Roteiro – Primitivas de Sincronização

- Introdução
- Semáforos
- Monitores
- Outros Exemplos
- Perigo de *Deadlocks*
- Exercícios

Introdução

- Proposta até aqui vistas sobre exclusão mútua não levaram em consideração o desperdício de processamento
- Se um processo não pode entrar numa região crítica, ele, **repetidamente**, checa uma condição de entrada até esta ser verdadeira
- Este modo é conhecido como *busy wait*, ou espera ocupada
- Se ao invés do processo checar **indefinidamente** uma condição antes de entrar em uma região crítica, ele apenas informar que tem interesse no acesso à esses dados, economizaria ciclos da CPU

Semáforos

- Dijkstra propôs o conceito de **semáforos** para resolver o problema da espera ocupada
- Um semáforo tem dois campos, um VALOR e uma FILA de processos bloqueados, além de duas operações associadas **P()** e **V()**
- O valor de um semáforo pode ser *true* ou *false*
- A fila de processos bloqueados está inicialmente vazia e um processo deve adiciona-lo na fila quando faz uma chamada a P()
- Quando um processo chama P() e VALOR é *true*, então o valor do semáforo torna-se *false*

Semáforos

```
public class BinarySemaphore {
    boolean value;
    BinarySemaphore(boolean initValue) {
        value = initValue;
    }
    public synchronized void P() {
        while (value == false)
            try{
            this.wait(); // referencia a esse objeto
          }catch(Exception e){ }// na fila de processos bloqueados
        value = false;
    }
    public synchronized void V() {
        value = true;
        notify();}
}
```

Semáforos

- Agora, implementar exclusão mútua é trivial

```
BinarySemaphore mutex = new BinarySemaphore(true);
mutex.P();
seçãoCrítica();
mutex.V();
```

- Uma variante de semáforo permite alocar inteiros como valor
- Esses semáforos são conhecidos como **semáforos contadores**
- **Java não tem semáforo** em sua construção de linguagem, logo é preciso implementar

Semáforos

```
public class CountingSemaphore {
    int value;
    public CountingSemaphore(int initValue) {
        value = initValue;
    }
    public synchronized void P() {
        value--;
        if (value < 0) try{
            this.wait();
      }catch(Exception e){}
    }
    public synchronized void V() {
        value++;
        if (value <= 0) notify();
    }
}
```

