

# Computação Concorrente (DCC/UFRJ)

## Aula 9: Monitores e mecanismos de sincronização em Java

Prof. Silvana Rossetto

29 de outubro de 2019

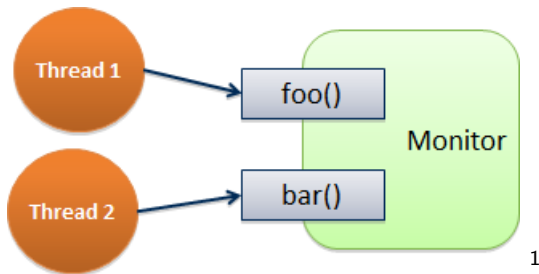
# Definição de monitor

Módulo de software que consiste de:

- 1 **um ou mais procedimentos**
- 2 **uma sequência de inicialização**
- 3 **variáveis de condição**
- 4 **estado interno**

- Um **monitor** é uma construção de Linguagem de Programação
- Proposto por Hoare, em 1974

# Visão geral de monitor



<sup>1</sup>Fonte: <http://lycog.com>

- 1 **Apenas uma operação interna pode estar ativa a cada instante de tempo:** se uma thread chama uma operação do monitor e ele está ocupado, a thread é bloqueada
  - permite implementar a **sincronização por exclusão mútua**
- 2 **Qualquer operação pode suspender a si mesma em uma variável de condição**
  - permite implementar a **sincronização por condição**

As variáveis locais (definidas dentro do monitor) são acessíveis apenas pelos procedimentos oferecidos pelo monitor

## **Todas as funções de sincronização são confinadas no monitor**

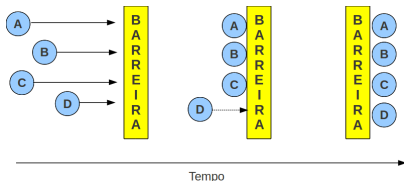
- ...mais fácil verificar a corretude da implementação
- ...uma vez que o monitor foi corretamente programado, o acesso protegido ao recurso compartilhado está garantido para todas as threads

# “Simulando” monitores em Java

- O acesso aos métodos de uma classe em Java pode ser com **exclusão mútua**: usar a palavra reservada **synchronized**
- Objetos Java dispõem das operações de bloqueio/desbloqueio: **wait**, **notify** e **notifyAll**

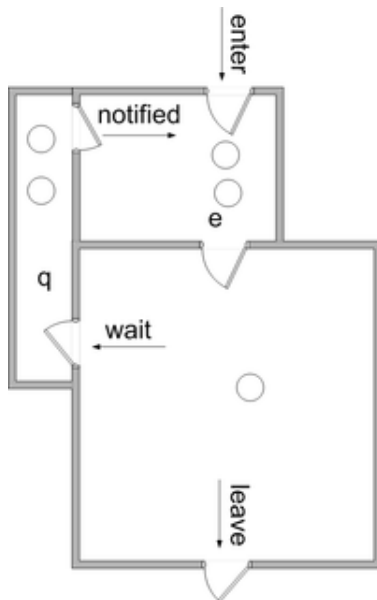
O uso combinado dessas funcionalidades com o **modelo de classes** da linguagem permite construir objetos Java com características de **monitores**

# Exemplo: barreira usando monitor



```
class Barreira {  
    private int numThreads;  
    private int cont;  
  
    //...construtor  
    Barreira(int n) {  
        this.cont = 0;  
        this.numThreads = n;  
    }  
  
    public synchronized void chegada() {  
        this.cont++;  
        if (this.cont < this.numThreads)  
            this.wait();  
        else {  
            this.cont = 0;  
            this.notifyAll();  
        }  
    }  
}
```

# Semântica dos monitores em Java



2



# Suporte parcial do conceito geral de monitores

- 1 Em Java **não existe suporte de compilação para checar e prevenir condições de corrida no programa** (se algum método de acesso a variáveis compartilhadas não é precedido de `synchronized`, pode ocorrer condições de corrida)
- 2 Em Java **não há variáveis de condição explícitas** (quando uma thread executa uma operação `wait`, ela fica bloqueada na fila de uma **variável de condição implícita**, associada com o objeto do bloco `synchronized`)

# Sincronização por exclusão mútua em Java

Uso da sentença **synchronized**:

- todas as execuções de sentenças *synchronized* que se referem ao mesmo **objeto compartilhado** excluem a execução simultânea de outras execuções
- funciona como um **lock**

```
void foo(){  
    synchronized(this) {  
        x++;  
        y = x;  
    }  
}
```

```
synchronized void bar(){  
    y++;  
    x += 3;  
}
```

# Sincronização condicional em Java

## Variáveis de condição em Java

- Dentro de um *bloco* (ou *método*) **synchronized**, uma thread pode suspender ela mesma chamando o método **wait()** *sem argumentos*
- Para retomar a execução de uma thread suspensa em um dado objeto, outra thread deve executar o método **notify()** *de dentro de um bloco **synchronized** que se refere ao mesmo objeto*

