

Programação Concorrente

Nelson Rosa – nsr@cin.ufpe.br



UNIVERSIDADE
FEDERAL
DE PERNAMBUCO

CIn.ufpe.br

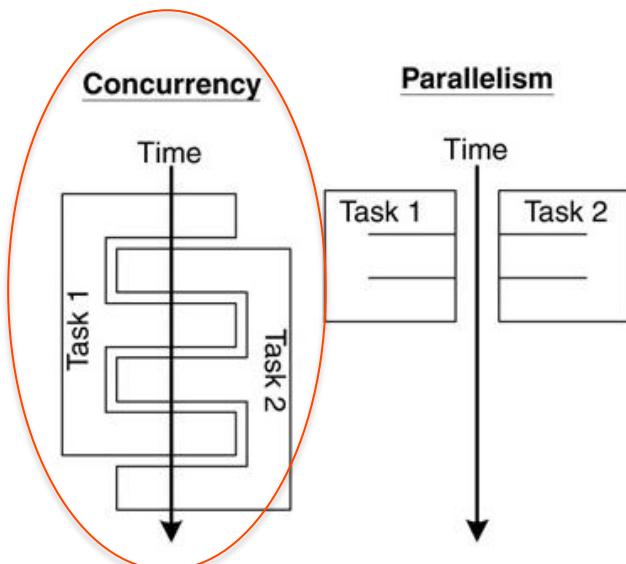
Objetivos

- **Estudar conceitos relacionados à concorrência**
- **Entender como estes conceitos são suportados por Java**
- **Importante: Não pretende ser exaustivo**

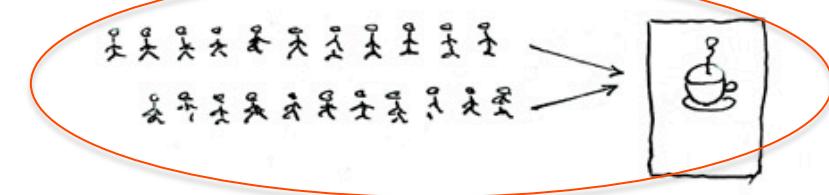
Concorrência

O que é concorrência?

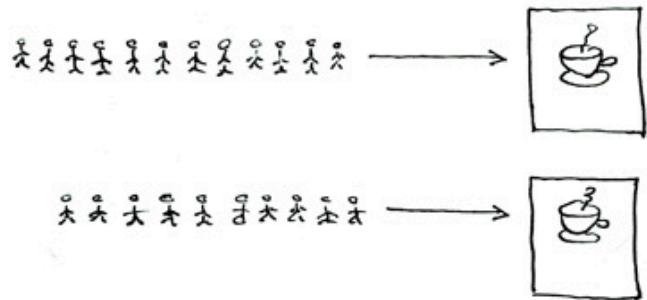
Concorrência é uma **propriedade** dos sistemas onde vários processos são **executados ao mesmo tempo** e podem ou não **interagir** uns com os outros



Concurrent = Two Queues One Coffee Machine



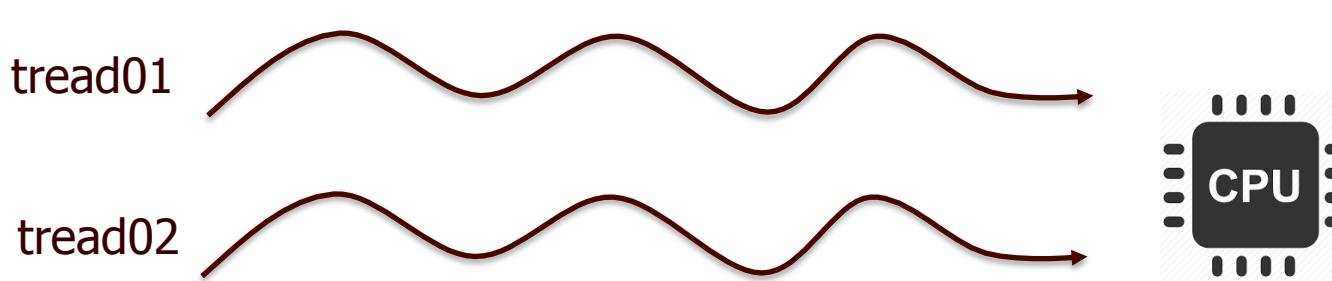
Parallel = Two Queues Two Coffee Machines



