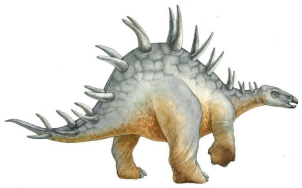
Capítulo 6: Sincronização de Processos

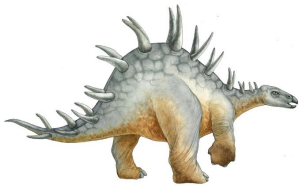
Capitulo 6: Sincronização de Processos

- Introdução
- O problema da Seção Crítica
- Sincronização de Hardware
- Semáforos
- Problemas Clássicos de Sincronização
- Monitores
- Exemplos de Sincronização



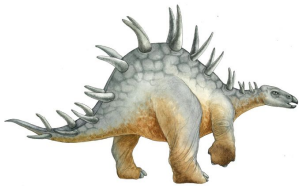
Introdução

- Acesso concorrente a dados compartilhados pode resultar em inconsistência;
- Manutenção de dados consistentes requer mecanismos para garantir a execução coordenada de processos que executam simultaneamente;
- Suponhamos que nós desejamos prover uma solução para o problema do produtor-consumidor quando as rotinas são executadas concorrentemente.



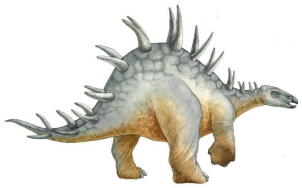
Produtor

```
while (count == BUFFER_SIZE)
   ; // do nothing

// add an item to the buffer
++count;
buffer[in] = item;
in = (in + 1) % BUFFER_SIZE;
```



Consumidor

```
while (count == 0)
   ; // do nothing

// remove an item from the buffer
--count;
item = buffer[out];
out = (out + 1) % BUFFER_SIZE;
```



Condição de Corrida (Race Condition)

- count++ pode ser implementado como

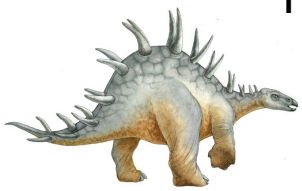
```
register1 = count
register1 = register1 + 1
count = register1
```

- count-- pode ser implementado como

```
register2 = count
register2 = register2 - 1
count = register2
```

- Considere esta execução intercalada com “count = 5” inicialmente:

P0: produtor executa register1 = count {register1 = 5}
P1: produtor executa register1 = register1 + 1 {register1 = 6}
P2: consumidor executa register2 = count {register2 = 5}
P3: consumidor executa register2 = register2 - 1 {register2 = 4}
P4: produtor executa count = register1 {count = 6 }
P5: consumidor executa count = register2 {count = 4}



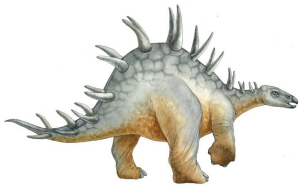
Problema da Seção Crítica

1. Condição de Corrida – Quando existe um acesso concorrente a um dado compartilhado e o resultado final depende da ordem da execução;

2. Seção Critica- Parte do código onde um dado compartilhado é acessado;

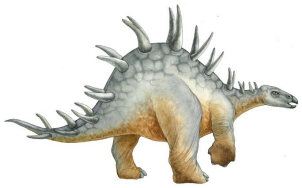
3. Seção de Entrada - Código que requisita a permissão para entrar em sua seção crítica;

4. Seção de Saída – Código que é executado após a saída da seção crítica.



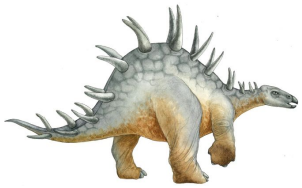
Estrutura Típica de um Processo

```
while (true) {
        entry section
                critical section
        exit section
                remainder section
}
```



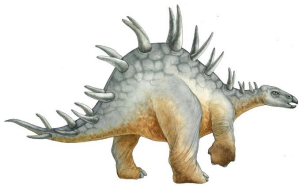
Solução para o problema da Seção Crítica

1. Exclusão Mutua – Se o processo P_i esta executando em sua seção critica, então nenhum outro processo pode estar executando em sua seção critica;
2. Progresso – Se nenhum processo está executando em sua seção critica e existem processos que desejam entrar em suas seções criticas, então somente os processos que não estão executando em sua seção de saída podem participar da seleção do processo que entrará na seção critica, esta seleção não pode ser postergada indefinidamente;
3. Espera Limitada - Deve existir um limite do número de vezes que outros processos são permitidos a entrar na suas seções críticas depois que um processo fez a solicitação para entrar em sua seção critica e antes desta requisição ser atendida



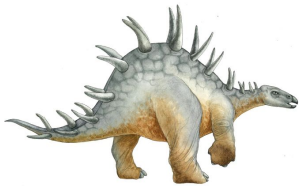
Solução de Peterson

- Solução para dois processos
- Assume que as operações de LOAD e STORE são atômicas, ou seja, não podem ser interrompidas;
- Os dois processos compartilham duas variáveis:
  - int turn;
  - Boolean flag[2]
- A variável turn indica de quem é a vez de entrar na seção critica;
- O array flag é usado para indicar se um processo está pronto para entrar na secao critica. flag[i] = true implica que o processo P_i está pronto



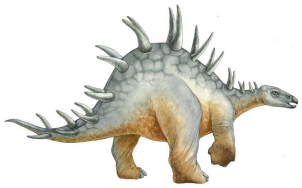
Algoritmo para processo P_i

```
while (true) {

    flag[i] = TRUE;
    turn = j;
    while (flag[j] && turn == j);

        critical section

    flag[i] = FALSE;

        remainder section

}
```



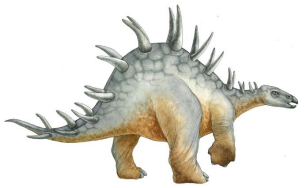
Seção critica usando locks

```
while (true) {

        acquire lock

            critical section

        release lock

            remainder section

}
```



