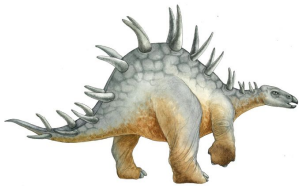
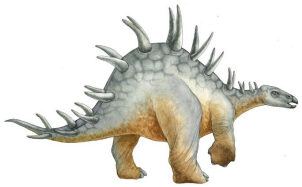
Sistemas Operacionais
Laboratório 6
Sincronização entre Processos

SINCRONIZAÇÃO EM JAVA



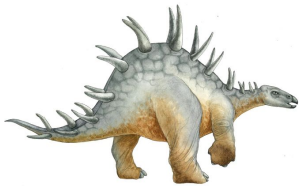
Arquivos para baixar

- Os arquivos que serão usados no laboratório estão disponíveis em https://github.com/josemacedo/sistemas-operacionais/tree/master/4_Lab_Sincronizacao



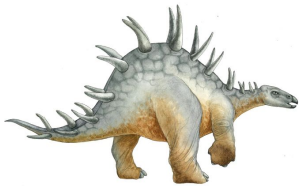
Threads

- Threads em Java
  - Java suporta concorrência a nível de linguagem
    - A linguagem C suporta concorrência em bibliotecas de funções
  - Adota um modelo preemptivo de execução
    - Intervalos do processador são alocados às threads em um modelo round-robin
  - Suporta o conceito de prioridade
  - Implementado na classe Thread e na interface Runnable



Threads

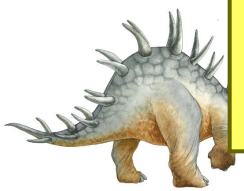
- Threads em Java
  - Todo programa consiste de pelo menos uma thread que executa o método `main`
    - thread principal (main thread)
  - Outras threads internas podem ser criadas internamente pela JVM
    - Depende de cada implementação de JVM
  - Outras threads a nível de usuário podem ser explicitamente criados pelo programa



Threads em Java

- Classe Thread
  - Deve-se estender a classe `Thread`
  - Sobrepor o método `run`
    - Método executado quando a thread é iniciada
  - Instanciar um ou vários objetos thread
  - Ativar o método `start` dos objetos thread para iniciar as threads correspondentes

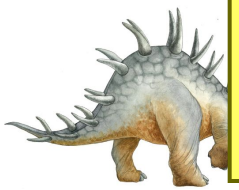
```
public class HelloWorldThread extends Thread {

   public void run() {
      System.out.println("Hello World");
   }

   public static void main(String[] args) {
      HelloWorldThread t = new HelloWorldThread();
      t.start();
   }
}
```



Threads em Java

- Classe Thread

```
class MyThread extends Thread {
   private String message;

   public MyThread(String m) {message = m;}

   public void run() {
      for(int r=0; r<20; r++)
         System.out.println(message);
   }

   public static void main(String[] args) {
      MyThread t1,t2;
      t1=new MyThread("primeira thread");
      t2=new MyThread("segunda thread");
      t1.start();
      t2.start();

}
}
```



Threads em Java

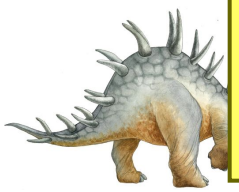
- Interface Runnable
  - Deve-se implementar a interface `Runnable`
  - Implementar o método `run`
    - Método executado quando a thread é iniciada
  - Instanciar um ou vários objetos thread
  - Ativar o método `start` dos objetos thread para iniciar as threads correspondentes

```
public class HelloWorldThread2 implements Runnable {

   public void run() {
      System.out.println("Hello World");
   }

   public static void main(String[] args) {
      HelloWorldThread2 h = new HelloWorldThread2();
      Thread t = new Thread(h);
      t.start();
   }
}
```

Threads em Java

- Interface Runnable

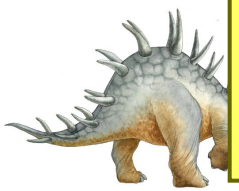
```
class MyRunnable implements Runnable {
   private String message;

   public MyRunnable (String m) {message = m;}

   public void run() {
      for(int r=0; r<20; r++)
         System.out.println(message);
   }


public static void main(String[] args) {
      MyRunnable r1,r2;
      Thread t1, t2;
      r1=new MyRunnable("primeira thread");
      r2=new MyRunnable("segunda thread");
      t1 = new Thread(r1);
      t2 = new Thread(r2);
      t1.start();
      t2.start();
   }
}
```