© Joe Armstrong 2013

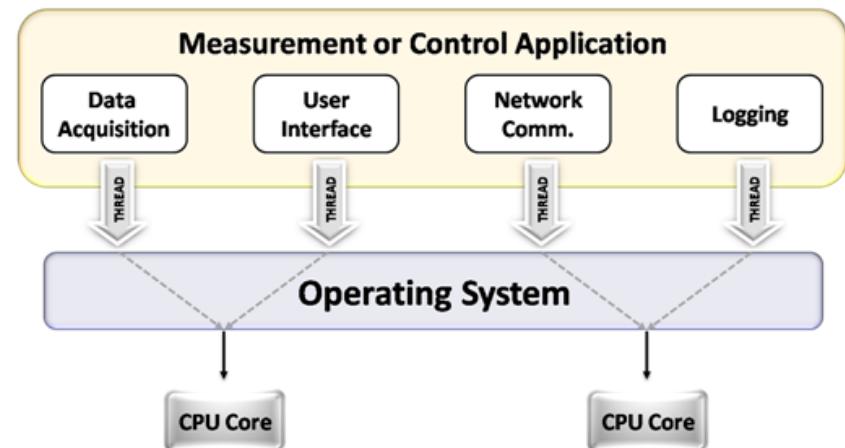
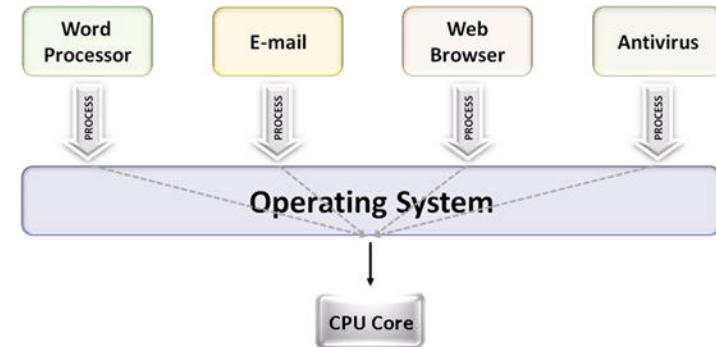
Concorrência

```
public class MyThreadExample {  
    public static Object lock1 = new Object();  
    public static Object lock2 = new Object();  
  
    public static void main(String args[]) {  
        Thread01 thread01 = new Thread01();  
        Thread02 thread02 = new Thread02();  
  
        thread01.start();  
        thread02.start();  
    }  
}
```



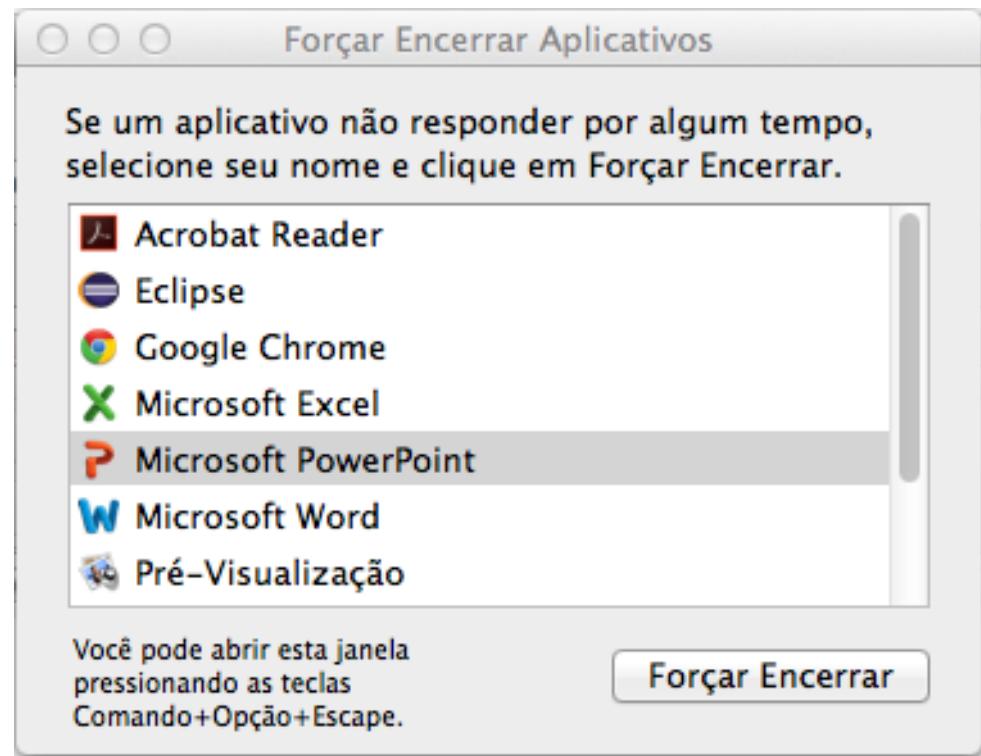
Concorrência:: Motivação

- Usuários querem executar diversas tarefas **simultaneamente** no computador
- Mesmo as aplicações mais simples executam mais de uma “coisa por vez”
 - e.g., formatar um documento no Word enquanto se digita
- Programas capazes disto são chamados **concorrentes**



Concorrência:: Motivação

- **Unidades de execução**
 - Processes e threads
- **Um sistema normalmente tem vários processos e threads ativos**
- **Mesmo em computadores que tem apenas um core, este core é compartilhado entre vários processos e threads (time slicing)**
- **Computadores com vários cores melhoram muito a capacidade de execução concorrente de processos e threads**



Concorrência:: Vantagens

■ Eficiência

- Se uma tarefa executa em um tempo t em um processador, ela poderia executar em t/n em n processadores?