Sincronização de Hardware

- Vários sistemas provêm suporte de hardware para seção crítica;
- Monoprocessadores – podem desabilitar interrupções
  - Código da seção critica poderia executar sem preempção
  - Geralmente muito ineficiente para multiprocessadores
    - SOs que usam isto não escalam
- Máquinas modernas provem instruções atômicas de hardware especiais
  - Atomic = não interrompível
  - Mesmo testar uma palavra na memória e definir seu valor
  - Ou trocar o conteúdo de duas palavras de memoria

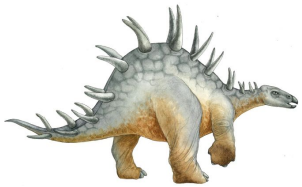


Estrutura de Dados para solução em Hardware

```
public class HardwareData
{
    private boolean value = false;

    public HardwareData(boolean value) {
       this.value = value;
    }

    public boolean get() {
       return value;
    }

    public void set(boolean newValue) {
       value = newValue;
    }

    public boolean getAndSet(boolean newValue) {
       boolean oldValue = this.get();
       this.set(newValue);

       return oldValue;
    }

   public void swap(HardwareData other) {
      boolean temp = this.get();

      this.set(other.get());
      other.set(temp);
    }
}
```

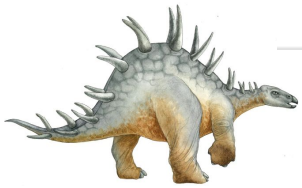
Solucao usando GetAndSet

```
// lock is shared by all threads
HardwareData lock = new HardwareData(false);

while (true) {
    while (lock.getAndSet(true))
       Thread.yield();

    criticalSection();
    lock.set(false);
    remainderSection();
}
```

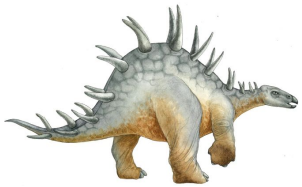


Solucao usando Swap

```
// lock is shared by all threads
HardwareData lock = new HardwareData(false);

// each thread has a local copy of key
HardwareData key = new HardwareData(true);

while (true) {
    key.set(true);

    do {
       lock.swap(key);
    }
    while (key.get() == true);

    criticalSection();
    lock.set(false);
    remainderSection();
}
```



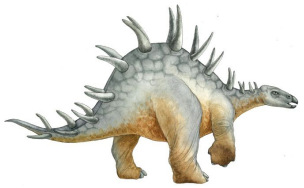
SEMÁFOROS



Semáforo

- Método de sincronização que não precisa de espera ocupada
- Semáforo *S* – variável inteira
- Duas operações padrões modificam S: acquire() e release()
  - Originalmente chamadas de P() e V()
- Menos complicada
- Só pode ser acessada via duas operações atômicas

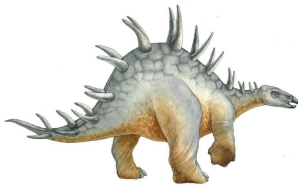
```
acquire() {
    while value <= 0
       ; // no-op
    value--;
}

    release() {
        value++;
    }
```



Semáforo como ferramenta de sincronismo

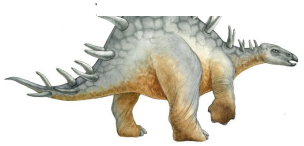
- Contador– valor inteiro que pode variar dentro de um dominio irrestrito
- Binário– valor inteiro que pode variar somente entre 0 e 1
  - Também conhecido como mutex locks

```
Semaphore S = new Semaphore();

S.acquire();

   // critical section

S.release();

   // remainder section
```



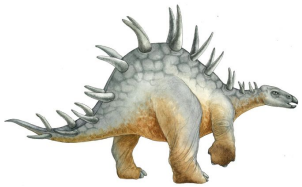
Exemplo em Java com uso de Semaforos

```
public class Worker implements Runnable
{
    private Semaphore sem;
    private String name;

    public Worker(Semaphore sem, String name) {
       this.sem = sem;
       this.name = name;
    }

    public void run() {
       while (true) {
          sem.acquire();
          MutualExclusionUtilities.criticalSection(name);
          sem.release();
          MutualExclusionUtilities.remainderSection(name);
       }
    }
}
```



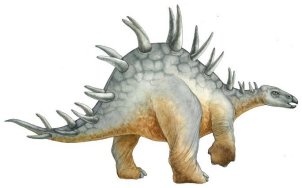
Java Example Using Semaphores

```
public class SemaphoreFactory
{
    public static void main(String args[]) {
       Semaphore sem = new Semaphore(1);
       Thread[] bees = new Thread[5];

       for (int i = 0; i < 5; i++)
          bees[i] = new Thread(new Worker
             (sem, "Worker " + (new Integer(i)).toString() ));

       for (int i = 0; i < 5; i++)
          bees[i].start();
    }
}
```



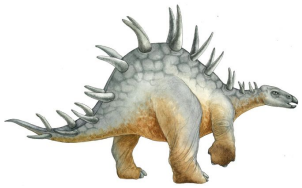
Implementação de semáforo

- Deve garantir que dois processos não podem executar acquire () e release () sobre o mesmo semáforo ao mesmo tempo;
- Consequentemente, a implementação se torna um problema de seção crítica onde o acquire e release são colocados na seção crítica
  - Pode agora ter uma espera ocupada na implementação da seção crítica
    - Mas a implementação pode ser pequena
    - Espera ocupada curta se a seção crítica for ramente ocupada
- Observe que aplicações podem gastar muito tempo em sua seção crítica e esta solução pode não ser boa



Implementação de Semáforo sem Espera Ocupada

- Com cada semáforo existe uma fila de espera. Cada entrada na fila de espera tem dois itens de dado:
  - valor (tipo inteiro)
  - ponteiro para o próximo registro da lista
- Duas operações:
  - block – coloca o processo solicitante na fila apropiada de espera.
  - wakeup – remove um dos processo da fila espera e o coloca na fila de prontos.



