

Programação Concorrente

Memória Compartilhada – Lock e Semáforo

(Versão 2019/2)

Prof. Edson F. da Fonseca
MBA, MsC, PMP, Cobit



UNIVERSIDADE CATÓLICA DE BRASÍLIA
Cursos de Tecnologia da Informação

Programação Concorrente e Distribuída
Prof. Edson Francisco da Fonseca

CONTROLE DE CONCORRÊNCIA

Mecanismos de Controle de Concorrência do Java:

- **Monitor**: protege trechos de código/métodos que manipulam dados/recursos compartilhados, impedindo o acesso concorrente
- **Lock** (ou Mutex): cria uma fila de acesso a um dado/recurso compartilhado, impedindo o acesso concorrente
- **Semáforo**: limita o número de usuários que acessam simultaneamente um recurso, criando filas de acesso se este número for excedido



LOCK E SEMÁFORO



LOCK

- Também conhecido como Mutex
- É outra forma muito comum de fazer o mesmo
- Mecanismo de Controle de Concorrência
- Uso de recurso compartilhado
- Seção crítica
 - Garante a exclusão mútua
 - Permite somente UM acesso por vez



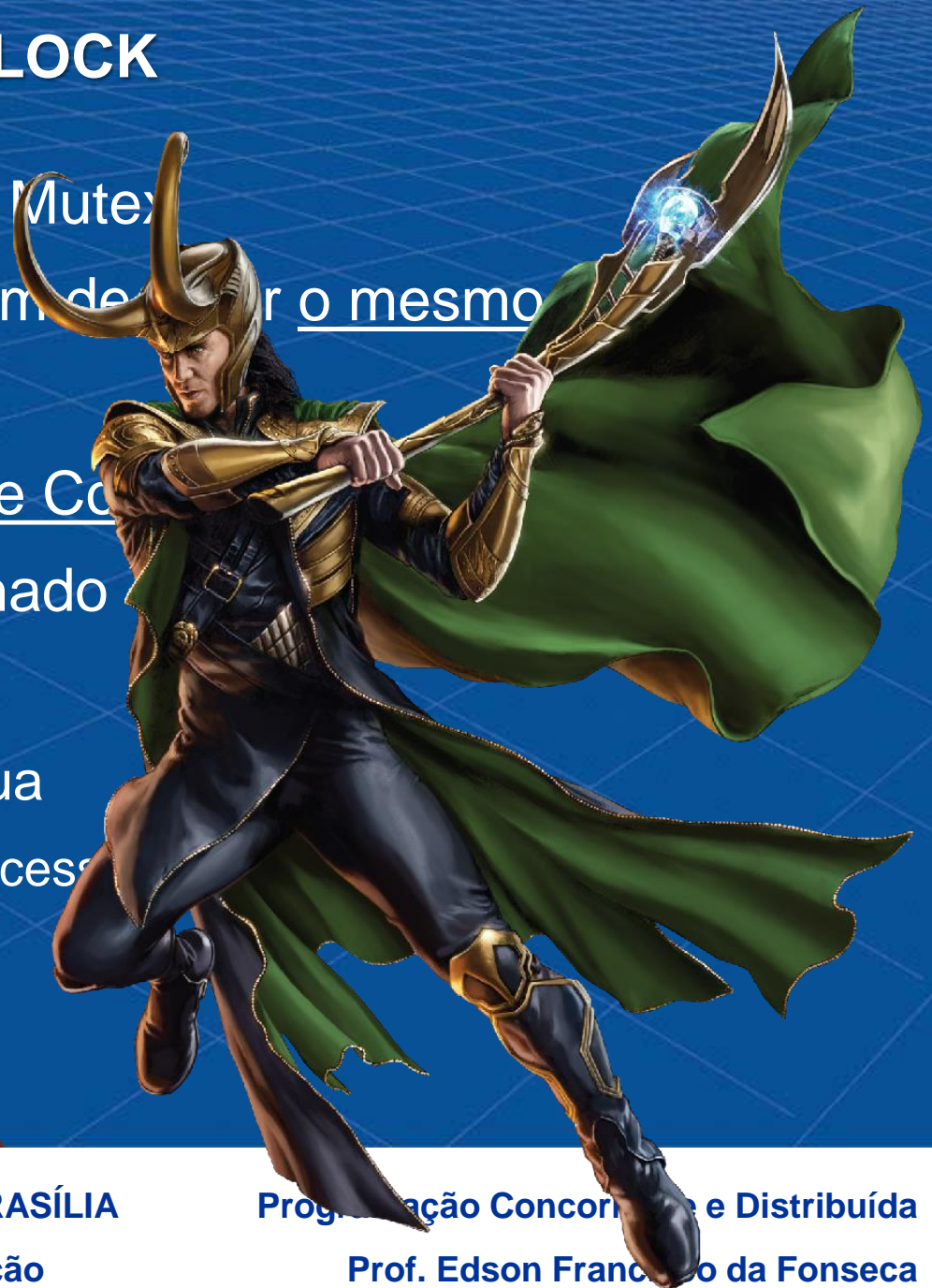
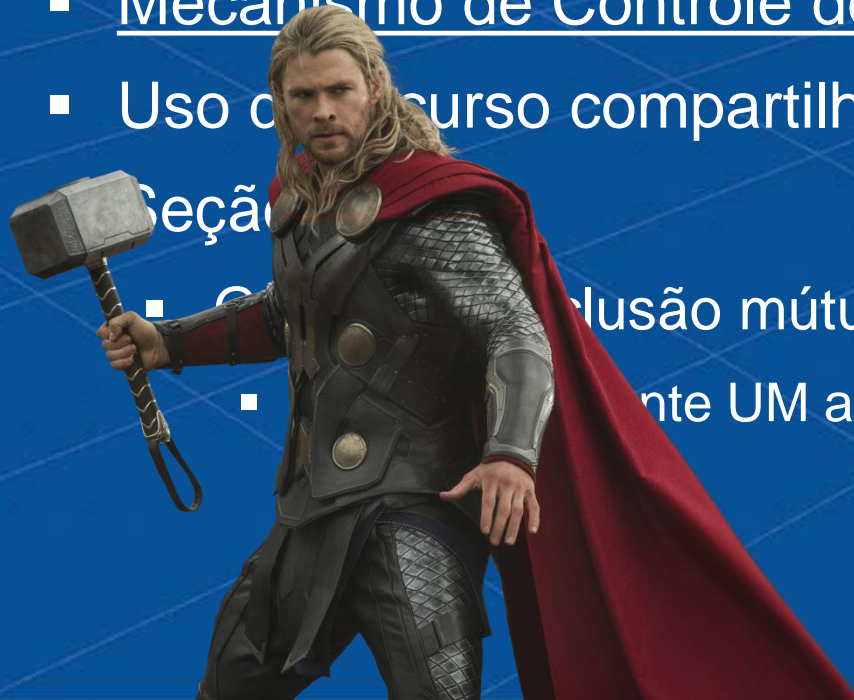
LOCK

