Programação Concorrente em Java

Aula 11

Sinalização de Threads

Introdução

- O intuito da sinalização de threads é possibilitar o envio de “sinais” entre elas.
- Adicionalmente, a sinalização permite que threads “esperem” (wait) sinais de outras threads.
- Por exemplo: a thread B pode estar esperando um sinal da thread A, indicando que um determinado está pronto para ser processado.

Sinalizando via objetos compartilhados

- Um modo simples de threads enviarem sinais umas para as outras é através de objetos sinalizados.
- Por exemplo: Thread A muda o valor de uma variável boolean dentro de um bloco sincronizado.Thread B lê o novo valor e toma alguma ação em relação a isso.

```
public class MySignal{
  protected boolean hasDataToProcess = false;
  public synchronized boolean hasDataToProcess(){
    return this.hasDataToProcess;
  }
  public synchronized void setHasDataToProcess(boolean hasData){
    this.hasDataToProcess = hasData;
  }
}
```

Sinalizando via objetos compartilhados

- Tanto Thread A quanto Thread B devem ter uma referência para uma instância de MySignal, se o código quiser funcionar.
- Obviamente, se Thread A e Thread B possuem referênciam para instâncias diferentes, elas nunca irão detectar o “sinal” uma da outra.
- O dado a ser processado pode ser também localizado em um buffer compartilhado separado da instância MySignal.

Espera Ocupada

- Thread B precisa processar dados mas está esperando os dados ficarem disponíveis (provavelmente outra thread está trabalhando neles).
- Em outras palavras, B está esperando que a thread A mande um sinal, fazendo que hasDataToProcess retorne true.

```
protected MySignal sharedSignal = ...
...
while(!sharedSignal.hasDataToProcess()){
  //do nothing... busy waiting
}
```

wait(), notify(), notifyAll()

- Espera ocupada (busy waiting) não é lá muito eficiente. Em alguns casos, onde o tempo de espera é muito pequeno, pode valer a pena.
- Seria interessante colocar a thread em estado **inativo** até que o recurso que ela quer se torne disponível. Desta forma, não se usaria ciclos de processamento comuns da espera ocupada.
- A classe Object define três métodos para solucinar esse problema: **wait(), notify()** e **notifyAll()**.

wait(), notify(), notifyAll()

- Uma thread que chama wait() em qualquer objeto, se torna inativa (sleep) até que uma **outra** thread chame o notify() do objeto ao qual a thread original acionou o wait().
- Para chamar os métodos wait() e notify() de qualquer objeto, a thread “chamante” deve primeiramente obter uma trava (lock) desse objeto.
- Em outras palavras, a thread chamante deve ativar o wait() e o notify() dentro de um bloco sincronizado!

wait(), notify(), notifyAll()

```
public class MonitorObject{
}

public class MyWaitNotify{
  MonitorObject myMonitorObject = new MonitorObject();
  public void doWait(){
    synchronized(myMonitorObject){
      try{
        myMonitorObject.wait();
      } catch(InterruptedException e){...}
    }
  }
  public void doNotify(){
    synchronized(myMonitorObject){
      myMonitorObject.notify();
    }
  }
}
```

Travas ou Locks

```
public class MsLunch {
    private long c1 = 0;
    private long c2 = 0;
    private Object lock1 = new Object();
    private Object lock2 = new Object();
    public void inc1() {
        synchronized(lock1) {
            c1++;
        }
    }
    public void inc2() {
        synchronized(lock2) {
            c2++;
        }
    }
}
```

```
public void addName(String name) {
    synchronized(this) {
        lastName = name;
        nameCount++;
    }
    nameList.add(name);
}
```

Sinais perdidos

- Chamada do notify()/notifyAll() **antes** da chamada do wait()
- Isso pode ser ou não um problema, mas em alguns casos, a thread que iniciou o wait() ficará esperando “eternamente”.
- A isso, é chamado de **missing signal.**
- Para evitar isso, deve-se armazenar os sinais.

Sinais perdidos

```
public class MyWaitNotify2{
  MonitorObject myMonitorObject = new MonitorObject();
  boolean wasSignalled = false;
  public void doWait(){
    synchronized(myMonitorObject){
      if(!wasSignalled){
        try{
          myMonitorObject.wait();
         } catch(InterruptedException e){...}
      }
      //clear signal and continue running.
      wasSignalled = false;
    }
  }
  public void doNotify(){
    synchronized(myMonitorObject){
      wasSignalled = true;
      myMonitorObject.notify();
    }
  }
}
```

Spurious Wakeups

- É possível ainda que threads acordem mesmo se os métodos notify() e notifyAll() não forem chamados.
- São chamados de **spurious wakeups,** ou seja, despertares sem razão aparente.
- Para previnir isso, basta checar a variável de sinalização em um **loop** ao contrário de um **if.**

Spurious Wakeup

```
public class MyWaitNotify3{
  MonitorObject myMonitorObject = new MonitorObject();
  boolean wasSignalled = false;
  public void doWait(){
    synchronized(myMonitorObject){
      while(!wasSignalled){
        try{
          myMonitorObject.wait();
         } catch(InterruptedException e){...}
      }
      //clear signal and continue running.
      wasSignalled = false;
    }
  }
...
```

```
...
public void doNotify(){
    synchronized(myMonitorObject){
      wasSignalled = true;
      myMonitorObject.notify();
    }
  }
}
```

Múltiplas Threads...

