Múltiplas Linhas de Execução Java (Threads)

SCC0604 - Programação Orientada a Objetos

Prof. Fernando V. Paulovich
http://www.icmc.usp.br/~paulovic
paulovic@icmc.usp.br

Instituto de Ciências Matemáticas e de Computação (ICMC)
Universidade de São Paulo (USP)

25 de julho de 2010

Sumário

Sumário

Introdução

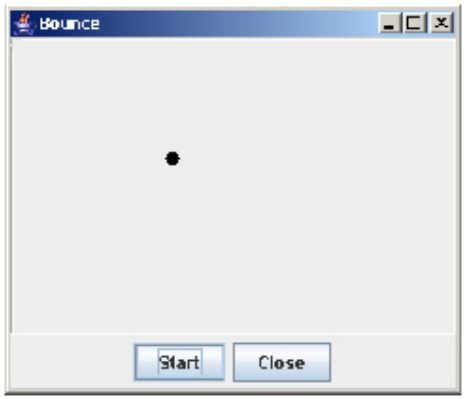
- Apesar de usarmos na maior parte das vezes um computador mono-processador, é possível "simular" o funcionamento paralelo do mesmo
- Os programas de múltiplas linhas de execução ampliam a ideia o paralelismo: os programas individuais parecerão realizar várias tarefas ao mesmo tempo
- Normalmente, cada tarefa é chamada de linha de execução (thread)

Introdução

- É dito que um programa que pode executar mais de uma linha de execução simultaneamente têm múltiplas linhas de execução
- Cada linha de execução é executada em um contexto separado, dessa forma é como se fosse possível obter paralelismo de execução
- Então qual a diferença entre múltiplos processos e múltiplas linhas de execução?

O que são linhas de execução?

- O seguinte programa mostra que a execução de múltiplas coisas ao mesmo tempo sem o uso de linhas de execução pode ser problemática



O que são linhas de execução?

```
1  public class Ball {
2
3      public void bounce() {
4          draw();
5          for(int i = 1; i <= 1000; i++) {
6              move();
7              try {
8                  Thread.sleep(5);
9              } catch (InterruptedException e) {}
10         }
11     }
12 }
```

O que são linhas de execução?

- Para acionar o sistema usa-se

```
1 Ball b = new Ball(painelBounce);
2 b.bounce();
```

Usando linhas de execução para dar uma chance às outras tarefas

- Para resolver esse problema, vamos usar duas linhas de execução: uma para a bola e outra para a linha de execução principal
- Dessa forma, é possível deixar a bola quicando e ainda manipular a janela de apresentação

Usando linhas de execução para dar uma chance às outras tarefas

- Uma forma simples de se executar um procedimento em uma linha de execução separado é por meio da inserção desse código em um método **run()** de uma classe derivada de **Thread**

Usando linhas de execução para dar uma chance às outras tarefas

```
1  public class Ball extends Thread
2
3      public void run() {
4          draw();
5          for(int i = 1; i <= 1000; i++) {
6              move();
7              try {
8                  Thread.sleep(5);
9              } catch (InterruptedException e) {}
10         }
11     }
12 }
```

Executando e iniciando linhas de execução

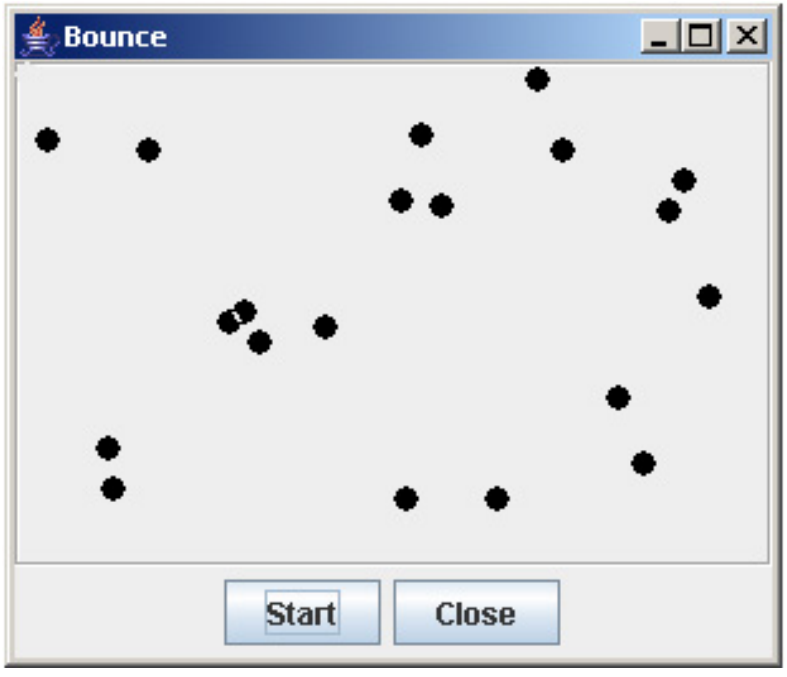
- Quando um objeto derivado de **Thread** é construído, o método **run()** não é chamado automaticamente
- Isso deve ser feito chamando o método **start()**

```
1 Ball b = new Ball(painelBounce);
2 b.start();
```