- Também conhecido como Mutex
- É outra forma muito comum de se obter o mesmo

- Mecanismo de Controle de Co
- Uso do curso compartilhado

Seção

- Exclusão mútua
- Permite que apenas UM acesso



LOCK

Java – Interface Lock :

- lock(): primitiva de bloqueio
- unlock(): primitiva de desbloqueio
- Outros métodos da interface Lock:
 - tryLock(): retorna true se conseguir bloquear ou false
 - getHoldCount(): retorna o número de threads que tentaram obter o lock
 - isHeldByCurrentThread(): retorna true se a thread que fez a chamada obteve
 - isLocked(): retorna true se estiver bloqueado
 - getQueueLength(): retorna o número de threads que aguardam pela liberação do lock



LOCK

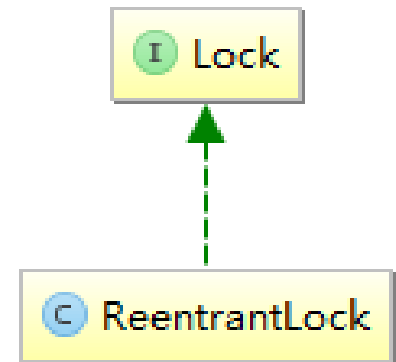
Java – Classe ReentrantLock :

- Implementa mecanismo de bloqueio exclusivo
- Por default a retirada de threads da fila não é ordenada, ou seja, não há garantias de quem irá adquirir o lock quando este for liberado
- O construtor ReentrantLock(true) cria um lock com fila FIFO, tornando o acesso significativamente mais lento




LOCK

```
public class Classe {  
    private Lock lock;  
  
    public Classe {  
        this.lock = new ReentrantLock();  
    }  
  
    public void metodo() {  
        this.lock.lock();  
        try {  
            // Seção crítica...  
        } finally {  
            this.lock.unlock();  
        }  
    }  
}
```




LOCK

Java – Interface `ReadWriteLock`

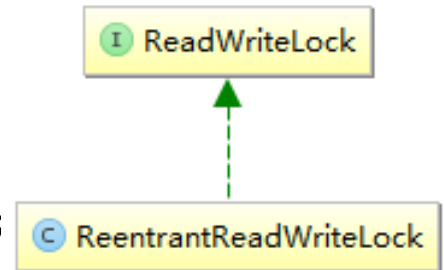
 `ReadWriteLock` :

- Possui dois Locks:
 - `readLock()`: acesso compartilhado com direito de leitura (acesso)
 - `writeLock()`: acesso exclusivo com direito de escrita (modificação)
- Implementada por `ReentrantReadWriteLock`
- Por default não garante a ordem de liberação nem preferência entre leitores e escritores
 - Ordenação FIFO é garantida passando `true` para o construtor

 `ReentrantReadWriteLock`

LOCK

```
public class Classe {  
    private ReadWriteLock lockRW;  
  
    public Classe {  
        this.lockRW = new ReentrantReadWriteLock();  
    }  
  
    public void metodo() {  
        this.lockRW.readLock().lock(); // OU .writeLock()  
        try {  
            // Seção crítica...  
        }  
        finally {  
            this.lockRW.readLock().unlock(); // OU .writeLock()  
        }  
    }  
}
```



LOCK

Java – Classe Condition:

