13

Fundamentos de Programação Concorrente

Helder da Rocha
www.argonavis.com.br

Programação concorrente

- O objetivo deste módulo é oferecer uma introdução a Threads que permita o seu uso em aplicações gráficas e de rede
- Tópicos abordados
  - A classe Thread e a interface Runnable
  - Como criar threads
  - Como controlar threads
  - Tópicos sobre deadlock
  - Exemplos de monitores: wait() e notify()
- Para mais detalhes, consulte as referências no final do capítulo

Múltiplas linhas de execução

- Múltiplos threads oferecem uma nova forma de dividir e conquistar um problema de computação
  - Em vez de dividir o problema apenas em objetos independentes ...
  - ... divide o problema em tarefas independentes
- Threads vs. Processos
  - Processos: tarefas em espaços de endereços diferentes se comunicam usando pipes oferecidos pelo SO
  - Threads: tarefas dentro do espaço de endereços da aplicação se comunicam usando pipes fornecidos pela JVM
- O suporte a multithreading de Java é nativo
  - Mais fácil de usar que em outras linguagens onde não é um recurso nativo

O que é um thread

- Um thread parece e age como um programa individual. Threads, em Java, são objetos.
- Individualmente, cada thread faz de conta que tem total poder sobre a CPU
- Sistema garante que, de alguma forma, cada thread tenha acesso à CPU de acordo com
  - Cotas de tempo (time-slicing)
  - Prioridades (preemption)
- Programador pode controlar parcialmente forma de agendamento dos threads
  - Há dependência de plataforma no agendamento

Por que usar múltiplos threads?

- Todo programa tem pelo menos um thread, chamado de Main Thread.
  - O método main() roda no Main Thread
- Em alguns tipos de aplicações, threads adicionais são essenciais. Em outras, podem melhorar o bastante a performance.
- Interfaces gráficas
  - Essencial para ter uma interface do usuário que responda enquanto outra tarefa está sendo executada
- Rede
  - Essencial para que servidor possa continuar a esperar por outros clientes enquanto lida com as requisições de cliente conectado.

Como criar threads

- Há duas estratégias
  - Herdar da classe java.lang.Thread
  - Implementar a interface java.lang.Runnable
- Herdar da classe Thread
  - O objeto é um Thread, e sobrepõe o comportamento padrão associado à classe Thread
- Implementar a interface Runnable
  - O objeto, que define o comportamento da execução, é passado para um Thread que o executa
- Nos dois casos
  - Sobreponha o método run() que é o "main()" do Thread
  - O run() deve conter um loop que irá rodar pelo tempo de vida do thread. Quando o run(), terminar, o thread morre.

Exemplo: herdar da classe Thread

```
public class Trabalhador extends Thread {
    String produto; int tempo;
    public Trabalhador(String produto,
                       int tempo) {
        this.produto = produto;
        this.tempo = tempo;
    }
    public void run() {
      for (int i = 0; i < 50; i++) {
        System.out.println(i + " " + produto);
        try {
          sleep((long)(Math.random() * tempo));
        } catch (InterruptedException e) {}
      }
      System.out.println("Terminei " + produto);
    }
}
```

Exemplo: implementar Runnable

```
public class Operario implements Runnable {
   String produto; int tempo;
   public Operario (String produto,
                    int tempo) {
      this.produto = produto;
      this.tempo = tempo;
   }
   public void run() {
     for (int i = 0; i < 50; i++) {
       System.out.println(i + " " + produto);
       try {
         Thread.sleep((long)
                      (Math.random() * tempo));
       } catch (InterruptedException e) {}
     }
     System.out.println("Terminei " + produto);
   }
}
```

