Azzolini Riccardo

Produttore-consumatore – Altre soluzioni

1 Soluzione con i semafori

Siccome monitor e semafori sono equivalenti, il problema del produttore-consumatore può essere risolto anche con i semafori. In particolare, ne servono tre:

- uno per la mutua esclusione sulla sezione critica;
- uno per bloccare il consumatore quando il buffer è vuoto;
- uno per bloccare il produttore quando il buffer è pieno.

I tre semafori vengono creati all'interno della classe `ProdCons` (quella contenente il main), che, per semplicità, li memorizza in degli attributi statici pubblici (cioè, di fatto, in delle variabili globali):

```
import java.util.concurrent.Semaphore;

public class ProdCons {
    public static final Semaphore mutex = new Semaphore(1);
    public static final Semaphore full = new Semaphore(0);
    public static final Semaphore empty = new Semaphore(1);

    public static void main(String[] args) {
        CellaCondivisa cella = new CellaCondivisa();
        new Produttore(cella).start();
        new Consumatore(cella).start();
    }
}
```

I valori di inizializzazione dei semafori sono importanti:

- `mutex` è inizializzato a 1 (come solitamente avviene per i semafori finalizzati alla mutua esclusione), perché la sezione critica è inizialmente vuota, quindi il primo thread che ci vorrà entrare potrà farlo, senza bisogno di essere autorizzato da un altro.

- Analogamente, `empty` è inizializzato a 1 perché all'inizio il buffer è vuoto, quindi il produttore potrà inserire subito un elemento. Siccome si sta considerando, per ora, un buffer di capacità 1, i tentativi di inserimento successivi al primo dovranno essere bloccati finché lo spazio non viene liberato dal consumatore, perciò è corretto che la prima produzione porti a 0 il semaforo, e un valore di inizializzazione superiore a 1 sarebbe invece errato.
- Al contrario, `full` è inizializzato a 0 per indicare che, in partenza, il buffer è vuoto, quindi il consumatore non potrà proseguire finché il produttore non avrà inserito un elemento.

In questa versione della soluzione, la cella condivisa si occupa solo della mutua esclusione, che realizza usando il semaforo `mutex`:

```
public class CellaCondivisa {
    private static final int BUFFER_SIZE = 1;
    private int numItems = 0;
    private int value;

    public int getItem() throws InterruptedException {
        ProdCons.mutex.acquire();
        int tmp = value;                    //
        numItems--;                         // Sezione critica
        System.out.println(tmp + " read"); //
        ProdCons.mutex.release();
        return tmp;
    }

    public void setItem(int v) throws InterruptedException {
        ProdCons.mutex.acquire();
        value = v;                           //
        System.out.println(v + " written"); // Sezione critica
        numItems++;                          //
        ProdCons.mutex.release();
    }

    public int getCurrentSize() {
        return numItems;
    }
}
```

Invece, il blocco in caso di buffer pieno/vuoto è gestito direttamente nel codice dei due tipi di thread:

```
public class Produttore extends Thread {
    private CellaCondivisa cella;

    public Produttore(CellaCondivisa cella) {
        this.cella = cella;
    }

    public void run() {
        int i = 0;
        while (true) {
            try {
                // Blocca se il buffer è pieno.
                ProdCons.empty.acquire();
                cella.setItem(i);
            } catch (InterruptedException e) { break; }
            // Adesso il buffer contiene almeno un elemento:
            // sblocca un eventuale consumatore in attesa.
            ProdCons.full.release();
            i++;
        }
    }
}

public class Consumatore extends Thread {
    private CellaCondivisa cella;

    public Consumatore(CellaCondivisa cella) {
        this.cella = cella;
    }

    public void run() {
        while (true) {
            int v;
            try {
                // Blocca se il buffer non contiene almeno un elemento.
                ProdCons.full.acquire();
                v = cella.getItem();
            } catch(InterruptedException e) { break; }
            // Adesso il buffer è sicuramente non pieno:
            // sblocca un eventuale produttore in attesa.
            ProdCons.empty.release();
        }
    }
}
```

Osservazione: Mentre la cella condivisa fa sempre `acquire` e `release` sullo stesso semaforo, `mutex`, qui le chiamate `acquire` e `release` in uno stesso metodo sono fatte su semafori diversi.

2 Realizzazione di una coda

Solitamente, il buffer usato dai produttori e consumatori ha dimensioni maggiori di uno, e gli elementi vengono consumati nello stesso ordine in cui sono prodotti. Per memorizzare gli elementi, serve allora una coda (cioè un buffer FIFO, *First In First Out*).