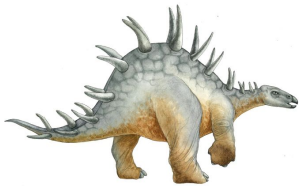
Implementação de Semáforo sem Espera Ocupada

- Implementation of **acquire()**:

```
acquire(){
    value--;
    if (value < 0) {
       add this process to list
       block;
    }
}
```

- Implementation of **release()**:

```
release(){
    value++;
    if (value <= 0) {
       remove a process P from list
       wakeup(P);
    }
}
```

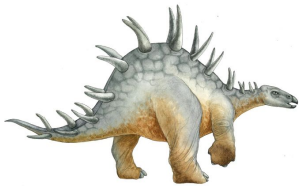


Deadlock and Starvation

- Deadlock – dois ou mais processos podem esperar indefinidamente por um evento, onde existe uma dependencia circular entre os processos
- Seja S e Q dois semáforos iniciados com o valor 1

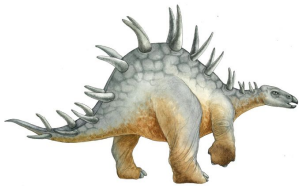
P_0	P_1
S.acquire();	Q.acquire();
Q.acquire();	S.acquire();
.	.
.	.
.	.
S.release();	Q.release();
Q.release();	S.release();

- Starvation – bloqueio indefinido. Um processo pode nunca ser removido da fila de semáforos.



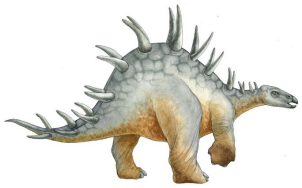
Problemas Classicos de Sincronizacao

- Problema de Buffer Limitado
- Problema dos Leitores e Escritores
- Problema do Jantar dos Filósofos



Introducao

- Acesso concorrente a dados compartilhados pode resultar em inconsistencia
- Manutencao de dados consistentes requer mecanismos para garantir a execucao coordenada de processos que executam simultaneamente
- Suponhamos que nos desejamos prover uma solucao para o problema do produtor-consumidor (cap. 3) quando as rotinas sao executadas concorrentemente.



EXERCICIO: PRODUTOR- CONSUMIDOR

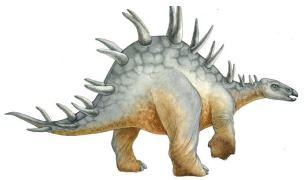
Prover uma solucao para o problema do produtor-consumidor usando semaforos. Use semaforos para controlar também o tamanho do buffer

Produtor

```
while (count == BUFFER_SIZE)
   ; // do nothing

// add an item to the buffer
++count;
buffer[in] = item;
in = (in + 1) % BUFFER_SIZE;
```

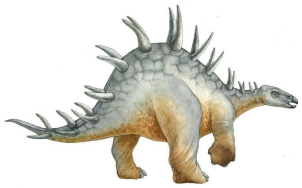
Consumidor

```
while (count == 0)
   ; // do nothing

// remove an item from the buffer
--count;
item = buffer[out];
out = (out + 1) % BUFFER_SIZE;
```



Problema do Buffer Limitado

- *N* buffers, cada um pode guardar um item de dado
- Semaforo mutex iniciado com valor 1
- Semaforo cheio iniciado com valor 0
- Semaforo vazio iniciado com valor N



Problema de Buffer Limitado (cont.)

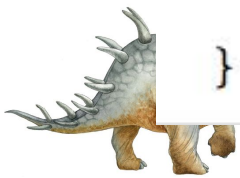
- A fabrica de produtor e consumidor

```
public class Factory
{
    public static void main(String args[]) {
       Buffer buffer = new BoundedBuffer();

       // now create the producer and consumer threads
       Thread producer = new Thread(new Producer(buffer));
       Thread consumer = new Thread(new Consumer(buffer));

       producer.start();
       consumer.start();
    }
}
```

Problema de Buffer Limitado (cont.)

- A estrutura do processo produtor

```
public class Producer implements Runnable
{
   private Buffer buffer;

   public Producer(Buffer buffer) {
      this.buffer = buffer;
   }

   public void run() {
      Date message;

      while (true) {
         // nap for awhile
         SleepUtilities.nap();
         // produce an item & enter it into the buffer
         message = new Date();
         buffer.insert(message);
      }
   }
}
```



Problema de Buffer Limitado (cont.)

- A estrutura do processo consumidor

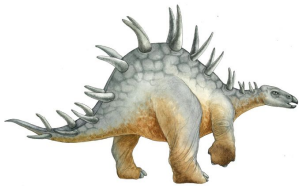
```
public class Consumer implements Runnable
{
   private Buffer buffer;

   public Consumer(Buffer buffer) {
      this.buffer = buffer;
   }

   public void run() {
      Date message;

      while (true) {
         // nap for awhile
         SleepUtilities.nap();
         // consume an item from the buffer
         message = (Date)buffer.remove();
      }
   }
}
```

Problema de Buffer Limitado (cont.)

```
public class BoundedBuffer implements Buffer
{
    private static final int BUFFER_SIZE = 5;
    private Object[] buffer;
    private int in, out;
    private Semaphore mutex;
    private Semaphore empty;
    private Semaphore full;

    public BoundedBuffer() {
       // buffer is initially empty
       in = 0;
       out = 0;
       buffer = new Object[BUFFER_SIZE];

       mutex = new Semaphore(1);
       empty = new Semaphore(BUFFER_SIZE);
       full = new Semaphore(0);
    }

    public void insert(Object item) {
       // Figure 6.9
    }

    public Object remove() {
       // Figure 6.10
    }
}
```

Problema de Buffer Limitado (cont.)

```
public void insert(Object item) {
    empty.acquire();
    mutex.acquire();

    // add an item to the buffer
    buffer[in] = item;
    in = (in + 1) % BUFFER_SIZE;

    mutex.release();
    full.release();
}
```



Problema de Buffer Limitado (cont.)

