

Sistemi Operativi

(modulo di Informatica II)

Laboratorio

Sincronizzazione in Java (Semafori)

Patrizia Scandurra

Università degli Studi di Bergamo

La sincronizzazione in Java

- **Metodi di sincronizzazione**
 1. **I semafori** -- `java.util.concurrent`
 2. **Monitor mediante metodi/blocchi sincronizzati** e segnali *wait/notify/notifyAll* -- a livello di linguaggio (modificatore *synchronized*)
 3. **Monitor mediante oggetti lock e condizioni** -- `java.util.concurrent`

(Non in programma!)

- Le barriere
- Variabili atomiche

I semafori

- Variabile intera per ottenere/rilasciare risorse in ambito concorrente
 - semaforo **binario** (0 o 1) -- *mutex lock*; istanza singola di una risorsa, ad esempio la *sezione critica*
 - semaforo **generalizzato** (contatore) -- il valore può variare in un dominio senza restrizioni; istanze multiple di una risorsa
- Un semaforo S è manipolato dalle funzioni atomiche:
 - *acquire*(S) → acquisisce l'uso della risorsa
 - *release*(S) → rilascia la risorsa

Protocollo di accesso alla sezione critica con un semaforo binario

Semaphore S; // Assumendo che sia
// inizializzato ad 1 (risorsa libera)

acquire(S);

criticalSection();

release(S);

I semafori in Java

- Oggetti **java.util.concurrent.Semaphore**
- Ogni semaforo è un contatore di permessi: consente di definire il numero iniziale di accessi consentiti: 1 se *semaforo binario*, un valore > 1 se *semaforo generalizzato*
- Ogni semaforo possiede una coda dei processi in attesa di accedere alla risorsa
- Operazioni sui semafori
 - **acquire**: sospende il processo in esecuzione in caso di risorsa non disponibile e lo inserisce nella coda di attesa del semaforo
 - **release**: rilascia la risorsa e riattiva un processo della coda di attesa cedendogli la risorsa

I Semafori: `java.util.concurrent.Semaphore`

Vedi anche

<https://docs.oracle.com/javase/8/docs/api/java/util/concurrent/Semaphore.html>

Constructor Summary

`Semaphore`(int permits) Creates a Semaphore with the given number of permits and nonfair fairness setting.

Method Summary

void	<code>acquire</code> (int permits) Acquires the given number of permits from this semaphore, blocking until all are available, or the thread is <code>interrupted</code> .
int	<code>availablePermits</code> () Returns the current number of permits available
void	<code>release</code> (int permits) Releases the given number of permits.
boolean	<code>tryAcquire</code> (int permits) Acquires the given number of permits from this semaphore, only if all are available at the time of invocation.
boolean	<code>tryAcquire</code> (long timeout, TimeUnit unit) Acquires a permit from this semaphore, if one becomes available within the given waiting time and the current thread has not been interrupted.

Uso del semaforo binario in Java

Protocollo di accesso alla sezione critica

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

....

Semaphore sem = new Semaphore(1);

try {
    sem.acquire();
    // critical section
}
catch (InterruptedException ie) { }
finally {
    sem.release();
}
```

Problema del buffer limitato – soluzione tramite semafori in Java

- Un buffer di N locazioni (capacità), ciascuna in grado di contenere un oggetto
- Un semaforo **mutex** inizializzato ad 1 che garantisce la mutua esclusione nell'accesso al buffer
- Un semaforo **full** inizializzato a 0
- Un semaforo **empty** inizializzato al valore N

Buffer limitato in Java

```
public class BoundedBuffer implements Buffer
{
    private static final int BUFFER_SIZE = 5;
    private Object[] buffer;
    private int in, out;
    private Semaphore mutex;
    private Semaphore empty;
    private Semaphore full;

    public BoundedBuffer() {
        // buffer is initially empty
        in = 0;
        out = 0;
        buffer = new Object[BUFFER_SIZE];

        mutex = new Semaphore(1);
        empty = new Semaphore(BUFFER_SIZE);
        full = new Semaphore(0);
    }

    public void insert(Object item) {
        // Figure 6.9
    }

    public Object remove() {
        // Figure 6.10
    }
}
```

Buffer limitato in Java

Metodo `insert(item)` – eseguito dal produttore

```
public void insert(Object item) {  
    empty.acquire();  
    mutex.acquire();  
  
    // add an item to the buffer  
    buffer[in] = item;  
    in = (in + 1) % BUFFER_SIZE;  
  
    mutex.release();  
    full.release();  
}
```

Buffer limitato in Java

Metodo `remove()` – eseguito dal consumatore

```
public Object remove() {  
    full.acquire();  
    mutex.acquire();  
  