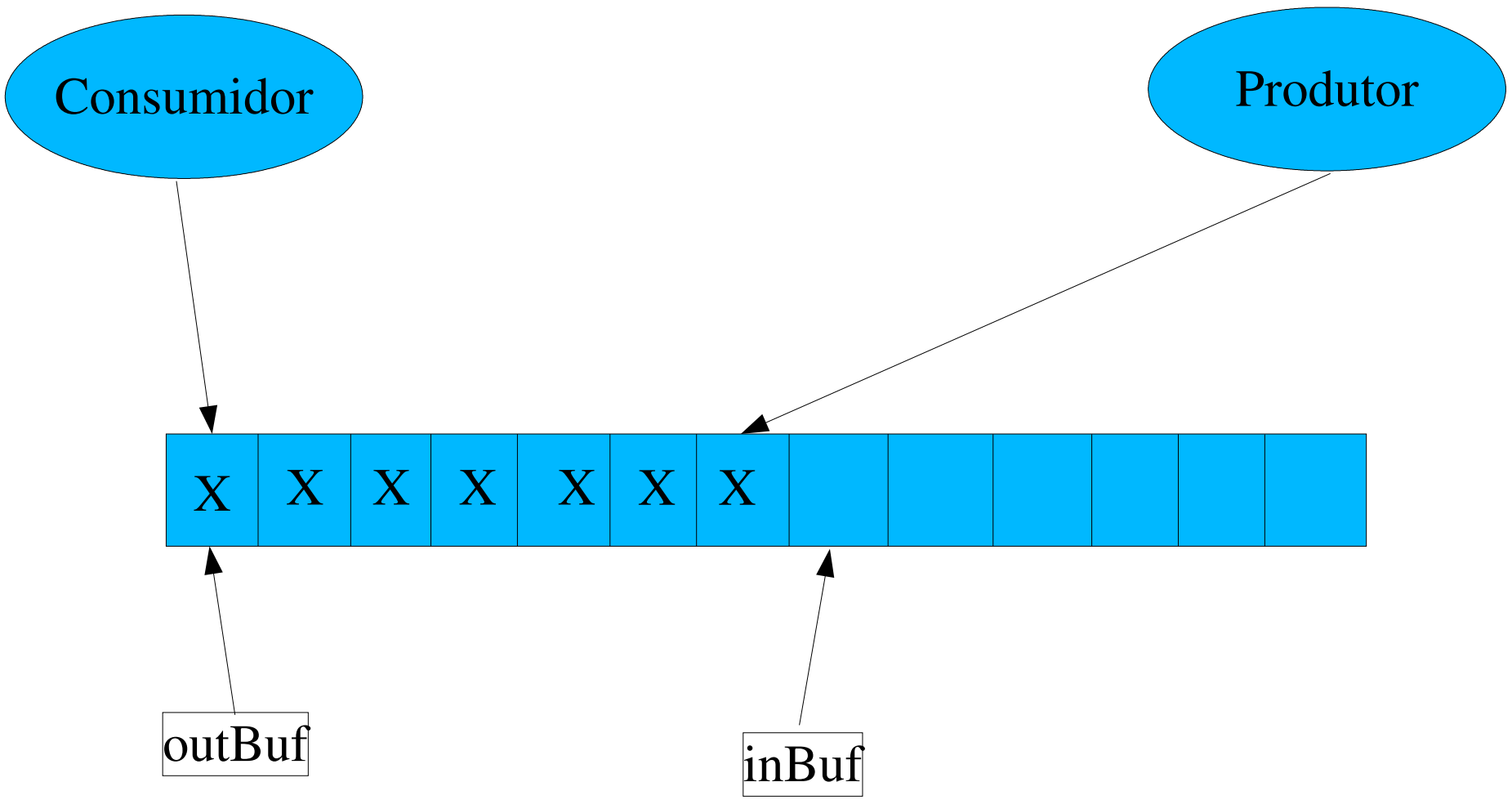
Semáforos – Produtor e Consumidor

- No problema clássico **Produtor-Consumidor**, há um *buffer* compartilhado entre dois processos
- O **Produtor** produz itens que serão colocados no *buffer* e o consumidor retirar tais itens
- Como o *buffer* é compartilhado, cada processo deve acessa-lo concorrentemente
- Será um usado um *array* de *double* de tamanho T, com dois ponteiros *inBuf* e *outBuf*, que indicam itens depositados e retirados respectivamente

Semáforos – Produtor e Consumidor

- Nesse problema, tem-se dois limites de sincronização que devem ser satisfeitos:
  - O **Consumidor** não deve retirar item de um *buffer* vazio
  - O **Produtor** não deve colocar item em um *buffer* cheio. O *buffer* se torna cheio se o **Produtor** colocar itens mais rápido do que o **Consumidor** consegue retirar
- Essa condicional é chamada de **sincronização condicional**, e requer que um processo espere alguma condição para continuar em operação