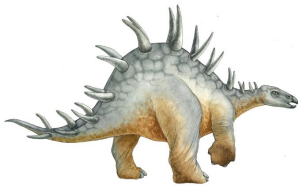
Metodo remove()

```
public Object remove() {
    full.acquire();
    mutex.acquire();

    // remove an item from the buffer
    Object item = buffer[out];
    out = (out + 1) % BUFFER_SIZE;

    mutex.release();
    empty.release();

    return item;
}
```

EXERCICIO:
Problema Leitores-Escritores

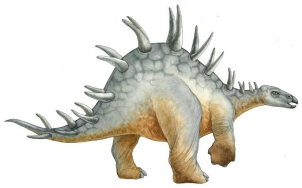
- Um conjunto de dados é compartilhado entre um certo número de processos concorrentes
  - Readers – só leem o conjunto de dados; eles não realizam atualizações
  - Writers – podem ler e gravar dados.

- Problema – permitir que múltiplos readers leiam ao mesmo tempo. Somente um único escritor pode acessar o conjunto de dados compartilhados em um determinado momento.



Problema Leitores-Escritores

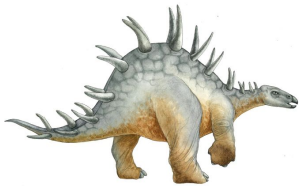
- Um conjunto de dados é compartilhado entre um certo numero de processos concorrentes
  - Readers – só leem o conjunto de dados; eles nao realizam atualizações
  - Writers – podem ler e gravar dados.
- Problema – permitir que múltiplos readers leiam ao mesmo tempo. Somente um único escritor pode acessar o conjunto de dados compartilhados em um determinado momento.
- Dados Compartilhados
  - Conjunto de dados
  - Semaforo mutex iniciado com 1.
  - Semaforo db iniciado com 1.
  - Inteiro readerCount iniciado com 0.



Problema Leitores-Escritores

Interface for read-write locks

```
public interface RWLock
{
   public abstract void acquireReadLock();
   public abstract void acquireWriteLock();
   public abstract void releaseReadLock();
   public abstract void releaseWriteLock();
}
```



Problema Leitores-Escritores

- The structure of a writer process

```
public class Writer implements Runnable
{
    private RWLock db;

    public Writer(RWLock db) {
       this.db = db;
    }

    public void run() {
       while (true) {
          // nap for awhile
          SleepUtilities.nap();

          db.acquireWriteLock();

          // you have access to write to the database
          SleepUtilities.nap();

          db.releaseWriteLock();
       }
    }
}
```

Problema Leitores-Escritores

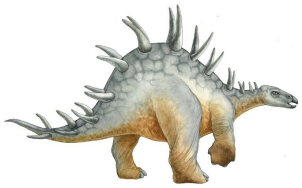
- The structure of a reader process

```
public class Reader implements Runnable
{
    private RWLock db;

    public Reader(RWLock db) {
       this.db = db;
    }

    public void run() {
       while (true) {
          // nap for awhile
          SleepUtilities.nap();

          db.acquireReadLock();

          // you have access to read from the database
          SleepUtilities.nap();

          db.releaseReadLock();
       }
    }
}
```

Problema Leitores-Escritores

Methods called by writers.

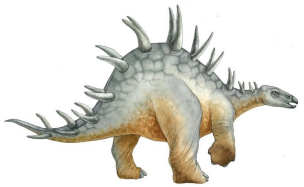
```
public void acquireWriteLock() {
   db.acquire();
}

public void releaseWriteLock() {
   db.release();
}
```



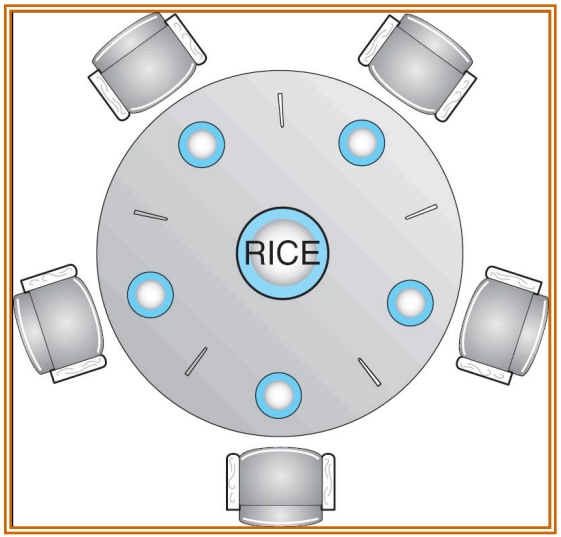
Problema Leitores-Escritores

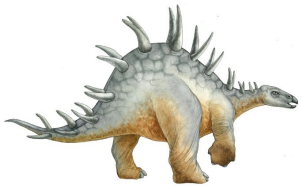
Methods called by readers.

```
public void acquireReadLock() {
    mutex.acquire();
    ++readerCount;

    // if I am the first reader tell all others
    // that the database is being read
    if (readerCount == 1)
       db.acquire();

    mutex.release();
}

public void releaseReadLock() {
    mutex.acquire();
    --readerCount;

    // if I am the last reader tell all others
    // that the database is no longer being read
    if (readerCount == 0)
       db.release();

    mutex.release();
}
```

