UFRPE

Thread

Richarlyson A. D'Emery
site: https://sites.google.com/site/profricodemery/mpoo
grupo: http://groups.google.com/group/**mpoo_uast**
email grupo: **mpoo_uast**@googlegroups.com
contato: rico_demery@yahoo.com.br

Sumário

UFRPE

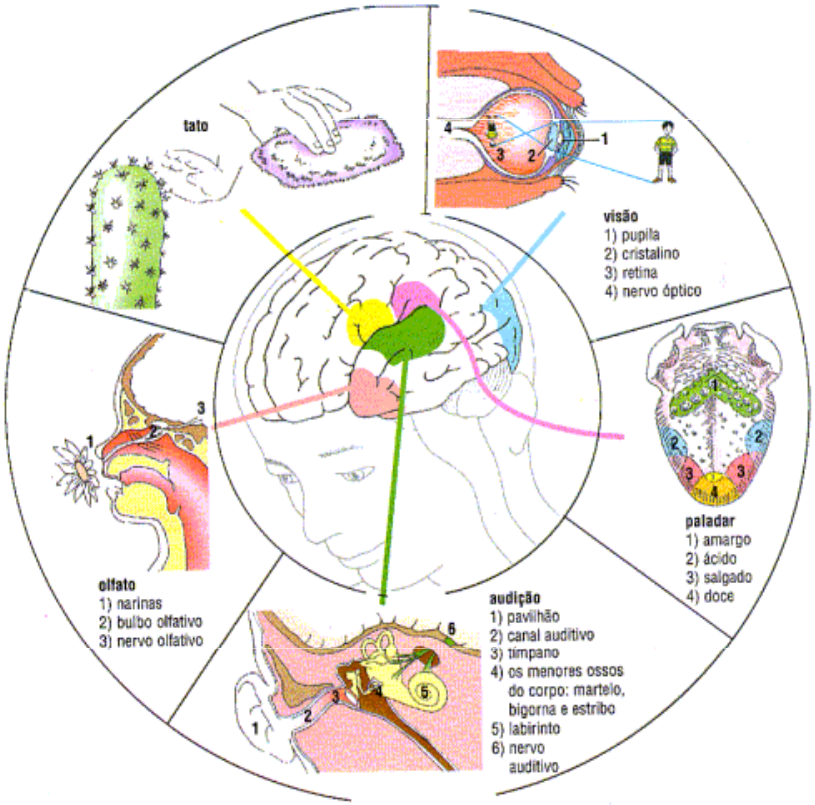
- Objetivos
- Introdução
- Complexidade
- Simutaneidade
- Processo versus Thread
- Programas e processos
- A classe Thread
- Ciclo de Vida
- Sincronização

Objetivos

- Entender Threads e sua utilização
- A classe Thread e seus métodos
- Ciclo de Vida
- Implementação
  - Estendendo da classe `Thread`
  - Implementando a interface `Runnable`
- Sincronização

O que essas coisas possuem em comum?

UFRPE



Resposta: SIMULTANEIDADE

Complexidade

UFRPE

- Experimente:
  - Abra 3 livros na página 1
  - Leia os livros simultaneamente
  - Leia algumas palavras do primeiro livro,
  - Leia algumas palavras do segundo livro,
  - Leia algumas palavras do terceiro livro,
  - Volte ao primeiro livro e leia as próximas palavras
  - E assim por diante

Complexidade

UFRPE

- Problemas
  - Trocar de livro
  - Ler rapidamente
  - Lembrar onde parou em cada livro
  - Mover o livro para mais perto a fim de enxergar melhor
  - Afastar o livro que não está lendo
  - E ainda entender o conteúdo dos livros

Simultaneidade

- A maioria das linguagens de programação não permite atividades simultâneas
  - C, C++ (linguagens de uma única thread)
    - Não possuem multithreading predefinido
- Java disponibiliza primitivas de simultaneidade
  - Fluxos de execução (threads)
  - Também chamado de multithreading ou multiescalonamento
  - Classes disponíveis – pacote `java.lang`
    - Thread
    - ThreadGroup
    - ThreadLocal
    - ThreadDeath