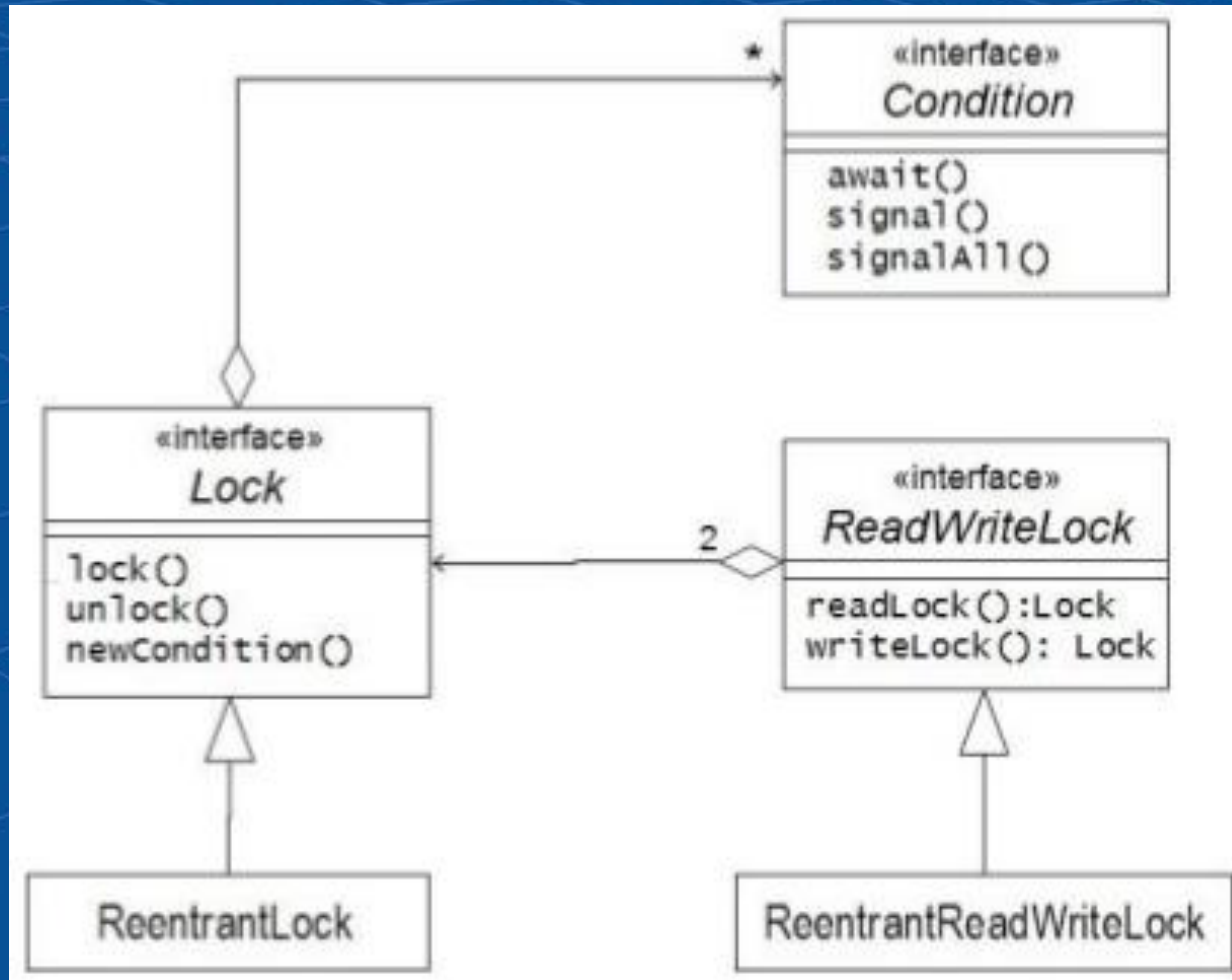
- Locks podem possuir condições de acesso
- São mais flexível que monitores
- Condições devem ser associadas a um Lock
 - Criada com o método `newCondition()` do Lock

Principais métodos:

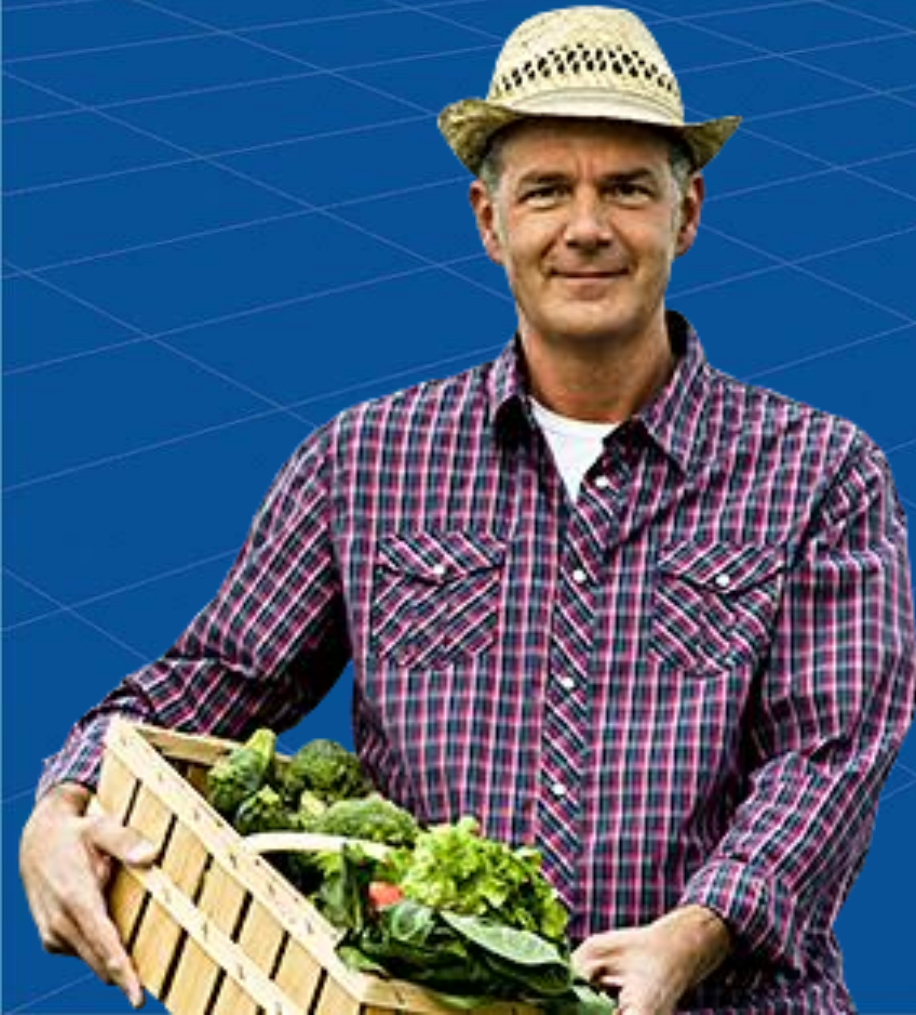
- `await()`: aguarda condição ser alterada
- `signal()`: sinaliza que houve alteração da condição
- `signalAll()`: retira todas as threads da fila