Executando e iniciando linhas de execução

- Em Java, uma linha de execução precisa dizer às outras linhas quando está ociosa, para que as outras linhas tenham a chance de executar seus códigos do método **run()**
- Normalmente isso é feito usando-se o método **sleep()**
- O que significa fazer **Ball** herdar de **Thread**? Isso não é estranho pelas características de uma herança?

Executando diversas linhas de execução



Sumário

Propriedades das linhas de execução

- As linhas de execução podem estar em um dos quatro estados
  - Nova
  - Passível de execução
  - Bloqueada
  - Morta

Linhas de execução novas

- Quando uma linha de execução é criada com o método **new** - por exemplo, **new Ball()** - a linha ainda não está em execução, mas está no estado novo

Linhas de execução passíveis de execução

- Quando o método **start()** é chamado, a linha de execução é passível de execução
- Uma linha de execução passível de execução pode não estar sendo executada ainda, isso fica por conta do sistema operacional
- Quando o código de uma linha de execução começa a ser executado, a linha está em execução

Linhas de execução bloqueadas

- Uma linha de execução está no estado de bloqueada quando
  - Alguém chama o método **sleep()** da mesma
  - A mesma chama uma operação que está bloqueando entrada/saída
  - A linha de execução chama o método **wait()**
  - A linha de execução tenta bloquear um objeto que está bloqueado por outra execução
  - Alguém chama o método **suspend()** da linha de execução