Processo x Thread - Idéia Básica

- Processos
  - programa em execução que contém um único fluxo de execução.
- Threads
  - programa em execução com múltiplos fluxos de execução
  - Processos leves
    - devido ao menor tempo gasto em atividades de criação e escalonamento de threads, se comparada aos processos

Programas e Processos

UFRPE

- Um programa é um conceito estático
  - um programa é um arquivo em disco que contém um código executável por uma CPU.
  - quando executado dizemos que é um processo.
- Processo é um programa em execução
  - Aloca recursos - memória, disco, impressora, CPU
- Um mesmo programa (vários processos) pode ser executado várias vezes simultaneamente
- O sistema operacional controla a execução dos vários processos
  - distribui tempo – algoritmos de prioridade.
  - garante sincronismo entre os processos quando os mesmo precisam trocar informações.

Threads em Java

UFRPE

- Java é interpretado
  - diferença básica entre os processos e programas
  - quem cuida dos vários Threads de um programa é o próprio interpretador Java.
- Vantagens em relação aos processos:
  - chaveamento entre threads é mais rápido que entre processos
  - troca de mensagens entre threads é mais eficiente.
- A eficiência ocorre porque o interpretador tem o controle maior sobre os threads.
- Mas existe grande ineficiência do interpretador

Threads em Java

UFRPE

- Coletor de Lixo
  - Em C e C++ o programador deve recuperar memória alocada dinâmicamente

```
#include <stdio.h>
#include <alloc.h>

int main (void) {
     int *p;
     int a;
     ...
     p=(int *)malloc(a*sizeof(int));
     if (!p) {
       printf ("** Erro: Memoria Insuficient e**");
       exit;
     }
     ...
   free(p);
   ...
   return 0;
}
```

Threads em Java

UFRPE

- Coletor de Lixo
  - Java possui uma thread automática que recupera a memórica alocada dinamicamente de que o programa não precisa mais
  - Thread de baixa prioridade
    - executado sempre que o sistema estiver sem memória
    - objeto não utilizado
      - Referência `null`

Thread em Java

- Problemas
  - Dependente de plataforma
  - Solaris
    - Threads de alta prioridade são executadas até o fim ou até surgir outra de prioridade maior (**preempção**)
  - Aplicações 32 bits
    - Possuem *quantum* de tempo
    - Threads ficam em espera até todas as outras de igual prioridade serem executadas.

A Classe Thread

- Construtor
  - `public Thread ()`
  - `public Thread (String nomeThread)`
- Métodos
  - `run()`
    - Implementado em um objeto `Runnable` (interface Java)
  - `start()`
    - Dispara uma `IllegalThreadStateException` se já estiver sido iniciada
  - `static sleep()`
    - Coloca uma thread para dormir e enquanto dorme não utiliza o processador
  - `yield()`
    - Pausa thread, permitindo outra ser despachada
  - `interrupt()`
    - Interrompe uma thread
    - Retorna `true` se a thread atual for interrompida e `false` caso contrário

