Blocos Guardados e Variáveis de Condição

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

Blocos Guardados (*Guarded Blocks*)

Threads frequentemente têm que coordenar/sincronizar suas ações

- *Guarded blocks* são um mecanismo comum para coordenação
  - Um bloco só é executado caso uma determinada **condição** seja verdadeira

Guarded Blocks - Exemplo

- Considere o seguinte trecho de código

```
public void guarded() {
    //Esse loop gasta tempo do processador.
    //Não faça isso!
    while(!condicao){
    }
    System.out.println("Condicao satisfeita!");
}
```

- Solução ineficiente

`Object.wait()`

- Suspende a *thread* corrente e libera o recurso
- A invocação do `wait()` não retorna até outra *thread* ter enviado uma notificação
  - Sinaliza a ocorrência de algum evento especial
- Não necessariamente o evento que a *thread* está esperando

Blocos Guardado com eficiência

```
public synchronized void guarda() {
    //O loop acontece uma vez para cada evento especial,
    // o qual pode não ser o evento esperado
    while(!condicao) {
        try {
            wait();
        } catch (InterruptedException e) {}
    }
    System.out.println("Condição e eficiência satisfeita!");
}
```

Por que usar o `synchronized`?

- Monitor precisa ter sido adquirido para chamar `wait()`
  - A *thread* **libera o monitor** e fica bloqueada
- `synchronized` é a maneira de adquirir o monitor
- Uma chamada a `wait()` sem a aquisição prévia de um monitor implica em `IllegalMonitorStateException`

```
~/tmp $ cat C.java
public class C {

    public static void main(String args[]) throws InterruptedException {
        Object o = new Object();
        o.wait();
    }

}
~/tmp $ java C
Exception in thread "main" java.lang.IllegalMonitorStateException
        at java.lang.Object.wait(Native Method)
        at java.lang.Object.wait(Object.java:485)
        at C.main(C.java:5)
~/tmp $
```

Notificando as *threads*

- Notificação
  - Retira *thread* notificada do estado de espera
  - Thread que notifica precisa ter controle do monitor
- `notify()`
  - Notifica **uma** *thread* que esteja esperando em um monitor
  - Não se especifica qual *thread* vai ser notificada
- `notifyAll()`
  - Notifica **todas** as threads que estejam esperando em um monitor

