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
- Para mais detalhes, consulte
  - o livro texto (capítulo 14) ou o "trail" sobre Threads do Java Tutorial em java.sun.com/tutorial/
  - o minicurso J220 (Programação Concorrente em Java)
  - Exemplos do CD

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

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

Por que usar múltiplos threads?

- Todo programa tem pelo menos um thread, chamado de thread Main.
- Em algumas áreas threads adicionais são essenciais. Em outras, podem melhorar o desempenho
- 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

- Herdar da classe Thread
  - Thread são objetos
- Implementar a interface Runnable
  - Método orientado a procedimento
- 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 loop terminar, e o run(), terminar, o thread morre.

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;
    }
}
```

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

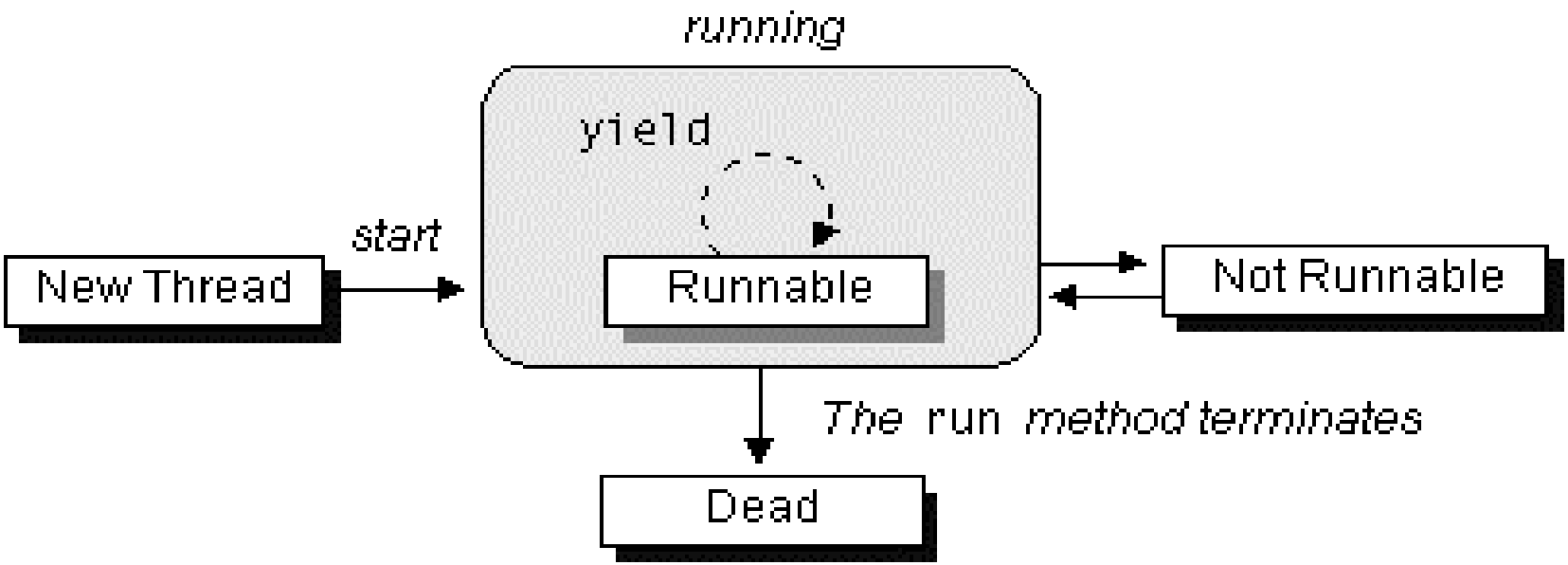
- *Para o Trabalhador que é Thread*

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

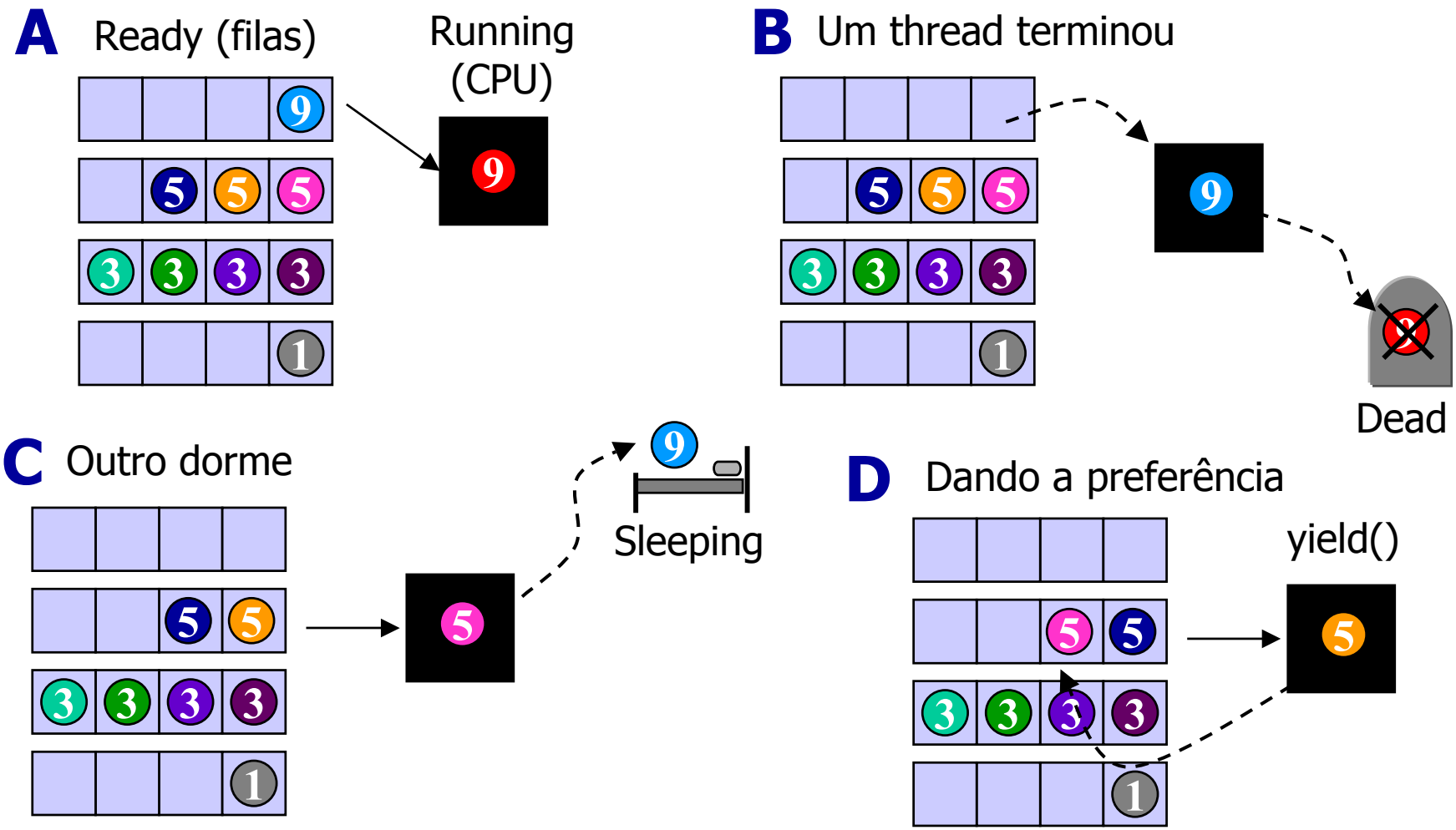
- *Para o Trabalhador que é Runnable*

