SISTEMI OPERATIVI

(MODULO DI INFORMATICA II)

Sincronizzazione in Java

(Semafori e barriere)

Patrizia Scandurra

Università degli Studi di Bergamo

a.a. 2012-13

La gestione dei thread in Java

- Creazione e avvio
- Terminazione e cancellazione dei thread
- La gestione dei segnali
- **Sincronizzazione**
  - **I semafori**
  - **Le barriere**
- Dati specifici dei thread
- Schedulazione
- Gruppi di thread

Riassumendo...

Meccanismi di sincronizzazione in Java

- **Monitor** (a livello di linguaggio con il modificatore synchronized)
  - **Sincronizzazione indiretta** con metodi/blocchi sincronizzati
  - **Sincronizzazione diretta** metodi/blocchi sincronizzati + segnali wait/notify/notifyAll
- **Monitor con variabili condizione** (java.utili.concurrent)
  - **lock** + segnali cond.await() e cond.signal()
- **Semafori, barriere** (java.util.concurrent)
  - Meccanismi più semplici, per problemi semplici

Protocollo di accesso alla sezione critica: 1. e 2. a confronto

Rule 1: Instead of using `synchronized` keyword, use `Lock.lock()` and `Lock.unlock()`

Existing API	New API
`Object monitorObject;` `synchronized(monitorObject){` `  //critical section` `}`	`Lock lockObject;` `try{` `  lockObject.lock();` `  //critical section` `}` `finally{` `  lockObject.unlock()` `}`

blocco sincronizzato (similmente con i **metodi sincronizzati**, contrassegnando come synchronized i metodi della classe "specifica" di monitorObject)

Cooperazione all'interno della sezione critica - 1. e 2. a confronto

Rule 2: Instead of using `wait()` and `notify()` in the critical section use `await()` and `signal()` on condition variables

Existing API	New API
`` //////////////////////////////// //Inside your critical section:// //////////////////////////////// boolean somecondition; //evaluate your wait criteria while(somecondition){ wait(); //re-evaluate somecondition } ```	``` //////////////////////////////// //Outside your critical section:// //////////////////////////////// Condition conditionVariable = lockObject.newCondition(); //////////////////////////////// //Inside your critical section:// //////////////////////////////// boolean somecondition; //evaluate your wait criteria while(somecondition){ conditionVariable.await(); //re-evaluate somecondition } ```
`` //////////////////////////////// //Inside of your critical section:// //////////////////////////////// boolean someothercondition; //evaluate your notify criteria if(someothercondition) { notify(); } ```	``` //////////////////////////////// //Inside your critical section:// //////////////////////////////// boolean someothercondition; //evaluate your signal criteria if(someothercondition) { conditionVariable.signal(); } ```

I semafori

- Richiamiamo il concetto di semaforo:
  - Variabile intera
    - semaforo **binario** (0 o 1) -- ***mutex lock***
    - semaforo **generalizzato** (contatore) -- il valore può variare in un dominio senza restrizioni

- Un semaforo *S* è manipolato dalle funzioni:
  - ***acquire(S)*** → acquisisce l'uso della risorsa
  - ***release(S)*** → rilascia la risorsa

sono operazioni *atomiche*

Protocollo di **accesso alla sezione critica** con un semaforo (binario)

```
Semaphore S;        //Assumendo che sia
                    //inizializzato ad 1

acquire(S);
criticalSection();
release(S);
```

I semafori in Java

- Un semaforo può essere facilmente definito uno usando i meccanismi di sincronizzazione di base in Java
  - Ad es. con una implementazione con **sospensione e rischedulazione**

- Tuttavia i semafori e altri meccanismi di sincronizzazione (lock, barriere, ecc..) sono forniti da Java JS2E 5.0 in poi nel package **java.util.concurrent**

Implementazione dei semafori – con sospensione e rischedulazione

- Strutture dati: ogni semaforo S ha una **coda dei processi in attesa** di acquisire la risorsa
- Metodo *acquire(S)*: sospende il processo in esecuzione in caso di risorsa non disponibile e lo inserisce nella coda di attesa del semaforo
- Metodo *release(S)*: rilascia la risorsa e riattiva il primo processo della coda di attesa cedendogli la risorsa
- Lo schedulatore dei processi in attesa della risorsa
  - definisce l'ordine di ottenimento della risorsa in base alla politica adottata per il semaforo

La classe Semaphore in Java

```
public class Semaphore{

  private int value;
  public Semaphore() {
            value = 1;
  }

  public Semaphore(int value) {
            this.value = value;
  }
 …
```

La classe Semaphore in Java (Cont.)

```
 public synchronized void acquire() {
      while (value == 0)
      try {
            wait();
      } catch (InterruptedException ie) { }
      value--;
 }

 public synchronized void release() {
      ++value;
      notify();
 }
}
```

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

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

Method Summary	
void	**acquire**(int permits) Acquires the given number of permits from this semaphore, blocking until all are available, or the thread is interrupted.
int	**availablePermits**() Returns the current number of permits available in this semaphore.
void	**release**(int permits) Releases the given number of permits, returning them to the semaphore.
boolean	**tryAcquire**(int permits) Acquires the given number of permits from this semaphore, only if all are available at the time of invocation.
boolean	**tryAcquire**(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.