Threads em Java

- Interface Runnable

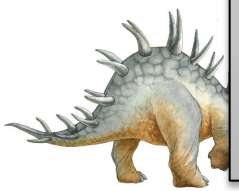
```
public class MySecondRunnable implements
Runnable{

    public void run() {
        System.out.printf("I'm running in thread %s \n",
       Thread.currentThread().getName());

    }
    public static void main(String[] args) {
Runnable runnable = new MySecondRunnable();

for(int i = 0; i < 25; i++){
    Thread thread = new Thread(runnable);
    thread.setName("Thread " + i);
    thread.start();
}
}

}
```



Threads em Java

- Interface Runnable (Rodando o algoritmo)

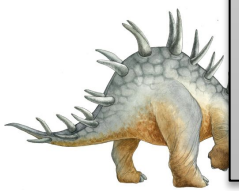
```
I'm running in thread Thread 1
I'm running in thread Thread 24
I'm running in thread Thread 22
I'm running in thread Thread 23
I'm running in thread Thread 18
I'm running in thread Thread 19
I'm running in thread Thread 13
I'm running in thread Thread 20
I'm running in thread Thread 21
I'm running in thread Thread 17
I'm running in thread Thread 16
I'm running in thread Thread 15
I'm running in thread Thread 14
I'm running in thread Thread 10
I'm running in thread Thread 12
I'm running in thread Thread 11
I'm running in thread Thread 9
I'm running in thread Thread 8
I'm running in thread Thread 7
I'm running in thread Thread 6
I'm running in thread Thread 5
I'm running in thread Thread 4
I'm running in thread Thread 3
I'm running in thread Thread 2
I'm running in thread Thread 0
```



Threads em Java

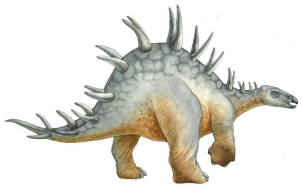
- Interface Runnable (Rodando o algoritmo pela 2ª vez)

```
I'm running in thread Thread 0
I'm running in thread Thread 17
I'm running in thread Thread 15
I'm running in thread Thread 11
I'm running in thread Thread 10
I'm running in thread Thread 24
I'm running in thread Thread 23
I'm running in thread Thread 22
I'm running in thread Thread 21
I'm running in thread Thread 20
I'm running in thread Thread 19
I'm running in thread Thread 18
I'm running in thread Thread 16
I'm running in thread Thread 14
I'm running in thread Thread 13
I'm running in thread Thread 12
I'm running in thread Thread 9
I'm running in thread Thread 8
I'm running in thread Thread 6
I'm running in thread Thread 7
I'm running in thread Thread 5
I'm running in thread Thread 4
I'm running in thread Thread 3
I'm running in thread Thread 2
I'm running in thread Thread 1
```



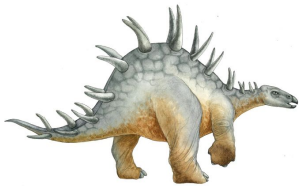
Threads em Java

- Thread x Runnable
  - Estender a classe `Thread` impossibilita herança de outra classe
    - Java não suporta herança múltipla
  - Implementar a interface `Runnable` permite a herança de outra classe
    - Java permite as primitivas `extends` e `implements` serem usadas conjuntamente



Threads em Java

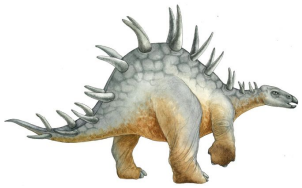
- Terminando Threads
  - Uma thread termina quando o método `run` é concluído
    - Método `run` retorna normalmente
    - Método `run` lança uma exceção não capturada
  - Threads não podem ser reinicializadas
    - Invocar o método `start` mais que uma vez gera a exceção `InvalidThreadStateException`
  - Método `isAlive` pode ser usado para verificar se a thread não foi terminada



```
t.start()
...
if t.isAlive()
```