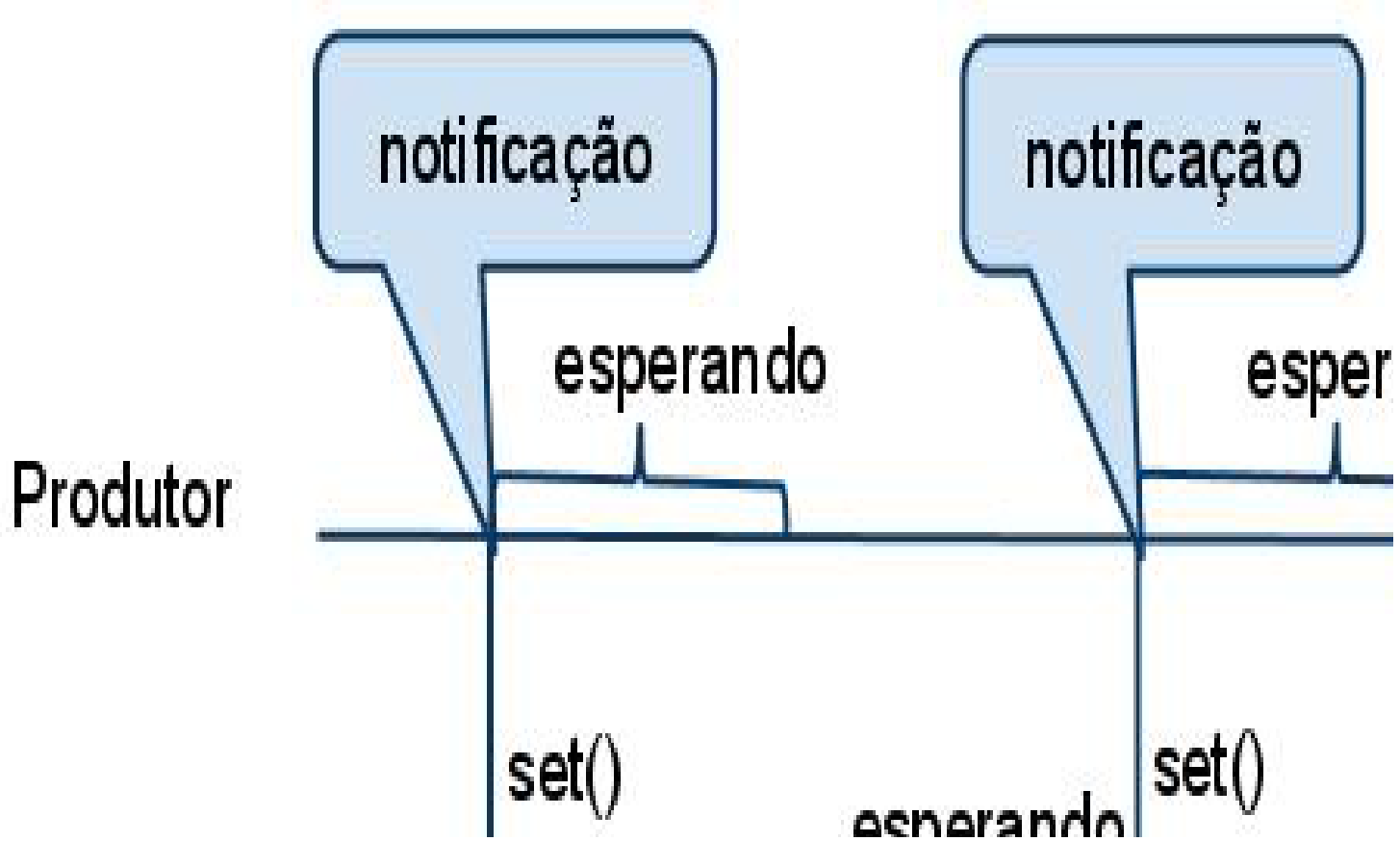
Exemplo de fluxo de notificação

1. *Thread* A adquire o monitor *M*
2. *Thread* A checa se a condição é verdadeira
   a. NÃO É!
   b. *Thread* A invoca o método `wait()`
      - O monitor *M* é liberado
      - Execução suspensa
3. Outra *thread* adquire o monitor *M*
4. Ela invoca o `notifyAll()`
5. Todas as threads que esperam em *M* são notificadas
   - Agora elas vão brigar para adquirir o monitor

Exemplo: Produtor/Consumidor

- Duas *threads* compartilham dados entre elas
- *Thread* Produtor -> Fornece os dados
- *Thread* Consumidor -> Pega os dados
- Comunicam-se através de um *buffer* finito e compartilhado
- Necessário Coordenação
  - Consumidor só pega o dado depois que o produtor tiver fornecido
  - Produtor só fornece dado depois que anterior for consumido

Exemplo de Fluxo



Mostrar código das classes `Drop`, `Produtor`, `Consumidor` e `ProdutorConsumidor`

Mostrar código das classes `Drop`, `Produtor`, `Consumidor` e `ProdutorConsumidor`

A implementação está correta?

