Evitando deadlocks

IFPE – TADS – LPOO 2

Professor: Paulo Guedes

Introdução – revisão rápida

- Se há concorrência, é preciso:
  - Proteger os dados durante o acesso
- Em Java, cada objeto possui um lock
  - Usar "synchronized" para definir seções críticas
  - Útil para proteger o acesso aos dados
  - Leitura/escrita protegidos: exclusão mútua.

Introdução – revisão rápida

- Locks: necessários para proteger dados
  - Mas podem introduzir problemas
- Mau uso de locks pode causar falhas de concorrência
  - Exemplo: deadlock e livelock
  - Geralmente são questões difíceis de detectar, reproduzir e resolver
- É melhor prevenir os deadlocks
  - Como? Vamos entender melhor

O que o synchronized faz?

- Objeto X chama um método synchr.
- JVM testa se o lock de X está disponível
  - Se não: thread espera ficar disponível
- Trava o objeto X (lock de X indisponível)
- Executa o método
  - Outros métodos podem ser chamados
  - Outros locks podem ser obtidos (Y, Z, etc.)
- Destrava o objeto X (lock disponível)

O que causa deadlocks?

- Deadlock ocorre quando o lock solicitado nunca se torna disponível
  - Faz uma thread esperar “pra sempre”
- Ocorre se uma thread já tem um lock X
  - E está esperando por outro lock Y...
  - Que depende por algum motivo de X
- Dependência pode ser direta ou não
  - Exemplo: A → B → C → A

Exemplo 1

```
public static Object cacheLock
= new Object();
public static Object tableLock =
new Object();
...
public void oneMethod() {
  synchronized (cacheLock) {
     synchronized (tableLock) {
 doSomething();
     }
   }
 }
```

```
  public void anotherMethod()
{
    synchronized (tableLock) {
      synchronized
(cacheLock) {
 doSomethingElse();
      }
    }
  }
```

Exemplo 1

- OneMethod → tenta obter cacheLock
  - Depois tenta tableLock
- AnotherMethod → tenta obter tableLock
  - Depois tenta cacheLock
- Pode causar deadlock se duas threads tentarem chamar um método cada

Deadlock no exemplo 1

- OneMethod → obtém cacheLock
- AnotherMethod → obtém tableLock
- OneMethod → tenta obter tableLock
  - Não consegue e espera (cacheLock preso)
- AnotherMethod → tenta obter cacheLock
  - Não consegue e espera (tableLock preso)

Deadlocks não são óbvios

- Testar se pode haver deadlocks é difícil
  - Dependem do escalonador, carga do sistema e do ambiente
- Reproduzir pode ser complicado
  - Condição que causa o deadlock depende de vários fatores
- Identificar deadlocks é difícil
  - É melhor prevenir

O que causa um deadlock

- Mais de uma thread, mais de um lock
- Ordem de aquisição dos locks em ciclo
  - Geralmente devido a chamadas com ciclos
  - Condição impossibilita obter um dos locks
- Mas se os locks forem obtidos **sempre** na mesma ordem por todas as threads...
  - Não há possibilidade de deadlocks!

MVC - Um exemplo mais sutil

```
public class Model {
  private View myView;
  public synchronized void updateModel(Object someArg) {
    doSomething(someArg);

myView.somethingChanged();
  }
  public synchronized Object getSomething() {
    return someMethod();
  }
}
```

```
public class View {
  private Model myModel;
  public synchronized void somethingChanged() {
      doSomething();
    }
    public synchronized void updateView() {
      Object o = myModel.getSomething();
    }
  }
}
```

Exemplo 2 - MVC

- UpdateModel → obtém o lock de Model
  - Depois chama view.somethingChanged...
  - O qual tenta obter o lock de View
- UpdateView → obtém o lock de View
  - Depois chama model.getSomething...
  - Que tenta obter o lock de Model
- Potencial para deadlock aqui
  - Mas deadlocks podem ser ainda mais sutis

Exemplo 3 – com objetos

```
public void transferMoney(Account fromAccount,
      Account toAccount, DollarAmount amountToTransfer) {
  synchronized (fromAccount) {
    synchronized (toAccount) {
      if (fromAccount.hasSufficientBalance(amountToTransfer) {
        fromAccount.debit(amountToTransfer);
        toAccount.credit(amountToTransfer);
      }
    }
  }
}
```

Exemplo 3 – com objetos

Considere a situação:

- Thread A:
  - transferMoney(accountOne, accountTwo, amount);
- Thread B:
  - transferMoney(accountTwo, accountOne, amount);
- Ainda há o risco de deadlock
  - Pois a ordem dos locks pode variar

Como evitar deadlocks

- Evitar obter mais de um lock por vez
  - Pode não ser possível
- Se há um número fixo de locks
  - Garantir que sempre são obtidos em ordem
- Reduzir tamanho de código synchroniz.
  - Não sincronizar se for desnecessário
- Garantir **por código** a ordem de obtenção dos locks

Ordenando aquisição dos locks

```
public void transferMoney(Account fromAccount,
      Account toAccount, DollarAmount amountToTransfer) {
 Account firstLock, secondLock;
 if (fromAccount.accountNumber() == toAccount.accountNumber())
   throw new Exception("Cannot transfer from account to itself");
 else if (fromAccount.accountNumber() < toAccount.accountNumber()) {
   firstLock = fromAccount;
   secondLock = toAccount;
 }
 else {
   firstLock = toAccount;
   secondLock = fromAccount;
 }
```

Testa se é a mesma conta

Garante que os locks serão obtidos sempre na mesma ordem pelo código

Ordenando aquisição dos locks

```
    else {
      firstLock = toAccount;
      secondLock = fromAccount;
    }
    synchronized (firstLock) {
      synchronized (secondLock) {
        if (fromAccount.hasSufficientBalance(amountToTransfer) {
          fromAccount.debit(amountToTransfer);
          toAccount.credit(amountToTransfer);
        }
      }
    }
}
```

Garante que os locks serão obtidos sempre na mesma ordem pelo código

Aqui não há chance de ocorrer deadlocks: todos os locks são obtidos em ordem

Referências

- Avoid synchronizatoin deadlocks
  - Use a consistent, defined synchronization ordering to keep your apps running
  - http://www.javaworld.com/javaworld/jw-10-2001/jw-1012-deadlock.html