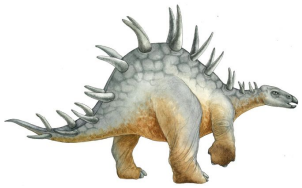
Threads em Java

- Prioridades
  - Pode ser usada para expressar a importância ou urgência de diferentes threads
  - Políticas de escalonamento de threads não são padronizadas em Java
    - Dependente da implementação da JVM
    - Em geral, preferência é dada a threads de maior prioridade



Threads em Java

- Prioridades
  - Cada thread possui uma prioridade que varia entre `Thread.MIN_PRIORITY` (1) e `Thread.MAX_PRIORITY` (10)
  - Por default, cada nova thread tem a prioridade da thread pai
    - Thread principal associada com o método `main` tem a prioridade `Thread.NORM_PRIORITY` (5)
  - Prioridade de uma thread pode ser identificada e modificada com os métodos `getPriority` e `setPriority`



Threads em Java

- Gerenciamento de Threads
  - A classe Thread define diversos métodos para gerenciamento das threads
    - Métodos estáticos ativados pela própria thread
      - Provêem informações sobre a própria thread
      - Alteram o estado da própria thread
    - Outros métodos que podem ser invocados por outras threads

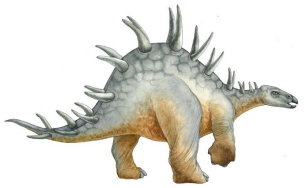
```
public static Thread currentThread()
  Retorna uma referência à thread atualmente em execução.
```

Threads em Java

- Gerenciamento de Threads
  - Interrupções
    - Uma interrupção é uma indicação de que a thread deve parar a execução
      - Programador decide como a thread responde a uma interrupção
      - Em geral, a thread termina sua execução
    - Interrupção é gerada invocando o método `interrupt` do objeto thread a ser interrompido

```
public void interrupt()
```

```
...
t.interrupt();
...
```



Threads em Java

- Gerenciamento de Threads
  - Tratando Interrupções
    - Ao receber uma interrupção, a thread pode . . .
      - Capturar a exceção InterruptedException
        - Somente é possível quando a thread executa métodos que lançam a exceção
      - Verificar se foi interrompida usando o método estático `Thread.interrupted`

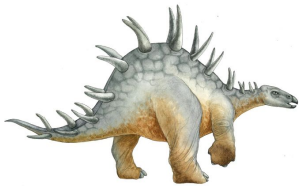
```
public static boolean interrupted()
```

```
...
if (Thread.interrupted()) {
   return;
}
...
```

```
...
if (Thread.interrupted()) {
   throw new
InterruptedException();
}
...
```

Threads em Java

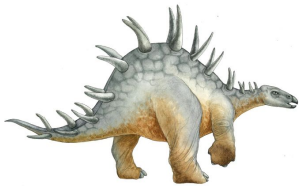
- Gerenciamento de Threads
  - Tratando Interrupções
    - Interrupção é implementada usando um flag denominado *interrupt*
      - Invocação do método `Thread.interrupt` liga o flag *interrupt status*
      - Invocação do método `interrupted` desliga o flag *interrupt status*
      - Qualquer método que lança a exceção `InterruptedException` desliga o flag *interrupt status*



Threads em Java

- Gerenciamento de Threads
  - Tratando Interrupções
    - O método `Thread.isInterrupted` pode ser usado por uma thread para verificar se outra thread foi interrompida
      - Não altera o flag da thread verificada
    - É possível ligar/desligar o flag várias vezes

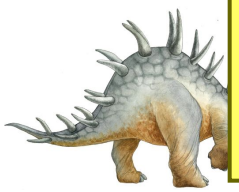
```
public boolean
isInterrupted()
```



Threads em Java

- Tratando interrupções

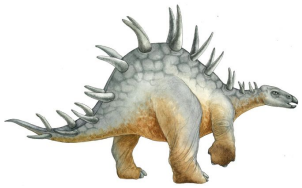
```
public class InterruptThread {
    public static void main(String[] args) {
        Thread thread1 = new Thread(new WaitRunnable());

        thread1.start();

        try {
            Thread.sleep(1000);
        } catch (InterruptedException e) {
            e.printStackTrace();
        }

        thread1.interrupt();
    }

    private static class WaitRunnable implements Runnable {
        @Override
        public void run() {
            System.out.println("Current time millis : " + System.currentTimeMillis());

            try {
                Thread.sleep(5000);
            } catch (InterruptedException e) {
                System.out.println("A thread has been interrupted");
                System.out.println("The thread is interrupted : " + Thread.currentThread().isInterrupted());
            }
            System.out.println("Current time millis : " + System.currentTimeMillis());
        }
        }
}
```



