le altre classi è di 1 a 1 nel caso del gestore (perché ce n'è solo uno) e 1 a n nel caso degli aerei, in quanto il loro numero è variabile a piacere.

Nota: i nomi tagliati sono "gestore" e "aeroporto".

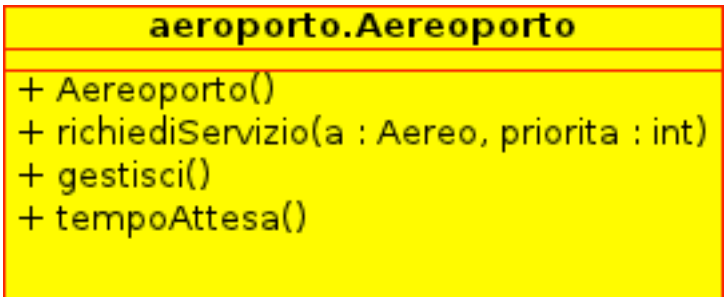
CLASSE MAIN



La classe di main è, ovviamente, il punto di lancio del programma. E' all'interno di questo metodo che viene fatta richiesta del numero di aerei da mandare in simulazione. Inoltre, l'attesa della terminazione e la stampa dei tempi medi d'attesa (calcolati però altrove) sono altresì funzioni che vengono lasciate fare a questa classe.

Non ci sono notazioni di rilievo particolari da fare: la creazione delle istanze viene fatta tramite l'invocazione dei costruttori. Una volta terminati i thread degli aerei viene mandato un thread interrupt al gestore che terminerà così la propria esecuzione. A questo punto viene fatta la stampa dei tempi d'attesa e terminerà anche la main, terminando così il programma.

CLASSE AEROPORTO



La classe Aeroporto comprende i metodi richiediServizio() che si occupa, come dice il nome, di inoltrare la richiesta di servizio degli aerei e farla passare per il gestore. Comprende poi il metodo gestisci() che fornisce il riferimento ai metodi atterra() e decolla() che sono contenuti nella classe Aereo e permettono la simulazione dei tempi di atterraggio e decollo da parte degli aerei. Il metodo tempoAttesa() si occupa del calcolo dei tempi medi d'attesa. E' presente inoltre anche un metodo privato sveglia() per svegliare gli aerei che devono eseguire le loro richieste e farli uscire dalla coda. E' qui che viene registrato il tempo di uscita. Il tempo di entrata viene invece registrato nel momento in cui l'aereo viene posto in coda (all'interno del metodo richiediServizio()).

All'interno della classe Aeroporto sono presenti quattro code LinkedList che gestiscono la priorità con cui vengono processate le richieste degli aerei. Inizialmente si era pensato di realizzarle con una HashMap (in modo da avere tempi più veloci, in quanto le operazioni sulle HashMap hanno complessità O(1)), ma il fatto di essere unsorted (e complicando quindi la gestione della priorità in base al peso) le ha rese poco consigliabili. Una implementazione di questo tipo implicava infatti la necessità di clonare l'HashMap o di trasferire tutti i suoi elementi in un ArrayList, cosa che diventava decisamente poco comoda e confusionaria. Così, si è deciso di utilizzare delle code LinkedList, ordinate tramite il MyComparator personalizzato.

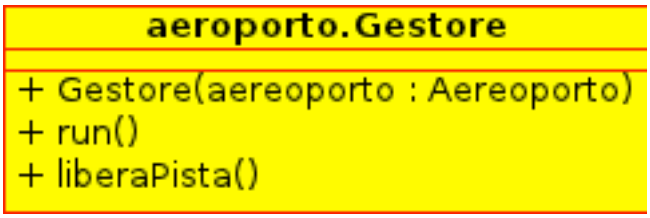
Le variabili presenti sono in gran parte necessarie per il calcolo dei tempi medi d'attesa: si tratta di contatori per il numero di aerei serviti in ogni coda e variabili long per la memorizzazione del tempo di entrata ed uscita per ogni coda.

Come già anticipato in precedenza, la mutua esclusione è garantita da un ReentrantLock.

E' presente un semaforo chiamato "aerei" per la gestione delle richieste degli aerei, utilizzato anche per permettere al gestore di dormire quando non sono presenti richieste.

