Especialização em Engenharia de Software

Programação Orientada a Objetos
Concorrência

Sérgio Soares
DSC/UPE
sergio@dsc.upe.br

FACAPE

Execução seqüencial

```
Conta c;
c = new Conta("12-3");

c.creditar(100);
c.debitar(80);

System.out.println(
  c.getSaldo());
```

Os comandos são executados em **seqüência**, um após o outro

O resultado da execução é sempre o mesmo: a conta tem R$ 20,00 de saldo

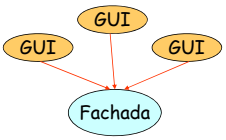
Nas linguagens concorrentes...

A execução também pode ser concorrente:
dois ou mais comandos podem ser executados ao mesmo tempo

Na execução seqüencial, só um comando é executado por vez

A execução concorrente é útil para...

- Permitir que vários usuários utilizem os serviços de um dado sistema ao mesmo tempo
- Aumentar a eficiência de um programa (mas cuidado!)



A execução concorrente é possível pois...

Pode-se

- executar cada um dos comandos em um processador diferente (paralelismo), ou
- dividir o tempo da execução de um processador para executar os vários comandos, parte a parte

Os comandos normalmente não são atômicos

Os comandos e suas partes

Até uma atribuição como

```
saldo = saldo + 5;
```

é executada por partes, similarmente à execução da seqüência de comandos abaixo:

```
double tmp = saldo;
tmp = tmp + 5;
saldo = tmp;
```

Execução concorrente

```
Conta c;
c = new Conta("12-3");

   c.creditar(100)
||
   c.debitar(80)

System.out.println(
  c.getSaldo());
```

Os comandos podem ser executados ao mesmo tempo

Pode-se executar parte de um, parte de outro, arbitrariamente, até completar a execução dos dois

Mais execução concorrente

```
Conta c;
c = new Conta("12-3");

   c.creditar(100)
||
   c.debitar(80)

System.out.print(
  c.getSaldo());
```

Pode-se também executar o primeiro por completo e depois o segundo por completo, ou vice-versa

A escolha é arbitrária, feita pelo interpretador

Concorrência e não determinismo

```
Conta c;
c = new Conta("12-3");

   c.creditar(100)
||
   c.debitar(80)

System.out.print(
  c.getSaldo());
```

A execução pode gerar mais de um resultado! Depende da ordem escolhida para executar as partes dos comandos

A escolha é arbitrária, feita pelo interpretador

`Conta`: segura em um ambiente seqüencial...

```
class Conta {...
  private double saldo = 0;
  void creditar(double vc) {
1   double tmpc = saldo;
2   tmpc = tmpc + vc;
3   saldo = tmpc;
  }
  void debitar(double vd) {
A   double tmpd = saldo;
B   tmpd = tmpd - vd;
C   saldo = tmpd;
  }
```


Mas em um ambiente concorrente...

```
c.creditar(100) || c.debitar(80)
```

Várias seqüências de execução dos comandos são possíveis:

- `1 2 3 A B C` (resultado: 20)
- `A B C 1 2 3` (resultado: 20)
- `1 A 2 3 B C` (resultado: -80)
- `A 1 2 B C 3` (resultado: 100)
- ...

`Repositorio`: seguro em um ambiente seqüencial...

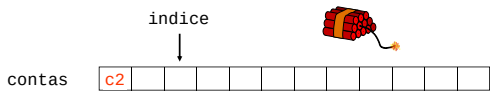
```
public class RepositorioContasArray {
  private Conta[] contas;
  private int indice;

  public void inserir(Conta c) {
    contas[indice] = c;
    indice = indice + 1;
  }
  ...
}
```

Mas em um ambiente concorrente...

```
RepositorioContasArray rep =  ...
```

	thread1	*thread2*
1	`rep.inserir(c1);`	`rep.inserir(c2);`
2	`contas[indice] = c1;`	`contas[indice] = c2;`
3	`indice = indice + 1;`	`indice = indice + 1;`



Ambiente concorrente e interferências

- A execução de um comando pode interferir na execução de um outro comando
  - As interferências ocorrem quando os comandos compartilham (concorrem) pelos mesmos recursos (atributos, variáveis, etc.)
  - Resultados não desejados podem ser gerados

Para evitar interferências indesejadas...