LOCK



PROBLEMA: PRODUTOR/CONSUMIDOR

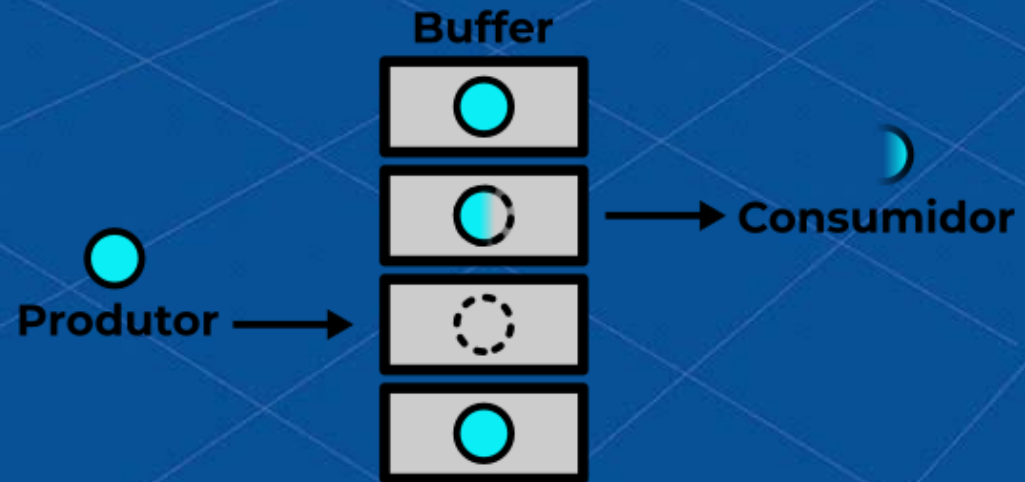


UNIVERSIDADE CATÓLICA DE BRASÍLIA
Cursos de Tecnologia da Informação

Programação Concorrente e Distribuída
Prof. Edson Francisco da Fonseca

PROBLEMA: PRODUTOR/CONSUMIDOR

- Um ou mais produtores criam produto e colocam no buffer
- Um ou mais consumidores consomem produto do buffer
- O produtor precisa esperar por espaço no buffer para produzir
- O consumidor precisa esperar por produtos no buffer para consumir
- O desafio é sincronizar o acesso ao recurso



PROBLEMA: PRODUTOR/CONSUMIDOR

```
public class Principal {

    public static void main(String args[]) {
        int qtdProdutor=5, qtdConsumidor=7;
        int maior=qtdProdutor>qtdConsumidor?qtdProdutor:qtdConsumidor;
        Produtor[] produtores = new Produtor[qtdProdutor];
        Consumidor[] consumidores = new Consumidor[qtdConsumidor];
        CircularBuffer buffer = new CircularBuffer();
        for (int i=0; i<maior; i++) {
            if (i < qtdProdutor) {
                produtores[i] = new Produtor("Produtor "+(i+1), buffer);
                produtores[i].start();
            }
            if (i < qtdConsumidor) {
                consumidores[i] = new Consumidor("Con. "+(i+1), buffer);
                consumidores[i].start();
            }
        }
    }
}
```

PROBLEMA: PRODUTOR/CONSUMIDOR

```
public class Produtor extends Thread {
    private CircularBuffer buffer;

    public Produtor(String nome, CircularBuffer buffer) {
        super(nome);
        this.buffer = buffer;
    }

    public void run() {
        int producao = (int) (Math.random() * 5);
        Sysout("==> " + this.getName() + " produzir " + producao);
        for (int i=0; i<producao; i++)
            this.produzir();
        Sysout("==> " + this.getName() + " fim da producao");
    }

    public void produzir() {
        int produto = (int) (Math.random() * 100);
        this.buffer.escrever(produto);
    }
}
```

PROBLEMA: PRODUTOR/CONSUMIDOR

```
public class Consumidor extends Thread {
    private CircularBuffer buffer;

    public Consumidor(String nome, CircularBuffer buffer) {
        super(nome);
        this.buffer = buffer;
    }

    public void run() {
        int consumo = (int) (Math.random() * 5);
        Sysout("==> " + this.getName() + " consumir " + consumo);
        for (int i=0; i<consumo; i++)
            this.consumir();
        Sysout("==> " + this.getName() + " fim do consumo");
    }

    public void consumir() {
        this.buffer.ler();
    }
}
```



```
public class CircularBuffer {
    private int[] buffer;
    private int bufferQtd;
    private int writePos;
    private int readPos;
    private Lock lock;
    private Condition podeWrite;
    private Condition podeRead;

    public CircularBuffer() {
        this.buffer = new int[3];
        this.lock = new ReentrantLock();
        this.podeWrite = this.lock.newCondition();
        this.podeRead = this.lock.newCondition();
    }

    public void escrever(int valor) {
        this.lock.lock();
        System.out.println("Escrever "+ valor +" - buffer["+this.bufferQtd+"]");
        try {
            while (this.bufferQtd == this.buffer.length) {
                System.out.println("--> Escrita esperando...");
                this.podeWrite.await();
            }
            this.buffer[this.writePos] = valor;
            this.writePos = (this.writePos+1)%this.buffer.length;
            this.bufferQtd++;
            this.podeRead.signal();
        } catch (InterruptedException e) { e.printStackTrace(); }
        finally {
            System.out.println("Escrito "+ valor +" - buffer["+this.bufferQtd+"]");
            this.lock.unlock();
        }
    }
}
```

```
public class CircularBuffer {  
    private int[] buffer;  
    private int bufferQtd;  
    private int writePos;  
    private int readPos;  
    private Lock lock;  
    private Condition podeWrite;  
    private Condition podeRead;  
  
    public CircularBuffer() {  
        this.buffer = new int[3];
```

```
        public int ler() {  
            int valor=0;  
            this.lock.lock();  
            System.out.println("Ler - buffer["+this.bufferQtd+"]");  
            try {  
                while (this.bufferQtd == 0) {  
                    System.out.println("--> Leitura esperando...");  
                    this.podeRead.await();  
                }  
                valor = this.buffer[this.readPos];  
                this.readPos = (this.readPos+1)%this.buffer.length;  
                this.bufferQtd--;  
                this.podeWrite.signal();  
            } catch (InterruptedException e) { e.printStackTrace(); }  
            finally {  
                System.out.println("Lido "+ valor +" - buffer["+this.bufferQtd+"]");  
                this.lock.unlock();  
            }  
            return valor;  
        }  
    }  
}
```