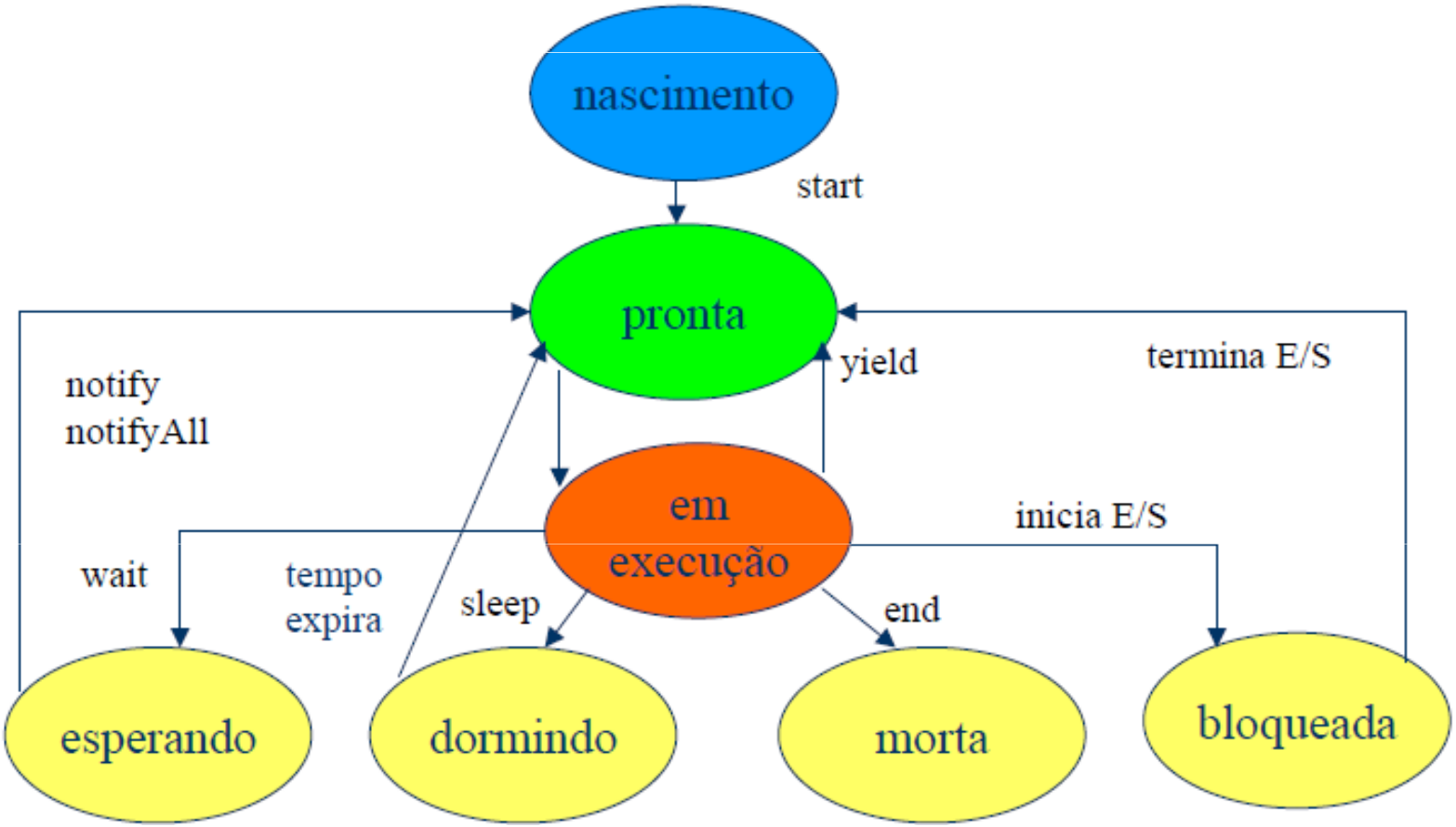
A Classe Thread

UFRPE

- Outros métodos
  - `isInterrupted()`
    - Informa se um thread foi interrompida
  - `isAlive ()`
    - Retorna `true` se `start()` foi chamado e a thread não estiver morta
      - `run()` continua em execução
  - `setName ()`
    - Configura o nome de uma thread
  - `getName ()`
    - Retorna o nome de uma thread
  - `toString ()`
    - Retorna uma String com o nome, a prioridade e o grupo

Ciclo de Vida

UFRPE



Criando Threads em Java

- Duas maneiras possíveis
  - Estendendo da classe `Thread`
  - Implementando a interface `Runnable`