    // remove an item from the buffer  
    Object item = buffer[out];  
    out = (out + 1) % BUFFER_SIZE;  
  
    mutex.release();  
    empty.release();  
  
    return item;  
}
```

..

Buffer limitato in Java

- La struttura del thread produttore

```
public class Producer implements Runnable
{
    private Buffer buffer;

    public Producer(Buffer buffer) {
        this.buffer = buffer;
    }

    public void run() {
        Date message;

        while (true) {
            // nap for awhile
            SleepUtilities.nap();
            // produce an item & enter it into the buffer
            message = new Date();
            buffer.insert(message);
        }
    }
}
```

Buffer limitato in Java

- La struttura del thread consumatore

```
public class Consumer implements Runnable
{
    private Buffer buffer;

    public Consumer(Buffer buffer) {
        this.buffer = buffer;
    }

    public void run() {
        Date message;

        while (true) {
            // nap for awhile
            SleepUtilities.nap();
            // consume an item from the buffer
            message = (Date)buffer.remove();
        }
    }
}
```

Buffer limitato in Java

- La classe principale

```
public class Factory
{
    public static void main(String args[]) {
        Buffer buffer = new BoundedBuffer();

        // now create the producer and consumer threads
        Thread producer = new Thread(new Producer(buffer));
        Thread consumer = new Thread(new Consumer(buffer));

        producer.start();
        consumer.start();
    }
}
```

Vedi applicazione completa in <boundedbuffer>

Prima degli esercizi

- Domande da porsi per **concepire una buona soluzione ad un dato problema di concorrenza**
 - Chi sono i **thread** (tipi di thread e ruoli lettori/scrittori)?
 - Chi sono le **risorse** (tipi e numeri) condivise dai thread?
 - Come realizziamo la **mutua esclusione** alla sezione critica (risorsa primaria di ogni problema di concorrenza!)?
 - accesso esclusivo mediante un semaforo *mutex*
 - E' necessario **coordinare** le mosse dei **thread** per accedere alle risorse?
 - attesa/risveglio dei thread mediante operazioni *acquire/release* su semafori

Esercizio 1

- **Contatore sincronizzato.** Si realizzi una applicazione Java che permetta a più thread di operare contemporaneamente su un *contatore* (oggetto di classe `Counter`) sfruttando i semafori. A tale scopo:
 - Si definisca la classe `Counter`. Un contatore (oggetto della classe `Counter`) è inizialmente a zero ed è in grado di contare fino a 10. La classe deve anche contenere dei metodi `incrementa()` / `decrementa()` per consentire di incrementare/decrementare di una unità il contatore; le operazioni di incremento/decremento devono essere possibili solo se mantengono rispettivamente il conteggio non superiore a 10/non negativo.
 - Si definiscano le classi dei thread che accedono ad uno stesso contatore. Ci sono due tipi di thread: 1. i thread con ruolo “TaskI” che all’interno di un ciclo infinito si alternano tra dormire un pò (un certo num. random di ms) e incrementare il contatore; 2. e thread con ruolo “TaskD” che all’interno di un ciclo infinito si alternano tra dormire un pò (un certo num. random di ms) e decrementare il contatore.
 - Per testare il corretto funzionamento dell’oggetto di classe `Counter`, si preveda un programma principale (una classe con metodo `main()`) che istanzi un certo numero di thread `TaskI` e `TaskD`.

Esercizio 2

Coke Machine:

Si implementi un programma Java per risolvere il problema di prelevare lattine di coca-cola da una macchinetta e di rifornirla nel caso in cui rimanga vuota. Si utilizzino i semafori come meccanismo di sincronizzazione. In particolare:

- Definire le classi per i thread con ruolo “Utente” e “Rifornitore”.
- Definire la classe `CokeMachine` contenente le lattine di coca-cola ed i metodi:
 - `preleva(...)`, eseguito dal generico utente per prelevare una lattina dalla macchinetta
 - `rifornisci(...)`, eseguito dal fornitore del servizio per caricare la macchinetta nel caso in cui rimane vuota
- Si assumi che inizialmente la macchinetta sia piena, e che un utente (a scelta: il primo a trovare la macchinetta vuota o l'utente che preleva l'ultima lattina) segnala al fornitore che la macchinetta è vuota, il quale a seguito di tale comunicazione provvede ad effettuare il rifornimento.

- Come piccolo aiuto, lo scheletro della classe di una possibile soluzione con semafori è riportato di seguito

```
import java.util.concurrent.Semaphore;
class CokeMachine{
    static final int N = 50; //Capacità della macchinetta
    int count ; //Numero effettivo di lattine presenti
    private Semaphore mutex; //binario, per la mutua
                                //esclusione

    private Semaphore empty; //semaforo binario per
    //segnalare al rifornitore che la macchinetta è vuota
    ... <COMPLETARE>...
}
```