Threads em Java

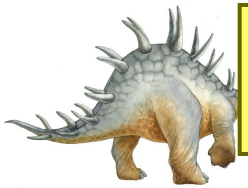
- Tratando interrupções (Executando o algoritmo)

```
Current time millis : 1411671363127
The thread has been interrupted
The thread is interrupted : false
Current time millis : 1411671364127
```



Threads em Java

- Gerenciamento de Threads
  - Pausando a Execução
    - O método `Thread.sleep` suspende a execução da própria thread por um período de tempo especificado
      - Torna o processador disponível para outras threads
      - Pode ser usado para regular o tempo de execução
      - Precisão depende dos temporizadores do sistema e do escalonador

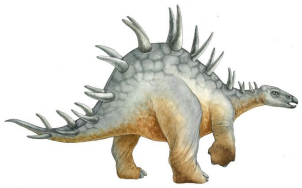


```
public static void sleep(long millis)
public static void sleep(long millis, int nanos)
```

Threads em Java

- Gerenciamento de Threads
  - Pausando a Execução
    - Tempo de pausa pode ser terminado por interrupções
      - Gera a exceção `InterruptedException`

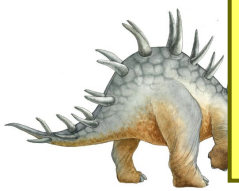
```
...
try {
   Thread.sleep(4000);
} catch (InterruptedException e) {
   return;
}
...
```



Threads em Java

- Pausando a execução

```
public class SleepThread {
   public static void main(String[] args) {
      System.out.println("Current time millis : " + System.currentTimeMillis());

      try {
         Thread.sleep(1000);
      } catch (InterruptedException e) {
         e.printStackTrace();
      }

      System.out.println("Current time millis : "       + System.currentTimeMillis());

      System.out.println("Nano time : " + System.nanoTime());

      try {
         Thread.sleep(2, 5000);
      } catch (InterruptedException e) {
         e.printStackTrace();
      }

      System.out.println("Nano time : " + System.nanoTime());
   }
}
```

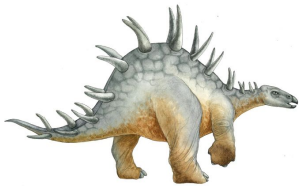


Threads em Java

- Pausando a execução (Rodando o algoritmo)

```
Current time millis : 1411670964563
Current time millis : 1411670965564
Nano time : 1124690940447899
Nano time : 1124690943306766
```

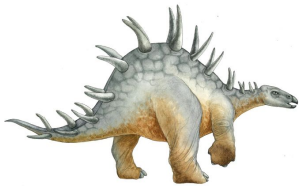
  - Diferença de precisão em milisegundos e nanosegundos. O resultado depende do computador, sistema operacional e configuração.



Threads em Java

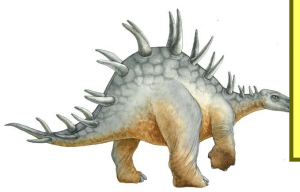
- Gerenciamento de Threads
  - Pausando a Execução
    - O método estático `Thread.yield` faz a thread pausar temporariamente
      - Permite que outras threads possam executar

```
public static void yield()
```



Threads em Java

- Gerenciamento de Threads
  - Aguardando o término
    - O método `join` permite uma thread aguardar o término de um outra thread
      - Invocação faz a thread pausar até o término da outra thread
    - Também é possível especificar um período de tempo para esperar o término da thread
      - Precisão depende dos temporizadores do sistema e do escalonador
    - Pode ser terminado por interrupções
      - Gera a exceção `InterruptedException`

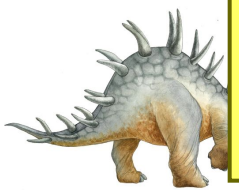


```
public final void join()
public final void join(long millis)
public final void join(long millis, int nanos)
```

