Documentazione

Andrea Vincentini

1 febbraio 2016

Indice

1 FairSem

Un *FairSem* è un semaforo FIFO realizzato utilizzando gli strumenti di basso livello offerti dal linguaggio Java.

1.1 Implementazione

La classe FairSem fornisce il *costruttore* e i metodi *P()* e *V()*.
Fornisce inoltre le variabili private *sem_val* (contenente il valore del semaforo) e la coda *blocked_thread_queue* (utilizzata per la memorizzazione FIFO dei thread in attesa di poter accedere alla risorsa).

1.1.1 FairSem(int value) o FairSem(int value, boolean test)

Sono presenti 2 costruttori simili tra loro discriminati dal numero di argomenti passati. Il costruttore inizializza il valore del semaforo con il valore passatogli come argomento.
Nel caso si stia eseguendo un test viene invocato il costruttore *FairSem(int value, boolean test)* che permette l'abilitazione del test.

1.1.2 P()

Questo metodo ricava l'identificatore del thread in esecuzione e lo inserisce preliminarmente all'interno della coda *blocked_thread_queue* e verifica la seguente condizione:

```
sem_val <= 0 || th_id != blocked_thread_queue.peek()
```

Se la condizione è verificata il semaforo viene asseganto al thread altrimenti il thread sospende la sua esecuzione e la condizione verrà ricontrollata ogni volta che il thread si sveglia.

1.1.3 V()

Questo metodo risveglia i thread che si sono bloccati in attesa di poter accedere al semaforo

1.2 Esecuzione e Test

Per eseguire il test basta dare i seguenti comandi:

```
make FairSem_Test
java FairSem_Test
```

Scopo del test è mostrare che in ogni istante durante l'esecuzione ci siano al più *value* thread che possiedono il semaforo e che l'ordine di esecuzione dei thread rispettino la politica FIFO.

2 SynchPort

La classe SynchPort rappresenta una generica porta sincrona sviluppata utilizzando solamente gli strumenti messi a disposizione dalla classe FairSem. All'interno di una porta vengo inviati dati di tipo *Message* contenenti l'informazione e la porta del mittente.

2.1 Implementazione

La classe fornisce un *costrutture* e i due metodi *send()* e *receive()*. Le variabili private della classe sono:

- mess (buffer per contiene il messaggio);
- empty (FairSem che indica se il buffer mess è vuoto);
- full (FairSem che indica se il buffer mess è pieno);
- synch (FairSem di sincronizzazione);

2.1.1 SynchPort()

Il costruttore della classe inizializza i semafori ai seguenti valori:

- empty = 1;
- full = 0;
- synch = 0;

2.1.2 send()

Per poter inviare un messaggio un mittente deve verificare se *empty = 1* altrimenti deve bloccarsi su quel semaforo. Se empty = 1 il messaggio viene inserito nel buffer e il mittente deve sincronizzarsi con il ricevente tramite il semaforo *synch*.

2.1.3 receive()

Per poter ricevere un messaggio il ricevente deve prima verificare se *full = 1* altrimenti deve bloccarsi su quel semaforo. Una volta ricevuto il messaggio il ricevente può risvegliare il mittente tramite il semaforo di sincronizzazione *synch*.

2.2 Esecuzione e Test

Nel test è prevista una SynchPort che riceve messaggi stringa da 6 consumatori che inviano con velocità differenti. Il test serve per dimostrare la corretta gestione FIFO di ricezione dei messaggi inviati e il ricevimento di tutti i messaggi inviati. Per eseguire il test sono necessari i seguenti comandi:

```
make SynchPort_Test
java SynchPort_Test
```

3 PortArray

La classe *PortArray* definisce un array di SynchPort fornendo i metodi *send()* e *receive()*. Questa classe sfrutta i meccanismo di sincronizzazione forniti dalla classe FarSem.

3.1 Implementazione

Le variabili private della classe sono:

- *port_array* (un array di tipo PortStruct contenente la porta e il numero di produttori bloccati su quella porta);
- *mutex* (semaforo di mutua esclusione);
- *data_available* (semaforo di sincronizzazione per indicare la presenza di un dato);
- *RR_index* (indice per gestire l'array di porte in modalità Round Robin);
- *cont_rcv_blocked* (contatore dei consumatori bloccati).

3.1.1 PortArray(int dim)

Construttore della classe che genera il vettore *port_array* di dimensione *dim* e inizializza le variabili a:

- *mutex* a 1 ;
- *data_available* a 0;
- *RR_index* a -1;
- *cont_rcv_blocked* a -1.

3.1.2 send(Message mess, int p)

Il metodo *send(Message¡T¿ mess, int p)* permette di inviare il messaggio *mess* alla porta di indice *p* appartenente all'array *port_array*.
Prima di inviare un messaggio viene incrementato il contatore dei produttori bloccati sulla porta e viene segnalato al consumatore tramite il semaforo di sincronizzazione *data_available* che un nuovo dato è disponibile.