Criando Threads em Java – Exemplo1

- Estendendo da classe `Thread`

```
public class PingPong extends Thread{
      private String palavra;
      private int tempoEspera;
      public PingPong (String texto, int tempo){
            palavra=texto;
            tempoEspera=tempo;
      }
      public void run(){
            try{
                  for (;;){
                        System.out.println(palavra);
                        Thread.sleep(tempoEspera);
                  }
            }catch (Exception e){
               System.err.println("Ocorreu um erro");
               return;
            }
      }
      public static void main(String [] args){
            PingPong ping = new PingPong("ping", 500);
            ping.start();

            PingPong pong = new PingPong("pong", 1000);
            pong.start();
      }
}
```

O `start()` faz com que a JVM invoque o `run` da Thread

Criando Threads em Java – Exemplo2

UFRPE

- Implementando a interface `Runnable`

```
public class PingPongRunnable implements Runnable{
      private String palavra;
      private int tempoEspera;

      public PingPongRunnable (String texto, int tempo){
            palavra=texto;
            tempoEspera=tempo;
      }
      public void run(){
            try{
                  for (;;){
                        System.out.println(palavra);
                        Thread.sleep(tempoEspera);
                  }
            }catch (Exception e){
                  System.err.println("Ocorreu um erro");
                  return;
            }
      }
      public static void main(String [] args){
            Runnable ping = new PingPongRunnable("ping", 500);
            Runnable pong = new PingPongRunnable("pong", 1000);
            new Thread(ping).start();
            new Thread(pong).start();
      }
}
```

Criando Threads em Java

UFRPE

- Múltiplas Threads
  - Exemplo3
    - Estendendo da classe `Thread`
      - `ImprimindoThread.java`
  - Exemplo4
    - Implementando a interface `Runnable`
      - `ImprimindoThreadRunnable.java`
  - Exemplo5
    - Com Prioridade
      - Estendendo da classe `Thread`
        - » `ThreadSimples.java`
      - Aplicação
        - » `TesteDuasThreadsComPrioridade.java`