Threads em Java

- Aguardando o Término

```
public class JoinThread {
   public static void main(String[] args) {
      Thread thread2 = new Thread(new WaitRunnable());
      Thread thread3 = new Thread(new WaitRunnable());

      System.out.println("Current time millis : " + System.currentTimeMillis());

      thread2.start();

      try {
         thread2.join();
      } catch (InterruptedException e) {
         e.printStackTrace();
      }

      System.out.println("Current time millis : " + System.currentTimeMillis());

      thread3.start();

      try {
         thread3.join(1000);
      } catch (InterruptedException e) {
         e.printStackTrace();
      }
```

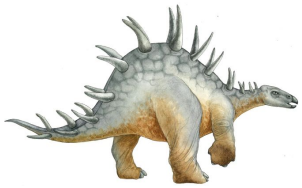


Threads em Java

- Pausando a execução (Rodando o algoritmo)

```
Current time millis : 1411672008793
Current time millis : 1411672013795
Current time millis : 1411672014796
```

  - O primeiro join() aguarda 5 segundos para a outra thread, e quando definimos um tempo limite, esperamos apenas 1 segundo e retornamos para o método join.



Threads em Java

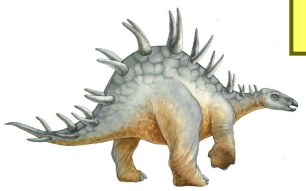
- Gerenciamento de Threads
  - Métodos Depreciados

```
public final void stop()
   Força a thread parar a execução e terminar.
```

```
public void destroy()
   Destrói a thread (nunca implementado).
```

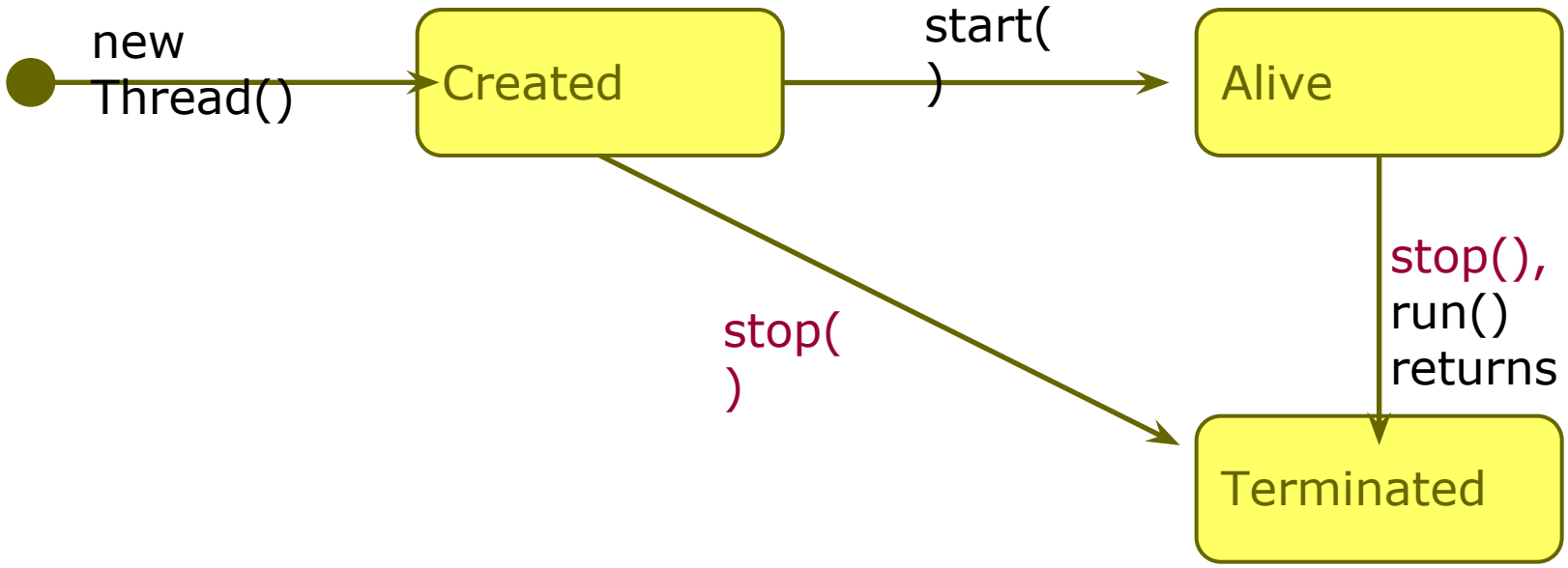
```
public final void suspend()
   Suspende a execução da thread.
```