■ Disponibilidade

- Se um processo está ocupado, outro poderia estar pronto para “ajudá-lo”

■ Distribuição

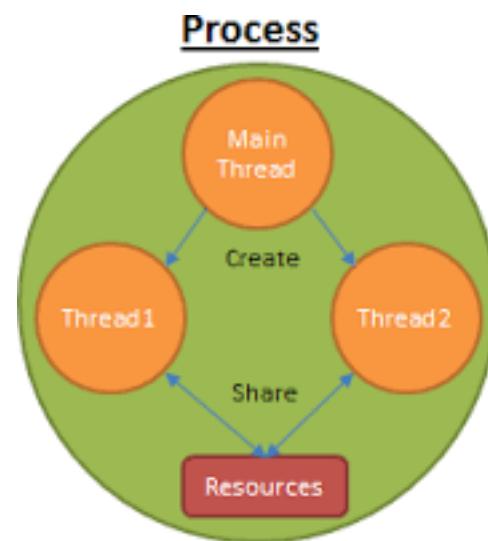
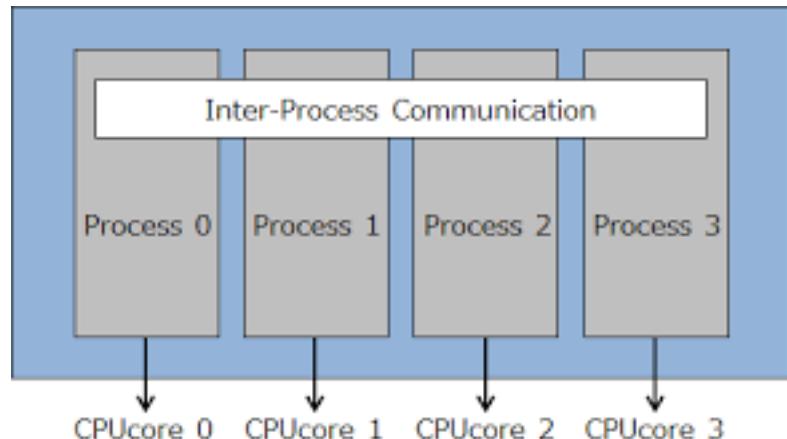
- Processadores em diferentes máquinas poderiam colaborar para resolver um problema ou trabalhar juntos

Concorrência:: Desafios

- É mais difícil implementar programas concorrentes corretos...
 - ... por outro lado, o ganho no desempenho vale o esforço
- Alguns problemas do dia-a-dia são inherentemente sequenciais
- Pontos específicos
 - **Comunicação:** enviar ou receber informação
 - **Sincronização:** esperar por outro processo para “agir”
 - **Atomicidade:** não parar no meio da tarefa e deixar uma bagunça

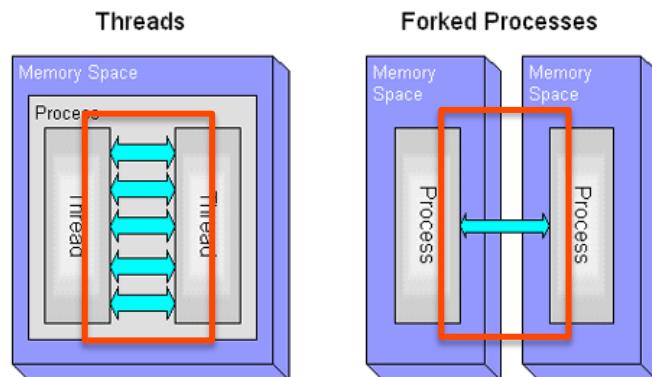
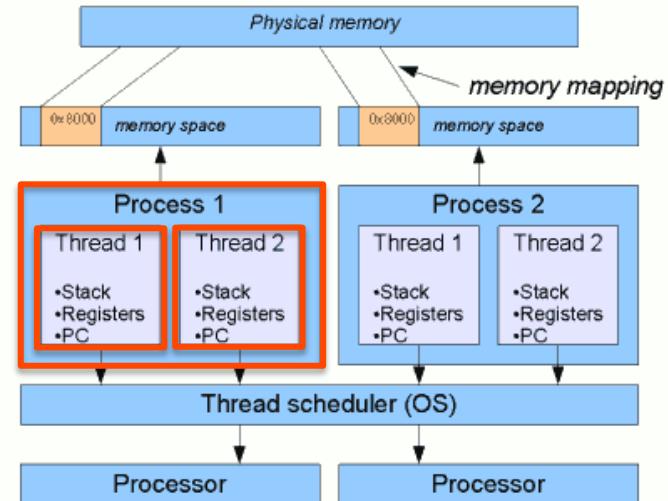
Concorrência:: Conceitos Básicos:: Processos