Il codice seguente usa un array per realizzare una coda "artigianale", che è thread-safe (cioè non soggetta a race condition), ma non bloccante: se un thread prova fare un inserimento quando la coda è piena, o a estrarre un valore quando la coda è vuota, esso non rimane in attesa di poter effettivamente compiere tale operazione; invece, viene visualizzato un messaggio d'errore, e il programma termina. Spetterà quindi al codice che vi accede evitare inserimenti quando la coda è piena ed estrazioni quando essa è vuota.

```
public class CellaCondivisaCoda {
    private final int BUFFER_SIZE;
    private int numItems = 0;
    private int[] values;
    private int first = 0; // Indice dell'elemento meno recente
    private int last = 0; // Indice dopo l'elemento più recente
    // first == last corrisponde a una coda vuota

    public CellaCondivisaCoda(int bufferSize) {
        BUFFER_SIZE = bufferSize;
        values = new int[BUFFER_SIZE];
    }

    public int getItem() throws InterruptedException {
        ProdCons.mutex.acquire();
        if (numItems == 0) {
            System.err.println("Lettura di buffer vuoto!");
            System.exit(1);
        }
        numItems--;
        int tmp = values[first];
        first = (first + 1) % BUFFER_SIZE;
        System.out.println(tmp + " read");
        ProdCons.mutex.release();
        return tmp;
```

```
    }

    public void setItem(int v) throws InterruptedException {
        ProdCons.mutex.acquire();
        if (numItems == BUFFER_SIZE) {
            System.err.println("Scrittura di buffer pieno!");
            System.exit(1);
        }
        values[last] = v;
        last = (last + 1) % BUFFER_SIZE;
        numItems++;
        System.out.println(v + " written");
        ProdCons.mutex.release();
    }

    public int getCurrentSize() {
        return numItems;
    }
}
```

Osservazione: Per realizzare una coda, l'array è gestito in modo circolare: sia gli inserimenti che le estrazioni procedono nella direzione dall'inizio alla fine dell'array, e ripartono dall'inizio quando arrivano alla fine.

Il codice del main e dei due tipi di thread rimane simile alla versione con il buffer di capacità 1. A parte l'uso della classe `CellaCondivisaCoda` al posto di `CellaCondivisa`, le differenze principali sono che:

- vengono creati due produttori e due consumatori, (cosa che prima non si faceva perché avere produttori/consumatori multipli ha senso solo con un buffer di capacità superiore a 1);
- cambia il valore di inizializzazione del semaforo `empty`: come prima, esso è uguale alla capacità del buffer, e questa non è più 1 (bensì, ad esempio, 4).

```
import java.util.concurrent.Semaphore;

public class ProdCons {
    private static final int BUFFER_SIZE = 4;
    public static final Semaphore mutex = new Semaphore(1);
    public static final Semaphore full = new Semaphore(0);
    public static final Semaphore empty = new Semaphore(BUFFER_SIZE);

    public static void main(String[] args) {
        CellaCondivisaCoda cella = new CellaCondivisaCoda(BUFFER_SIZE);
```

```
        new Produttore(cella).start();
        new Produttore(cella).start();
        new Consumatore(cella).start();
        new Consumatore(cella).start();
    }
}

public class Produttore extends Thread {
    private CellaCondivisaCoda cella;

    public Produttore(CellaCondivisaCoda cella) {
        this.cella = cella;
    }

    public void run() {
        int i = 0;
        while (true) {
            try {
                ProdCons.empty.acquire();
                cella.setItem(i);
            } catch (InterruptedException e) { break; }
            ProdCons.full.release();
            i++;
        }
    }
}

public class Consumatore extends Thread {
    private CellaCondivisaCoda cella;

    public Consumatore(CellaCondivisaCoda cella) {
        this.cella = cella;
    }

    public void run() {
        while (true) {
            int v;
            try {
                ProdCons.full.acquire();
                v = cella.getItem();
            } catch(InterruptedException e) { break; }
            ProdCons.empty.release();
        }
    }
```

```
}
```

3 BlockingQueue

Poiché il problema del produttore-consumatore è molto comune, i progettisti di Java hanno messo a disposizione un'apposita struttura dati, corrispondente all'interfaccia `java.util.concurrent.BlockingQueue` (letteralmente "coda bloccante"). In particolare, si tratta di una struttura dati FIFO (appunto, una coda) thread-safe, che è in grado di mettere in attesa (bloccare) un produttore che cerca di inserire quando la coda è piena, o un consumatore che cerca di prelevare un valore quando questa è vuota, e può anche essere usata con produttori/consumatori multipli.

Siccome `BlockingQueue` è un'interfaccia, per usarla bisogna scegliere una delle sue implementazioni. Alcune delle principali sono:

- `ArrayBlockingQueue`: memorizza gli elementi in un array, e ha una capacità limitata;
- `LinkedBlockingQueue`: memorizza gli elementi in una linked list, con capacità illimitata o, opzionalmente, limitata.