Como usar o Thread

- *Para o Trabalhador que é Thread*

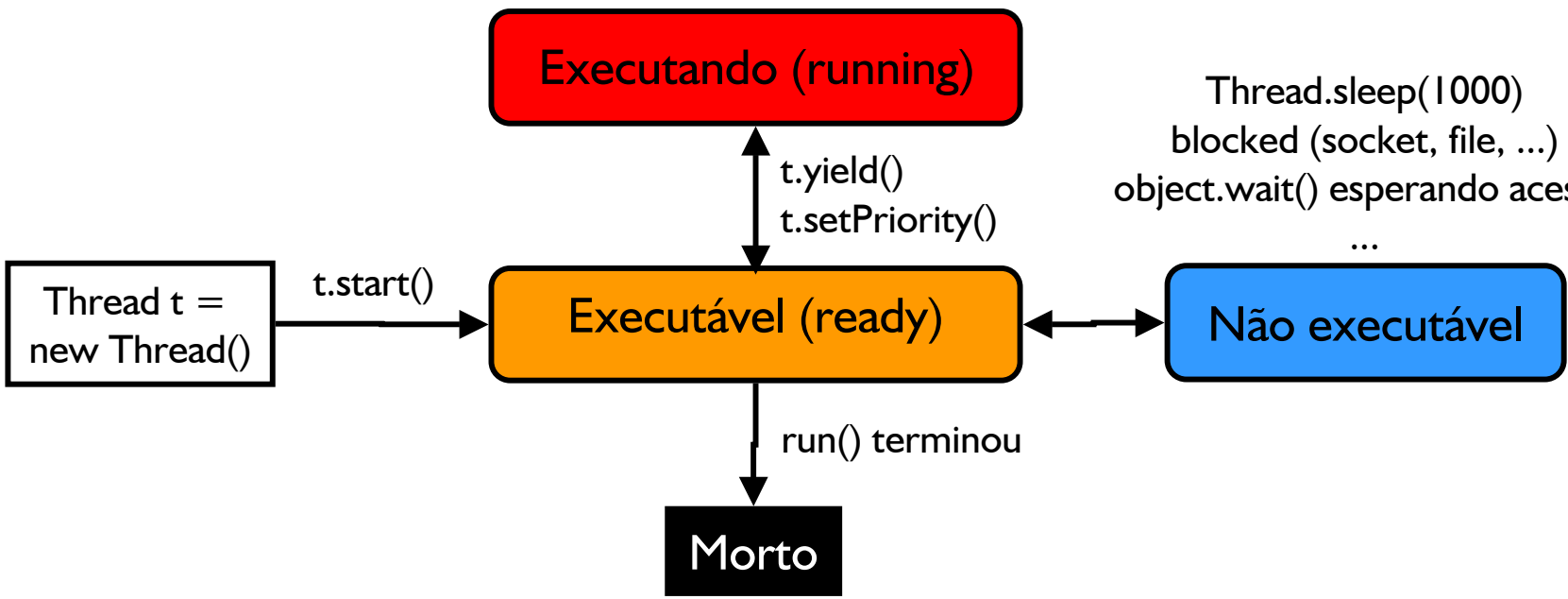
```
Trabalhador severino =
        new Trabalhador("sapato", 100);
Trabalhador raimundo =
        new Trabalhador("bota", 500);
severino.start();
raimundo.start();
```

- *Para o Trabalhador que é Runnable*

```
Operario biu = new Operario ("chinelo", 100);
Operario rai = new Operario ("sandalia", 500);
Thread t1 = new Thread(biu);
Thread t2 = new Thread(rai);
t1.start();
t2.start();
```

