Threads

Fernando Castor

Professor Adjunto

Centro de Informática

Universidade Federal de Pernambuco

castor@cin.ufpe.br

Elaborado conjuntamente por:

Benito Fernandes, João Paulo Oliveira e Weslley Torres

Processos

“Um processo é basicamente um programa em execução...”

Contém seu próprio espaço de memória.

Competem por tempo de CPU

Threads estão intimamente ligadas aos processos

- “Processos leves”
- Duas ou mais coisas acontecendo em um sistema ao mesmo tempo
- Espaço de endereçamento compartilhado

Processos vs. Threads

THREADS

Mais leves

Endereçamento compartilhado

Cooperam ou Cooperam

Existem dentro de um processo

PROCESSOS

Mais pesados

Endereçamento próprio

Competem por recursos

Possuem ao menos um thread

Benefícios de Threads

Mais leves

- criação e terminação mais rápidas (quando comparadas a processos)
- Comutação de contexto entre duas threads é mais rápida
- menos memória

Benefícios de Threads

A comunicação entre threads é mais rápida do que a comunicação entre processos.

Multiprogramação usando threads é mais portável do que usando múltiplos processos.

- Pode ser mais simples também

Suporte de linguagens

Linguagens modernas como Java e C# possuem suporte nativo a threads

- Ou conceitos parecidos, como atores em Erlang

C e C++ necessitam de biblioteca específica para o mesmo fim

Implementação de threads

Existem duas formas de criar explicitamente um thread em Java:

- Estendendo a classe `Thread`
- Implementando a interface `Runnable`

A tarefa a ser executado pelo thread é implementada pelo método `run()`

A Interface `Runnable`

Através da utilização da interface `Runnable`, também é possível criar threads

- sem precisar estender a classe `Thread`

A criação de uma nova thread é feita através da instanciação de um objeto thread

- usando o objeto que implementa a interface `Runnable`

Herdando da Classe Thead

```
public class HelloThread extends Thread {

    public void run() {
        System.out.println("Hello from a thread!");
    }

    public static void main(String args[]) {
        (new HelloThread()).start();
    }

}
```

Interface Runnable

```
public class HelloRunnable implements Runnable {

    public void run() {
        System.out.println("Hello from a thread!");
    }

    public static void main(String args[]) {
        (new Thread(new HelloRunnable())).start();
    }

}
```

Exemplo

```
3  class Cavalo3 extends Thread {
4
5      // cada cavalo possui uma qde passos atual e
6      // uma colocação
7      private int passos = 1, colocacao;
8
9      // contador conta a qde de cavalos que
10     // já ultrapassaram a linha de chegada
11     private static int contador = 1, qde_cavalos=0;
12
13     // podemos ter um páreo de no máximo 10 cavalos
14     private static Cavalo3[] cavalos = new Cavalo3[10];
15
16     public Cavalo3(String nome) {
17             // a classe Thread pai pode receber um nome;
18             super(nome);
19             cavalos[qde_cavalos] = this;
20             qde_cavalos++;
21     }
22
23     // imprime a posição atual dos cavalos
24     public static void imprime() {
25             for (int i=0; i< qde_cavalos; i++) {
26                     System.out.print("\nCavalo (" +
27                     cavalos[i].getName() + "): ");
28                     for (int j=0; j< cavalos[i].passos; j++)
29                     System.out.print('>');
30             }
31             System.out.println();
32     }
33
```

Exemplo (cont)

```
34      // trecho executável de cada thread
35      public void run() {
36              int tempo;
37              while(true) {
38                      imprime();
39                      passos++;
40                      try {
41                              if (passos == 51) {
42                                      colocacao = contador++;
43                                      break;
44                              }
45                              // pausa por um tempo randomico
46                              // entre 0 a 500 ms
47                              tempo = (int) (Math.random() * 500);
48                              Thread.sleep(tempo);
49                      }
50                      catch (InterruptedException e) { }
51              }
52      }
53
54      public int getColocacao() {
55              return colocacao;
56      }
57  }
```

```
3   public class CorridaDeCavalo3 {
4       public static void main(String[] args)
5       throws InterruptedException {
6               Cavalo3 c1 = new Cavalo3("1");
7               Cavalo3 c2 = new Cavalo3("2");
8               c1.start();
9               c2.start();
10              System.out.println("Classificacao final:");
11              System.out.println("Cavalo (1): "+c1.getColocacao());
12              System.out.println("Cavalo (2): "+c2.getColocacao());
13      }
14  }
```