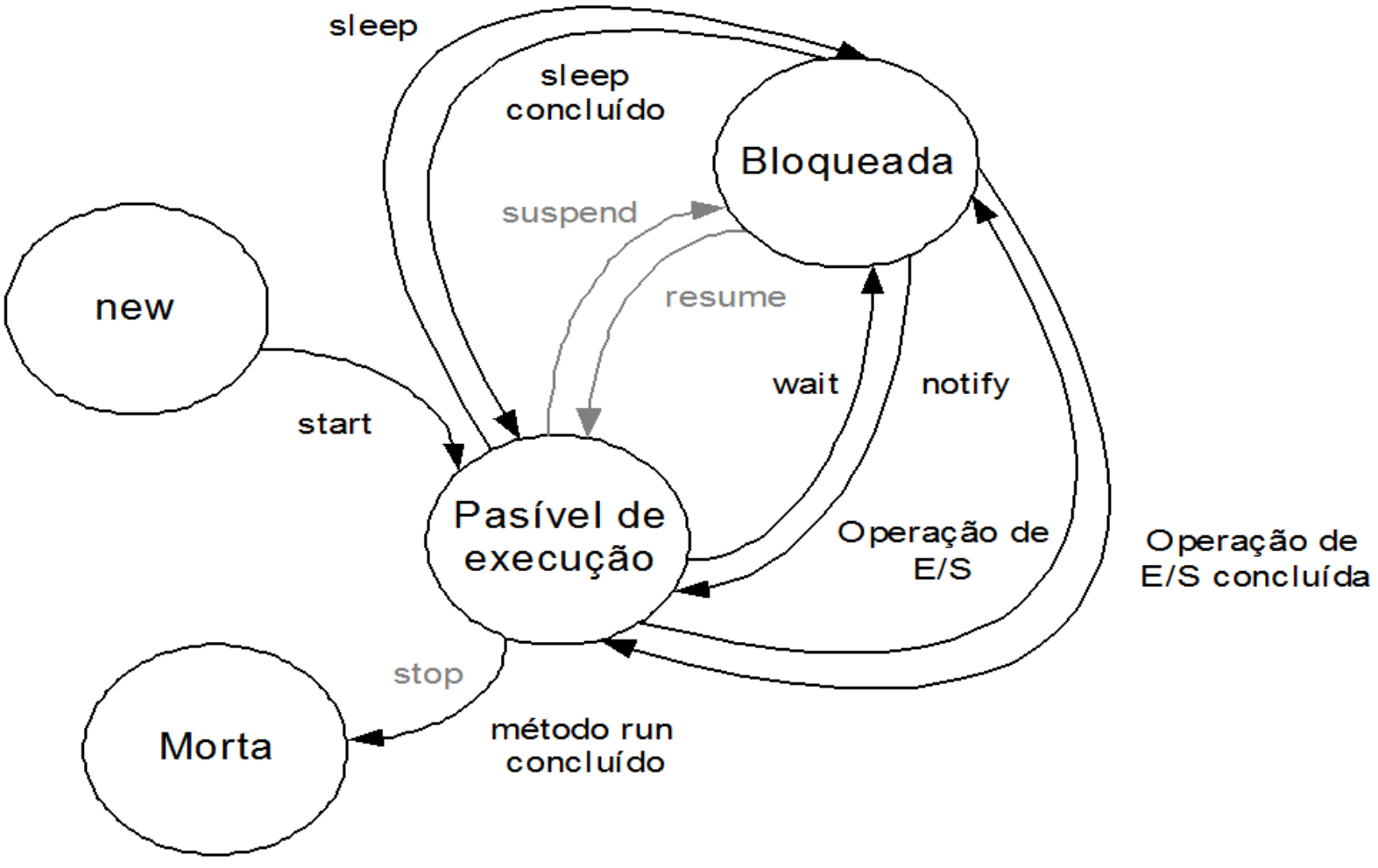
Saindo do estado de bloqueio

- Uma linha de execução sai do estado bloqueado e volta para o passível de execução quando
  - O número especificado de milissegundos expirou em uma chamada **sleep()**
  - Uma operação de E/S esperada terminou
  - A mesma está em **wait()** e alguém chama **notify()/notifyAll()**
  - A mesma está esperando um bloqueio de um objeto pertencente a outra linha, e essa linha cede esse bloqueio
  - A mesma está suspensa (**suspend()**) e o método **resume()** foi chamado

Linhas de execução mortas

- Uma linha de execução morre por duas razões
  - Quando o método **run()** encerra
  - Quando uma exceção não capturada encerra o método **run()**
- Uma outra forma, depreciada, de "matar" uma linha de execução é chamar o método **stop()**
  - O método **stop()** não é para ser usado, nada garante que o mesmo funcionará!

Estados de uma linha de execução



Prioridades da linha de execução

- Em Java, todo linha de execução tem prioridade
- A prioridade de uma linha de execução pode ser aumentada ou diminuída aplicando-se o método **setPriority()**
- Os valores de prioridade podem estar entre 1 (**MIN_PRIORITY**) e 10 (**MAX_PRIORITY**) – **NORM_PRIORITY** é igual a 5

Prioridades da linha de execução

- Uma linha de execução com alta prioridade executa até que
  - A mesma cede a vez, chamando o método **yield()**
  - Sai do estado de passível de execução
  - Outra linha de execução de prioridade mais alta torna-se passível de execução

Prioridades da linha de execução

```
1  private void botaoExpressActionPerformed(ActionEvent evt) {
2    for(int i=0; i < 5; i++) {
3      Ball b = new Ball(painelBounce, Color.RED);
4      b.setPriority(Thread.NORM_PRIORITY+2);
5      b.start();
6      }
7    }
8
9  private void botaoStartActionPerformed(ActionEvent evt) {
10   for(int i=0; i < 5; i++) {
11     Ball b = new Ball(painelBounce, Color.BLACK);
12     b.setPriority(Thread.NORM_PRIORITY);
13     b.start();
14   }
15 }
```

Sumário

Sincronismo

- Na maioria dos programas com múltiplas linhas de execução, duas ou mais linhas precisam compartilhar o acesso aos mesmos objetos
- Dependendo da ordem em que os dados foram acessados, o resultado pode danificar os objetos
- Tal situação é frequentemente chamada de condição de corrida (*race condition*)

Comunicação de linhas de execução sem sincronismo

- Para evitar acesso simultâneo de um objeto compartilhado por diversas linhas de execução, você deve sincronizar o acesso

Comunicação de linhas de execução sem sincronismo

```
1  class Bank {
2    ...
3    public void transfer(int from, int to, int amount) {
4      if (accounts[from] < amount) return;
5        accounts[from] -= amount;
6        accounts[to] += amount;
7        ntransacts++;
8        if (ntransacts % NTEST == 0) test();
9    }
10 }
```

