Azzolini Riccardo 2020-03

Problema del produttore-consumatore

1 Problema

Il problema del **produttore-consumatore**, già introdotto nella lezione precedente, è caratterizzato da due task, il **produttore** e il **consumatore**, che comunicano attraverso un **buffer**, il quale ha una capacità limitata:

- il produttore genera dati e li deposita nel buffer;
- il consumatore utilizza i dati prodotti, rimuovendoli di volta in volta dal buffer;
- i due task producono e consumano continuamente, ma con una frequenza variabile e non nota a priori (quindi la soluzione non si potrà basare su una conoscenza di quando i due task effettueranno le loro operazioni).

Gli aspetti da gestire in questa situazione sono:

- l'accesso concorrente al buffer (che costituisce una sezione critica, perciò si applicano semplicemente le tecniche di mutua esclusione, già viste);
- il comportamento del produttore quando il buffer è pieno;
- il comportamento del consumatore quando il buffer è vuoto.

2 Schema della soluzione

Quando il buffer è pieno, il produttore non ha spazio per inserire dati, quindi deve sospendere la sua esecuzione. Successivamente, quando il consumatore preleva un elemento dal buffer pieno, liberando uno spazio, esso "sveglia" il produttore, che potrà allora ricominciare a depositare elementi nel buffer.

Analogamente, il consumatore si deve sospendere quando il buffer è vuoto, poiché non ci sono dati da prelevare, e viene poi risvegliato da un produttore, dopo che quest'ultimo ha depositato un elemento nel buffer vuoto.

Questa soluzione può essere implementata tramite le primitive di comunicazione tra thread: `synchronized`, `wait()`, `notify()` e `notifyAll()`. Bisogna però fare attenzione a non introdurre race condition, deadlock, o anche semplicemente errori di logica applicativa.

3 Esempio di implementazione errata

Un esempio di soluzione errata è quella realizzata a partire dalla seguente classe:

```
public class CellaCondivisa {
    private int numItems = 0;
    private int value;

    public synchronized int getItem() {
        numItems--;
        return value;
    }

    public synchronized void setItem(int v) {
        value = v;
        numItems++;
    }
}
```

Questa classe rappresenta un buffer di capacità 1, dato che la singola variabile `value` può memorizzare solo un elemento. Siccome i metodi sono `synchronized`, non ci sono race condition su tale variabile.

Si realizzano poi i thread di produttore e consumatore, che condivideranno un'istanza di `CellaCondivisa`:

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

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

    public void run() {
        for (int i = 1; i <= 10; i++) {
            cella.setItem(i);
            System.out.println("P: prodotto " + i);
        }
        System.out.println("P terminato");
    }
}

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

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

    public void run() {
        for (int i = 1; i <= 10; i++) {
            int v = cella.getItem();
            System.out.println("C: consumato " + v);
        }
        System.out.println("C terminato");
    }
}

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

Il difetto di questa soluzione è che:

- il produttore non verifica se il buffer è pieno prima di effettuare un inserimento, rischiando così di sovrascrivere un valore già presente;
- il consumatore non controlla se il buffer è vuoto, e di conseguenza potrebbe prelevare un valore già letto.

Ad esempio, il seguente output mostra che si sono verificati entrambi i problemi:

```
P: prodotto 1
P: prodotto 2
P: prodotto 3
P: prodotto 4
C: consumato 1
P: prodotto 5
C: consumato 5
P: prodotto 6
C: consumato 6
P: prodotto 7
C: consumato 7
P: prodotto 8
C: consumato 8
P: prodotto 9
```

```
C: consumato 9
P: prodotto 10
C: consumato 10
P terminato
C: consumato 10
C: consumato 10
C: consumato 10
C terminato
```

Osservazione: In questo esempio di output, sembra che il consumatore abbia prelevato il valore 1 dopo che il produttore ha scritto 4. Ciò sarebbe impossibile, poiché il buffer ha spazio per un solo elemento: in realtà, l'1 è stato consumato prima che venisse prodotto il 2, e solo la stampa "consumato 1" è avvenuta dopo la produzione di ulteriori valori (per motivi di scheduling dei thread e/o buffering dell'output). Questo particolare non rappresenta quindi un errore.

4 Esempio di implementazione corretta

Per ottenere un'implementazione corretta, si definisce la classe `CellaCondivisa` in modo che:

- realizzi la mutua esclusione (come prima);
- regoli automaticamente l'accesso dei task produttore e consumatore in base al numero di elementi disponibili, mettendoli in attesa quando è opportuno, senza che questi neanche "se ne accorgano".