LOCK E SEMÁFORO



SEMÁFORO

- É uma variável especial que controla acessos a recursos compartilhados
- Criada por Edsger Dijkstra em 1965
- Utilizado no SO THEOS
- Seu valor indica quantos processos podem acessar ao mesmo tempo o recurso



[https://pt.wikipedia.org/wiki/Sem%C3%A1foro_\(computa%C3%A7%C3%A3o\)](https://pt.wikipedia.org/wiki/Sem%C3%A1foro_(computa%C3%A7%C3%A3o))



SEMÁFORO

Operações:

- Inicialização: Recebe o limite de processos
- P: Decrementa o valor do semáforo. Se o semáforo está com 0 o processo é posto para dormir
- V: Se o semáforo estiver com 0 e existir algum processo adormecido, um processo é acordado. Caso contrário, o valor do semáforo é incrementado.
- Dijkstra utilizou P e V devido às palavras holandesas *proberen* (testar), e *verhogen* (incrementar)
- As operações devem ser atômicas
- Se estabelece um limite de 1 (só um processo por vez) é um semáforo binário = lock



[https://pt.wikipedia.org/wiki/Sem%C3%A1foro_\(computa%C3%A7%C3%A3o\)](https://pt.wikipedia.org/wiki/Sem%C3%A1foro_(computa%C3%A7%C3%A3o))



SEMÁFORO

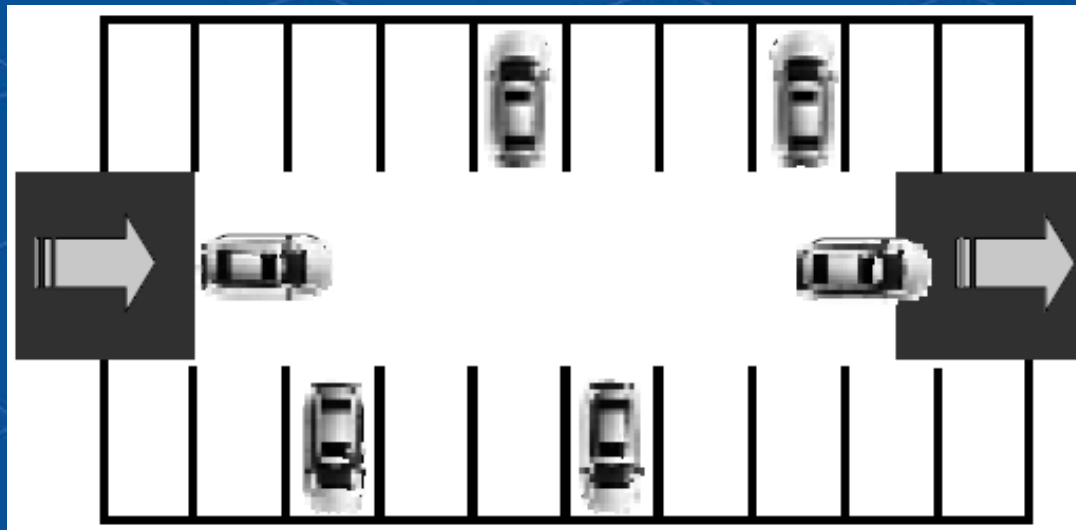
Java – Classe Semaphore:

- Semaphore(int acessos [, boolean ordem]): número de acessos simultâneos; [fila FIFO]
- acquire: solicita acesso, faz o papel do P
- release(): libera acesso, faz o papel do V



PROBLEMA: ESTACIONAMENTO

- Estacionamento possui um determinado número de vagas
- Carros entram, ficam diferentes tempos e saem
- Desafio é garantir que a quantidade de carros estacionados seja compatível com o número de vagas



PROBLEMA: ESTACIONAMENTO

```
public class Carro extends Thread {
    private static Semaphore estacionamento = new Semaphore(10, true);

    public Carro(String nome) {
        super(nome);
    }

    public void run() {
        try {
            estacionamento.acquire();
            System.out.println("--> " + this.getName() + " entrou");
            sleep((long) (Math.random() * 10000));
            System.out.println(this.getName() + " saiu");
            estacionamento.release();
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
    }

    public static void main(String args[]) {
        for (int i=1; i<=20; i++)
            new Carro("Carro "+i).start();
    }
}
```