Semáforos – Produtor e Consumidor



Semáforos – Produtor e Consumidor

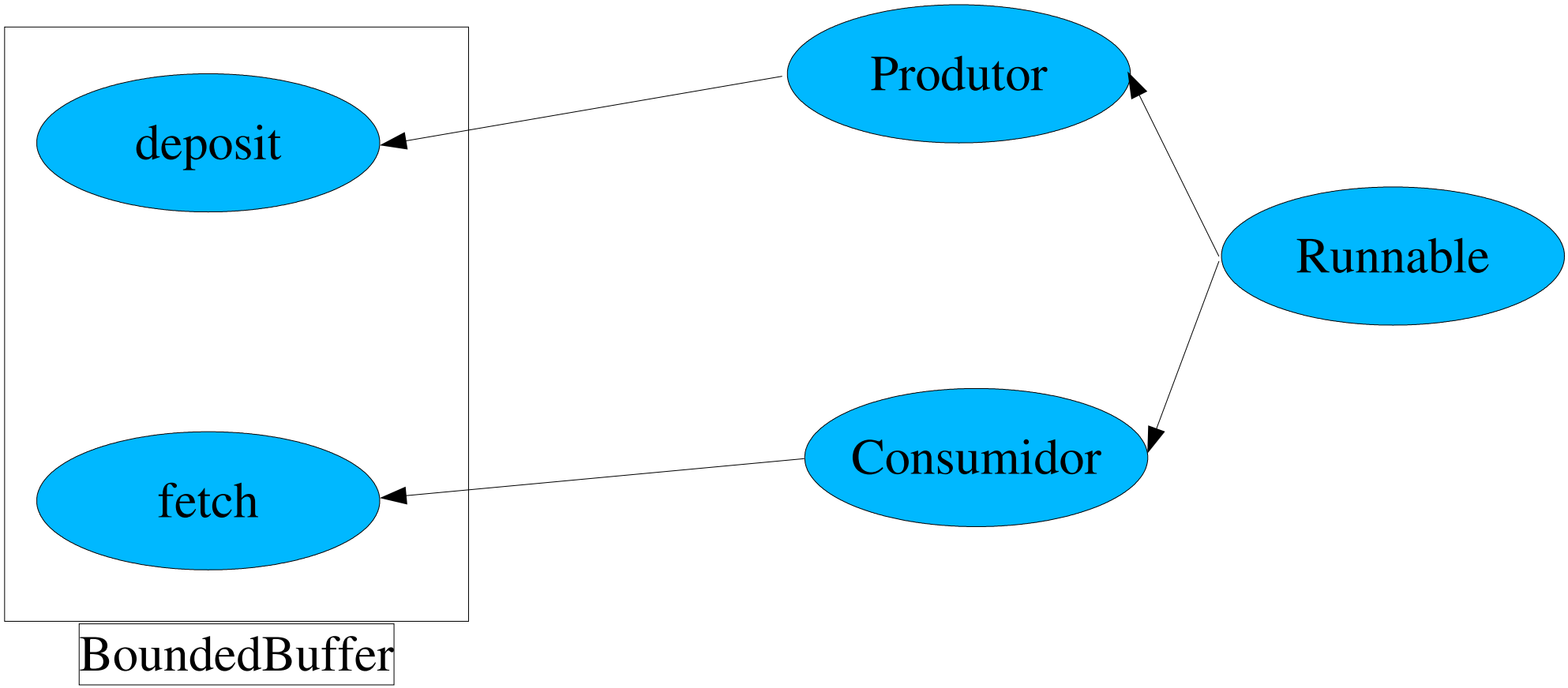
```
class BoundedBuffer {
    final int size = 10;
    double[] buffer = new double[size];
    int inBuf = 0, outBuf = 0;
    BinarySemaphore mutex = new BinarySemaphore(true);
    CountingSemaphore isEmpty = new CountingSemaphore(0);
    CountingSemaphore isFull = new CountingSemaphore(size);
```

Semáforos – Produtor e Consumidor

```
public void deposit(double value) {
    isFull.P(); // espera se buffer cheio
    mutex.P(); // garante a exclusão mútua
    buffer[inBuf] = value; // atualiza o buffer
    inBuf = (inBuf + 1) % size;
    mutex.V();
    isEmpty.V();  // notifica algum processo esperando
}
  public double fetch() {
    double value;
    isEmpty.P(); // espera se buffer vazio
    mutex.P();  // garante a exclusão mútua
    value = buffer[outBuf]; //lê do buffer
    outBuf = (outBuf + 1) % size;
    mutex.V();
    isFull.V(); // notifica algum produtor esperando
    return value;}}
```

Semáforos – Produtor e Consumidor - Exercício

- Construa as classes **Produtor** e **Consumidor** para testar a classe *BoundedBuffer*