O método **notify** *não recebe argumentos*: uma thread qualquer suspensa na fila do objeto é desbloqueada

## Desbloqueio para todas as threads

O método **notifyAll()** desbloqueia todas as threads suspensas no **objeto de sincronização**

# Locks e variáveis de condição em Java

- Todo objeto Java pode agir como um **lock** e como uma **variável de condição**
- Os métodos **wait**, **notify** e **notifyAll** (da classe Object) são a API para acesso às **filas de condição implícitas** de cada objeto

# A operação Object.wait

- **wait()** ativamente libera o *lock* e pede ao Sistema Operacional para suspender a thread corrente permitindo que outras threads adquiram o *lock* e modifiquem o estado do objeto protegido
- Depois de ser desbloqueada, a thread **readquire o lock antes de retomar a sua execução** a partir do ponto em que foi bloqueada

## Produtor/Consumidor em Java

```
class BufferLimitado {  
    static final int N = 10;    //tamanho do buffer  
    private int[] buffer = new int[N];  
    private int count=0, in=0, out=0;  
  
    // Construtor  
    BufferLimitado() {  
        for (int i=0; i<N; i++)  
            ... //inicia o buffer  
    }  
    ...  
}
```

## Inserir

```
public synchronized void Inserir (int elemento) {  
    try {  
        while (count==N) {  
            wait(); } //bloqueio condição lógica  
        buffer[in] = elemento;  
        in=(in+1)%N; count++;  
        notifyAll();  
    } catch (InterruptedException e) { }  
}
```

**notifyAll() poderia ser substituído por notify()?**



## Retira

```
public synchronized int Remove () {  
    int elemento;  
    try {  
        while (count==0) {  
            wait(); } //bloqueio condição lógica  
        elemento = buffer[out%N];  
        out=(out+1)%N;  count--;  
        notifyAll();  
        return elemento;  
    } catch(InterruptedException e) {return -1;}  
}
```

**notifyAll() poderia ser substituído por notify()?**

- Como há apenas **uma variável de condição implícita associada a um objeto de locação**, pode ocorrer de **duas ou mais threads estarem esperando na mesma variável**, mas por condições lógicas distintas
- Por isso, o uso das operações `notify` e `notifyAll` deve ser feito com cuidado

# Impacto de desempenho com notifyAll

- Uma chamada `notifyAll` acorda (desbloqueia/sinaliza) **todas as threads** esperando naquele objeto, mesmo que estejam em subgrupos de espera distintos
- Esse tipo de *semi-espera-ocupada* pode causar impactos no desempenho da aplicação (ex., uma thread é acordado, ganha o controle da CPU e verifica que deve voltar a se bloquear, todo esse processamento poderia ser economizado)

## Possibilidade de erro com notify

Por outro lado, se **notify** for usado ao invés de **notifyAll**, a única thread acordada pode ser membro de um subgrupo errado (que não tem a condição lógica para prosseguir naquele momento)

# Quando usar notify() ao invés de notifyAll()

O uso de **notify** (ao invés de **notifyAll**) pode ser feito quando os seguintes requisitos são atendidos:

- 1 todas as threads esperam pela mesma condição lógica;
- 2 cada notificação deve permitir que apenas uma thread volte a executar.

# Leitores e escritores com monitores

```
class Leitor extends Thread {
    //objeto monitor para coordenar a lógica de execução
    LE monitor;
    Leitor (LE m) {
        this.monitor = m;
    }
    public void run () {
        try {
            for (;;) {
                this.monitor.EntraLeitor();
                // !!! parte do código que faz a leitura !!!
                this.monitor.SaiLeitor();
            }
        } catch (InterruptedException e) { return; }
    }
}
```

# Leitores e escritores com monitores

```
class Escriitor extends Thread {
    //objeto monitor para coordenar a lógica de execução
    LE monitor;
    Escriitor (LE m) {
        this.monitor = m;
    }
    public void run () {
        try {
            for (;;) {
                this.monitor.EntraEscriitor();
                // !!! parte do codigo que faz a escrita !!!
                this.monitor.SaiEscriitor();
            }
        } catch (InterruptedException e) { return; }
    }
}
```

# Leitores e escritores com monitores

```
class LeitorEscritor {
    static final int L = 4;
    static final int E = 3;
    public static void main (String[] args) {
        int i;
        LE monitor = new LE(); // monitor
        Leitor[] l = new Leitor[L];    // leitores
        Escritor[] e = new Escritor[E]; // escritores
        for (i=0; i<L; i++) {
            l[i] = new Leitor(monitor); l[i].start();
        }
        for (i=0; i<E; i++) {
            e[i] = new Escritor(monitor); e[i].start();
        }
    }
}
```

Implemente o código da classe **LE** com os métodos:

- ❶ **EntraLeitor**
- ❷ **SaiLeitor**
- ❸ **EntraEscritor**
- ❹ **SaiEscritor**



# Referências bibliográficas

- ① *Programming Language Pragmatics*, Scott, Morgan-Kaufmann, ed. 2, 2006
- ② *Java Concurrency in Practice*, Goetz et. al., Addison Wesley, 2006