Escolhendo uma forma de implementação

Herdar de `Thread` é fácil

Java não permite herança múltipla, porém

Deve ser feito através da implementação da interface `Runnable`.

Applets devem ser sub-classes da classe Applet

- applets só podem implementar threads através da implementação da interface `Runnable`.

Estados de uma thread

New

Runnable

Running

Not Runnable

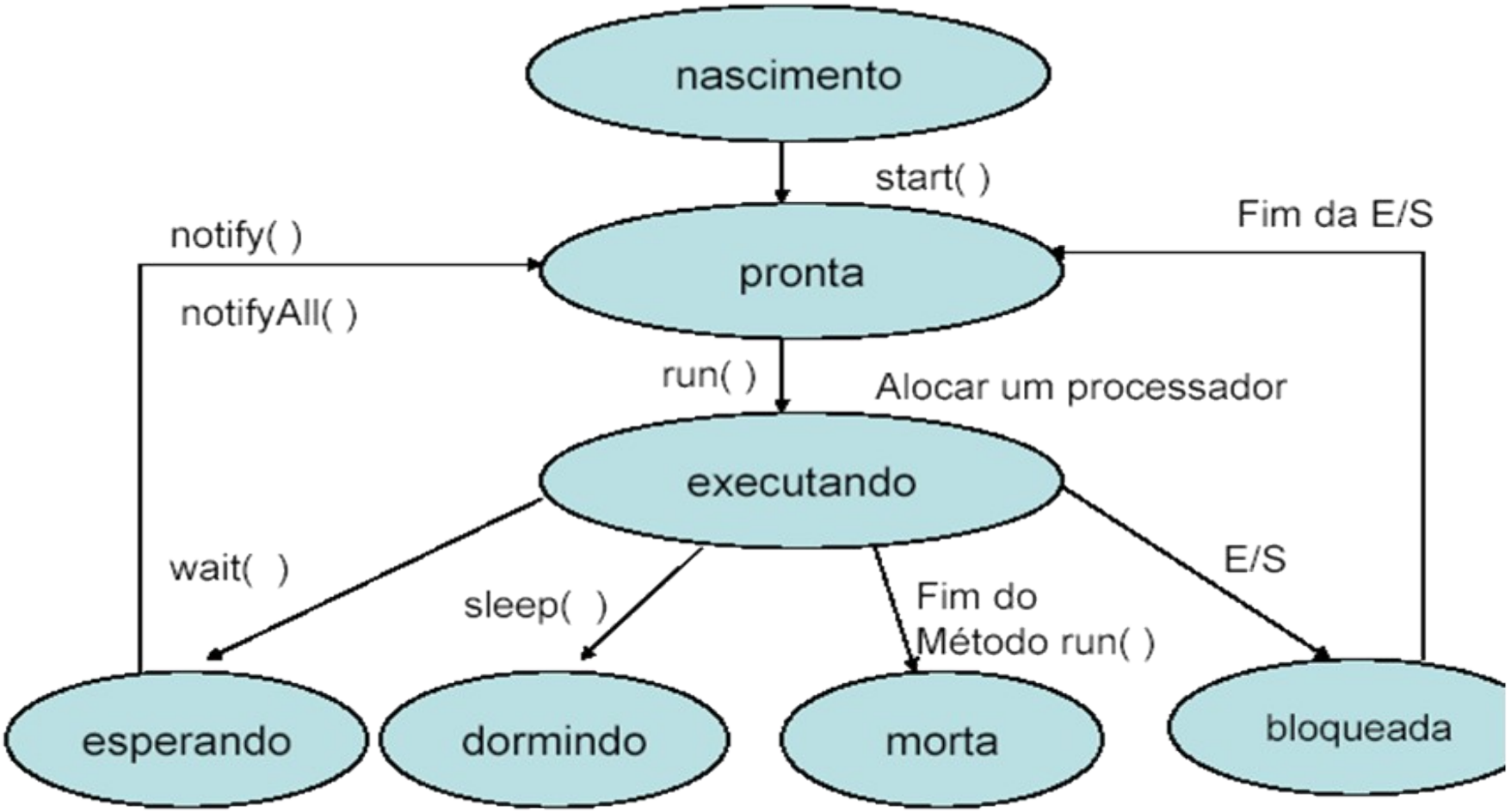
- Bloqueada por uma operação de E/S
- Esperando em uma variável de condição
- Dormindo