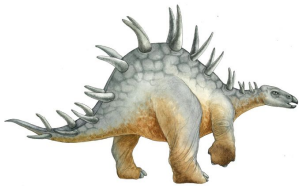
```
public final void resume()
   Continua a execução da thread suspensa.
```



Threads em Java

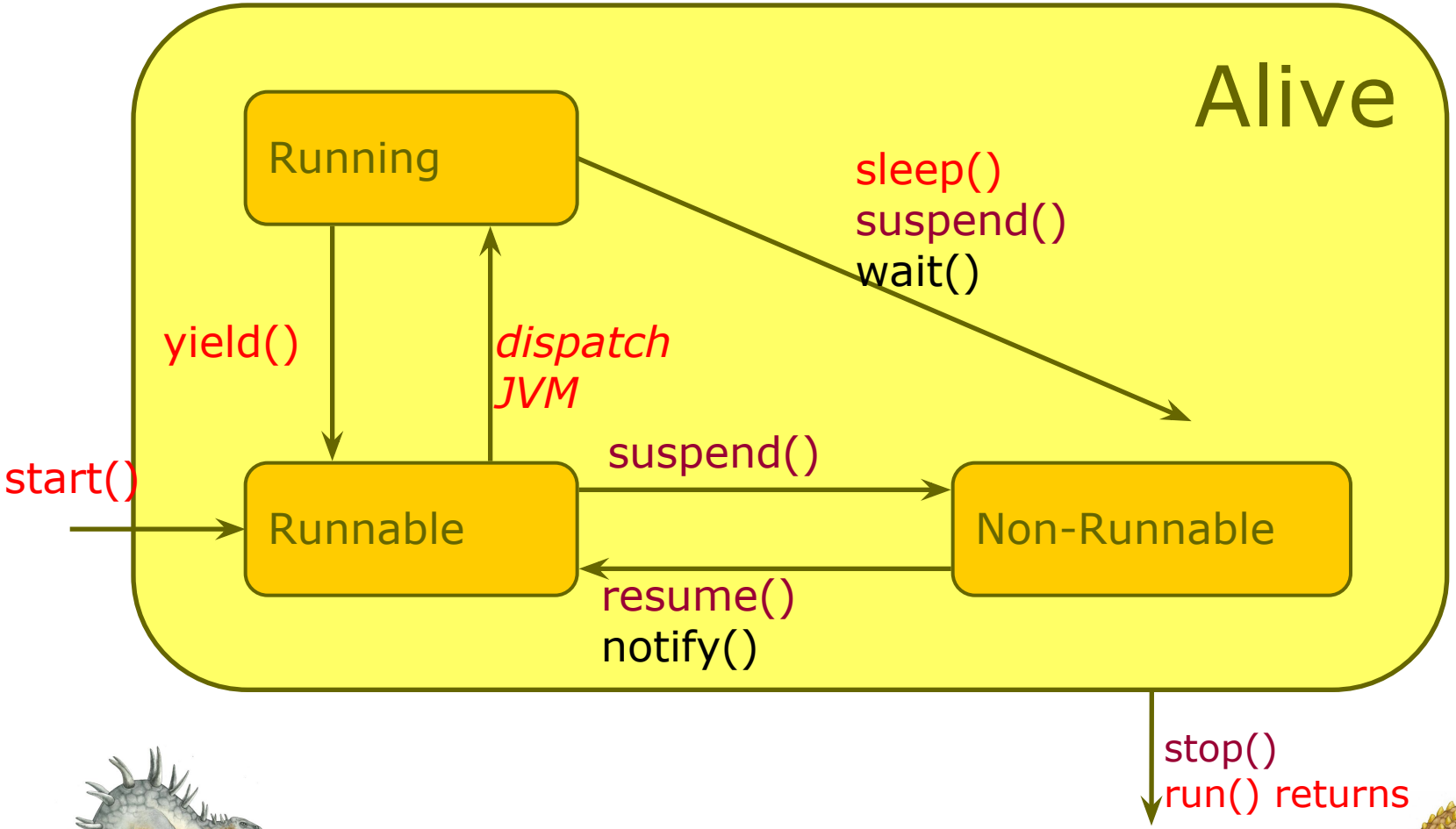
- Ciclo de Vida





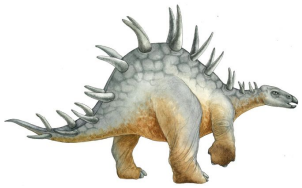
Threads em Java

- Ciclo de Vida



Sincronização

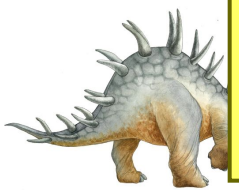
- Cada thread possui uma pilha independente -> duas threads que executam o mesmo método possuem versões diferentes das variáveis locais.
- A memória dinâmica (heap) de um programa é compartilhada por todas as threads -> duas threads poderão portanto acessar os mesmos atributos um objeto de forma concorrente.
- Devido ao mecanismo de preempção, não há como controlar o acesso a recursos compartilhados sem um mecanismo específico de sincronização.



Sincronização

- Exemplo

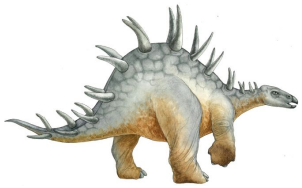
```
public class ContaCorrente
{
    float saldo = 0;
    float tmp;

public float getSaldo()
{
   return saldo;
}

public void depositar(float valor)
{
    tmp = getSaldo();
    saldo = tmp + valor;
}

public void sacar(float valor)
{
    tmp = getSaldo();
    saldo = tmp - valor;
}
}
```



Sincronização

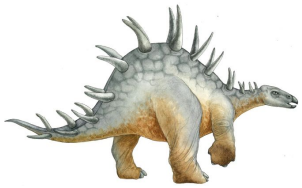
- Considere duas threads acessando uma mesma conta simultaneamente, uma invocando o método depositar e outra o método sacar

Thread 1	Thread 2	saldo
---	---	0