A scopo illustrativo, viene aggiunto anche un metodo che restituisce il numero di elementi attualmente nel buffer; essendo di sola lettura, questo non ha bisogno di essere `synchronized`.

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

    public synchronized int getItem() throws InterruptedException {
        if (numItems == 0) {
            // Se non c'è niente da leggere,
            // chi cerca di farlo viene fermato.
            wait();
        }
        numItems--;
        // Siccome il buffer non è più pieno,
```

```
        // viene svegliato un eventuale produttore in attesa.
        notify();
        return value;
    }

    public synchronized void setItem(int v) throws InterruptedException {
        if (numItems == BUFFER_SIZE) {
            // Se il buffer è pieno,
            // chi cerca di scrivere va in attesa.
            wait();
        }
        value = v;
        numItems++;
        // Adesso il buffer non è più vuoto,
        // quindi si sveglia un eventuale consumatore in attesa.
        notify();
    }

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

In questo modo, l'unica modifica da apportare al codice dei thread rispetto alla soluzione precedente è l'aggiunta della gestione delle `InterruptedException`, e il main rimane invariato:

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

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

    public void run() {
        for (int i = 1; i <= 10; i++) {
            try {
                cella.setItem(i);
            } catch(InterruptedException e) { break; }
            System.out.print(" P" + i);
        }
    }
}
```

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

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

    public void run() {
        for (int i = 1; i <= 10; i++) {
            int v;
            try {
                v = cella.getItem();
            } catch(InterruptedException e) { break; }
            System.out.print(" C" + v);
        }
    }
}

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

Grazie ai meccanismi di sincronizzazione implementati da `CellaCondivisa`, e siccome il buffer ha capacità 1, nell'esecuzione di questo programma si alternano per forza una produzione e un consumo. Ci si aspetterebbe allora che l'unico possibile output sia il seguente:

```
P1 C1 P2 C2 P3 C3 P4 C4 P5 C5 P6 C6 P7 C7 P8 C8 P9 C9 P10 C10
```

In realtà, ne sono possibili anche altri, a causa degli effetti di "riordinamento" delle stampe causati dallo scheduling e/o dal buffering dell'output. Può ad esempio sembrare che vengano prodotti/consumati più elementi di fila, e addirittura che un elemento venga consumato prima di essere prodotto:

```
C1 P1 P2 P3 C2 C3 P4 C4 P5 C5 P6 C6 P7 C7 P8 C8 P9 C9 P10 C10
```

Anche un output di questo tipo corrisponde a un'esecuzione corretta: si osserva infatti che ogni elemento prodotto è stato consumato una e una sola volta. Per visualizzare il "vero" ordine di esecuzione delle operazioni, si possono aggiungere delle stampe all'interno delle sezioni critiche, in modo che esse vengano sempre eseguite insieme letture/scritture:

```
public synchronized int getItem() throws InterruptedException {
    if (numItems == 0) {
        wait();
    }
    numItems--;
    System.out.print(" Out");
    notify();
    return value;
}

public synchronized void setItem(int v) throws InterruptedException {
    if (numItems == BUFFER_SIZE) {
        wait();
    }
    value = v;
    numItems++;
    System.out.print(" In");
    notify();
}
```

Così, si vede che, effettivamente, non ci sono mai due "In" o due "Out" consecutivi, nonostante l'ordine delle stampe "Px" e "Cx":

```
In Out P1 In C1 Out P2 In P3 C2 Out In C3 Out P4 In P5 C4 Out In
C5 Out P6 In C6 Out C7 P7 In P8 Out C8 In P9 Out In P10 C9 Out C10
```

4.1 Posizione delle chiamate `wait` e `notify`

È molto importante stare attenti a dove si fanno le chiamate `wait` e `notify`. In generale, valgono le seguenti regole:

- chiamare `wait` prima di aver effettuato qualsiasi operazione sull'oggetto (cioè, di solito, all'inizio di una sezione critica), perché, mentre si è in attesa, viene rilasciato il lock, e quindi altri thread possono accedere all'oggetto;
- chiamare `notify` dopo aver completato le operazioni sull'oggetto (quindi, solitamente, subito prima di uscire dalla sezione critica).

Ad esempio, la seguente implementazione dei metodi di `CellaCondivisa` è incorretta a causa di un posizionamento errato di queste chiamate:

```
public synchronized int getItem() throws InterruptedException {
    if (numItems == 0) {
        wait();
```