LOCK E SEMÁFORO



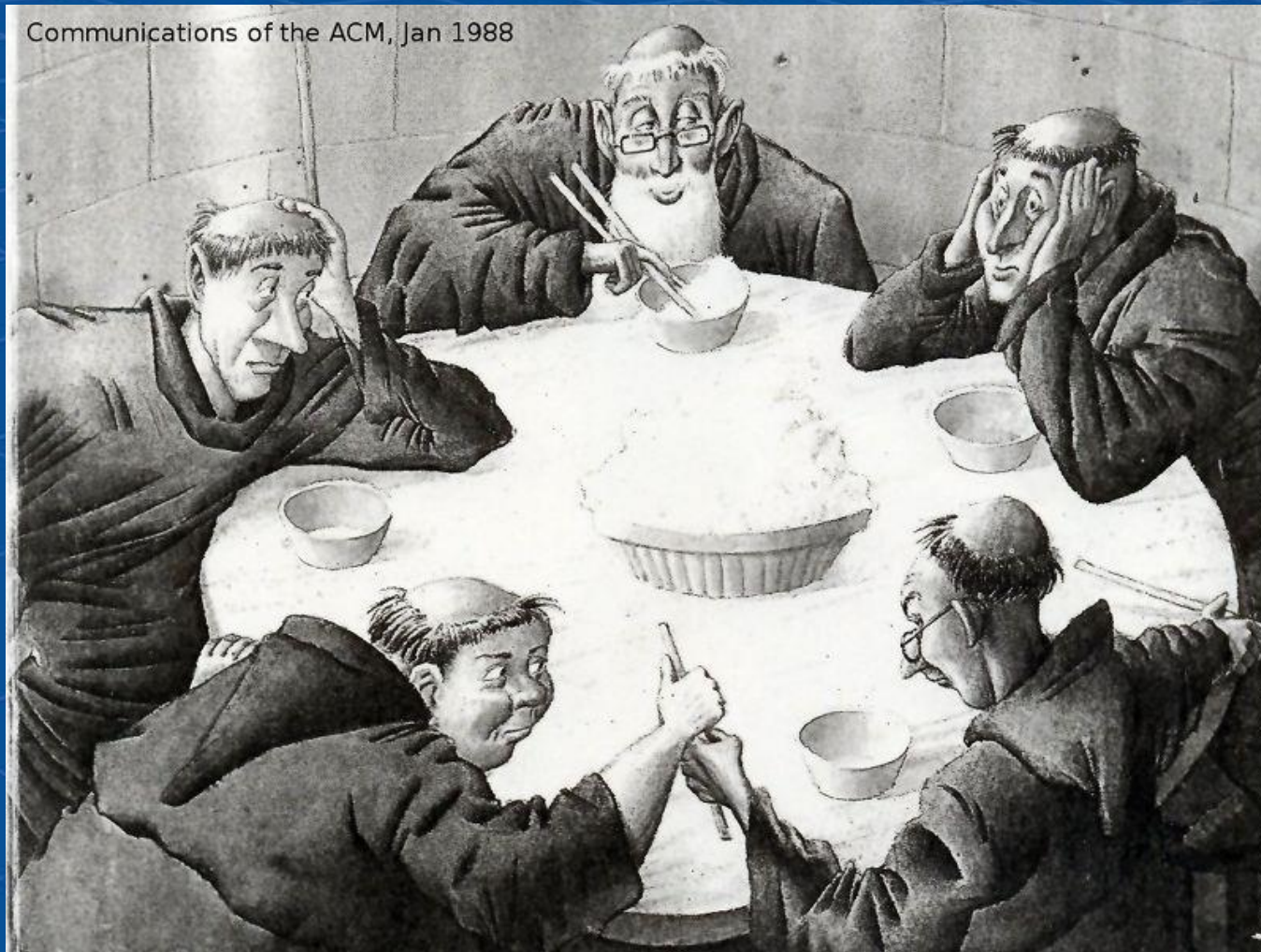
Perguntas?



UNIVERSIDADE CATÓLICA DE BRASÍLIA
Cursos de Tecnologia da Informação

Programação Concorrente e Distribuída
Prof. Edson Francisco da Fonseca

PROBLEMA: JANTAR DOS FILÓSOFOS (DINING PHILOSOPHERS)



UNIVERSIDADE CATÓLICA DE BRASÍLIA
Cursos de Tecnologia da Informação

Programação Concorrente e Distribuída
Prof. Edson Francisco da Fonseca

PROBLEMA: JANTAR DOS FILÓSOFO (DINING PHILOSOPHERS)

- Cinco filósofos estão sentados numa mesa redonda
- Na frente de cada filósofo, há uma tigela de arroz
- Entre cada filósofo há um hashi (pauzinho chinês)
- Para poder comer um pouco de arroz, um filósofo deve ter 2 hashis: o da esquerda e o da direita
- Um filósofo sempre pega o hashi da esquerda primeiro
- Tendo o esquerdo, o filósofo tenta pegar o hashi da direita
- Tendo ambos, ele come um pouco de arroz
- Ele larga os hashis e pausa um pouco
- Tenta tudo novamente



DEADLOCK

Starvation:

- Ocorre quando um, ou mais, processo não consegue obter recursos no sistema e não pode progredir

Deadlock:

- É uma forma especialmente drástica de starvation em que dois ou mais processos estão esperando por uma condição que nunca vai ocorrer
- Um deadlock é causado pela situação onde um conjunto de processos está bloqueado permanentemente



DEADLOCK



UNIVERSIDADE CATÓLICA DE BRASÍLIA
Cursos de Tecnologia da Informação

Programação Concorrente e Distribuída
Prof. Edson Francisco da Fonseca

PROBLEMA: JANTAR DOS FILÓSOFO (DINING PHILOSOPHERS)