conta.depositar(100);	---	0
tmp = saldo;	---	0
---	conta.sacar(50);	0
---	tmp = conta;	0
--- (0)	saldo = tmp - 50;	-50
saldo = tmp + 100;	---	100



Sincronização

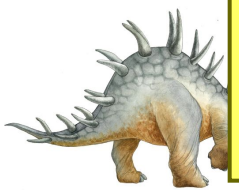
- Java permite restringir o acesso a métodos de um objeto ou trechos de código através do modificador synchronized.
- Cada objeto Java possui um (e apenas um) lock.
- Para invocar um método synchronized é necessário adquirir (implicitamente) o lock associado ao objeto.
- Se dois ou mais métodos de um objeto forem declarados como synchronized, apenas um , apenas um poderá ser acessado de cada vez.



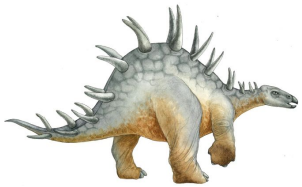
Sincronização

- Exemplo

```
public class ContaCorrente
{
    float saldo = 0;
    float tmp;

public float getSaldo()
{
   return saldo;
}

public void synchronized depositar(float valor)
{
    tmp = getSaldo();
    saldo = tmp + valor;
}

public void synchronized  sacar(float valor)
{
    tmp = getSaldo();
    saldo = tmp - valor;
}
}
```

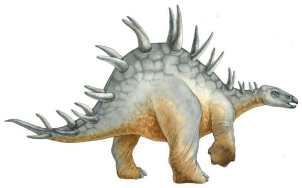


SEMÁFOROS



Semáforos em Java

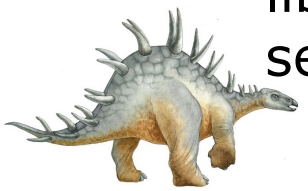
- Implementado pela classe Semaphore,
- Composto basicamente de:
  - variável inteira,
  - lista de espera,
- Restringir o número de *threads* que podem acessar determinado recurso,
- Semáforos binários são usados para o problema de exclusão mútua.



