
UNIVERSITY INSTITUTE - ISCTE - INSTITUTO UNIVERSITÁRIO - LISBOA -
PROGRAMAÇÃO
CONCORRENTE E DISTRIBUÍDA
ESTRUTURAS DE COORDENAÇÃO
VEM ESTUDAR CONNOSCO
WWW.ISCTE-IUL.PT

Sumário

- Variáveis condicionais
- Semáforo
- Barreira
- Fila Bloqueante (BlockingQueue)
- Problemas avançados da sincronização

Variáveis Condicionais

- A thread acquires the lock associated with the condition variable.
- Tests the condition while owning the lock.
- If the condition is true it performs the task and releases the lock when it is done.

Variáveis Condicionais

- If the condition is false the lock is released and the thread waits for a change on the condition.
- When another thread changes the logical value of the condition and releases the lock, one of the waiting threads is notified (woken up) so that it can test the condition again.
- When a thread wakes up *it has to test the condition again*. Why?

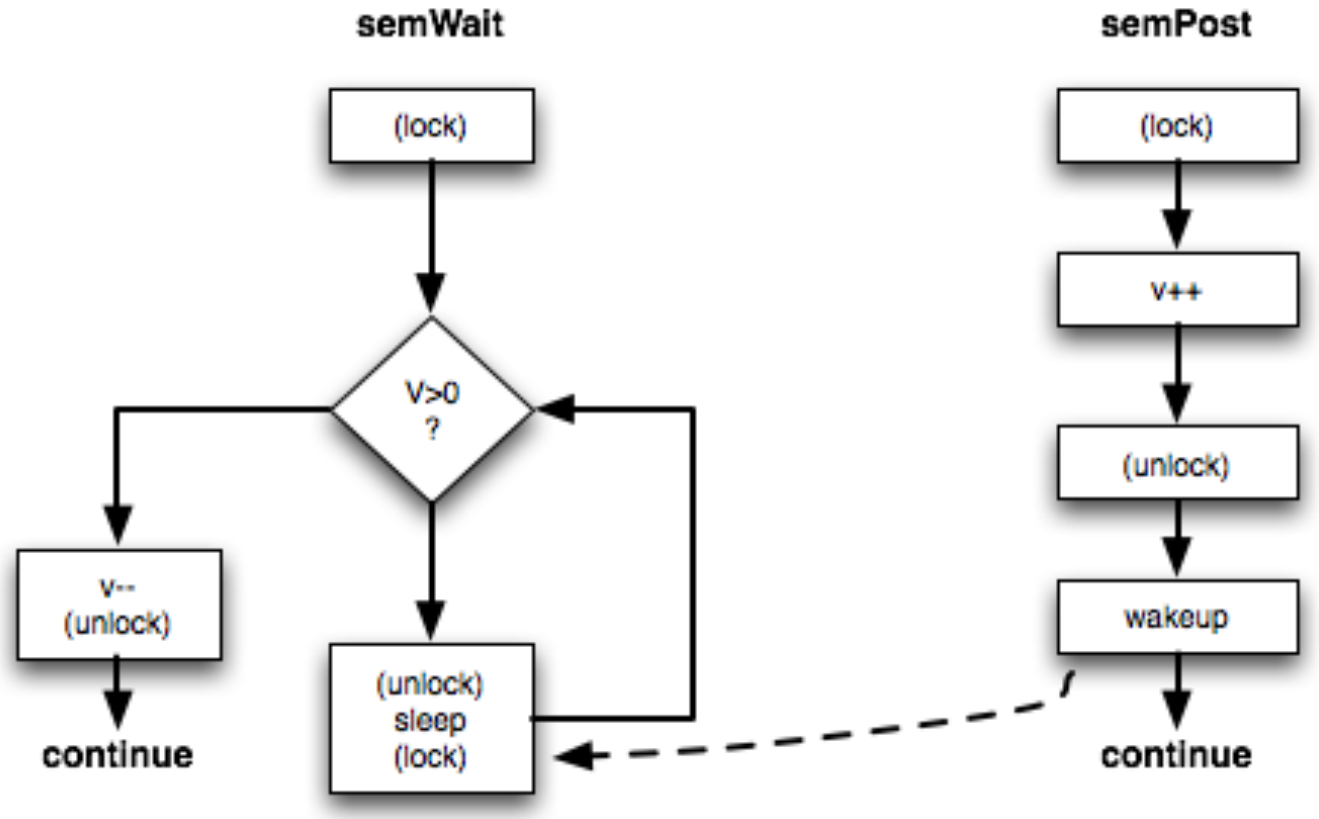
Semáforo

- The wait/notify is a particular case of the general concept of semaphore
- E. Dijkstra, Dutch scientist and world reference in computer science generalized the semaphore concept for computer programming.
- A counting semaphore is a variable that can be arbitrarily incremented but can't be decremented below zero.
- Useful when one wants threads to wait for some event that happens in succession or that can be counted.

semWait e semPost

- ***semWait*** : a thread tries to decrement the count of the semaphore. If successful, the method returns. If the count is zero, the thread will wait until another thread increments the count of the semaphore
- ***semPost***: increments the value of a semaphore and wakes up waiting threads (if any)

semWait e semPost



Implementação de um semáforo

```
// Extensions/Semaphore.java
package Extensions; // vide http://www.lambdacs.com/

public class Semaphore {
  int count = 0;

  public Semaphore(int i) {
    count = i;
  }

  public Semaphore() {
    count = 0;
  }

  public synchronized void init(int i) {
    count = i;
  }
  ...
}
```

