

LPRM
Laboratório de Pesquisa em Redes e Multimídia



Threads em Java

(Aula 17)

Universidade Federal do Espírito Santo
Departamento de Informática

LPRM Laboratório de Pesquisa em Redes e Multimídia

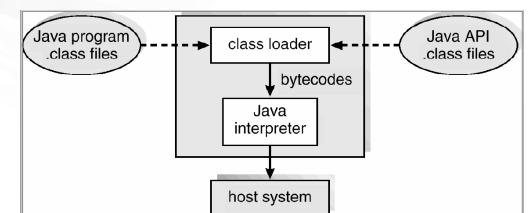
Java Threads

- Difícil de classificar com user thread ou kernel thread
- As threads Java são gerenciadas pela JVM.
- Threads em Java podem ser criadas das seguintes maneiras:
 - Fazendo "extend" da classe Thread.
 - Implementando a interface Runnable.
- A JVM só suporta um processo
 - Criar um novo processo em java implica em criar uma nova JVM p/ rodar o novo processo

Prof.ª Patrícia D. Costa LPRM/DI/UFES 2 Sistemas Operacionais 2008/1

LPRM Laboratório de Pesquisa em Redes e Multimídia

Java

- Componentes:
 - Especificação da linguagem de programação
 - API Java
 - JVM: Máquina Virtual Java
- JVM:
 

Prof.ª Patrícia D. Costa LPRM/DI/UFES 3 Sistemas Operacionais 2008/1

LPRM Laboratório de Pesquisa em Redes e Multimídia

JVM

- Programas Java compilados são bytecodes independentes de plataforma de execução da JVM
- A JVM consiste de:
 - Carregador de classes (Class loader)
 - Verificador de classes (Class verifier)
 - Interpretador runtime (Runtime interpreter)

Prof.ª Patrícia D. Costa LPRM/DI/UFES 4 Sistemas Operacionais 2008/1

Java Threads

- Toda aplicação Java tem pelo menos uma thread (sem considerar o system thread)
- Do ponto de vista do desenvolvedor, o programa começa com uma thread, chamada de *main thread*.
- A main thread cria novas threads.

Java Threads ⁽²⁾

- Cada thread é associada com uma instância da classe Thread.
- Duas estratégias possíveis para criar uma thread:
 - Instanciando a classe Thread;
 - Delegando criação/gerência da thread para "executor" (high-level concurrency objects)

Exemplo de criação de Thread em Java

- A aplicação que cria instância de Thread deve fornecer o código a ser executado na thread.
- Passando um objeto Runnable para o construtor da classe Thread:

```
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 de criação de Thread em Java ⁽²⁾

- Fazendo subclass de Thread (que também implementa Runnable)

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

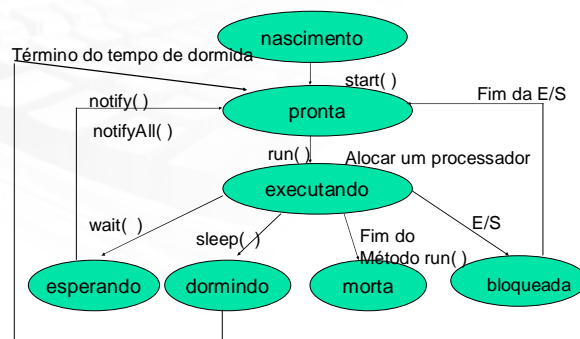
Principais métodos da Classe Thread

- `run()`: é o método que executa as atividades de uma thread. Quando este método finaliza, a thread também termina.
- `start()`: método que dispara a execução de uma thread. Este método chama o método `run()` antes de terminar.
- `sleep(int x)`: método que coloca a thread para dormir por x milissegundos.

Principais métodos da Classe Thread

- `join()`: método que espera o término da thread para qual foi enviada a mensagem para ser liberada.
- `interrupt()`: método que interrompe a execução de uma thread.
- `isInterrupted()`: método que testa se uma thread está ou não interrompida.

Estados da thread



Sincronização de Threads em Java

- O controle de execução de sessões críticas em Java é garantido por um mecanismo de monitores
- Sessão de código (RC) é executada por apenas uma thread em um determinado instante de tempo
- Thread que deseja executar a RC, pede permissão para execução;
 - Se tiver permissão, executa região bloqueando entrada para outras threads
 - Se não tiver a permissão (outra thread está executando), bloqueia até liberar

<http://java.sun.com/docs/books/tutorial/essential/concurrency/index.html>

Lprm Laboratório de Pesquisa em Redes e Multimídia

Monitores em Java

Baseada na abordagem de Hansen, mas sem garantias de que uma *waiting thread* entrará no monitor! Portanto, um signal é apenas uma indicação (para os processos em espera) de que o estado desejado existe. Isto significa que os processos que estão esperando devem sempre checar a condição antes começar a executar no monitor.

Prof.ª Patrícia D. Costa LPRM/DI/UFES 13 Sistemas Operacionais 2008/1

Lprm Laboratório de Pesquisa em Redes e Multimídia

Sincronização de Threads em Java (cont.)

- Todos os objetos Java são potencialmente *monitores*
- Objetos e classes possuem um lock intrínseco
- Para fazer chamada a algum método de um objeto compartilhado (monitor), deve-se obter o lock
- O mesmo para métodos de classe (static)
- A palavra *synchronized* é usada para se obter o lock de um objeto monitor (ou classe) em java

Prof.ª Patrícia D. Costa LPRM/DI/UFES 14 Sistemas Operacionais 2008/1

Lprm Laboratório de Pesquisa em Redes e Multimídia

Granularidade de Sincronização

- Classe
- Objeto
- Bloco de código

Prof.ª Patrícia D. Costa LPRM/DI/UFES 15 Sistemas Operacionais 2008/1

Lprm Laboratório de Pesquisa em Redes e Multimídia

Sincronização de Classe

- Garante que apenas uma thread vai executar um (*synchronized*) método no escopo de uma classe (static method)