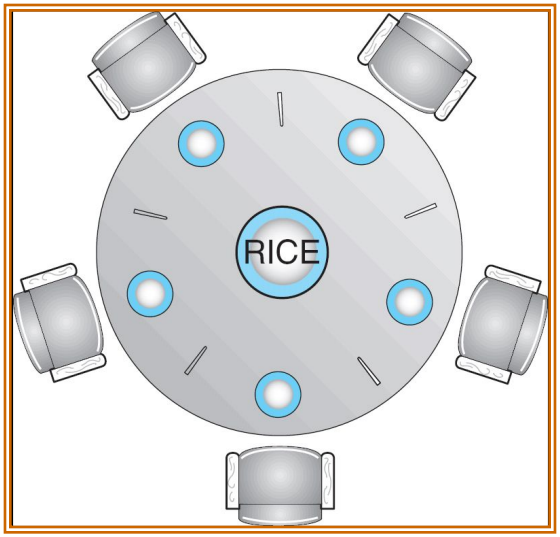


EXERCICIO:
Problema do Jantar dos Filosofos

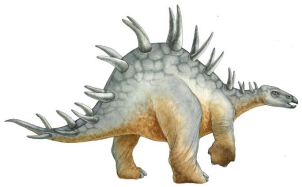




Problema do Jantar dos Filosofos



- Dado compartilhado
  - Prato de arroz (conjunto de dado)
  - Semaforo chopStick [5] iniciado com 1

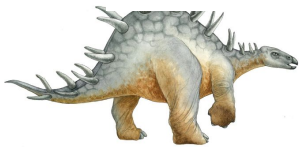


Problema do Jantar dos Filosofos (Cont.)

- The structure of Philosopher *i*:

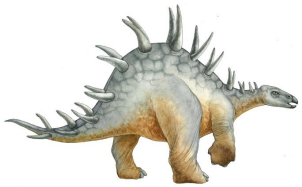
```
while (true) {
    // get left chopstick
    chopStick[i].acquire();
    // get right chopstick
    chopStick[(i + 1) % 5].acquire();

    eating();

    // return left chopstick
    chopStick[i].release();
    // return right chopstick
    chopStick[(i + 1) % 5].release();

    thinking();
}
```

Existe algum problema



Problemas com semaforos

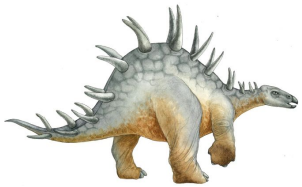
- Uso correto de operacoes com semaforos:
  - mutex.acquire() …. mutex.release()
  - mutex.wait() … mutex.wait()
  - Omitindo o mutex.wait () ou mutex.release() (ou ambos)



Monitores

Monitores

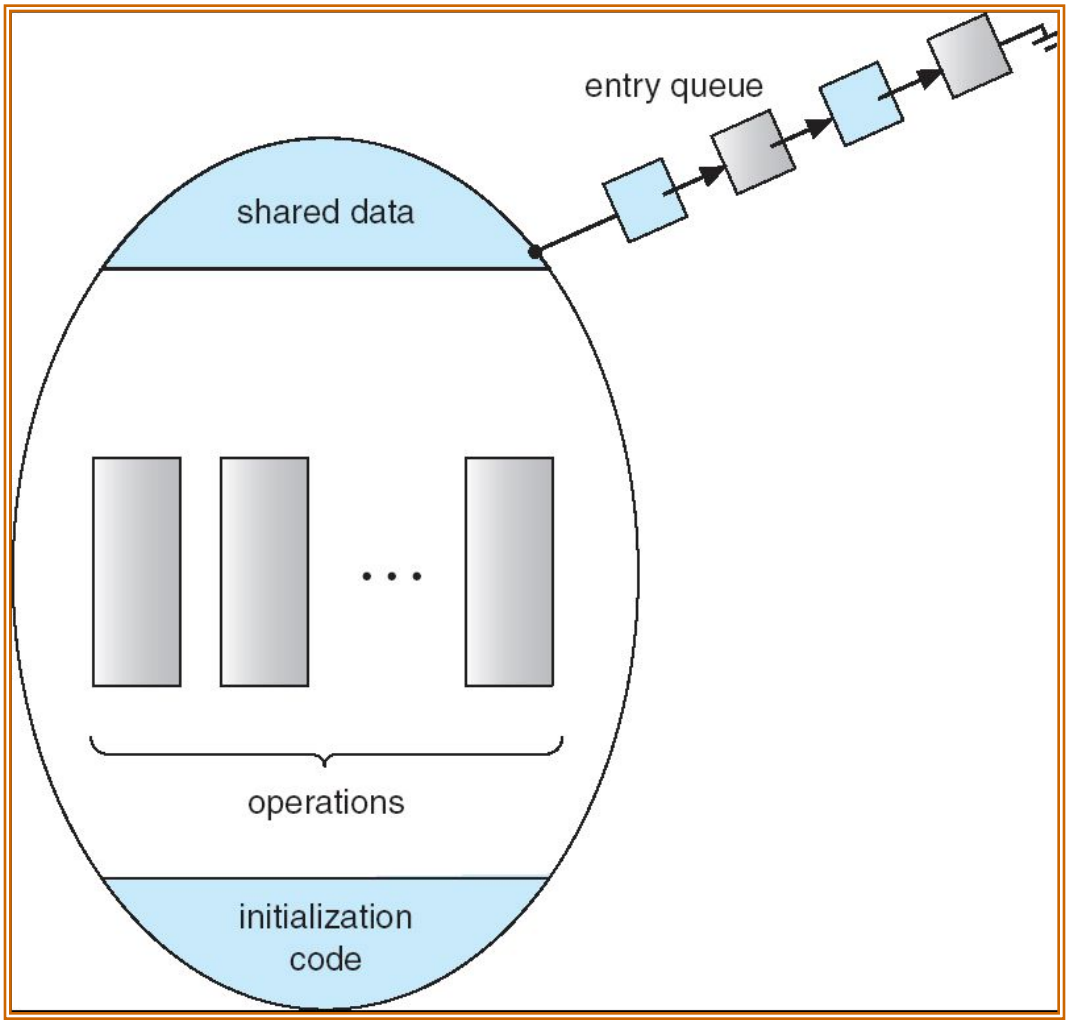
- Uma abstracao de alto-nivel que prove um mecanismo eficiente e conveniente para sincronizacao de processos
- Um unico processo pode estar ativo dentro do monitor a cada momento