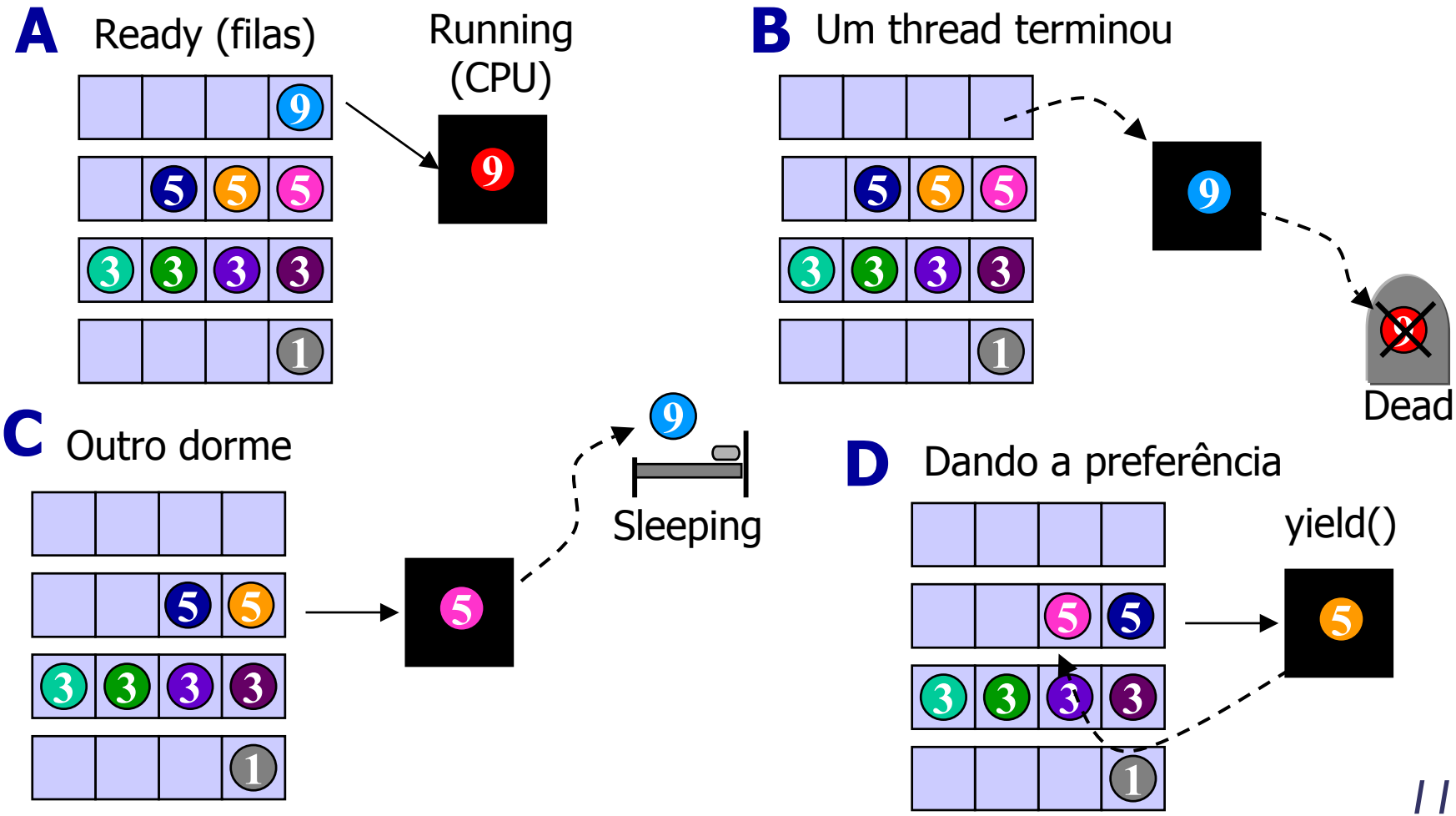
Ciclo de vida

- *Um thread está geralmente em um dentre três estados: executável (e possívelmente executando) e não-executável (esperando, bloqueado, etc.)*
- *O thread entre no estado executável com t.start(), que causa o início de run(), e passa para o estado morto quando run() chega ao fim.*



Filas de prioridades

- *O estado ready não é garantia de execução. Threads são regidos por prioridades. Threads de baixa prioridade têm menos chances*



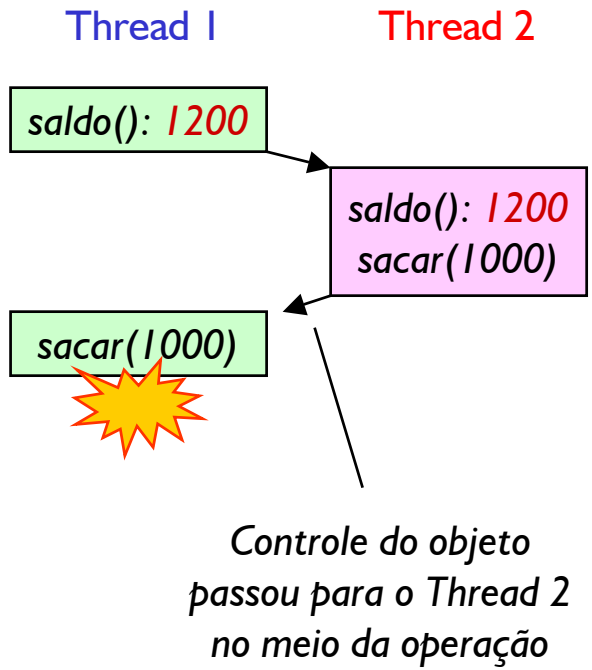
Principais métodos da classe Thread

- Estáticos
  - Thread currentThread(): retorna referência para o thread que está executando no momento
  - void sleep(long tempo): faz com que o thread que está executando no momento pare por tempo milissegundos no mínimo
  - void yield(): faz com que o thread atual pare e permita que outros que estão na sua fila de prioridades executem
- De instância
  - void run(): é o "main" do Thread. Deve ser implementado no Thread ou no objeto Runnable passado ao thread
  - void start(): é um bootstrap. Faz o JVM chamar o run()
  - boolean isAlive(): verifica se thread está vivo