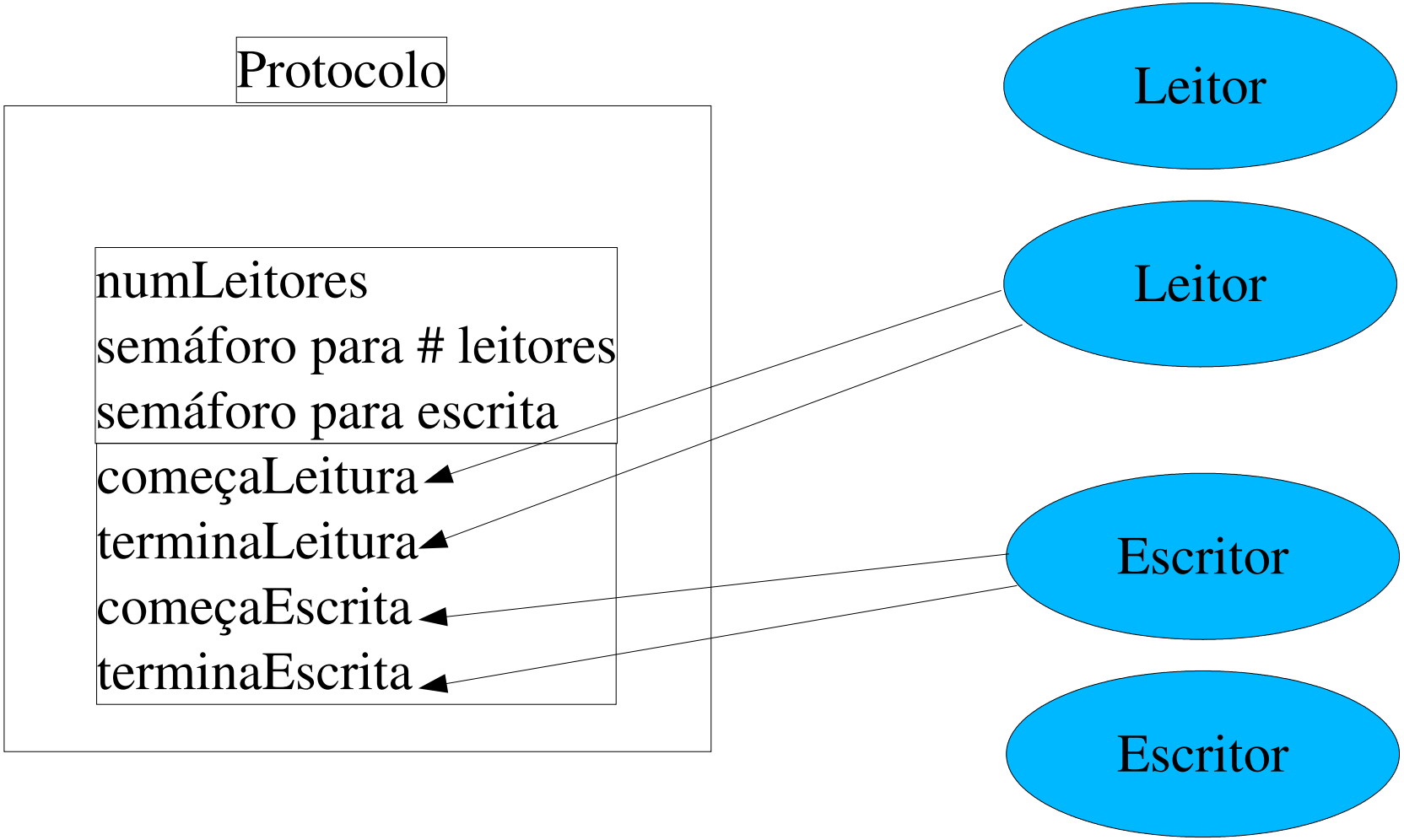


Semáforos – Leitores e Escritores

- O problema clássico dos **Leitores** e **Escritores** requer um projeto de protocolo para **coordenar o acesso** à áreas compartilhadas
- Condições:
  - Sem conflitos entre **Leitores** e **Escritores**. O protocolo deve assegurar que um **Leitor** e um **Escritor** não acessem juntos a base de dados
  - Sem conflito de **Escritor** e **Escritor**. O protocolo deve assegurar que dois **Escritores** não acessem juntos a base de dados

Semáforos – Leitores e Escritores - Exercício

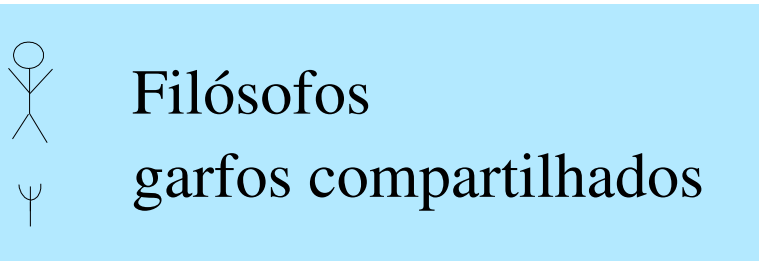
- Faça o protocolo para o acesso e as classes de **Leitores** e **Escritores**