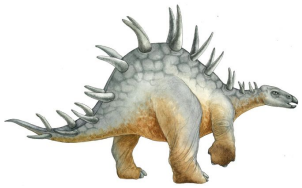


Sintaxe de um Monitor

```
monitor monitor name
{
   // shared variable declarations

   initialization code ( . . . ) {
      . . .
   }

   public P1 ( . . . ) {
      . . .
   }

   public P2 ( . . . ) {
      . . .
   }
         .
         .
         .
   public Pn ( . . . ) {
      . . .
   }
}
```

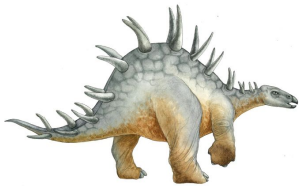
Visao esquematica de um monitor



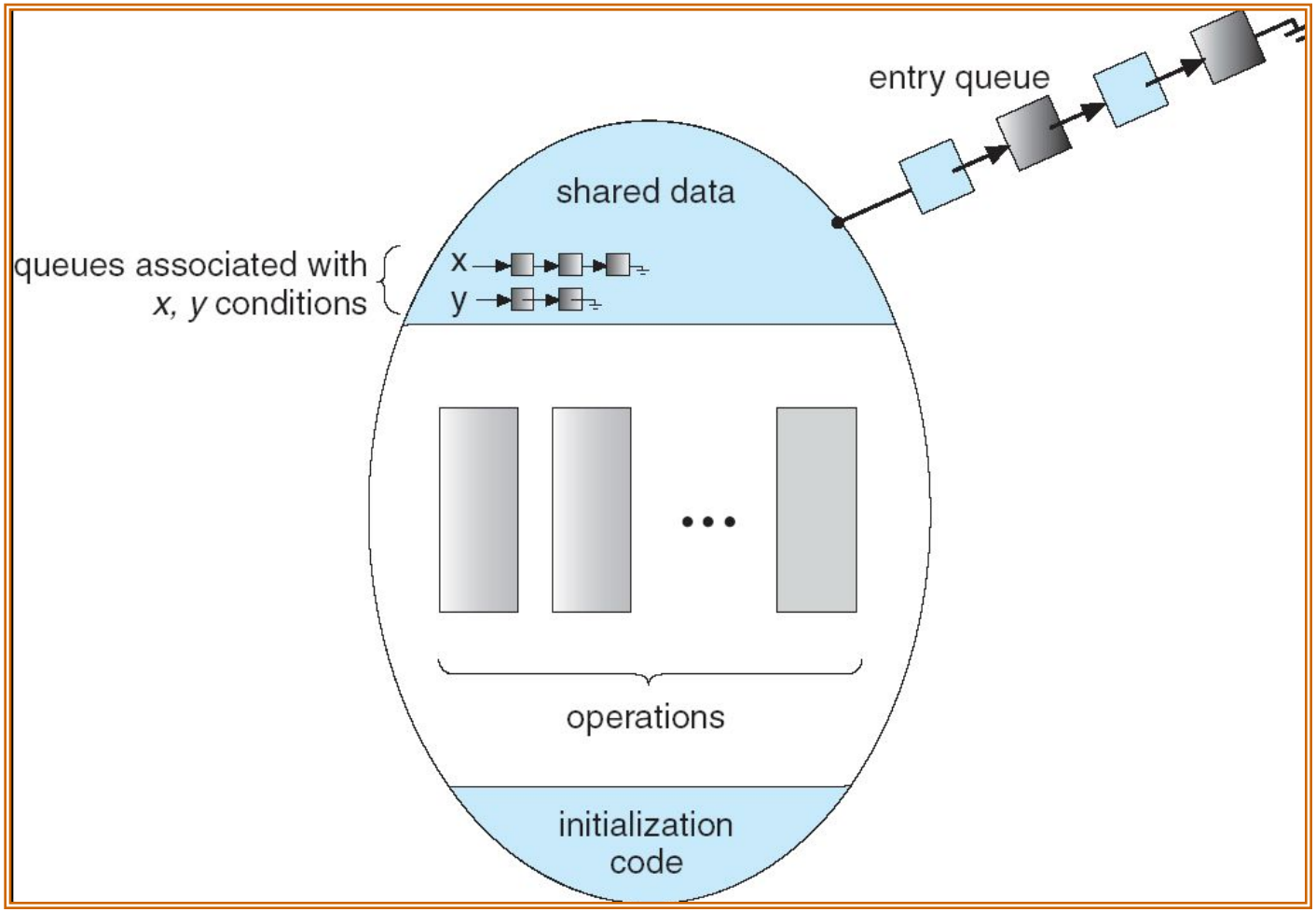


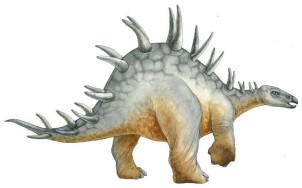
Variaveis de Condicao

- Condition x, y;
- Duas operacoes sobre a variavel de condicao:
  - x.wait () – um processo que invoca a operacao é suspendido.
  - x.signal () – acorda um dos processos que invocou x.wait ()



Monitor com Variaveis de Condicao





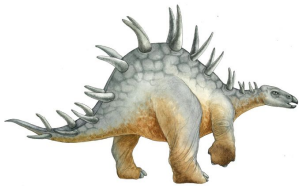
Solucao para o jantar dos filosofos

```
monitor DiningPhilosophers
{
    enum State {THINKING, HUNGRY, EATING};
    State[] states = new State[5];
    Condition[] self = new Condition[5];

    public DiningPhilosophers {
       for (int i = 0; i < 5; i++)
          state[i] = State.THINKING;
    }

    public void takeForks(int i) {
       state[i] = State.HUNGRY;
       test(i);
       if (state[i] != State.EATING)
          self[i].wait;
    }

    public void returnForks(int i) {
       state[i] = State.THINKING;
       // test left and right neighbors
       test((i + 4) % 5);
       test((i + 1) % 5);
    }

    private void test(int i) {
       if ( (state[(i + 4) % 5] != State.EATING) &&
          (state[i] == State.HUNGRY) &&
          (state[(i + 1) % 5] != State.EATING) ) {
             state[i] = State.EATING;
             self[i].signal;
       }
    }
}
```