Compartilhamento de recursos limitados

- *Recursos limitados podem ser compartilhados por vários threads simultaneamente*
  - *Cada objeto têm um bloqueio que pode ser acionado pelo método que o modifica para evitar corrupção de dados*
- *Dados podem ser corrompidos se um thread deixar um objeto em um estado incompleto e outro thread assumir a CPU*

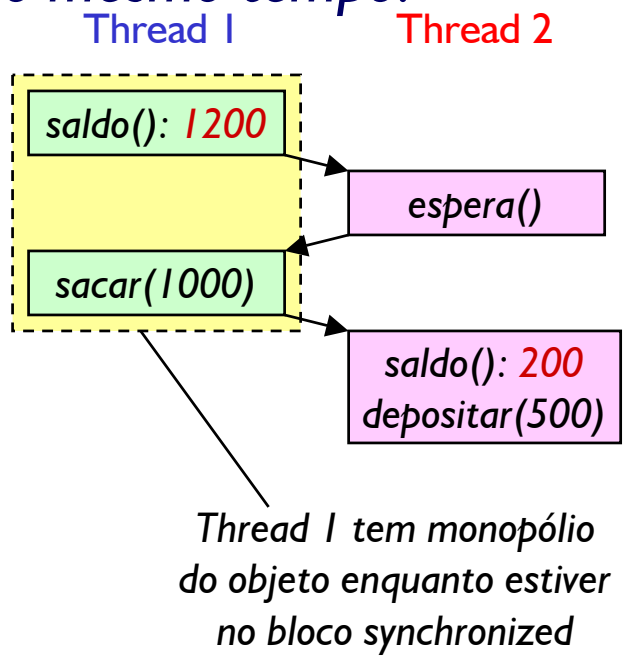
```
void operacaoCritica() {
   if (saldo() > 1000)
     sacar(1000);
   else depositar(500);
}
```



Corrupção

- *Recursos compartilhados devem ser protegidos*
  - *A palavra-reservada synchronized permite que blocos sensíveis ao acesso simultâneo sejam protegidos de corrupção impedindo que objetos os utilizem ao mesmo tempo.*
- *Synchronized deve limitar-se aos trechos críticos (performance!)*

```
void operacaoCritica() {
  // ... trechos thread-safe
  synchronized (this) {
    if (saldo() > 1000)
      sacar(1000);
    else depositar(500);
  } // (... trechos seguros ...)
}
```



- *Métodos inteiros podem ser synchronized*

```
synchronized void operacaoCritica() {}
```

Comunicação entre threads

- Se um recurso crítico está sendo usado, só um thread tem acesso. É preciso que
  - Os outros esperem até que o recurso esteja livre
  - O thread que usa o recurso avise aos outros que o liberou
- Esse controle é possível através de dois métodos da classe Object, que só podem ser usados em blocos synchronized
  - *wait()*: faz com que o Thread sobre o qual é chamado espere por um tempo indeterminado, até receber um...
  - *notify()*: notifica o próximo Thread que o recurso bloqueado foi liberado. Se há mais threads interessados, deve-se usar o
  - *notifyAll()*: avisa a todos os threads.

Exemplo clássico de comunicação (I)

- *A seguinte classe é uma pilha compartilhada por dois threads. Como os métodos push() e pop() contém código que pode corromper os dados, caso não sejam executados atomicamente, eles são synchronized*

```
public class PilhaSincronizada {
   private int index = 0;
   private int[] buffer = new int[10];

   public synchronized int pop() {
      index--;
      return buffer[index];
   }

   public synchronized void push(int i) {
      buffer[index] = i;
      index++;
   }
}
```

Exemplo de comunicação: (2) Produtor

- *O objeto abaixo produz 40 componentes em intervalos de 0 a 1 segundo e os tenta armazenar na pilha.*

```
public class Producer implements Runnable {
   PilhaSincronizada pilha;

   public Producer(PilhaSincronizada pilha) {
      this.pilha = pilha;
   }

   public void run() {
     int colorIdx;
     for (int i = 0; i < 40; i++) {
       colorIdx = (int)(Math.random() * Colors.color.length);
       pilha.push(colorIdx);
       System.out.println("Criado: "+ Colors.color[colorIdx]);
       try {
           Thread.sleep((int)(Math.random() * 1000));
       } catch (InterruptedException e) {}
     }
   }
}
```