Centro de Informática U·F·P·E

Variáveis de Condição

Variáveis de condição

- Outra forma de se implementar **escopos guardados**
- Para `locks` explícitos, cria-se variáveis de condição
- Objetos que implementam a interface `Condition`.
- Usa-se `Lock.newCondition()` para criar uma `Condition`
  - Possível criar **múltiplas `Conditions`** para um `Lock`
- Provê os métodos
  - `await()` para esperar até a condição ser verdadeira
  - `signal()` e `signalAll()` avisam aos `threads` que a condição ocorreu

Exemplos

```
import java.util.concurrent.locks.*;

public class CubbyHole{
    private int quantidade;
    private boolean disponibilidade = false;
    private Lock aLock = new ReentrantLock();
    private Condition variavelCondicional = aLock.newCondition();

    public int get(int who) {
        aLock.lock();
        try {
            while (disponibilidade == false) {
                try {
                    variavelCondicional.await();
                } catch (InterruptedException e) { }
            }
            disponibilidade = false;
            System.out.println("Consumidor " + who + " pegou: " +
                               quantidade);
            variavelCondicional.signalAll();
        }finally{
            aLock.unlock();
            return quantidade;
        }
    }
```

Continua

Exemplos

```
public void put(int who, int value) {
                aLock.lock();
try {
    while (disponibilidade == true) {
        try {
            variavelCondicional.await();
        } catch (InterruptedException e) { }
    }
    quantidade = value;
    disponibilidade = true;
    System.out.println("Produtor " + who + " colocou: " +
                       quantidade);
    variavelCondicional.signalAll();
    } finally {
        aLock.unlock();
    }
}
}
```

Exemplos

```
public class Consumidor extends Thread {
        private CubbyHole cubbyhole;
        private int numero;

        public Consumidor(CubbyHole c, int numero) {
            cubbyhole = c;
            this.numero = numero;
        }

        public void run() {
            int value = 0;
            for (int i = 0; i < 10; i++) {
                value = cubbyhole.get(numero);
            }
        }
}
```

Exemplos

```
public class Produtor extends Thread {
    private CubbyHole cubbyhole;
    private int numero;

    public Produtor(CubbyHole c, int numero) {
        cubbyhole = c;
        this.numero = numero;
    }

    public void run() {
        for (int i = 0; i < 10; i++) {
            cubbyhole.put(numero, i);
            try {
                sleep((int)(Math.random() * 100));
            } catch (InterruptedException e) { }
        }
    }
}
```

```
Produtor 1 colocou: 0
Consumidor 1 pegou: 0
Produtor 1 colocou: 1
Consumidor 1 pegou: 1
Produtor 1 colocou: 2
Consumidor 1 pegou: 2
Produtor 1 colocou: 3
Consumidor 1 pegou: 3
Produtor 1 colocou: 4
Consumidor 1 pegou: 4
Produtor 1 colocou: 5
Consumidor 1 pegou: 5
Produtor 1 colocou: 6
Consumidor 1 pegou: 6
Produtor 1 colocou: 7
Consumidor 1 pegou: 7
Produtor 1 colocou: 8
Consumidor 1 pegou: 8
Produtor 1 colocou: 9
Consumidor 1 pegou: 9
```

Implemente uma classe chamada `CountDownLatch`. Essa classe deve ter dois métodos, `await()` e `countDown()`. Seu construtor recebe um número inteiro positivo. Se uma thread chama o método `await()` de um objeto desse tipo, ela fica bloqueada esperando **até que o contador chegue a zero**. Chamadas a `countDown()` decrementam o contador interno do objeto, caso ele seja maior que zero. Se o contador chegar a zero, todas as threads que chamaram (ou chamarem) `await()` são desbloqueadas.

Implemente uma classe chamada `BlockingQueue` que representa uma fila bloqueante segura para múltiplas threads. Essa classe tem dois métodos, `take()` e `put()`, que incluem e removem um elemento da fila. O construtor da classe recebe sua capacidade máxima. Chamadas a `take()` removem um elemento da fila, se houver. Se a fila estiver vazia em uma chamada a `take()`, a thread que invocou o método fica bloqueada. Analogamente para uma chamada a `put()` quando o buffer está cheio. Sua implementação deve funcionar corretamente para múltiplos produtores e consumidores, deve garantir que produtores conseguem colocar itens em um buffer não-cheio, se assim o desejarem, que consumidores conseguem remover itens de um buffer não-vazio se assim o desejarem e que, a qualquer momento, não mais que um produtor e não mais que um consumidor estão usando a fila.