- Cada processo é um ambiente de execução autocontido, e.g., tem seu próprio espaço de memória
- Sinônimo de programa/ aplicação
- Comunicação através de IPCs (e.g., sockets, pipes)



Concorrência:: Conceitos Básicos:: Threads

- “processos leves”
- Criar novos threads é mais “barato” do que criar novos processos
- Threads “existem” dentro de processos, todo processo tem um+ threads
- Threads **compartilham os recursos do processo** (memória, arquivos abertos)
- Mais eficientes, mais problemáticos



Concorrência:: Conceitos Básicos

■ Multiprogramação

- 01 processador
- Vários programas prontos para executar
- Apenas um programa executa por vez
- Todos os outros programas esperam sua vez para executar

■ Multiprocessamento

- 02 ou mais processadores
- Vários programas prontos para executar
- Refere-se mais ao hardware, enquanto a multiprogração refere-se ao software

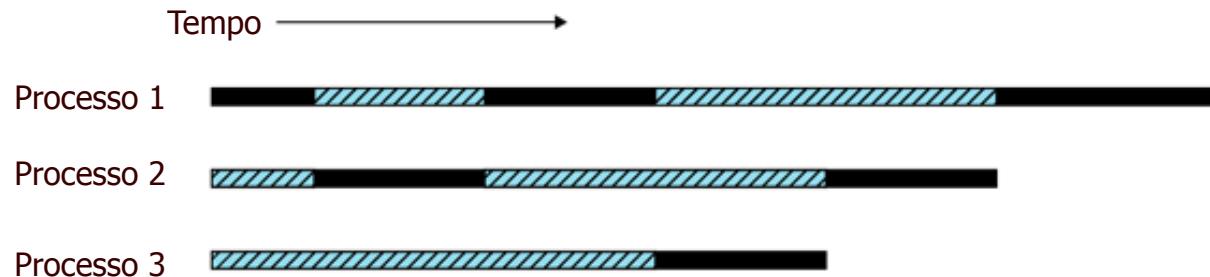
■ Multitarefa

- Similar a multiprogramação, mas referindo-se a múltiplos programas, threads, processos, e tarefas executando ao mesmo tempo

■ Multithreading

- Modelo de execução que permite um único programa ter vários segmentos (threads) executando concorrentemente

Concorrência



Interleaving (**multiprogramação**, 01 processador)



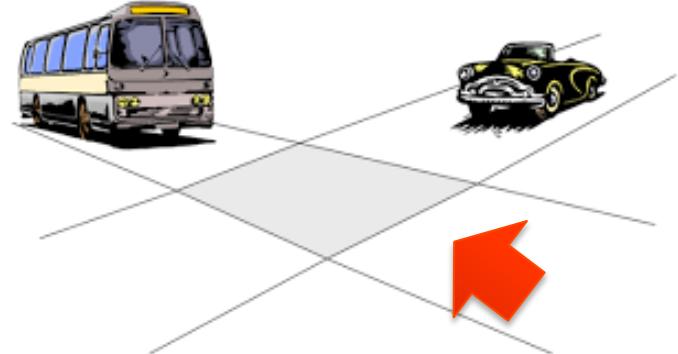
Interleaving (**multiprocessamento**, 02+ processadores)

■ Bloqueado ■ Executando

Concorrência:: Conceitos Básicos

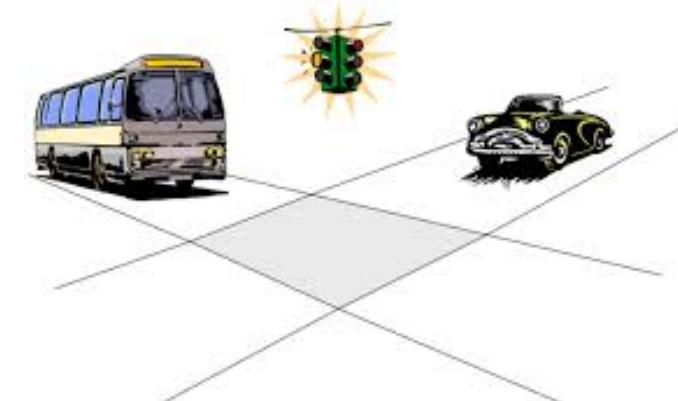
■ Condições de corrida

- Ocorre quando o valor de uma variável depende da ordem de execução de dois ou mais processos concorrentes



■ Região crítica

- Dois processos concorrentes podem acessar um recurso **compartilhado**
- Comportamento **inconsistente** se não houver controle no acesso
- Apenas **um** processo na seção crítica por vez

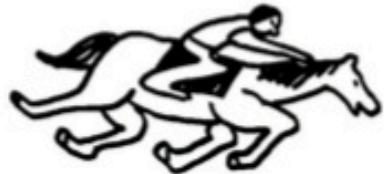


■ Questões básicas

- Como escolher o processo a entrar na seção crítica?
- O que acontece com os **outros** processos?