Solucao para o jantar dos filosofos (cont)

- Cada Filosofo invoca as operacoes takeForks(i) e returnForks(i) na seguinte sequencia:

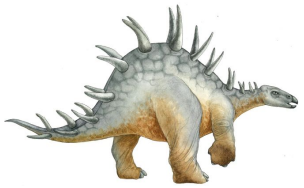
```
dp.takeForks (i)

    EAT

dp.returnForks (i)
```



Sincronização Java

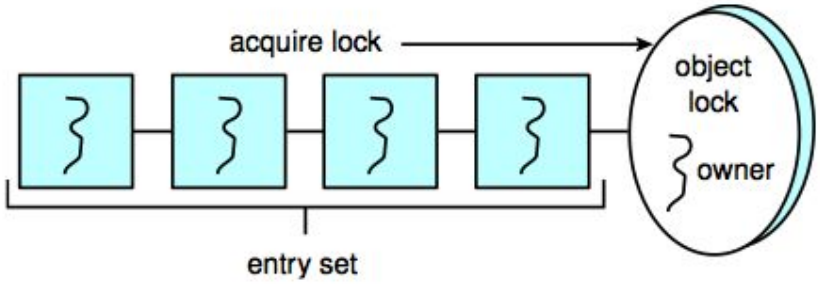
Java Synchronization

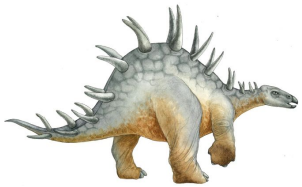
- Java provides synchronization at the language-level.
- Each Java object has an associated lock.
- This lock is acquired by invoking a **synchronized** method.
- This lock is released when exiting the synchronized method.
- Threads waiting to acquire the object lock are placed in the **entry set** for the object lock.



Java Synchronization

Each object has an associated **entry set**.

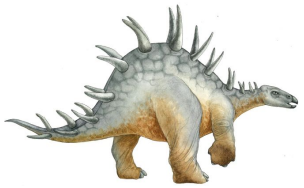




Java Synchronization

Synchronized insert() and remove() methods

```
public synchronized void insert(Object item) {
    while (count == BUFFER_SIZE)
       Thread.yield();

    ++count;
    buffer[in] = item;
    in = (in + 1) % BUFFER_SIZE;
}


public synchronized Object remove() {
    Object item;

    while (count == 0)
       Thread.yield();

    --count;
    item = buffer[out];
    out = (out + 1) % BUFFER_SIZE;

    return item;
}
```



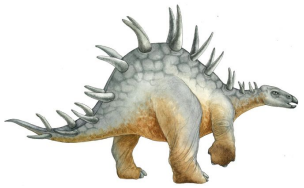
Java Synchronization wait/notify()

- When a thread invokes **wait():**

  1. The thread releases the object lock;
  2. The state of the thread is set to Blocked;
  3. The thread is placed in the **wait set** for the object.

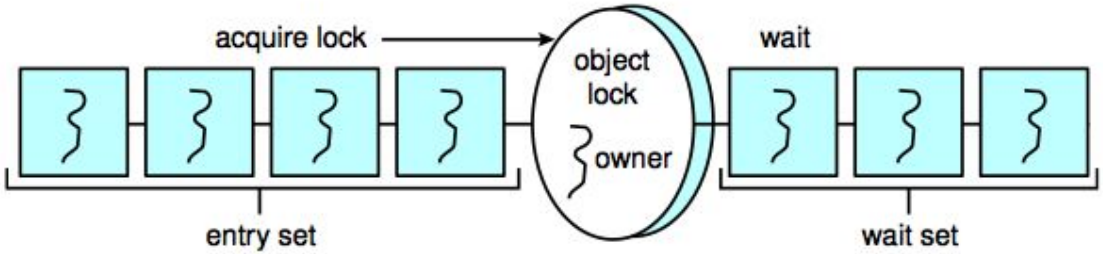
- When a thread invokes **notify()**:

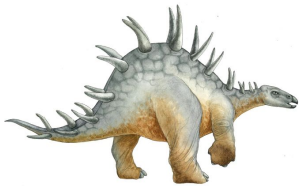
  1. An arbitrary thread T from the wait set is selected;
  2. T is moved from the wait to the entry set;
  3. The state of T is set to Runnable.



Java Synchronization

Entry and wait sets





Java Synchronization - Bounded Buffer

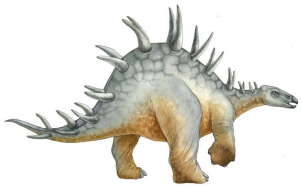
```
public class BoundedBuffer implements Buffer
{
    private static final int BUFFER_SIZE = 5;

    private int count, in, out;
    private Object[] buffer;

    public BoundedBuffer() {
      // buffer is initially empty
      count = 0;
      in = 0;
      out = 0;
      buffer = new Object[BUFFER_SIZE];
    }

    public synchronized void insert(Object item) {
       // Figure 6.28
    }

    public synchronized Object remove() {
       // Figure 6.28
    }
}
```

Java Synchronization - Bounded Buffer

```
public synchronized void insert(Object item) {
    while (count == BUFFER_SIZE) {
       try {
          wait();
       }
       catch (InterruptedException e) { }
    }

    ++count;
    buffer[in] = item;
    in = (in + 1) % BUFFER_SIZE;

    notify();
}

public synchronized Object remove() {
    Object item;

    while (count == 0) {
       try {
          wait();
       }
       catch (InterruptedException e) { }
    }

    --count;
    item = buffer[out];
    out = (out + 1) % BUFFER_SIZE;

    notify();

    return item;
}
```

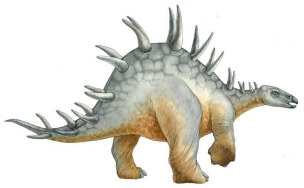
Java Synchronization

- The call to **notify()** selects an aribitrary thread from the wait set. It is possible the selected thread is in fact not waiting upon the condition for which it was notified.
- The call **notifyAll()** selects all threads in the wait set and moves them to the entry set.
- In general, **notifyAll()** is a more conservative strategy than **notify()**.