```
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



Filas de prioridades



Compartilhamento de recursos limitados

- Recursos limitados podem ser compartilhados por vários threads
  - Monitores são usados para determinar quem e quando poderá usar o recurso
  - Cada objeto têm uma trava que pode ser acionado pelo método que o modifica para evitar corrupção de dados
  - Métodos wait(), notify() e notifyAll(), de Object, garantem espera e notificação para qualquer objeto

Corrupção

- *Dados podem ser corrompidos se um thread deixar um objeto em um estado incompleto e outro thread assumir a CPU*
- *Para evitar que isto aconteça, objeto deve ser "travado" usando synchronized*

```
synchronized (this) {
    ... procedimentos críticos
}
```

- *Métodos inteiros podem ser synchronized*

```
public synchronized double
  sacarDinheiro(double grana) {}
```

Problemas de sincronização

- Quando métodos sincronizados chamam outros métodos sincronizados há risco de deadlock
- Exemplo: para alterar valor no objeto C:
  - O objeto A espera liberação de lock que está com objeto B
  - O objeto B aguarda que A faça a alteração em C para que possa liberar o lock
- Solução
  - Evitar que métodos sincronizados chamem outros métodos sincronizados
  - Se isto não for possível, controlar liberação e retorno dos locks

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 é atendido (recebe lock do caixa), deposita R$100.00 e libera o caixa*
  - *Cliente 2 é atendido (recebe o lock 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 lock 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!*

Métodos importantes

- *Da classe Thread*
  - *static sleep(tempo)*
  - *static yield()*
  - *setPriority()*
  - *start()*
- *De Object*
  - *wait()*
  - *notify()*
  - *notifyAll()*

Veja exemplos de monitores e sincronização com wait() e notify() no livro-texto (cap 14)

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*