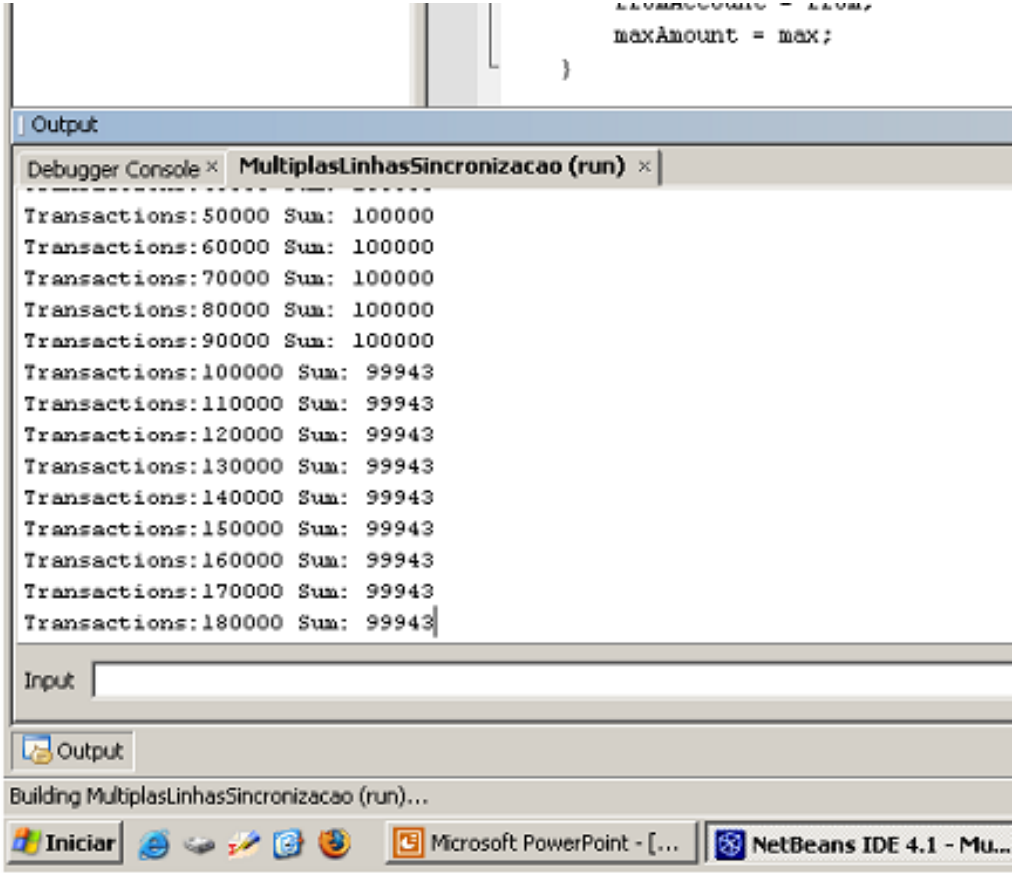
Comunicação de linhas de execução sem sincronismo

```
1  class TransferThread extends Thread {
2    public TransferThread(Bank b, int from, int max) {
3      ...
4    }
5
6    public void run() {
7      try {
8        while (!interrupted()) {
9          int toAccount = (int)(bank.size() * Math.random());
10         int amount = (int)(maxAmount * Math.random());
11         bank.transfer(fromAccount, toAccount, amount);
12         sleep(1);
13       }
14     } catch (InterruptedException e) {}
15   }
16 }
```

Comunicação de linhas de execução sem sincronismo

```
1  public class UnsynchBankTest {
2    public static void main(String[] args) {
3      Bank b = new Bank(NACCOUNTS, INITIAL_BALANCE);
4      int i;
5      for (i = 0; i < NACCOUNTS; i++) {
6        TransferThread t = new TransferThread(b, i, INITIAL_BALANCE);
7        t.setPriority(Thread.NORM_PRIORITY + i % 2);
8        t.start();
9      }
10   }
11
12   public static final int NACCOUNTS = 1000;
13   public static final int INITIAL_BALANCE = 100;
14 }
```

Comunicação de linhas de execução sem sincronismo



Sincronizando o acesso a recursos compartilhados

- O problema ocorrido anteriormente se dá quando duas linhas de execução estão tentando atualizar uma conta simultaneamente

```
1 accounts[to] += amount;
```

- O problema é que essa operação não é atômica!