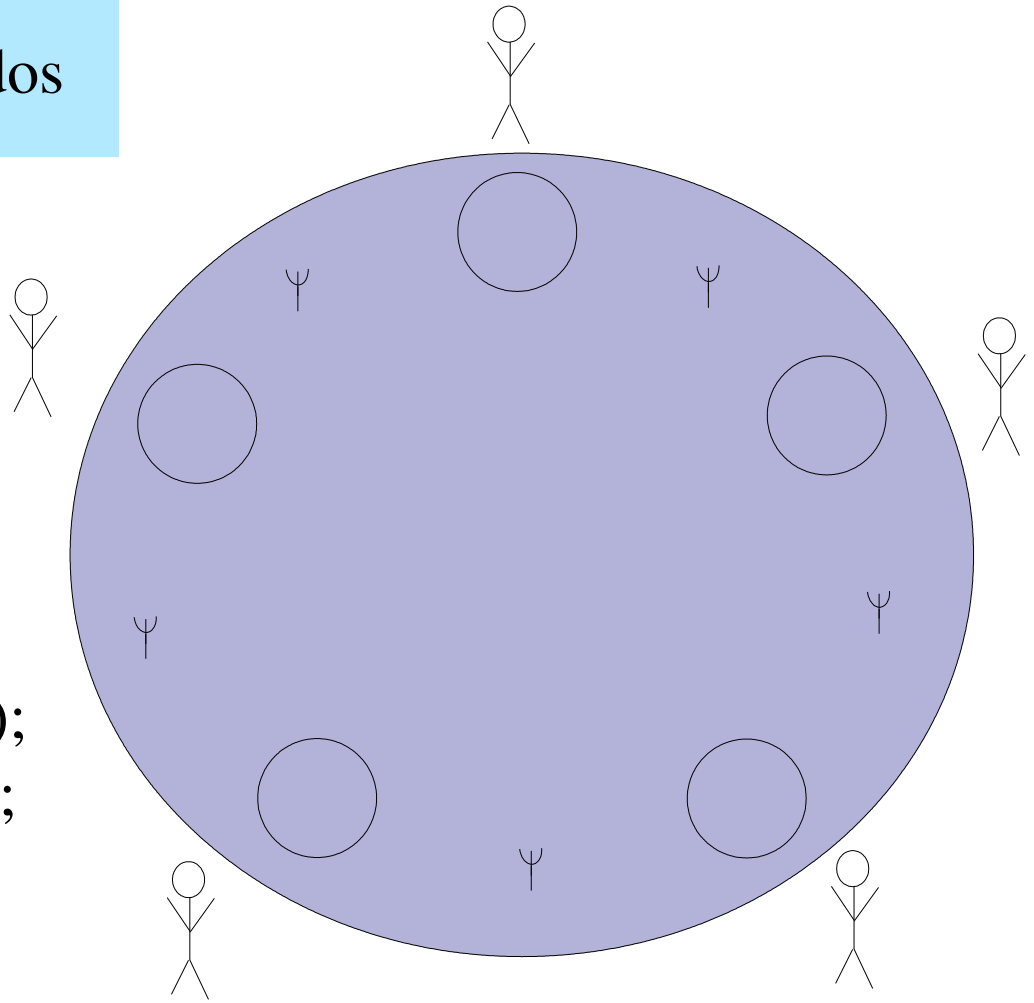


Semáforos – Filósofos Glutões

- Problema associado com concorrência e simetria
- Consiste de múltiplos filósofos que consomem tempo pensando e comendo spaghetti
- Um filósofo requer recursos compartilhados, como garfos para comer o spaghetti
- Logo, é necessário coordenar o acesso dos filósofos aos talheres
- A tarefa de comer corresponde à uma operação que acessa arquivos compartilhados

Semáforos – Filósofos Glutões





```
interface Resource {
    public void acquire(int i);
    public void release(int i);
}
```

Semáforos – Filósofos Glutões

```
class Philosopher implements Runnable {
    int id = 0;
    Resource r = null;
    public Philosopher(int initId, Resource initr) {
        id = initId; r = initr; new Thread(this).start();
    }
    public void run() {
        while (true) {
            try {
                System.out.println("Phil " + id + " thinking"); Thread.sleep(30);
                System.out.println("Phil " + id + " hungry"); r.acquire(id);
                System.out.println("Phil " + id + " eating"); Thread.sleep(40);
                r.release(id);
            } catch (InterruptedException e) { return; }
        }
    }
}
```

Semáforos – Filósofos Glutões

```
class DiningPhilosopher implements Resource {
   int n = 0;
   BinarySemaphore[] fork = null;
   public DiningPhilosopher(int initN) {
      n = initN;
      fork = new BinarySemaphore[n];
      for (int i = 0; i < n; i++) {  fork[i] = new BinarySemaphore(true);} }
   public void acquire(int i) {
      fork[i].P();
      fork[(i + 1) % n].P();     }
   public void release(int i) {
      fork[i].V();
      fork[(i + 1) % n].V(); }
   public static void main(String[] args) {
      DiningPhilosopher dp = new DiningPhilosopher(5);
      for (int i = 0; i < 5; i++)
         new Philosopher(i, dp);}}
```

Semáforos – Filósofos Glutões – Outros problemas

- Esse problema ilustra a diferença entre deadlock e starvation
- Filósofo pegar um garfo e seu vizinho o outro
- Todos pegam um garfo
- Como resolver????