- Múltiplas threads esperando o mesmo **sinal.**
  - O loop anterior também é uma ótima solução no caso de múltiplas threads esperando as quais são todas acordadas pelo notifyAll().
  - Only one thread at a time will be able to obtain the lock on the monitor object, meaning only one thread can exit the wait() call and clear the wasSignalled flag. Once this thread then exits the synchronized block in the doWait() method, the other threads can exit the wait() call and check the wasSignalled member variable inside the while loop. However, this flag was cleared by the first thread waking up, so the rest of the awakened threads go back to waiting, until the next signal arrives.

Constantes e Objetos Globais

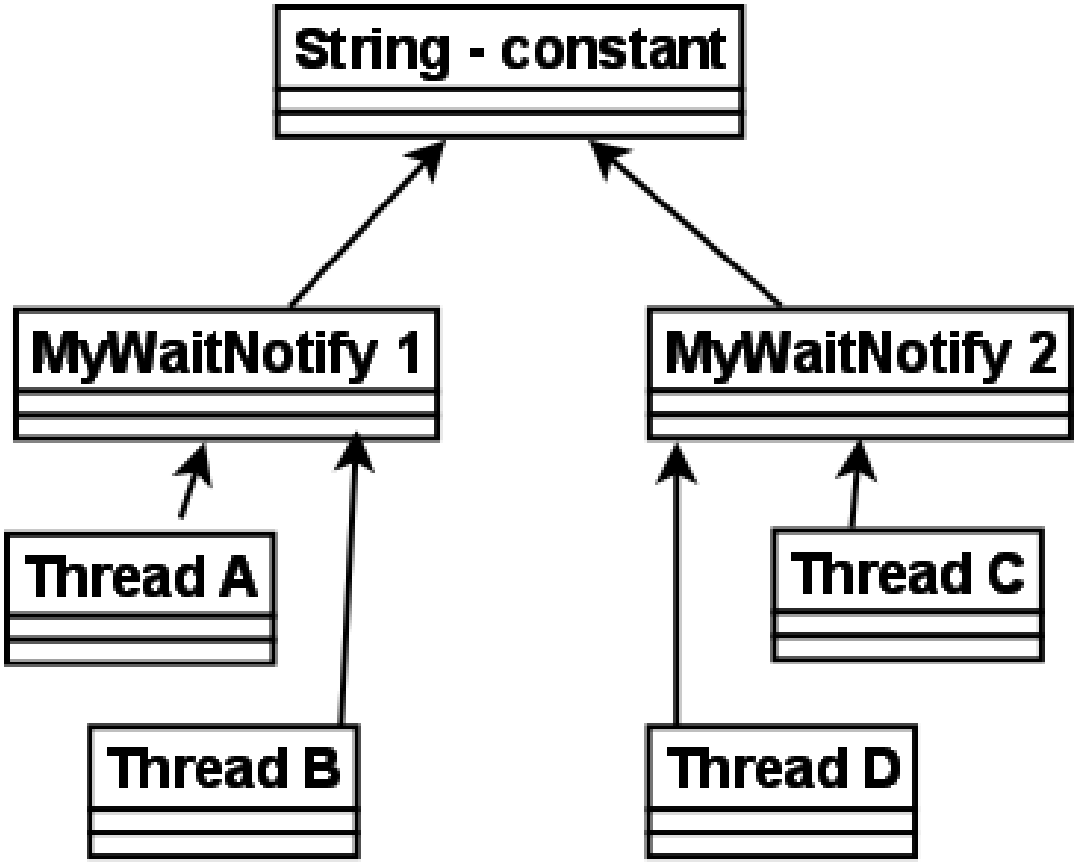
- Não chame o wait() em Strings constantes ou objetos globais!

```
public class MyWaitNotify{
  String myMonitorObject = "";
  boolean wasSignalled = false;
  public void doWait(){
    synchronized(myMonitorObject){
      while(!wasSignalled){
        try{
          myMonitorObject.wait();
         } catch(InterruptedException e){...}
      }
      //clear signal and continue running.
      wasSignalled = false;
    }
  }
  public void doNotify(){
    synchronized(myMonitorObject){
      wasSignalled = true;
      myMonitorObject.notify();
    }
  }
}
```

Constantes e Objetos Globais

- O problema de Strings é que elas são constantes. Logo, pra JVM, mesmo objetos diferentes com variáveis do tipo String com o mesmo valor, **apontam** para a mesma instância.
- Logo, uma thread que consegue uma trava para uma variável do tipo String em uma instância MyWaitNotify, outra thread que deseje fazer o mesmo com uma outra instância de MyWaitNotify, **ambas** estarão referenciando o **mesmo** objeto String.

Constantes e Objetos Globais



Referêcia

- http://tutorials.jenkov.com/java-concurrency/thread-signaling.html