Exemplo de comunicação (3) Consumidor

- *O objeto abaixo consome os 40 componentes da pilha mais lentamente, esperando de 0 a 5 segundos*

```
public class Consumer implements Runnable {

   PilhaSincronizada pilha;

   public Producer(PilhaSincronizada pilha) {
      this.pilha = pilha;
   }

   public void run() {
     int colorIdx;
     for (int i = 0; i < 20; i++) {
       colorIdx = pilha.pop();
       System.out.println("Usado: "+ Colors.color[colorIdx]);
       try {
         Thread.sleep((int)(Math.random() * 5000));
       } catch (InterruptedException e) {}
     }
   }
}
```

Monitor com wait() e notify()

- *A pilha foi modificada e agora faz com que os threads executem wait() e notify() ao utilizarem seus métodos*

```
public class PilhaSincronizada {
   private int index = 0;
   private int[] buffer = new int[10];

   public synchronized int pop() {
      while (index == 0) {
         try { this.wait();
         } catch (InterruptedException e) {}
      }
      this.notify();
      index--;
      return buffer[index];
   }

   public synchronized void push(int i) {
      while (index == buffer.length) {
         try { this.wait();
         } catch (InterruptedException e) {}
      }
      this.notify();
      buffer[index] = i;
      index++;
   }
}
```

Apesar de aparecer antes, a notificação só terá efeito quando o bloco synchronized terminar

Problemas de sincronização

- *Quando métodos sincronizados chamam outros métodos sincronizados há risco de deadlock*
- *Exemplo: para alterar valor no objeto C:*
  - *O Thread A espera liberação de acesso a objeto que está com Thread B*
  - *O Thread B aguarda que alguém (A, por exemplo) faça uma alteração no objeto para que possa liberá-lo (mas ninguém tem acesso a ele, pois B o monopoliza!)*
- *Solução*
  - *Evitar que métodos sincronizados chamem outros métodos sincronizados*
  - *Se isto não for possível, controlar liberação e retorno dos acessos (hierarquia de chaves de acesso)*

Exemplo de deadlock

```
public class Caixa {
    double  saldoCaixa = 0.0;
    Cliente clienteDaVez = null;

    public synchronized void atender(Cliente c, int op, double v) {
        while (clienteDaVez != null) { wait(); } //espera vez
        clienteDaVez = c;
        switch (op) {
            case -1:  sacar(c, v); break;
            case  1:  depositar(c, v); break;
        }
    }

    private synchronized void sacar(Cliente c, double valor) {
        while (saldoCaixa <= valor) { wait(); } //espera saldo, vez
        if (valor > 0) {
           saldoCaixa -= valor; clienteDaVez = null;
           notifyAll();
        }
    }

    private synchronized void depositar(Cliente c, double valor) {
        if (valor > 0 ) {
           saldoCaixa += valor; clienteDaVez = null;
           notifyAll();
        }
    }
}
```

Deadlock (2)

- Cenário 1:
  - Saldo do caixa: R$0.00
  - Cliente 1 (thread) é atendido (recebe acesso do caixa), deposita R$100.00 e libera o caixa
  - Cliente 2 (thread) é atendido (recebe o acesso do caixa), saca R$50.00 e libera o caixa
- Cenário 2: cliente 2 chega antes de cliente 1
  - Saldo do caixa: R$0.00
  - Cliente 2 é atendido (recebe acesso do caixa), tenta sacar R$50.00 mas não há saldo. Cliente 2 espera haver saldo.
  - Cliente 1 tenta ser atendido (quer depositar R$100.00) mas não é sua vez na fila. Cliente 1 espera sua vez.
  - DEADLOCK!

Exercícios

- *1. Implemente e rode o exemplo Trabalhador mostrado neste capítulo*
- *2. Altere a classe para que o Thread rode para sempre*
  - *Crie um método parar() que altere um flag que faça o loop infinito terminar*
- *3. Implemente e rode o exemplo Produtor - Consumidor*
- *4. Implemente o exemplo de deadlock e encontre um meio de evitá-lo.*

Curso J100: Java 2 Standard Edition

Revisão 17.0

(helder@acm.org)

argonavis.com.br