Concorrência:: Conceitos Básicos

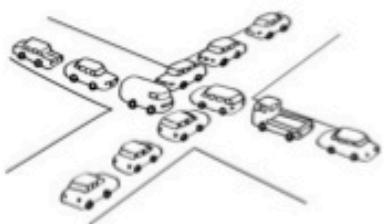
... and its effects



Race conditions



Starvation



Deadlocks



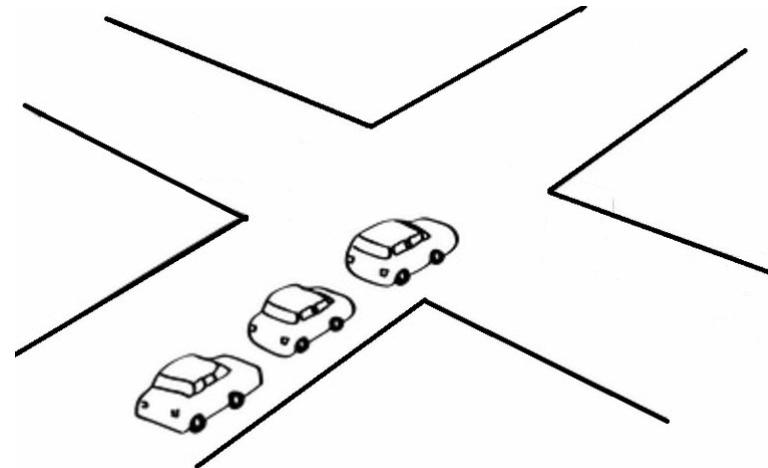
Livelocks

Concorrência:: Conceitos Básicos:: Deadlock

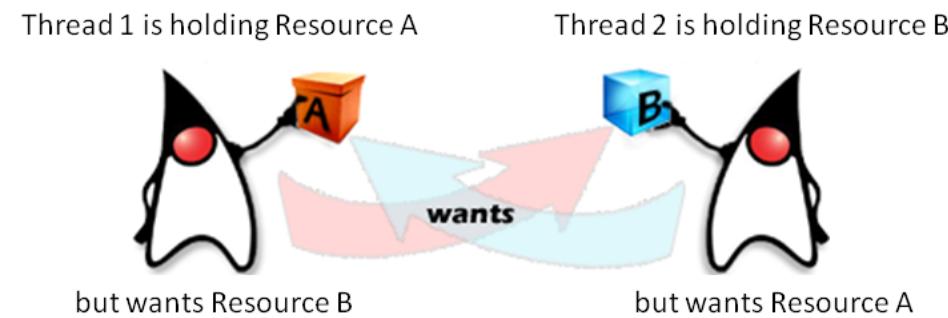
- Deadlock ocorre quando um processo está esperando por um evento que **nunca** ocorrerá

- Condições necessárias para o deadlock existir

- **Exclusão mútua:** Processo requisita acesso exclusivo a recursos
- **Manutenção do recurso:** Processo mantém o recurso enquanto espera por outros recursos
- **Sem preempção:** Recursos não podem ser retirados dos processos que estão esperando
- **Espera circular:** Cadeia circular de processos onde cada processo mantém o recurso necessário para outro processo na cadeia