Implementação de um semáforo

```
public class Semaphore {
  ...
  public synchronized void semWait() {
    boolean interrupted = false;
    while (count == 0) {
      try {wait();}
      catch (InterruptedException ie) {interrupted=true;}
    }
    count--;
    if (interrupted) Thread.currentThread().interrupt();
  }

  public synchronized void semPost() {
    count++;
    notify();
  }
}
```

Implementação de um semáforo

```
public class Semaphore {
  ...
  public synchronized void semWait() {
    boolean interrupted = false;
    while (count == 0) {
      try {wait();}
      catch (InterruptedException ie) {interrupted=true;}
    }
    count--;
    if (interrupted) Thread.currentThread().interrupt();
  }

  public synchronized void semPost() {
    count++;
    notify();
  }
}
```

Barreiras

- Uma barreira permite que um conjunto de threads esperem umas pelas outras. Esta coordenação é conseguida chamando um método da barreira.
- A barreira é inicializada com o número de threads a bloquear (N).
- As threads são bloquedas na barreira até todas as N threads terem chegado. Nesse momento são todas libertadas.
- A ideia é definir um ponto de coordenação onde um conjunto de threads só avança após todas as N threads terem chegado a esse ponto

Implementação da barreira

```
classe Barrier{
    private int numberWaiters;
    private int currentWaiters = 0;
    private int passedWaiters = 0;

    public synchronized void barrierWait()
                        throws InterruptedException{
        currentWaiters ++;
        while(currentWaiters < numberWaiters){
                wait();
        }
        if(passedWaiters == 0) notifyAll();
        passedWaiters++;
        if(passedWaiters == numberWaiters){
                passedWaiters = 0;
                currentWaiters = 0;
        }
    }
}
```

Barreiras Única

- A barreira única implementa um conceito similar ao de uma barreira

- A barreira única permite que um conjunto de threads (barradas) C1 espere pela chegada de um conjunto C2 de threads (expeditores). Após todos os C2 chegarem os threads do conjunto C1 avançam.

Barreiras Única

- A barreira única implementa um conceito similar ao de uma barreira

- A barreira única permite que um conjunto de threads (barradas) C1 espere pela chegada de um conjunto C2 de threads (expeditores). Após todos os C2 chegarem os threads do conjunto C1 avançam.

Barreiras Única

- No caso especial de uma única thread barrada:
  - cada thread expeditor incrementa a barreira única à medida que completa o seu trabalho, enquanto o thread barrado espera por todos eles
  - Uma vez que todos os threads expeditores completem o seu trabalho, o thread barrado é notificado e avança

Implementação da barreira única

- `currentPosters, totalposters` – threads expeditores

- `currentWaiters, totalWaiters` – threads barrados

Implementação da barreira única

```
class SingleBarrier {

   int currentPosters = 0;
   int totalPosters = 0;

   int passedWaiters = 0;
   int totalWaiters = 1;

   // ...

}
```

Implementação da barreira única

```
Class SingleBarrier {
    public SingleBarrier (int i) {
     totalPosters = i;
    }
    public SingleBarrier (int i, int j) {
           totalPosters = i; totalWaiters = j;
    }
    public SingleBarrier () {
    }
    public synchronized void init(int i) {
           totalPosters = i; currentPosters=0;
    }
    public synchronized void barrierSet(int i) {
           totalPosters = i; currentPosters=0;
    }
```

Implementação da barreira única

```
class SingleBarrier {
    public synchronized void barrierWait() {
      boolean interrupted = false;
      while (currentPosters != totalPosters) {
           try {wait();}
           catch (InterruptedException ie)
                      {interrupted=true;}
      }
      passedWaiters++;
      if (passedWaiters == totalWaiters) {
           currentPosters = 0; passedWaiters = 0;
           notifyAll();
      }
      if (interrupted)
            Thread.currentThread().interrupt();
  }
```

Implementação da barreira única

```
public synchronized void barrierPost() {
  boolean interrupted = false;
    // In case a poster thread beats barrierWait,
    // keep count of posters.
  while (currentPosters == totalPosters) {
      try {wait();}
      catch (InterruptedException ie)
         {interrupted=true;}
  }
  currentPosters++;
  if (currentPosters == totalPosters) notifyAll();
  if (interrupted)
        Thread.currentThread().interrupt();
}
```

BlockingQueue

- Uma blockinQueue é uma fila que coloca em wait as threads se a queue estiver vazia ou se não existir mais capacidade para colocar um novo elemento

- Disponibiliza os seguintes métodos:
  - offer(e): permite adicionar um elemento e à queue. Caso a queue esteja cheia põe a thread em wait até que seja retirado um elemeneto.
  - take(): retira o primeiro elemento da queue. Caso a queue esteja vazia coloca a thread em wait até que um novo elemento seja colocado.

Implementação da bloquingQueue de inteiros

```
classe IntBloquingQueue{
   private Queue<Interger> data = new ...
   public synchronized boolean offer(Integer value){

   }
   public synchronized Integer take() throws InterruptedException{

   }
}
```

Problemas avançados da sincronização

- Recursividade em métodos sincronizados;
- Exceções em métodos sincronizados;
- Métodos sincronizados estáticos;
- O perigo de atributos públicos;
- Objetos imutáveis;
- Sincronização e performance;
- O modificador volatile.

Sumário

- Variáveis condicionais
- Semáforo
- Barreira
- Fila Bloqueante (BlockingQueue)
- Problemas avançados da sincronização