Sincronização

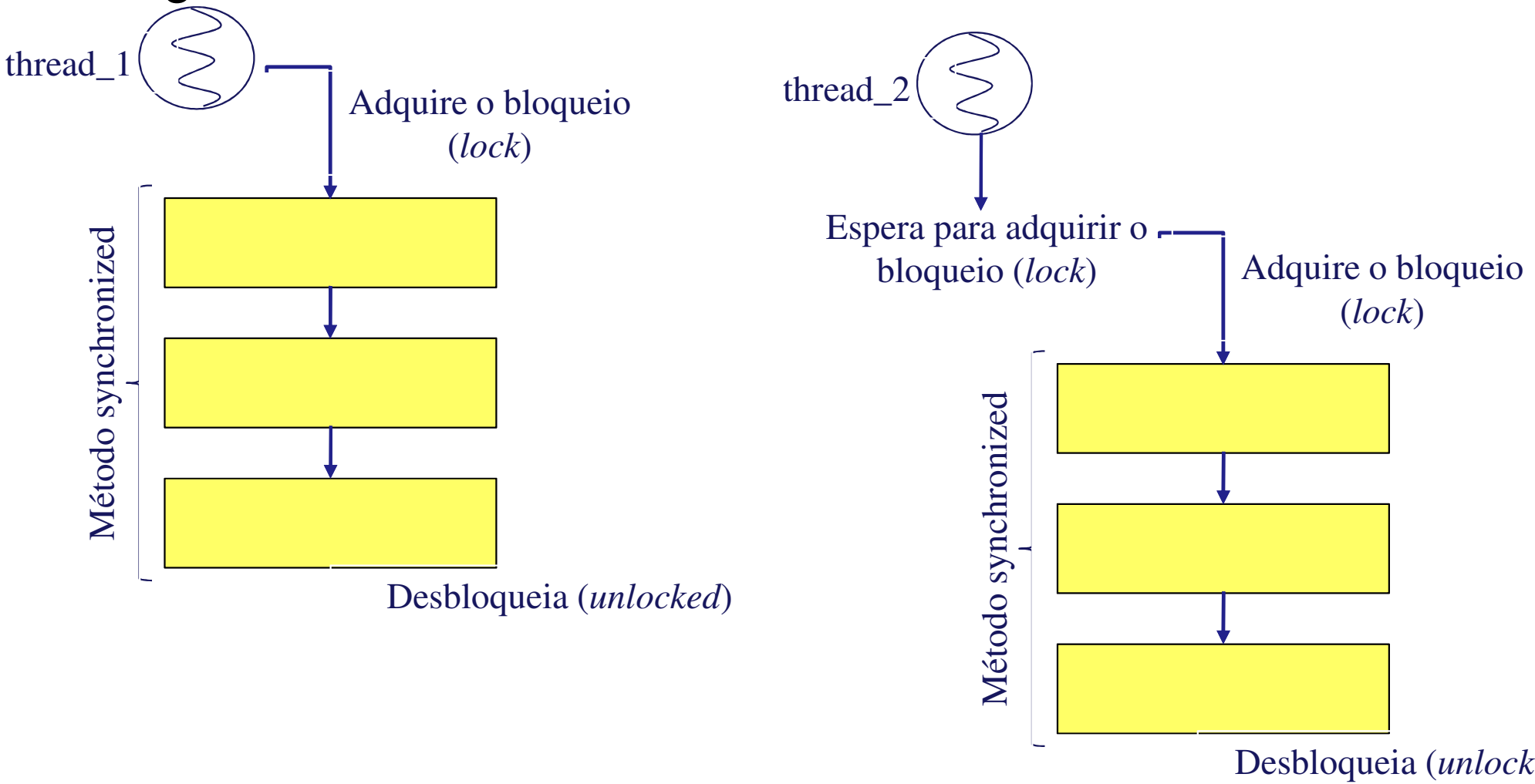
UFRPE

- Quando duas *threads* precisam utilizar ao mesmo tempo um objeto existe a possibilidade de haver corrupção de dados.
  - *Regiões críticas*
- Solução: ACESSO SINCRONIZADO

Sincronização

UFRPE

- Algoritmo:



Sincronização

- Algoritmo:
  - As *threads* devem estabelecer um acordo de forma que antes que qualquer instrução de uma região crítica seja executada um *bloqueio* do objeto deve ser adquirido.
  - Estando o objeto *bloqueado* qualquer outra *thread* fica impossibilitada de acessá-lo até que o objeto fique liberado (*desbloqueado).*
  - Cada objeto tem seu próprio *bloqueio.*
  - O *bloqueio* pode ser adquirido ou liberado através do uso de métodos ou instruções *synchonized.*
  - O objeto fica atomicamente bloqueado quando o seu método *synchronized* é invocado.

Sincronização

UFRPE

- Idéia do Algoritmo:
  - A sincronização força com que as execuções de duas ou mais *threads* sejam mutuamente exclusivas no mesmo espaço de tempo.
  - O *bloqueio é* automaticamente liberado assim que o método *synchronized* termina.

Sicronizando Threads em Java

UFRPE

- Exemplo6

```
public class ContaBancaria {
     private int numeroConta;
     private double saldoAtual;

     public ContaBancaria (int numero, double depositoInicial){
          numeroConta+=numero;
          saldoAtual=depositoInicial;
     }

     public synchronized double getSaldo() {
          return saldoAtual;
     }

     public synchronized void deposito (double valor) {
          saldoAtual+=valor;
     }
}
```

Sincronização

UFRPE

- *synchonized Statements*
  - Permite que a sincronização seja feita apenas em uma porção do código.
  - Como a sincronização afeta a performance, este processo é mais eficaz.
  - Com *synchronized statements* somente fica *bloqueado* apenas o necessário.
  - Sintaxe:

```
synchonized (objeto que será locked) {
    statements
}
```

UFRPE

FIM

Richarlyson D'Emery
rico_demery@yahoo.com.br