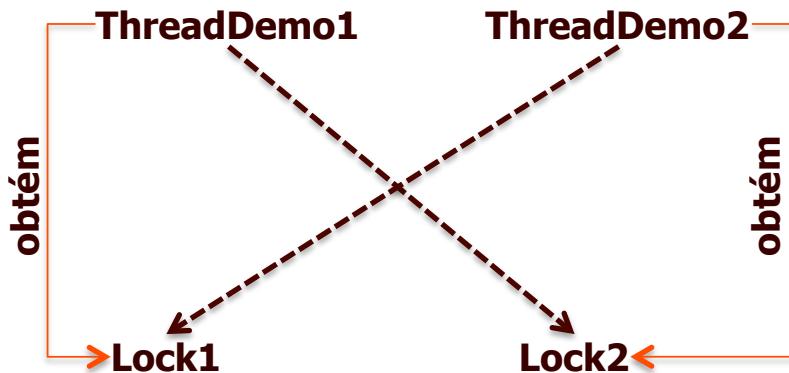


MakeAGIF.com



Concorrência:: Conceitos Básicos:: Deadlock

```
public class ThreadDeadlockSolved {  
    public static Object Lock1 = new Object();  
    public static Object Lock2 = new Object();  
  
    public static void main(String args[]) {  
        ThreadDemo1 thread01 = new ThreadDemo1();  
        ThreadDemo2 thread02 = new ThreadDemo2();  
        thread01.start();  
        thread02.start();  
    }  
}
```



```
...  
private static class ThreadDemo1 extends Thread {  
    public void run() {  
        synchronized (Lock1) {  
            System.out.println("Thread 1: Mantem o lock 1...");  
            try {Thread.sleep(10);}  
            catch (InterruptedException e){}  
            System.out.println("Thread 1: Esperando lock 2...");  
            synchronized (Lock2) {  
                System.out.println("Thread 1: Mantém locks 1&2");  
            }  
        }  
    }  
}  
  
private static class ThreadDemo2 extends Thread {  
    public void run() {  
        synchronized (Lock2) {  
            System.out.println("Thread 2: Mantém o lock 2...");  
            try {Thread.sleep(10);}  
            catch (InterruptedException e) {}  
            System.out.println("Thread 2: Esperando lock 1...");  
            synchronized (Lock1) {  
                System.out.println("Thread 2: Mantém locks 1&2");  
            }  
        }  
    }  
}
```

Concorrência:: Conceitos Básicos:: Starvation & Livelock

■ **Starvation**

- Quando o thread não consegue acesso ao recurso compartilhado e não consegue progredir

■ **Livelock**

- Threads são incapazes de progredir porque eles estão ocupados tentando sincronizar
- e.g., duas pessoas que tentam passar por um corredor estreito e ficam mudando de lado