Dead

Operações de uma thread

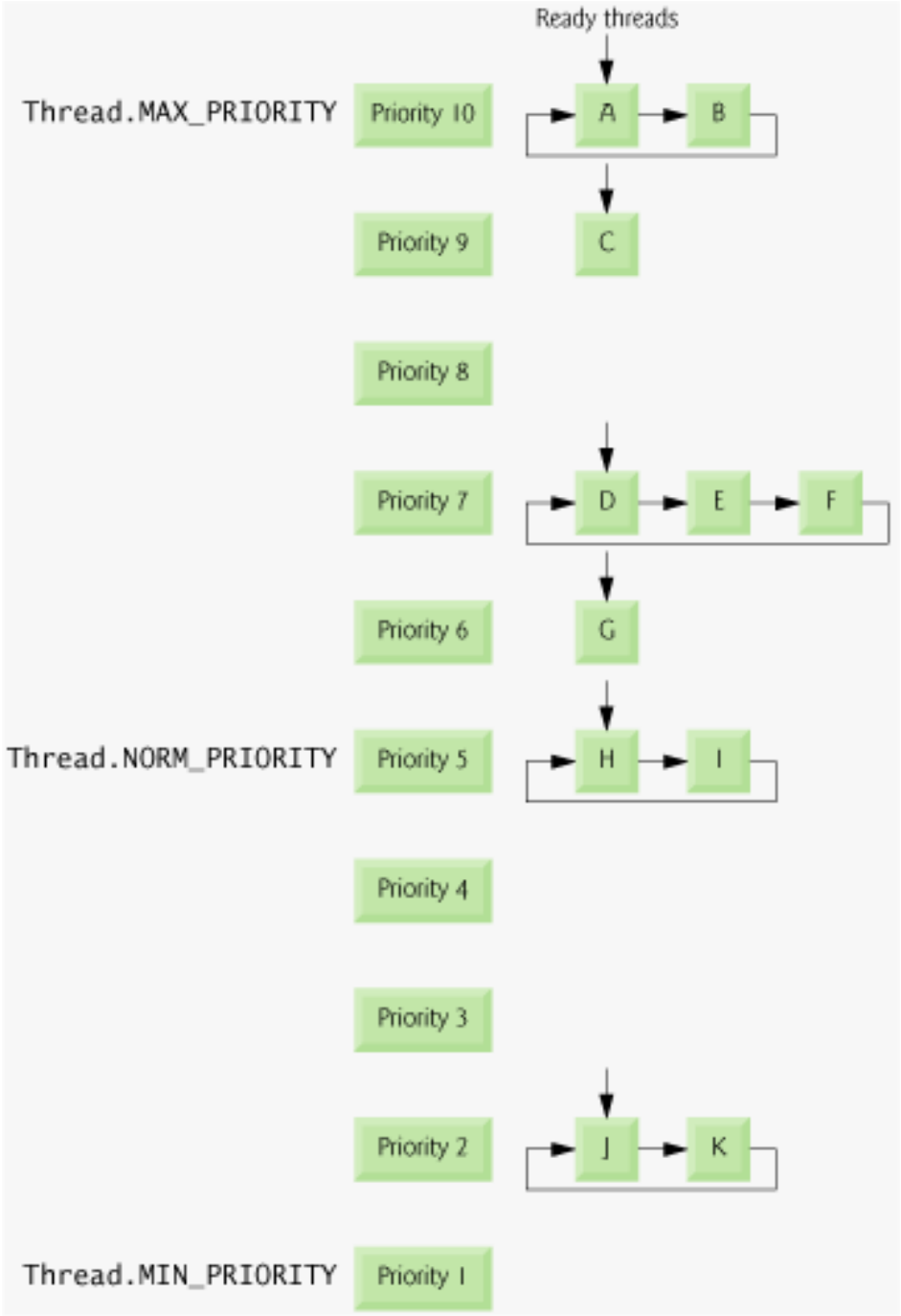
```
start()
run()
join()
sleep()
yield()
setPriority(int)
getPriority()
setName(String)
getName()
```