Communications of the ACM, Jan 1988

Mesa

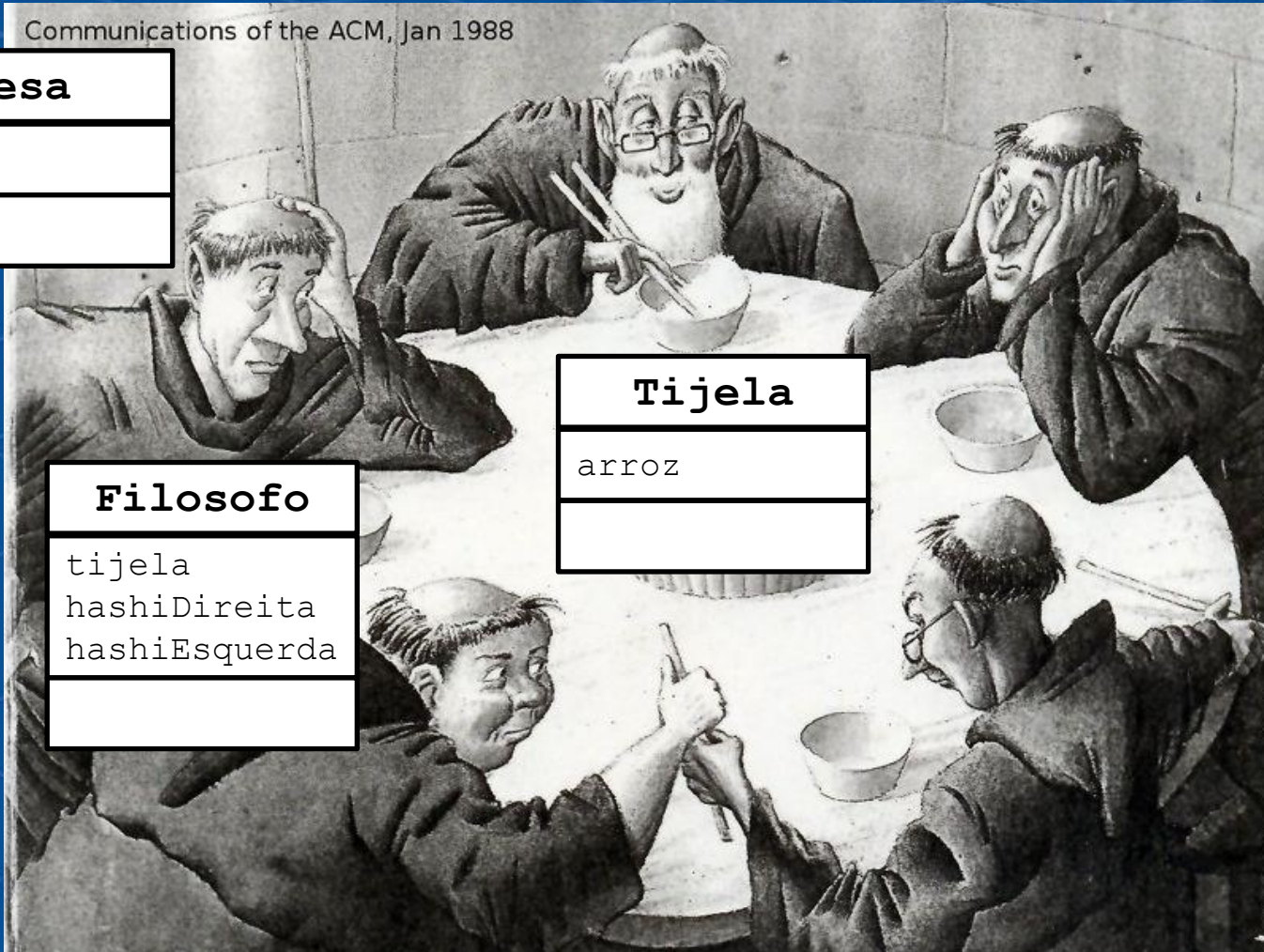
`main()`

Filosofo

`tijela`
`hashiDireita`
`hashiEsquerda`

Tijela

`arroz`



UNIVERSIDADE CATÓLICA DE BRASÍLIA
Cursos de Tecnologia da Informação

Programação Concorrente e Distribuída
Prof. Edson Francisco da Fonseca

JANTAR DOS FILÓSOFOS – LOCK

```
public class Mesa {
    private Filosofo[] filosofos;

    public Mesa() {
        Lock[] hashis = new Lock[5];
        for (int i=0; i<5; i++)
            hashis[i] = new ReentrantLock();
        this.filosofos = new Filosofo[5];
        for (int i=0; i<5; i++)
            this.filosofos[i] = new Filosofo("Filosofo "+(i+1), hashis[i], hashis[(i+1)%5]);
    }

    public static void main(String[] args) {
        Mesa mesa = new Mesa();
        System.out.println("*** JANTAR DOS FILÓSOFOS - LOCK ***");
        mesa.iniciar();
        System.out.println("*** FIM DO JANTAR - TODOS ACABARAM ***");
    }

    public void iniciar() {
        for (int i=0; i<5; i++)
            this.filosofos[i].start();
        try {
            for (int i=0; i<5; i++)
                this.filosofos[i].join();
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
    }
}
```


JANTAR DOS FILÓSOFOFOS – LOCK

```
public class Filosofo extends Thread {
    private Tijela tijela;
    private Lock hashiDireita;
    private Lock hashiEsquerda;

    public Filosofo(String nome, Lock hashiDir, Lock hashiEsq) {
        super(nome);
        this.tijela = new Tijela();
        this.hashiDireita = hashiDir;
        this.hashiEsquerda = hashiEsq;
    }

    public void run() {
        while (!this.tijela.vazia()) {
            while (!this.hashiEsquerda.tryLock());
            this.esperar(1000);
            while (!this.hashiDireita.tryLock());
            this.tijela.comer();
            Sysout(getName() + " comeu! Sobrou " + tijela.getArroz());
            this.hashiEsquerda.unlock();
            this.hashiDireita.unlock();
            this.esperar(3000);
        }
        Sysout("--> " + this.getName() + " ACABOU DE COMER!!!");
    }

    private void esperar(long tempo) {
        try { Thread.sleep(tempo); } catch (InterruptedException e) { e.printStackTrace(); }
    }
}
```

JANTAR DOS FILÓSOFOS – LOCK

```
public class Tijela {
    private int arroz;

    public Tijela() {
        this.arroz = 100;
    }

    public boolean comer() {
        if (this.arroz == 0)
            return false;
        try { Thread.sleep(3000); }
        catch (InterruptedException e) { e.printStackTrace(); }
        this.arroz -= 20;
        return true;
    }

    public boolean vazia() {
        return (this.arroz <= 0);
    }

    public int getArroz() {
        return arroz;
    }
}
```



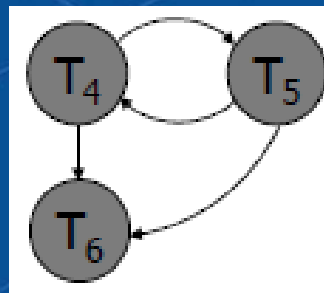
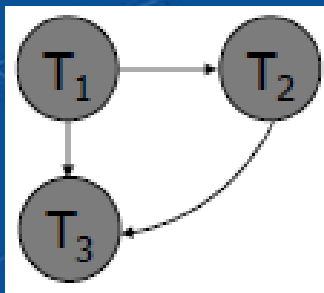
JANTAR DOS FILÓSOFOS – LOCK

Testar...