- Temos que controlar o acesso a recursos compartilhados
- Temos que proibir a realização de determinadas seqüências de execução
- Temos que **sincronizar** as execuções:
  - uma espera pelo término da outra

Em Java, para sincronização, use `synchronized`

- Só um método `synchronized` pode ser executado em um objeto por vez
  - Os outros ficam esperando!
- Vários métodos não sincronizados podem ser executados, no mesmo objeto, ao mesmo tempo que um método `synchronized`

`Conta`: segura em um ambiente concorrente...

```
class Conta {...
  private double saldo = 0;
  synchronized void creditar(double vc) {
1   double tmpc = saldo;
2   tmpc = tmpc + vc;
3   saldo = tmpc;
  }
  synchronized void debitar(double vd) {
A   double tmpd = saldo;
B   tmpd = tmpd - vd;
C   saldo = tmpd;
  }
```


Com sincronização, eliminamos interferências...

```
c.creditar(100) || c.debitar(80)
```

Só algumas seqüências de execução são permitidas:
- **1 2 3 A B C** (resultado: 20)
- **A B C 1 2 3** (resultado: 20)

O programa é determinístico: só um resultado é possível

O comando **synchronized** de Java

O modificador

```
synchronized void m(...) {
  corpo
}
```

Equivalentes

O comando; impede a execução simultânea de trechos de códigos sincronizados no mesmo argumento (um objeto)

```
void m(...) {
  synchronized(this) {
    corpo
  }
}
```


Como solicitar a execução concorrente em Java?

- O operador de execução concorrente **||** não existe em Java!
- Mas Java permite criar **threads**
  - Seqüências ou fluxos de execução que podem ser executados concorrentemente
  - Os comandos a serem executados são "empacotados" como threads
  - Objetos da classe **java.lang.Thread**

Concorrência implícita em Java...

- Aplicações web
  - servlets criam threads para executar cada requisição feita por um cliente
- Aplicações distribuídas
  - servidores RMI (EJB, CORBA, etc.) criam threads para executar cada requisição feita por um cliente

A classe **Thread** de Java

- Subclasses de **Thread** redefinem o método **run**
- Os comandos a serem executados por um thread são os comandos do corpo do método **run**

Execução concorrente
=
Criação e início de threads

c.creditar(1)||c.debitar(8) em Java

```
Thread c = new Credito(...);
Thread d = new Debito(...);
c.start();
d.start();
try {
  c.join();
  d.join();
} catch(InterruptedException e) {...}
```

Subclasses de **Thread**

Inicia a execução do thread

Espera pelo término da execução do thread

Simulando o operador de execução concorrente

```
A || B
```

corresponde a

```
Thread a = new ThreadA(...);
Thread b = new ThreadB(...);
a.start(); b.start();
try {
  a.join(); b.join();
} catch(InterruptedException e) {...}
```

Um comando a ser executado concorrentemente...

```
class Credito extends Thread {
  private Conta conta;
  private double val;
  Credito(Conta c,
          double v) {
    conta = c; val = v;
  }
  public void run() {
    conta.creditar(val);
  }
}
```

O comando a ser executado ao mesmo tempo que outro

E o outro comando...

```
class Debito extends Thread {
  private Conta conta;
  private double val;
  Debito(Conta c,
         double v) {
    conta = c; val = v;
  }
  public void run() {
    conta.debitar(val);
  }
}
```

Aqui poderia ter vários comandos, em seqüência ou concorrentes também

Mas quando herança é um problema...

```
class Debito implements Runnable {
  private Conta conta;
  private double val;
  Creditar(Conta c,
           double v) {
    conta = c; val = v;
  }
  public void run() {
    conta.debitar(val);
  }
}
```

Uma alternativa é implementar a interface `Runnable`...