```
        notify();
    }
    numItems--;
    return value;
}

public synchronized void setItem(int v) throws InterruptedException {
    if (numItems == BUFFER_SIZE) {
        wait();
        notify();
    }
    value = v;
    numItems++;
}
```

Eseguendo questa versione del programma, si possono infatti manifestare dei deadlock:

```
P1 C1 <deadlock>
P1 C1 P2 P3 C2 C3 <deadlock>
```

Per capire perché il programma si blocca, è utile aggiungere delle stampe all'interno delle sezioni critiche:

```
public synchronized int getItem() throws InterruptedException {
    System.out.print(" Entered get");
    if (numItems == 0) {
        System.out.print(" Going to wait on get");
        wait();
        System.out.print(" Going to notify in get");
        notify();
    }
    numItems--;
    System.out.print(" Exiting get");
    return value;
}

public synchronized void setItem(int v) throws InterruptedException {
    System.out.print(" Entered set");
    if (numItems == BUFFER_SIZE) {
        System.out.print(" Going to wait on set");
        wait();
        System.out.print(" Going to notify in set");
        notify();
    }
```

```
        value = v;
        numItems++;
        System.out.print(" Exiting set");
    }
```

Un possibile output è

```
Entered set Exiting set P1 Entered set Going to wait on set
Entered get Exiting get C1 Entered get Going to wait on get
```

che mostra che il deadlock è avvenuto perché:

1. il produttore ha eseguito `setItem(1)`, riempiendo il buffer, poi ha provato subito a eseguire `setItem(2)`, e, trovando il buffer pieno, si è messo in attesa;
2. successivamente, il consumatore ha eseguito `getItem()`, prelevando l'unico elemento disponibile (1), senza eseguire né `wait` né `notify` (dato che il buffer non era vuoto), e poi ha rieseguito immediatamente `getItem()`, trovando il buffer vuoto e mettendosi quindi in attesa;
3. a questo punto, i due task si sono trovati in attesa l'uno dell'altro.

L'errore di questa soluzione è infatti quello di aver inserito la chiamata `notify` nello stesso `if` che contiene la `wait`. Così:

- un produttore che trova subito il buffer vuoto e vi inserisce un elemento, senza bisogno di aspettare, non esegue la `notify`, e perciò un eventuale consumatore rimarrà in attesa, nonostante sia in realtà diventato disponibile un elemento da consumare;
- analogamente, un consumatore che consuma dal buffer pieno senza prima bloccarsi non sbloccherà un eventuale produttore in attesa, nonostante abbia reso disponibile uno spazio in cui inserire un elemento.

4.2 Buffer di dimensioni maggiori

Finora, il buffer `CellaCondivisa` conteneva un solo elemento. Se si ingrandisce il buffer, aumentando la costante `BUFFER_SIZE`,[1] ad esempio a 4,

[1] Per quest'esempio, `CellaCondivisa` rimane capace di memorizzare un solo valore: in pratica, la dimensione è "aumentata" solo ai fini della sincronizzazione. Memorizzare concretamente più valori è comunque abbastanza semplice: basta usare una qualsiasi struttura dati adeguata (tipicamente una coda).

```
public class CellaCondivisa {
    private static final int BUFFER_SIZE = 4;
    // ...
}
```

potranno essere effettuati fino a 4 “In” consecutivi (e di conseguenza sono possibili anche, al massimo, 4 “Out” consecutivi):

```
In Out P1 In P2 In P3 In C1 Out C4 Out C4 Out C4 P4 In P5 In P6 In
P7 In P8 Out C8 Out C8 Out C8 Out C8 In P9 In P10 Out C10 Out C10
```

5 Più produttori e consumatori

Per esaminare lo scenario in cui ci sono tanti thread che producono e tanti che consumano, tutti usando lo stesso buffer (che deve avere capienza maggiore di 1, altrimenti l’esecuzione sarebbe serializzata), si effettuano alcune modifiche al codice dei thread e del main. Le principali sono:

- ogni costruttore riceve come argomento aggiuntivo un nome da assegnare al thread, che verrà poi usato nelle stampe, in modo da poter distinguere i vari thread dello stesso tipo;
- i cicli di produzione e consumo diventano infiniti, per aumentare il numero di interazioni tra thread;
- vengono creati due thread produttori e due consumatori.

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

    public Produttore(String name, CellaCondivisa cella) {
        super(name);
        this.cella = cella;
    }

    public void run() {
        int i = 1;
        while (true) {
            try {
                cella.setItem(i);
            } catch(InterruptedException e) { break; }
            i++;
        }
    }
```

```
}

public class Consumatore extends Thread {
    private CellaCondivisa cella;

    public Consumatore(String name, CellaCondivisa cella) {
        super(name);
        this.cella = cella;
    }

    public void run() {
        while (true) {
            int v;
            try {
                v = cella.getItem();
            } catch(InterruptedException e) { break; }
        }
    }
}

public class ProdCons {
    public static void main(String[] args) {
        CellaCondivisa cella = new CellaCondivisa();
        new Produttore("p1", cella).start();
        new Produttore("p2", cella).start();
        new Consumatore("c1", cella).start();
        new Consumatore("c2", cella).start();
    }
}
```