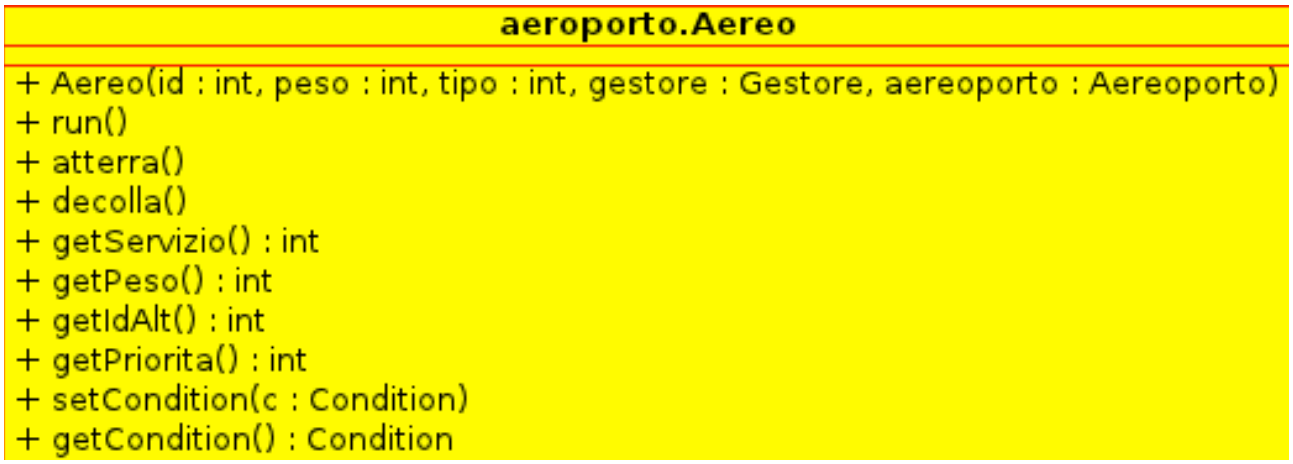
Un altro semaforo utilizzato è "piste" utilizzato per permettere l'accesso contemporaneo a solo due (in questo caso) piste.

CLASSE GESTORE



La classe gestore è composta di un metodo run() per mandare in esecuzione il gestore e un metodo liberaPista() che permette, ovviamente, di liberare la pista rilasciando il semaforo, una volta terminato il servizio fornito all'aereo.

CLASSE AEREO



La classe Aereo è fornita di diversi metodi pubblici principalmente orientati al passaggio degli attributi per l'utilizzo in altri metodi.

Il metodi principali è, come sempre, run() che si occupa di determinare pseudo-casualmente le caratteristiche dell'aereo e attribuirgli la giusta priorità, provvedendo poi all'invocazione della richiesta del servizio. Altrettanto importanti sono atterra() e decolla() che simulano il tempo necessario ad eseguire il servizio richiesto.

Nota importante: notare come all'interno della run si sia utilizzato un ciclo for per chiedere due volte i servizi. Le norme di progetto includevano infatti anche la necessità, da parte dell'aereo, di richiedere il servizio complementare a quello già ottenuto (decollo->atterraggio o atterraggio->decollo). All'interno del ciclo for e dopo la prima richiesta di servizio è presente un if che permette di invertire il servizio richiesto e modulare nella maniera correttamente la nuova priorità.

TESTING

Si allegano, di seguito, alcuni risultati di testing.

TESTING NUMERO 1:

```
Inserisci il numero di Aerei:
4

-----

Aerei serviti e terminati. Esecuzione completata.

-----

Richieste a priorità 4 servite= 1 tempo medio=1 ms
Richieste a priorità 3 servite= 3 tempo medio=73 ms
Richieste a priorità 2 servite= 1 tempo medio=164 ms
Richieste a priorità 1 servite= 3 tempo medio=192 ms
```

TESTING NUMERO 2:

```
Inserisci il numero di Aerei:
10

-----

Aerei serviti e terminati. Esecuzione completata.

-----

Richieste a priorità 4 servite= 7 tempo medio=56 ms
Richieste a priorità 3 servite= 3 tempo medio=263 ms
Richieste a priorità 2 servite= 7 tempo medio=798 ms
Richieste a priorità 1 servite= 3 tempo medio=1236 ms
```

TESTING NUMERO 3:

```
Inserisci il numero di Aerei:
20

-----

Aerei serviti e terminati. Esecuzione completata.

-----

Richieste a priorità 4 servite= 10 tempo medio=0 ms
Richieste a priorità 3 servite= 10 tempo medio=1401 ms
Richieste a priorità 2 servite= 10 tempo medio=1583 ms
Richieste a priorità 1 servite= 10 tempo medio=2224 ms
```

TESTING NUMERO 4:

```
Inserisci il numero di Aerei:
50

-----

Aerei serviti e terminati. Esecuzione completata.

-----

Richieste a priorità 4 servite= 27 tempo medio=30 ms
Richieste a priorità 3 servite= 23 tempo medio=2812 ms
Richieste a priorità 2 servite= 27 tempo medio=4304 ms
Richieste a priorità 1 servite= 23 tempo medio=6106 ms
```

CONCLUSIONI

Il testing rispecchia fedelmente le conclusioni che ci aspettavamo. Ovviamente i tempi di attesa devono crescere con il dimuire della priorità e questo è esattamente il tipo di comportamento che ci aspettavamo dal programma.

NOTE FINALI

Il codice sorgente è corredato di commenti dettagliati per le parti più importanti e i vari metodi.

Sono altresì presenti, all'interno del codice e opportunamente contrassegnate come commenti, istruzioni per la stampa di variabili, utili per un eventuale debug del programma. Essendosi rivelate molto utili in fase di sviluppo, si è ritenuto opportuno non eliminarle e lasciarle a disposizione dell'esaminatore.