Suporte à Concorrência em Java



UNIVERSIDADE
FEDERAL
DE PERNAMBUCO

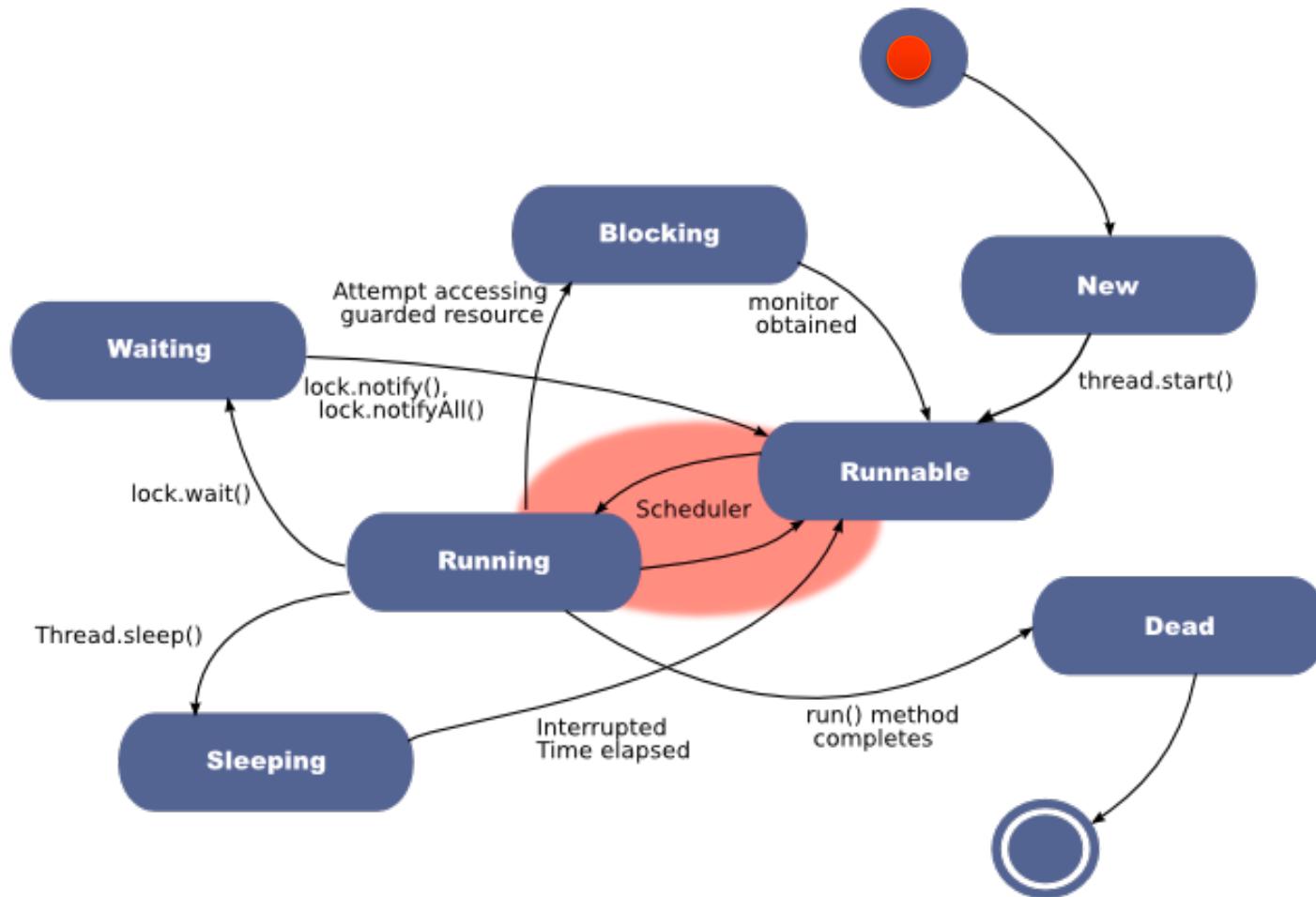
CIn.ufpe.br

Java:: Threads

■ Thread

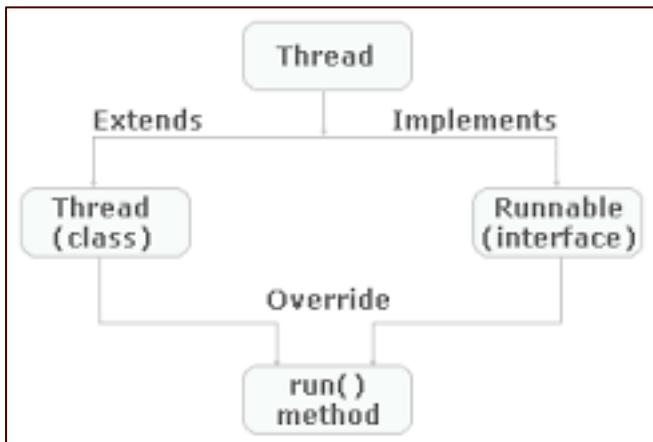
- Conjunto de instruções a serem executadas **uma por vez** e em uma **ordem específica**
- Classe Thread
- Todo programa Java tem **pelo menos um** thread
- Executa independentemente de outros threads, mas pode **acessar recursos compartilhados** de outros threads do mesmo **processo**

Java:: Threads:: Ciclo de Vida



Java:: Threads:: Como usar

- ✓ A aplicação deve fornecer o código que irá executar no thread.
- ✓ Duas formas de usar threads em Java.
- ✓ Objeto Runnable pode ser subclasse de qualquer outra classe além de Thread



```
public class ThreadSubclass extends Thread {  
    public void run() {  
        System.out.println("Hello from a thread!");  
    }  
}  
  
public static void main(String args[]) {  
    Thread thread01 = new ThreadSubclass();  
    thread01.start();  
}
```

```
public class ThreadRunnable implements Runnable {  
    public void run() {  
        System.out.println("Hello from a thread!");  
    }  
}  
  
public static void main(String args[]) {  
    Thread thread1 = new Thread(new ThreadRunnable());  
    thread1.start();  
}
```

Mais fácil de usar

Mais geral

Java:: Threads:: Prioridades

- Todo thread tem uma prioridade
- Threads com prioridade mais alta são preferencialmente executados em relação aqueles com menor prioridade
- Threads criados por outros threads tem a mesma prioridade do thread que o criou
- Thread.MIN_PRIORITY e Thread.MAX_PRIORITY dependem do SO e da JVM

```
public static void main(String args[]) {  
    ThreadPriority thread01 = new ThreadPriority();  
    ThreadPriority thread02 = new ThreadPriority();  
  
    thread01.setPriority(Thread.MAX_PRIORITY);  
    thread02.setPriority(Thread.MIN_PRIORITY);  
    thread01.start();  
    thread02.start();  
}
```

Java:: Threads:: Alguns métodos

- A execução do thread pode ser suspensa por um período de tempo com **Thread.sleep**
- Um thread pode esperar pela execução de outro com **Thread.join**
- Um thread pode ser interrompido com **Thread.interrupt**
- Sinalização de threads: **Wait, notify, notifyAll**

Java:: Threads:: Sincronização

- Threads podem acessar variáveis de forma **compartilhada**
- Se dois threads compartilham uma variável e a ordem de acesso ao recurso é importante, tem-se a “**race condition**”
 - Variável = “seção crítica”
- É preciso **sincronizar** este acesso
- A sincronização pode introduzir “**contenção**”
 - quando dois+ threads tentam acessar o mesmo recurso simultaneamente
 - provocar a execução mais lenta de um deles ou mesmo a suspensão
- A sincronização pode levar a ...
 - **Starvation**
 - **Livelock**
 - **Deadlock**

```
private int c = 0;  
  
public void increment() {  
    c++;  
}  
  
public void decrement() {  
    c--;  
}  
  
public int value() {  
    return c;  
}
```

Java:: Threads:: Sincronização:: Locks intrínsecos

■ Locks intrínsecos

- Garantem acesso exclusivo ao estado dos objetos

■ Todo objeto java tem um lock intrínseco

■ Todo thread que precisa de acesso exclusivo ao objeto precisa “obter” um lock antes de acessá-lo e liberá-lo após o acesso