Monitores

- Monitor é um alto nível de construção para sincronização em programação concorrente
- Um monitor pode ser visto como uma classe que pode ser usada em programas concorrentes
- Como uma classe, um monitor tem métodos e atributos para manipular dados
- Várias threads podem acessar um monitor ao mesmo tempo, logo ele possui suporte à métodos que garantem a exclusão mútua
- Todo monitor tem uma fila associada com threads aguardando para entrar

Monitores

- Monitores possuem variáveis de condição
- Um variável de condição possui duas operações: ***wait*** e ***notify***
- Para uma variável de condição ***x***, qualquer thread, ***t1***, que faça chamada para ***x.wait()*** é bloqueada e colocada em uma fila associada com ***x***
- Quando outra thread, ***t2***, fizer uma chamada ***x.notify()***, se a fila associada com ***x*** não estiver vazia, uma thread é removida da fila e inserida na fila de threads que estão eleitas para executar

Monitores

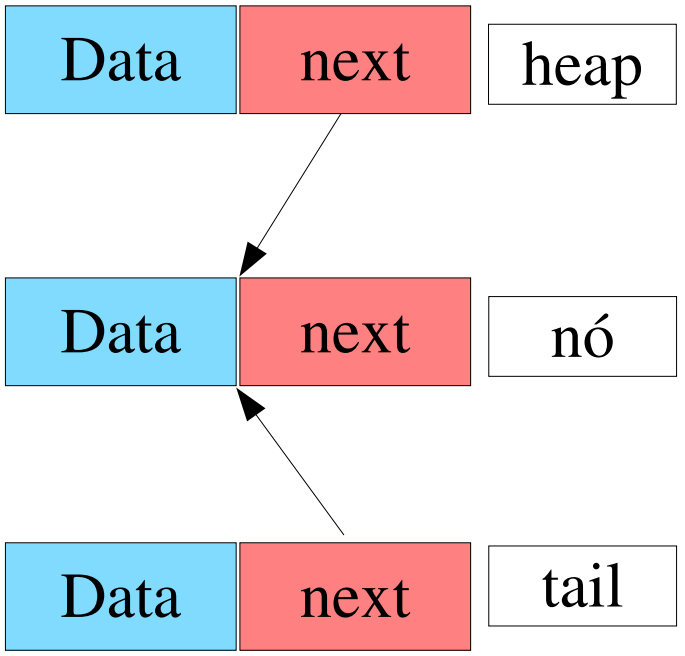
```
class DiningMonitor implements Resource {
    int n = 0;
    int state[] = null;
    static final int thinking = 0, hungry = 1, eating = 2;
    public DiningMonitor(int initN) {
        n = initN;
        state = new int[n];
        for (int i = 0; i < n; i++) state[i] = thinking;
    }
    int left(int i) {
        return (n + i - 1) % n;
    }
    int right(int i) {
        return (i + 1) % n;
    }
```

Monitores

```
 public synchronized void acquire(int i) {
        state[i] = hungry;
        test(i);
        while (state[i] != eating)try{wait();
                }catch(Exception e){} }
public synchronized void release(int i) {
        state[i] = thinking;
        test(left(i));
        test(right(i));
    }
    void test(int i) {
        if ((state[left(i)] != eating) &&   (state[i] == hungry) &&
        (state[right(i)] != eating)) {
            state[i] = eating;
            notifyAll();
        }
    }
```

Monitores – Outros exemplos

```
public class ListQueue {
    class Node {  public String data;    public Node next; }
    Node head = null, tail = null;
    public synchronized void enqueue(String data) {
        Node temp = new Node();
        temp.data = data;  temp.next = null;
        if (tail == null) { tail = temp; head = tail;
        } else {
            tail.next = temp;
            tail = temp; }
        notify();}
    public synchronized String dequeue() {
        while (head == null)
          try{wait();}catch(Exception e){} }
        String returnval = head.data;
        head = head.next;
        return returnval; }}
```



Exercícios

- Outro problema clássico é o dos Barbeiros. Há uma thread chamada Barber. O Barber corta o cabelo de algum cliente que estiver esperando. Se não há clientes, o Barber dorme. Pode acontecer de existir muitos clientes, então os clientes esperam sua vez nas cadeiras disponíveis. Escreva um programa em Java que faça esse problema clássico.