http://java.sun.com/javase/6/docs/api/java/util/concurrent/Semaphore.html

Uso del semaforo binario in Java 5

- Protocollo di accesso alla sezione critica

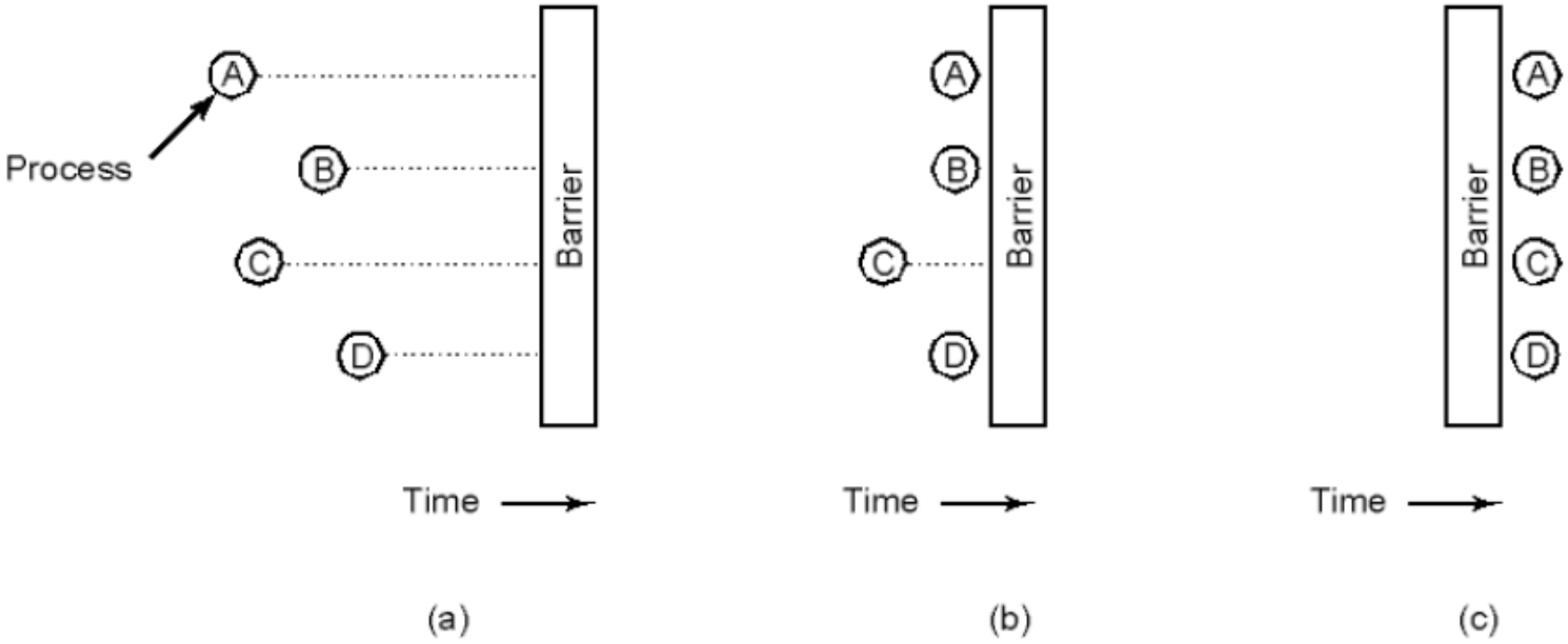
```
Semaphore sem = new Semaphore(1);

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

Problema del buffer limitato – soluzione tramite semafori in Java

- Strutture dati necessarie:
  - Un buffer di N locazioni (capacità), ciascuna in grado di contenere un oggetto
  - Un semaforo binario 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
- Vedi demo 1

Barriera

- Meccanismo di sincronizzazione che forza tutti i thread ad aspettare finchè tutti non raggiungono un determinato punto



Usare CyclicBarrier

- Costruire **l'oggetto CyclicBarrier** (la **barriera**) specificando:
  - Il numero di thread che partecipano all'operazione parallela;
  - (opzionale) una routine di "merge" da eseguire alla fine di ogni ciclo (iterazione) per combinare i risultati parziali
- Ad ogni iterazione, ogni thread:
  - esegue una porzione del suo lavoro;
  - una volta finito, invoca il metodo await() sulla barriera;
  - Il metodo await() ritorna **solo quando**:
    - tutti i thread hanno invocato await();
    - L'eventuale routine di merge è stata avviata (la barriera la invoca sull'ultimo thread che invoca await() prima di rilasciare i thread in attesa)
- Se un thread viene interrotto o supera il time out di attesa per la barriera, allora la barriera viene "interrotta" bruscamente e tutti gli altri thread in attesa ricevono una **BrokenBarrierException**

`java.util.concurrent.CyclicBarrier` (1)

- Consente di fermare e sincronizzare dei thread presso una *barriera* in attesa che tutti i partecipanti la raggiungono
- Utile per la decomposizione di problemi in sottoproblemi da risolvere concorrentemente

Constructor Summary
CyclicBarrier(int parties) Creates a new CyclicBarrier that will trip when the given number of parties (threads) are waiting upon it, and does not perform a predefined action when the barrier is tripped.
CyclicBarrier(int parties, Runnable barrierAction) Creates a new CyclicBarrier that will trip when the given number of parties (threads) are waiting upon it, and which will execute the given barrier action when the barrier is tripped, performed by the last thread entering the barrier.

`CyclicBarrier` (2)

- I sotto-problemi si affidano a thread distinti
- La barriera serve per sincronizzarsi sulla loro fine e ricomporne i risultati

Method Summary	
int	**await**() Waits until all parties have invoked await on this barrier.
int	**await**(long timeout, TimeUnit unit) Waits until all parties have invoked await on this barrier, or the specified waiting time elapses.
int	**getNumberWaiting**() Returns the number of parties currently waiting at the barrier.
int	**getParties**() Returns the number of parties required to trip this barrier.
boolean	**isBroken**() Queries if this barrier is in a broken state (se uno dei thread associati alla bariera ha superato il timeout o è stato interrotto).
void	**reset**() Resets the barrier to its initial state.

CyclicBarrier: un esempio

- Vedi demo 2