```

class Buffer {
    private static Pilha p;
    initialize { p = new Pilha(); }
    public Buffer(){}
    synchronized static public void Insere( Object _o ) {
        p.Push( _o );
    }
    synchronized static public Object Retira() {
        return p.Pop();
    }
}

```

Prof.ª Patrícia D. Costa LPRM/DI/UFES 16 Sistemas Operacionais 2008/1

Sincronização de Objeto

- Garante que apenas uma thread vai executar um (*synchronized*) método no escopo de um objeto

```
class Buffer {
    private Pilha p;
    public Buffer(){
        p = new Pilha();
    }
    synchronized public void Inserir( Object _o ) {
        p.Push( _o );
    }
    synchronized public Object Retira() {
        return p.Pop();
    }
}
```

Sincronização de Bloco

- Garante que apenas uma thread vai executar o conjunto de instruções definidas dentro do bloco
- É usado um objeto auxiliar (e.x. *this*) para garantir exclusão mútua

```
class Buffer {
    private Pilha p;
    public Buffer(){
        p = new Pilha();
    }
    public void Inserir( Object _o ) {
        synchronized(this)
        { p.Push( _o ); }
    }
    synchronized public Object Retira()
    {
        Object aux;
        synchronized(this)
        { aux = p.Pop(); }
        return aux;
    }
}
```

Sincronização de Bloco (métodos independentes)

- São usados objetos auxiliares distintos para evitar exclusão mútua (*interleave*) entre métodos

```
public class MsLunch
{
    private long c1 = 0;
    private long c2 = 0;
    private Object lock1 = new Object();
    private Object lock2 = new Object();
    public void incl() {
        synchronized(lock1){
            c1++;
        }
    }
    public void inc2() {
        synchronized(lock2){
            c2++;
        }
    }
}
```

Sincronização Reentrante

- Uma thread não pode obter um lock que está com outra thread
- Mas uma thread pode obter um lock que ela já possui
- Permite um método (*synchronized*) chamar outro método também *synchronized*
- Evita que uma thread cause o bloqueio de si mesma

Guarded Blocks

- Usados para coordenar as ações das threads
- Um "guarded block" começa perguntando (*polling*) se uma condição é verdadeira

```
// joy é uma variável compartilhada
public synchronized guardedJoy(){
    while(!joy){
        try {
            wait(); /* bloqueia a thread até algum evento ser
notificado. Note que pode ser qualquer evento, não
necessariamente o que esperamos */
        } catch (InterruptedException e) {}
    }
    System.out.println("Joy and efficiency have been achieved!");
}
```

Guarded Blocks (cont)

- Sempre use o "wait" dentro de um loop que testa se a condição esperada foi satisfeita (Abordagem de Hansen). Não assuma que a interrupção foi para a condição específica que estávamos esperando
- Mesmo que o evento tenha sido o que estávamos esperando, pode ser que a condição não seja mais a mesma quando a thread ganhar permissão de executar

Wait

- Quando uma thread executa *o.wait*, ela precisa ter o lock intrínseco do objeto *o*
- Executar o wait dentro de um método synchronized é uma maneira simples de obter o lock
- Quando uma thread chama o wait, a thread libera o lock e suspende a execução

NotifyAll (cont.)

- Uma outra thread que obtenha o lock vai chamar *o.notifyAll*, que informa a todas as threads que estejam esperando por aquele lock que algo importante aconteceu

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

Exemplo dos Filósofos

```
class DiningServer {
    private boolean checkStarving = false;
    private int numPhils = 0;
    private int[] state = null;
    private static final int
        THINKING = 0, HUNGRY = 1, STARVING = 2, EATING = 3;

    public DiningServer(int numPhils, boolean checkStarving) {
        this.numPhils = numPhils;
        this.checkStarving = checkStarving;
        state = new int[numPhils];
        for (int i = 0; i < numPhils; i++) state[i] = THINKING;
        System.out.println("DiningServer: checkStarving="
            + checkStarving);
    }

    private final int left(int i) { return (numPhils + i - 1) % numPhils; }

    private final int right(int i) { return (i + 1) % numPhils; }
}
```

Exemplo dos Filósofos (cont.)

```
private void seeIfStarving(int k) {
    if (state[k] == HUNGRY && state[left(k)] != STARVING &&
        state[right(k)] != STARVING) {
        state[k] = STARVING;
        System.out.println("philosopher " + k + " is STARVING");
    }
}

private void test(int k, boolean checkStarving) {
    if (state[left(k)] != EATING && state[left(k)] != STARVING &&
        (state[k] == HUNGRY || state[k] == STARVING) &&
        state[right(k)] != STARVING && state[right(k)] != EATING)
        state[k] = EATING;
    else if (checkStarving)
        seeIfStarving(k); // simplistic naive check for starvation
}
```

Exemplos

```
public synchronized void takeForks(int i) {
    state[i] = HUNGRY;
    test(i, false);
    while (state[i] != EATING)
        try {wait();} catch (InterruptedException e) {}
}

public synchronized void putForks(int i) {
    state[i] = THINKING;
    test(left(i), checkStarving);
    test(right(i), checkStarving);
    notifyAll();
}
}
```