DEADLOCK

- No Java threads podem ficar em espera indefinidamente
- Algumas linguagens/sistemas detectam o deadlock e reportam exceções
- Detecção de Deadlock:
 - Verifica o estado do sistema periodicamente
 - Precisa saber quais bloqueios estão ativos
 - Deadlocks são detectados por ciclos



LIVELOCK

- *Similar to a deadlock, except that the states of the processes involved in the livelock constantly change with regard to one another, none progressing.*
 - Similar ao deadlock, exceto que os estados dos processos envolvidos no livelock mudam constantemente em relação um ao outro, nenhum progredindo.
- *The term was defined formally at some time during the 1970s.*
 - O termo foi definido formalmente em algum momento durante a década de 1970.
- *Livelock is a special case of resource starvation; the general definition only states that a specific process is not progressing.*
 - Livelock é um caso especial **starvation**; a definição geral apenas afirma que um processo específico não está progredindo.



<https://en.wikipedia.org/wiki/Deadlock#Livelock>



JANTAR DOS FILÓSOFOS – SEM **LIVELock**

```
public class Filosofo extends Thread {  
    :  
    :  
    public void run() {  
        while (!this.tijela.vazia()) {  
            while (!this.hashiEsquerda.tryLock());  
            this.esperar(1000);  
            while (!this.hashiDireita.tryLock());  
            this.tijela.comer();  
            Sysout(getName() + " comeu! Sobrou " + tijela.getArroz());  
            this.hashiEsquerda.unlock();  
            this.hashiDireita.unlock();  
            this.esperar(3000);  
        }  
        Sysout("--> " + this.getName() + " ACABOU DE COMER!!!");  
    }  
    :  
    :  
}
```

Livelock



JANTAR DOS FILÓSOFOFOS – SEM **LIVELOCK**

```
public class Filosofo extends Thread {
    :
    :
    public void run() {
        while (!this.tijela.vazia()) {
            while (!this.hashiEsquerda.tryLock())
                this.esperar(1000);
            if (this.hashiDireita.tryLock()) {
                this.tijela.comer();
                Sysout(getName() + " comeu! Sobrou " + tijela.getArroz());
                this.hashiEsquerda.unlock();
                this.hashiDireita.unlock();
                this.esperar(3000);
            }
            else
                this.hashiEsquerda.unlock();
        }
        Sysout("--> " + this.getName() + " ACABOU DE COMER!!!");
    }
}
```



DEADLOCK



Perguntas?



UNIVERSIDADE CATÓLICA DE BRASÍLIA
Cursos de Tecnologia da Informação

Programação Concorrente e Distribuída
Prof. Edson Francisco da Fonseca

Obrigado!!!



Prof. Edson F. da Fonseca
edsonf@ucb.br

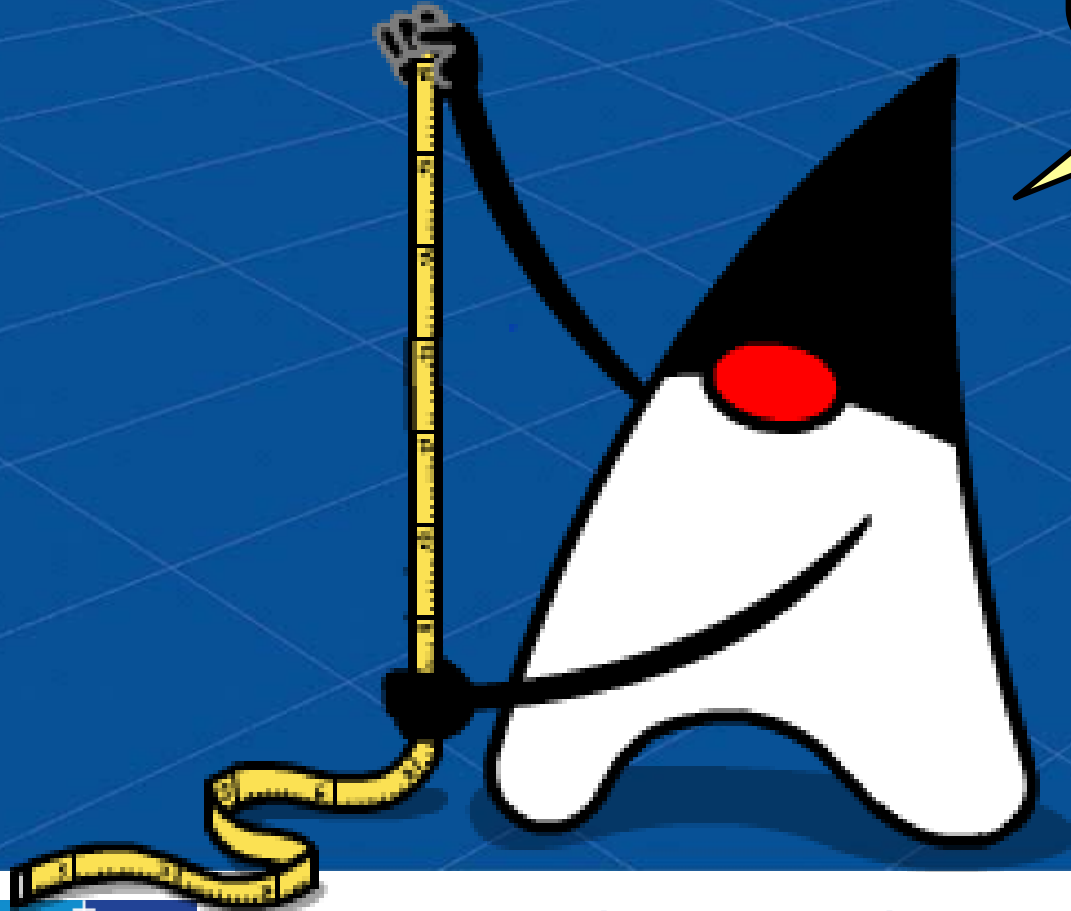


UNIVERSIDADE CATÓLICA DE BRASÍLIA
Cursos de Tecnologia da Informação

Programação Concorrente e Distribuída
Prof. Edson Francisco da Fonseca

LOCK E SEMÁFORO

Atividade
Supervisionada!!!



LOCK E SEMÁFORO

Lista de
Exercícios 3