`c.creditar(1)||c.debitar(8)` em Java, com `Runnable`

```
Conta x = new Conta("1234-5");
Thread c = new Credito(x, 100.0);
Thread d;
d = new Thread(new Debito(x, 80.0));
c.start();
d.start();
try {
  c.join();
  d.join();
} catch(InterruptedException e) {...}
```

Subclasse de `Thread`

Implementação de `Runnable`

Sincronização com `synchronized`

- A execução de um thread t tem que esperar pela execução de outro thread u quando
  - u está executando um método sincronizado e t quer começar a executar um método sincronizado do mesmo objeto

Pode ser o mesmo método ou não

Sincronização com `wait`

- A execução de um thread t tem que esperar pela execução de outro thread u quando
  - A semântica da aplicação requer que uma operação de t só seja executada em condições que podem ser garantidas por uma operação de u

A aplicação solicita a espera

Impedindo débitos além do limite

```
class Conta {...
  private double saldo = 0;
  synchronized void debitar(double v) {
    while (saldo < v){
      wait();
    }
    saldo = saldo - v;
  } ...
```

Assumindo que débitos além do limite não geram erros, mas espera ...

Interrompe a execução do método e do thread associado, que só volta a executar quando notificado por outro thread

Por que `while` e não `if`?

```
class Conta {...
  private double saldo = 0;
  synchronized void debitar(double v) {
    while (saldo < v){
      wait();
    }
    saldo = saldo - v;
  } ...
```

A notificação avisa que talvez seja possível realizar o débito, mas não garante! Temos que testar de novo...

Avisando que o saldo aumentou

```
class Conta {...
  synchronized void creditar(double v) {
    saldo = saldo + v;
    notifyAll();
  }
}
```

Avisa (notifica) **a todos** os threads que estavam esperando pela execução de um método desse (**this**) objeto

O método `wait...`

- Só pode ser chamado dentro de um método **synchronized**
- Coloca o thread executando o comando para esperar por uma notificação
- Libera a restrição de sincronização
  - outro método sincronizado vai poder começar a executar no mesmo objeto

Os métodos `notify` e `notifyAll`

- O método `notifyAll` acorda todos os threads esperando para executar um método do objeto que executou o `notifyAll`
- O método `notify` só acorda um dos processos
  - a escolha é arbitrária, feita pelo interpretador

Cuidado com concorrência!

- Os mecanismos para sincronização diminuem a eficiência do sistema e podem complicar o código
- O sistema pode entrar em deadlock!
  - Todos os threads parados, um esperando pelo outro
- O sistema pode entrar em livelock!
  - Threads executando mas não gerando nada que possa ser observado pelo usuário

Controle nas classes básicas

- Impedir a atualização concorrente de cópias de um objeto básico
  - ao utilizar um meio persistente, sem cache
  - ao retornar clones em coleções voláteis
- Usar timestamp
  - objetos só podem ser atualizados se não houver uma atualização mais recente

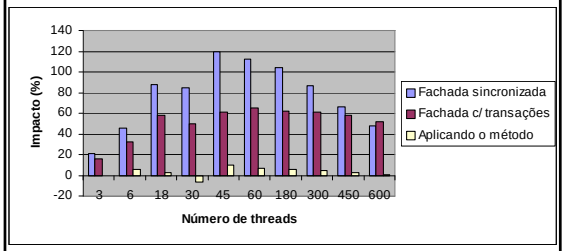
Controle nas coleções de dados

- Com SGBD
  - certificar-se do uso correto de SQL
    - cada comando SQL é tratado como uma transação pelo SGBD
  - utilizar transações quando necessário
- Sem SGBD
  - tratar o acesso a estrutura de dados
  - implementar o serviço de transações

Controle nas coleções de negócio e fachada

- Identificar regras de negócio sujeitas a interferência
  - sincronizar métodos
  - usar o gerenciador de concorrência
    - sincroniza apenas os *threads* que podem interferir entre si

Impacto em eficiência



Referências

- Sérgio Soares. *Desenvolvimento Progressivo de Programas Concorrentes Orientados a Objetos*. Dissertação de Mestrado. Fevereiro de 2001.
- Sérgio Soares e Paulo Borba. *Controle de Concorrência com Java e Bancos de Dados Relacionais*. V SBLP. Maio de 2001.
- Sérgio Soares e Paulo Borba. *Concurrency Manager*. SugarLoafPLoP 2001. Outubro de 2001.
- Sérgio Soares e Paulo Borba. *Concurrency Control with Java and Relational Databases*. COMPSAC 2002. Agosto de 2002.

Especialização em Engenharia de Software

Programação Orientada a Objetos

Concorrência

Sérgio Soares
DSC/UPE
sergio@dsc.upe.br

FACAPE