Java Synchronization - Readers-Writers

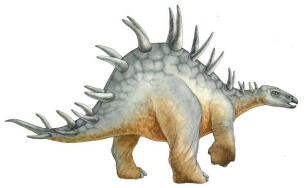
```
public class Database implements RWLock
{
    private int readerCount;
    private boolean dbWriting;

    public Database() {
       readerCount = 0;
       dbWriting = false;
    }

    public synchronized void acquireReadLock() {
       // Figure 6.33
    }

    public synchronized void releaseReadLock() {
       // Figure 6.33
    }

    public synchronized void acquireWriteLock() {
       // Figure 6.34
    }

    public synchronized void releaseWriteLock() {
       // Figure 6.34
    }
}
```



Java Synchronization - Readers-Writers

Methods called by readers

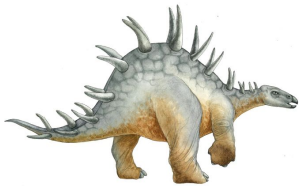
```
public synchronized void acquireReadLock() {
    while (dbWriting == true) {
       try {
          wait();
       }
       catch(InterruptedException e) { }
    }

    ++readerCount;
}

public synchronized int releaseReadLock() {
    --readerCount;

    // if I am the last reader tell writers
    // that the database is no longer being read
    if (readerCount == 0)
       notify();
}
```



Java Synchronization - Readers-Writers

Methods called by writers

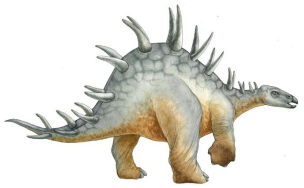
```
public synchronized void acquireWriteLock() {
    while (readerCount > 0 || dbWriting == true) {
       try {
          wait();
       }
       catch(InterruptedException e) { }
    }

    // once there are either no readers or writers
    // indicate that the database is being written
    dbWriting = true;
}

public synchronized void releaseWriteLock() {
    dbWriting = false;

    notifyAll();
}
```



Java Synchronization

Rather than synchronizing an entire method, blocks of code may be declared as synchronized

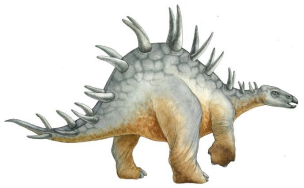
```
Object mutexLock = new Object();
. . .
public void someMethod() {
    nonCriticalSection();

    synchronized(mutexLock) {
       criticalSection();
    }

    remainderSection();
}
```



Java Synchronization

Block synchronization using wait()/notify()

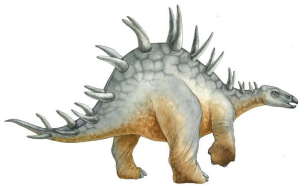
```
Object mutexLock = new Object();
. . .
synchronized(mutexLock) {
    try {
       mutexLock.wait();
    }
    catch (InterruptedException ie) { }
}

synchronized(mutexLock) {
    mutexLock.notify();
}
```



Concurrency Features in Java 5

Semaphores

```
Semaphore sem = new Semaphore(1);

try {
   sem.acquire();
   // critical section
}
catch (InterruptedException ie) { }
finally {
   sem.release();
}
```

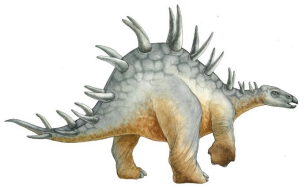


Concurrency Features in Java 5

A condition variable is created by first creating a **ReentrantLock** and invoking its **newCondition()** method:

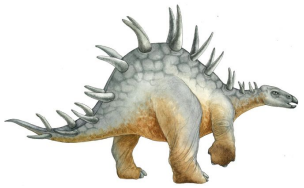
```
Lock key = new ReentrantLock();
Condition condVar = key.newCondition();
```

Once this is done, it is possible to invoke the **await()** and **signal()** methods.



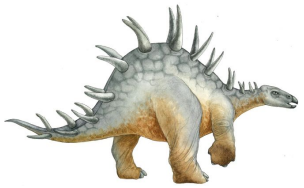
Monitor em Java

```
import java.util.concurrent.locks.Condition;
import java.util.concurrent.locks.Lock;
import java.util.concurrent.locks.ReentrantLock;

public class BoundedBuffer {
    private final String[] buffer;
    private final int capacity;

    private int front;
    private int rear;
    private int count;

    private final Lock lock = new ReentrantLock();

    private final Condition notFull = lock.newCondition();
    private final Condition notEmpty = lock.newCondition();
```



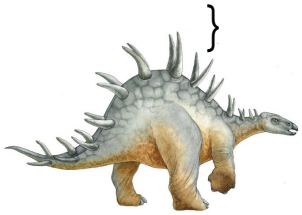
Monitor em Java

```
public BoundedBuffer(int capacity) {
        super();

        this.capacity = capacity;

        buffer = new String[capacity];
    }
```



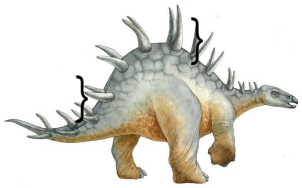
Monitor em Java

```
public void deposit(String data) throws
InterruptedException {
        lock.lock();

        try {
            while (count == capacity) {
                notFull.await();
            }

            buffer[rear] = data;
            rear = (rear + 1) % capacity;
            count++;

            notEmpty.signal();
        } finally {
            lock.unlock();
        }
    }
```



Monitor em Java

```
public String fetch() throws InterruptedException {
        lock.lock();

        try {
            while (count == 0) {
                notEmpty.await();
            }

            String result = buffer[front];
            front = (front + 1) % capacity;
            count--;

            notFull.signal();

            return result;
        } finally {
            lock.unlock();
        }
```



End of Chapter 6