Ciclo de Vida



Escalonamento

- Mecanismo que determina como os threads irão utilizar tempo de CPU.
- Somente os threads no estado *runnable* são escalonados para ser executados.
- Java permite a atribuição de prioridades para as threads.
- Threads com menor prioridade são escalonados com menor freqüência.



Fonte: Java, Como programar. Deitel, 6ª Edição

Centro de Informática U·F·P·E

Interferência entre *Threads*

Contexto

Threads se comunicam principalmente compartilhando variáveis e referências para objetos

Eficiente, porém pode causar dois tipos de erros

- Interferência entre threads
- Erros de consistência na memória

Interferência entre Threads

Erros introduzidos quando múltiplas *threads* acessam um dado compartilhado

Essa interferência pode fazer com que o resultado não seja esperado

Exemplo – código perigoso para *threads*

```
public class Contador {
    private int contador = 0;

    public void incrementar()
    {
        contador++;
    }
    public void decrementar()
    {
        contador--;
    }
    public int getValor()
    {
        return contador;
    }
}
```

Exemplo – código perigoso para *threads*

```
public class Contador {
    private int contador = 0;

    public void incrementar()
    {
        contador++;
    }
    public void decrementar()
    {
        contador--;
    }
    public int getValor()
    {
        return contador;
    }
}
```

Operação não-atômica!

1. Valor de contador é lido (possivelmente guardado em um registrador)
2. Valor lido é incrementado e armazenado (na memória)

Exemplo – código perigoso para *threads*

```
public class Contador {
    private int contador = 0;

    public void incrementar()
    {
        contador++;
    }
    public void decrementar()
    {
        contador--;
    }
    public int getValor()
    {
        return contador;
    }
}
```

Entre a leitura e o armazenamento, a *thread* atual pode ser interrompida!