3.1.3 receive(int v[], int n)

Il metodo *receive(int v[], int n)* permette ad un consumatore di ricevere un dato da una delle porte specificate all'interno del vettore *v*. Ogni volta che il consumatore può ricevere un dato da una di queste porte viene decrementato il contatore dei produttore bloccati sulla porta. Questo metodo restituisce una struttura dati *Msg_Rcv* contenente il messaggio e il numero della porta. Per evitare *starvation* il vettore delle porte viene esaminato con un politica Round Robin.

3.2 Esecuzione e Test

Il test ha lo scopo di verificare che i messaggi vengano ricevuti in modalità FIFO e che un consumatore legga solamente dalle porte specificate all'interno dell'array passato come argomento.
Il test prevede due casi:

- 3 produttori e 1 consumatore;
- 3 produttori e 2 consumatori.

Nel primo caso il consumatore leggerà da tutte e 3 le porte disponibili mentre nel secondo caso il consumatore1 si metterà in ascolto delle porte w[] = {0, 2} e il consumer2 sulla porta z[] = {1}. Per eseguire il test sono necessari i seguenti comandi:

```
make SynchPort_Test
java SynchPort_Test
```

4 Mailbox

La classe *Mailbox*, sviluppata a partire dalle classi PortArray e SynchPort è eseguita come un processo server e fornisce due metodi utilizzabili dai produttori e dai consumatori.

4.1 Implementazione

La classe Mailbox fornisce ai produttori il metodo *request_to_insert()* e ai consumatori il metodo *request_to_remove()*. Le variabili private della classe sono:

- i due buffer *int buffer[]* e *int buffer_id_sender[]*;
- il PortArray *ports* contenente le 2 porte del server;
- la lista *waiting_senders*;
- gli indici *index*, *last*, *count* per la gestione del buffer.

La lista *waiting_senders* assume una struttura differente nel caso sia eseguito l'assegnamento 4a o l'assegnamento 4b. Nel caso di assegnamento 4a ogni elemento della lista è una *SynchPort* mentre nel caso di assegnamento 4b ogni elemento della lista è un elemento di tipo *Priority_Port_List* contenete la SynchPort e la priorità del thread.

4.1.1 Mailbox_A()

Il costruttore inizializza le variabili private e inizializza il PortArray *ports* a dimensione 2 (una per le richieste di inserimento e l'altra per le richieste di dimensione) e la *waiting_senders* come una lista di SynchPort.

4.1.2 Mailbox_B()

Il costruttore inizializza le variabili private e inizializza il PortArray *ports* a dimensione 2 (una per le richieste di inserimento e l'altra per le richieste di dimensione) e la *waiting_senders* come una lista di Priority_Port_List.

4.1.3 request_to_insert(SynchPort port)

Questo metodo, invocato dai produttore, permette di inviare un messaggio alla Mailbox con il quale si richiede di inserire un dato prelevabile dalla porta *port*.

4.1.4 request_to_insert(SynchPort port, int priority)

Questo metodo, invocato dai produttore, permette di inviare un messaggio alla Mailbox con il quale si richiede di inserire un dato prelevabile dalla porta *port* inviato da un produttore di priorità *priority*.

4.1.5 request_to_remove(SynchPort port)

Questo metodo, invocato dai consumatori, permette di inviare un messaggio alla Mailbox con il quale si richiede di rimuovore un dato dal buffer (se ne presente almeno uno) inviandolo alla SynchPort *port* del consumatore.

4.2 Esecuzione e Test

Durante l'esecuzione del corpo della Mailbox viene letto un messaggio su una delle due porte appartenti all'array *ports*.
La gestione della lista *waiting_senders* varia a seconda dell'assegnamento eseguito:

- *4a* la lista viene gestita con una politca FIFO;
- *4b* la lista viene gestita con una politca a base prioritaria (gli elementi a priorità piu alta sono in cima alla lista e il prelievo coinvolge l'elemento a priorità maggiore).

Nel caso si riceva un messaggio sulla porta *INSERT_DATA* se il buffer non è pieno il dato ricevuto viene inserito nel buffer altrimenti si inserisce all'interno della lista *waiting_senders*. Nel caso si riceva un messaggio sulla porta *REMOVE_DATA* se la lista *waiting_senders* è vuota si rimuove semplicemnte un elemento dal buffer mentre se la lista non è vuota si rimuove un elemento dal buffer e lo si sostituisce con il valore letto dalla porta contenuta nel primo elemento della lista *waiting_senders*.
Il test prevede l'utilizzo di 1 consumatore ch effettua 50 operazioni per ricevere tutti i messaggi inviati dai 10 produttori (ogni produttore invia 5 messaggi).
Nell'assegnamento 4a si assume di avere produttori con uguale priorità mentre nel 4b si assume di avere produttori con priorità differente compresa tra 1 e 10.
Scopo del test è mostrare che tutti i messaggi inviati siano ricevuti e che i messaggi inseriti in lista siano estratti secondo le modalità di gestione specificate nell'assegnamento.
Per eseguire il test dell'assegnamento 4a sono necessari i seguenti comandi:

```
make Mailbox_A_Test
java Mailbox_A_Test
```

Per eseguire il test dell'assegnamento 4b sono necessari i seguenti comandi:

```
make Mailbox_B_Test
java Mailbox_B_Test
```