Spesso, la `BlockingQueue` viene usata per inviare tra thread diversi dei "messaggi", rappresentati da istanze di una classe appositamente definita. Ad esempio, un semplice messaggio potrebbe essere

```
public class Message {
    private final String msg;

    public Message(String msg) {
        this.msg = msg;
    }

    public String getMsg() {
        return msg;
    }
}
```

che contiene semplicemente una stringa, la quale può essere estratta tramite un apposito metodo (ma non modificata). A scopo illustrativo, nella soluzione del problema del produttore-consumatore che verrà presentata successivamente, gli elementi saranno appunto istanze di `Message`.[1]

[1] La definizione della classe `Message` serve puramente a illustrare il concetto di "messaggio". Infatti, in una `BlockingQueue` possono essere inseriti valori di qualsiasi tipo (non primitivo), quindi si potrebbero usare semplicemente degli oggetti di tipo `String`, oppure `Integer` (se si volessero trattare ancora i numeri interi, come nelle soluzioni precedenti), o altro ancora.

3.1 Metodi

Tra i metodi di `BlockingQueue`, quelli corrispondenti al comportamento richiesto dal problema del produttore-consumatore sono

```
public void put(E e) throws InterruptedException;
public E take() throws InterruptedException;
```

(dove `E` indica il tipo degli elementi), che eseguono, rispettivamente, l'inserimento e la rimozione di un elemento, mettendo il thread chiamante in attesa se l'operazione non può essere effettuata subito.

Esistono però anche altri metodi, che si comportano in modi diversi quando l'operazione richiesta non può essere effettuata:

- `add(e)` e `remove()` sollevano un'eccezione;
- `offer(e)` e `poll()` restituiscono dei *valori speciali* (`false` e `null`, rispettivamente) che indicano l'esito dell'operazione;
- `offer(e, time, unit)` e `poll(time, unit)` possono bloccare il thread chiamante fino, al massimo, allo scadere del *timeout* specificato, dopodiché indicano l'esito dell'operazione mediante la restituzione di valori speciali.

È anche possibile ottenere ("esaminare") un elemento senza rimuoverlo, attraverso i metodi `element()` e `peek()`; se la coda è vuota, `element()` solleva un'eccezione, mentre `peek()` restituisce il valore speciale `null`.

Riassunto dei metodi di `BlockingQueue`

	Eccezione	Valore speciale	Blocca	Timeout
Inserisci	`add(e)`	`offer(e)`	`put(e)`	`offer(e, time, unit)`
Rimuovi	`remove()`	`poll()`	`take()`	`poll(time, unit)`
Esamina	`element()`	`peek()`		

3.2 Uso per il problema del produttore-consumatore

Usando una `BlockingQueue` (che sostituisce la classe `CellaCondivisa` delle soluzioni precedenti), il produttore e il consumatore possono essere implementati senza preoccuparsi di attendere che la coda sia non piena o non vuota: ciò viene gestito automaticamente dalla coda stessa. Il codice dei due thread diventa quindi:

```
import java.util.concurrent.BlockingQueue;

public class Producer extends Thread {
    private BlockingQueue<Message> queue;
```

```
    public Producer(BlockingQueue<Message> queue) {
        this.queue = queue;
    }

    public void run() {
        try {
            for (int i = 0; i < 100; i++) {
                Message msg = new Message(String.valueOf(i));
                // Simula il tempo di produzione del messaggio
                Thread.sleep(10);
                queue.put(msg);
                System.out.println("Produced " + msg.getMsg());
            }
        } catch (InterruptedException e) {}

        try {
            queue.put(new Message("exit"));
        } catch (InterruptedException e) {}
    }
}

public class Consumer extends Thread {
    private BlockingQueue<Message> queue;

    public Consumer(BlockingQueue<Message> queue) {
        this.queue = queue;
    }

    public void run() {
        try {
            while (true) {
                Message msg = queue.take();
                if (msg.getMsg().equals("exit")) { break; }
                // Simula il tempo di consumo del messaggio
                Thread.sleep(10);
                System.out.println("Consumed " + msg.getMsg());
            }
        } catch (InterruptedException e) {}
        System.out.println("Consumer finished");
    }
}
```

Infine, nel main, è sufficiente creare la coda (qui viene scelta una `ArrayBlockingQueue` con capacità 10) e avviare i vari thread (in quest'esempio, uno solo per tipo):

```java
import java.util.concurrent.ArrayBlockingQueue;
import java.util.concurrent.BlockingQueue;

public class ProdConsService {
    public static void main(String[] args) {
        BlockingQueue<Message> queue = new ArrayBlockingQueue<>(10);
        new Producer(queue).start();
        new Consumer(queue).start();
    }
}
```

Nota: In questa soluzione, è mostrato anche un meccanismo mediante il quale il produttore, dopo aver finito di produrre messaggi, indica al consumatore di terminare, inviando di un messaggio "speciale" che contiene la stringa `"exit"`. Allora, se si aggiungessero più consumatori, sarebbe necessario emettere tanti messaggi `"exit"` quanti siano i consumatori, dato che ciascun messaggio verrebbe ricevuto da un consumatore solo (perché la lettura comporta la rimozione dalla coda).