Interleaving

Acontece quando operações em *threads* diferentes atuam sobre o mesmo objeto

Interleaving significa que os passos realizados pelas operações se sobrepõem

- De forma não-determinística

Condições de Corrida

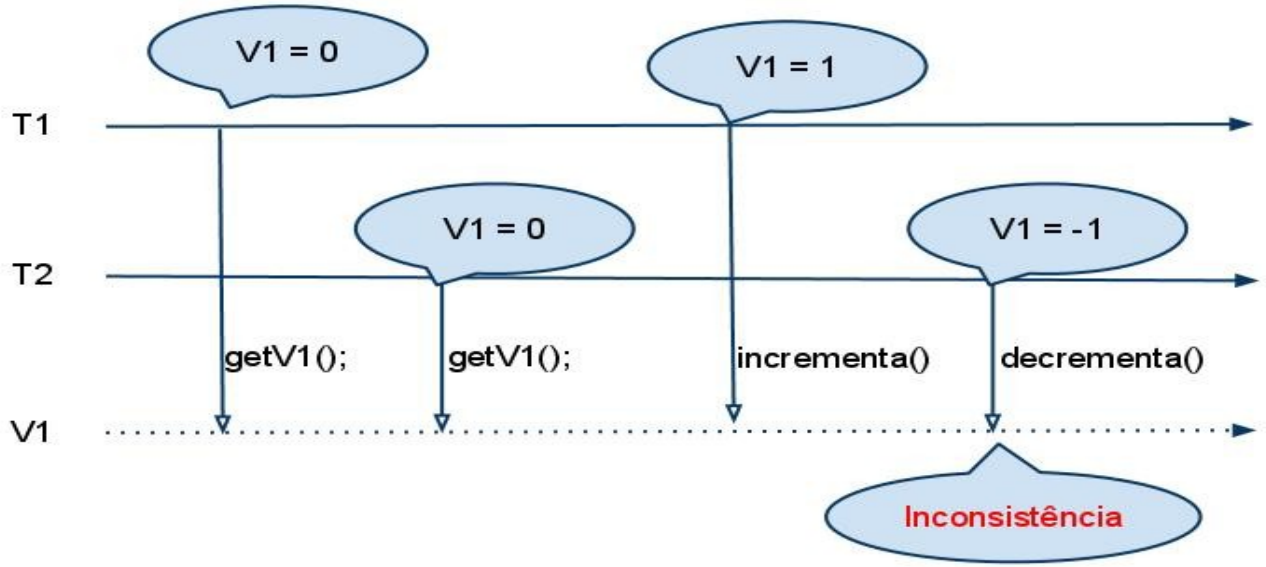
Threads acessam uma variável compartilhada ao mesmo tempo

As *threads* escrevem e lêem a variável

O resultado de uma *thread* pode sobrescrever o da outra.

O resultado da computação pode variar

Condições de corrida - Exemplo

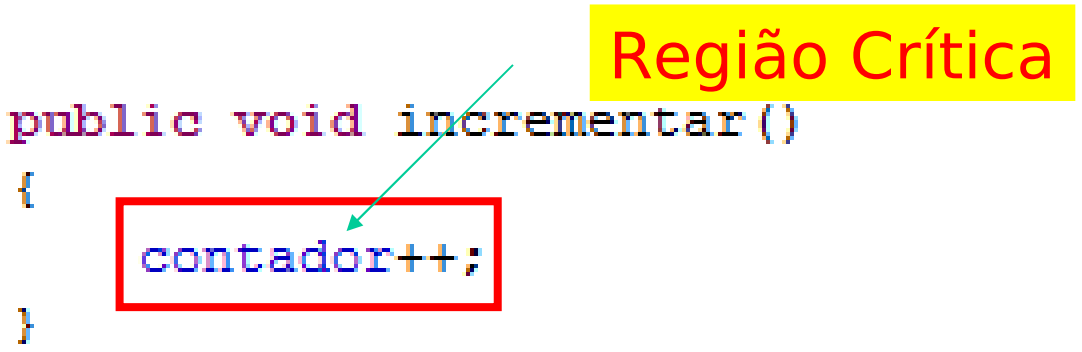


Regiões Críticas

Trechos de código que podem gerar condições de corrida

Para evitar condições de corrida:

- Exclusão mútua nas regiões críticas
- Sincronização entre as *threads* envolvidas



Exclusão mútua

Evita o acesso simultâneo a um recurso compartilhado

- Técnica usada em programação concorrente

Em certas regiões do programa, não mais que uma *thread* de cada vez

Pode não haver nenhuma

Exclusão mútua (cont.)

- Monitores, travas e mutexes...
  - Indicam se uma *thread* tem acesso à região crítica
  - Travamento feito antes do uso o recurso
    - Destravamento é realizado após esse uso
- Enquanto o recurso estiver em uso, qualquer outras threads precisam esperar
  - Pode causar a espera infinita por recursos

Erros de consistência na memória

Threads com diferentes visões sobre um dado

- Por exemplo, por causa de *caching*

Ex. variável inteira compartilhada entre as *threads*

- int contador = 0
- *thread* A incrementa a variável: contador++;
- *thread* B imprime a variável: System.out.println(contador);

O modificador `volatile` evita esse problema