■ Quando o thread está com o lock, nenhum outro thread pode obtê-lo

- O thread que tentar obter o lock fica bloqueado

■ Lock é “manipulado” pelo synchronized

```
...
private static class ThreadDemo1 extends Thread {
    public void run() {
        synchronized (Lock1) {
            System.out.println("Thread 1: Mantem o lock 1...");
            try {Thread.sleep(10);}
            catch (InterruptedException e){}
            System.out.println("Thread 1: Esperando lock 2...");
            synchronized (Lock2) {
                System.out.println("Thread 1: Mantem locks 1&2");
            }
        }
    }
}
```

Java:: Threads:: Sincronização:: Métodos sincronizados

Como sincronizar threads

- Métodos **synchronized**
- Comandos **synchronized**

```
public class CounterSynchronised {  
    private int c = 0;  
  
    public synchronized void increment() {  
        c++;  
    }  
  
    public synchronized void decrement() {  
        c--;  
    }  
  
    public synchronized int value() {  
        return c;  
    }}
```

```
...  
private static class ThreadDemo1 extends Thread {  
    public void run() {  
        synchronized (Lock1) {  
            System.out.println("Thread 1: Mantem o lock 1...");  
            try {Thread.sleep(10);}  
            catch (InterruptedException e){}  
            System.out.println("Thread 1: Esperando lock 2...");  
            synchronized (Lock2) {  
                System.out.println("Thread 1: Mantem locks 1&2");  
            }  
        }  
    }  
}
```

Java:: Threads:: Sincronização:: Acesso atômico

- **Ações atômicas são executadas por completo de uma vez, i.e., “não podem parar no meio”**
 - Executam completamente ou não são executadas

- **Exemplos**
 - Read/write de varíaveis de referência e a maioria dos tipos primitivos (exceto `long` e `double`)
 - Read/Write de todas as variáveis declaradas como `volatile`

Java:: Threads:: Deadlock

```
public class ThreadDeadlockSolved {  
    public static Object Lock1 = new Object();  
    public static Object Lock2 = new Object();  
  
    public static void main(String args[]) {  
  
        ThreadDemo1 T1 = new ThreadDemo1();  
        ThreadDemo2 T2 = new ThreadDemo2();  
        T1.start();  
        T2.start();  
    }  
}
```

```
...  
private static class ThreadDemo1 extends Thread {  
    public void run() {  
        synchronized (Lock1) {  
            System.out.println("Thread 1: Mantem o lock 1...");  
            try {Thread.sleep(10);}  
            catch (InterruptedException e){}  
            System.out.println("Thread 1: Esperando lock 2...");  
            synchronized (Lock2) {  
                System.out.println("Thread 1: Mantem locks 1&2");  
            }  
        }  
    }  
}  
  
private static class ThreadDemo2 extends Thread {  
    public void run() {  
        synchronized (Lock2) {  
            System.out.println("Thread 2: Mantém o lock 2...");  
            try {Thread.sleep(10);}  
            catch (InterruptedException e) {}  
            System.out.println("Thread 2: Esperando lock 1...");  
            synchronized (Lock1) {  
                System.out.println("Thread 2: Mantém locks 1&2");  
            }  
        }  
    }  
}
```

Java:: Threads:: Deadlock (resolvido)

```
public class ThreadDeadlockSolved {  
    public static Object Lock1 = new Object();  
    public static Object Lock2 = new Object();  
  
    public static void main(String args[]) {  
  
        ThreadDemo1 T1 = new ThreadDemo1();  
        ThreadDemo2 T2 = new ThreadDemo2();  
        T1.start();  
        T2.start();  
    }  
}
```

```
...  
private static class ThreadDemo1 extends Thread {  
    public void run() {  
        synchronized (Lock1) {  
            System.out.println("Thread 1: Mantem o lock 1...");  
            try {Thread.sleep(10);}  
            catch (InterruptedException e){}  
            System.out.println("Thread 1: Esperando lock 2...");  
            synchronized (Lock2) {  
                System.out.println("Thread 1: Mantem locks 1&2");  
            }  
        }  
    }  
}  
  
private static class ThreadDemo2 extends Thread {  
    public void run() {  
        synchronized (Lock1) {  
            System.out.println("Thread 2: Mantém o lock 1...");  
            try {Thread.sleep(10);}  
            catch (InterruptedException e) {}  
            System.out.println("Thread 2: Esperando lock 2...");  
            synchronized (Lock2) {  
                System.out.println("Thread 2: Mantém locks 1&2");  
            }  
        }  
    }  
}
```

Java:: Threads:: Blocos guardados

- Usados para coordenar as ações dos threads
- Bloco inicia com uma condição que deve ser verdadeira para o bloco continuar

```
public void guardedJoy() {  
    while(!joy) {}  
    System.out.println("Joy has been achieved!");  
}
```

```
public synchronized void SyncguardedJoy() {  
    while(!joy) {  
        try {  
            wait();  
        }  
        catch (InterruptedException e) {}  
    }  
    System.out.println("Joy and efficiency have been achieved!");  
}
```

```
public synchronized notifyJoy() {  
    joy = true;  
    notifyAll();  
}
```

Fim dos Slides