Sincronizando o acesso a recursos compartilhados

- O problema dessa forma consiste no fato do método **transfer()** poder ser interrompido no meio
- Para corrigir esse problema é necessário garantir que esse método seja executado por inteiro antes da linha de execução perder o controle de execução
- Isso pode ser feito simplesmente declarando tal método como **synchronized**

Sincronizando o acesso a recursos compartilhados

```
1  class Bank {
2      ...
3      public synchronized void transfer(int from, int to, int amount) {
4          if (accounts[from] < amount) return;
5          accounts[from] -= amount;
6          accounts[to] += amount;
7          ntransacts++;
8          if (ntransacts % NTEST == 0) test();
9      }
10 }
```

Sincronizando o acesso a recursos compartilhados

- Quando uma linha de execução chama um método **synchronized** é garantido que o método concluirá sua execução, antes que outra possa executar qualquer outro método **synchronized** no mesmo objeto

Sumário

A interface Runnable

- Quando é necessário usar múltiplas linhas de execução em uma classe que já é derivada de uma classe diferente de **Thread**, a interface **Runnable** deve ser usada

A interface Runnable

```
1  class TransferThread implements Runnable {
2
3      public TransferThread(Bank b, int from, int max) {
4          ...
5      }
6
7      public void run() {
8          ...
9      }
10
11     private Bank bank;
12     private int fromAccount;
13     private int maxAmount;
14 }
```

A interface Runnable

- Para criar uma linha de execução com base em um objeto que implementa **Runnable**, deve-se criar um objeto **Thread** e passar no seu construtor o objeto que implementa a interface **Runnable**

```
1 Thread t = new Thread(new TransferThread(...));
2 t.setPriority(Thread.NORM_PRIORITY);
3 t.start();
```

Sumário

Esperando um Thread Terminar

- Ao se chamar o método **start()** em uma Thread, essa passa a ser passível de execução e o restante do código após o **start()** continua a ser executado
- Esse pode ser um efeito inconveniente se em algum ponto do código se faça o uso do que é executado (calculado) dentro de ume Thread

Esperando um Thread Terminar

```java
public class TesteJoin extends Thread {

  public TesteJoin(String id) {
    this.id = id;
  }

  @Override
  public void run() {
    try {
      Thread.sleep(3000);
      System.out.println(id);
    } catch (InterruptedException ex) {
    }
  }

  private String id;
}
```

```java
public static void main(String[] args) {
  TesteJoin t1 = new TesteJoin("teste 1");
  TesteJoin t2 = new TesteJoin("teste 2");

  t1.start();
  t2.start();

  System.out.println("passou...");
}
```

Esperando um Thread Terminar

- Para se criar um ponto no código que fique esperando o término de uma Thread, chama-se o método **join()** da Thread

```
1  public static void main(String[] args) throws InterruptedException {
2    TesteJoin t1 = new TesteJoin("teste 1");
3    TesteJoin t2 = new TesteJoin("teste 2");
4
5    t1.start();
6    t2.start();
7
8    t1.join();
9    t2.join();
10
11   System.out.println("passou...");
12 }
```

Sumário

Introdução

- Desde Java 1.5, o modo mais seguro de se manipular **Threads** é por meio do pacote **java.util.concurrent**
- Usando-se esse pacote é possível criar conjuntos (*pool*) de **Threads** e gerenciar o comportamento desse conjunto como um todo
- Uma das principais classes para o gerenciamento de **Threads** é a **ExecutorService**

Criando um Thread

```
1  public class LinhaExecucao extends Thread {
2
3    public LinhaExecucao(String nome) {
4      this.nome = nome;
5    }
6
7    @Override
8    public void run() {
9      try {
10       for (int i = 0; i < 100; i++) {
11         System.out.println(nome);
12         Thread.sleep(50);
13       }
14     } catch (InterruptedException ex) {
15     }
16   }
17
18   private String nome;
19 }
```

Usando a ExecutorService

- Para se controlar um grupo de **Threads**, cria-se um **ExecutorService**
- Para se executa uma Thread, chama-se o método **execute(...)** do **ExecutorService**

```
1  public static void main(String[] args) {
2    //cria um pool para a execuação simultânea de no máximo 3 Threads
3    ExecutorService executor = Executors.newFixedThreadPool(3);
4
5    //cria Threads e põe para execução
6    for(int i=0; i < 10; i++) {
7      LinhaExecucao le = new LinhaExecucao("linha: "+i);
8      executor.execute(le);
9    }
10 }
```

Usando a ExecutorService

- A classe **ExecutorService** ainda fornece outros métodos
  - **shutdown()** : não permite que mais nenhuma **Thread** seja adicionada ao **ExecutorService**
  - **shutdownNow()** : para imediatamente a execução das **Threads** sendo executadas pelo **ExecutorService**

- É possível se criar outros tipos de **ExecutorService**
  - **Executor.newSingleThreadExecutor()**: cria um executor de uma única **Thread**
  - etc.

Maiores informações

- Para maiores informações sobre as linhas de execução e seu funcionamento consulte o Capítulo 1 do livro Core Java 2, Volume II - Recursos Avançados