5.1 Cella condivisa errata

L'implementazione di `CellaCondivisa` usata in precedenza,

```
public class CellaCondivisa {
    private static final int BUFFER_SIZE = 4;
    private int numItems = 0;
    private int value;

    public synchronized int getItem() throws InterruptedException {
        if (numItems == 0) {
            wait();
        }
```

```
        numItems--;
        System.out.println(
            Thread.currentThread().getName() + ": " + value + " read"
        );
        notify();
        return value;
    }

    public synchronized void setItem(int v) throws InterruptedException {
        if (numItems == BUFFER_SIZE) {
            wait();
        }
        value = v;
        System.out.println(
            Thread.currentThread().getName() + ": " + v + " written"
        );
        numItems++;
        notify();
    }
}
```

era corretta con un solo produttore e un solo consumatore, ma *non* lo è in presenza di produttori/consumatori multipli. In particolare, nonostante la capacità del buffer sia 4, possono essere scritti e/o letti più di 4 elementi di fila, come mostrato nel seguente esempio di output:

```
...
p1: 5 written
c1: 5 read
c2: 5 read
c1: 5 read
c1: 5 read
c1: 5 read
c1: 5 read
...
c1: 5 read
c2: 5 read
p1: 6 written
p1: 7 written
p1: 8 written
p1: 9 written
p1: 10 written
...
p1: 22 written
```

```
p2: 18 written
p2: 19 written
...
p2: 25 written
c2: 25 read
c2: 25 read
c2: 25 read
c2: 25 read
p1: 23 written
...
```

Questi errori si hanno perché un thread svegliato da una `notify` non torna immediatamente in esecuzione, ma diventa semplicemente ready, e nel frattempo ne possono essere schedulati altri. Ad esempio, se il consumatore `c1` va in attesa perché il buffer è vuoto, e poi viene svegliato (da un produttore che ha inserito un elemento), potrebbe capitare che venga schedulato prima un *altro* consumatore, `c2`, e questo "ruberebbe" allora l'elemento disponibile nel buffer. Ormai, però, `c1` è stato svegliato: quando verrà schedulato, la sua esecuzione proseguirà come se fosse ancora disponibile un elemento nel buffer, e perciò esso leggerà il valore già prelevato da `c2`. La soluzione consisterà allora nel fare in modo che, in questo caso, `c1` torni in attesa dopo essere stato svegliato.

5.2 Cella condivisa corretta

Per rendere la classe `CellaCondivisa` adatta all'uso con produttori/consumatori multipli, è necessario sostituire gli `if` usati per l'attesa con dei `while`: così, se la condizione di risveglio viene "annullata" da un altro thread mentre il thread che è stato svegliato aspetta di essere schedulato, quest'ultimo ricontrollerà la condizione e tornerà subito in attesa.

Inoltre, siccome possono esserci contemporaneamente molti thread in attesa, le chiamate `notify` vengono sostituite con `notifyAll`, per garantire di svegliarli sempre tutti ogni volta che cambia lo stato del buffer: alcuni di essi riusciranno concretamente a eseguire, mentre altri torneranno subito in attesa (grazie ai cicli `while`). Questo è il modo più semplice di essere sicuri che non ci sia il rischio di svegliare solo un thread "sbagliato", che in realtà non potrebbe proseguire, e tornerebbe quindi in attesa, potenzialmente provocando un deadlock.

```
public class CellaCondivisa {
    private static final int BUFFER_SIZE = 4;
    private int numItems = 0;
    private int value;

    public synchronized int getItem() throws InterruptedException {
```

```
        while (numItems == 0) {
            wait();
        }
        numItems--;
        System.out.println(
            Thread.currentThread().getName() + ": " + value + " read"
        );
        notifyAll();
        return value;
    }

    public synchronized void setItem(int v) throws InterruptedException {
        while (numItems == BUFFER_SIZE) {
            wait();
        }
        value = v;
        System.out.println(
            Thread.currentThread().getName() + ": " + v + " written"
        );
        numItems++;
        notifyAll();
    }
}
```