Classe Semaphore
Método Construtor

- `Semaphore(int permits)`
- `Semaphore(int permits, boolean fair)`

- Cria um semáforo com o número de permissões fornecido como parâmetro,
- Se o parâmetro "fair" for:
  - TRUE → a lista de espera é uma FIFO,
  - FALSE (ou não setado) → não é possível garantir a ordem da lista de espera.
- **OBS:** se o número de permissões for negativo as liberações devem ocorrer antes de qualquer permissão ser concedida.

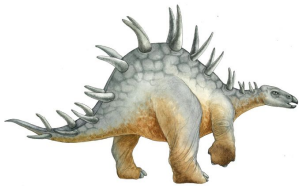


Classe Semaphore
Método acquire()

- `acquire() // Adquire uma permissão`
- `acquire(int permits)`

- Se existir ao menos uma (ou *n*) permissão disponível:
  - é liberada uma (ou *n*) permissão para a thread,
  - e é reduzido o número de permissões disponíveis no semáforo.

- Senão: a thread fica fora do escalonamento (em estado inativo) até que:
  - alguma outra thread realize uma liberação e esta thread seja a próxima a conseguir uma (ou *n*) permissão,
  - alguma outra thread interrompa a thread atual.

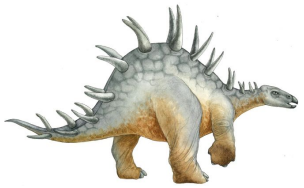


Classe Semaphore
Método release()

- `release() // Libera uma permissão`
- `release(int permits)`

- Libera uma (ou *n*) permissão, a qual retorna para o semáforo,
- Número de permissões do semáforo é incrementado,
- Se algumas threads estavam na lista de espera, uma delas é escolhida para receber a permissão,
- A thread que receber a permissão retorna para o escalonamento.

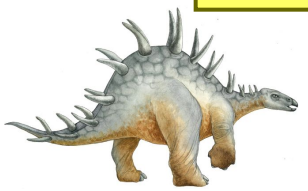
OBS: Não existe nenhuma obrigação de que a thread que realiza a liberação deve ter adquirido a permissão.



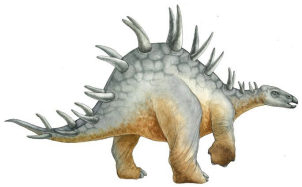
Threads em Java

- Exemplo de Semáforo em Java

```
import java.util.concurrent.Semaphore;

public class ExemploSemaforo
{

   private int value = 0;

   private final Semaphore mutex = new Semaphore(1);

   public int getNextValue() throws InterruptedException {
      try {
         mutex.acquire();
         return value++;
      } finally {
         mutex.release();
      }
   }
}
```

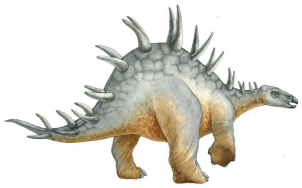


MONITORES



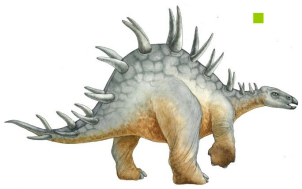
Sincronização

- Monitores
  - Construção que pode ser usada para exclusão mútua e sincronização de processos
  - Proposto por Dijkstra e posteriormente implementado por Hoare e Hansen
  - Explicita e centraliza as sessões críticas em uma parte especial do código
    - Exclusão mútua é automaticamente forçada
    - Facilita o entendimento
  - Não adota primitivas para demarcar regiões críticas
    - Evita o esquecimento do uso das primitivas



Sincronização

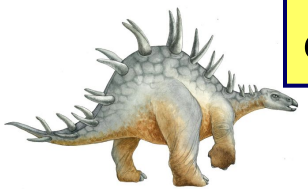
- Monitores
  - Um monitor possui . . .
    - Nome que tem o propósito de identificação
    - Variáveis globais que são compartilhadas entre os processos que usam do monitor
    - Procedimentos de entrada (procedure entries) que podem ser ativados pelos processos
      - Podem possuir variáveis locais e parâmetros
      - Um único processo poder ativar os procedimentos do monitor, a cada instante
      - Impõe a exclusão mútua entre processos
      - Variáveis globais somente podem ser acessadas a partir dos procedimentos
    - Código de inicialização das variáveis globais



Sincronização

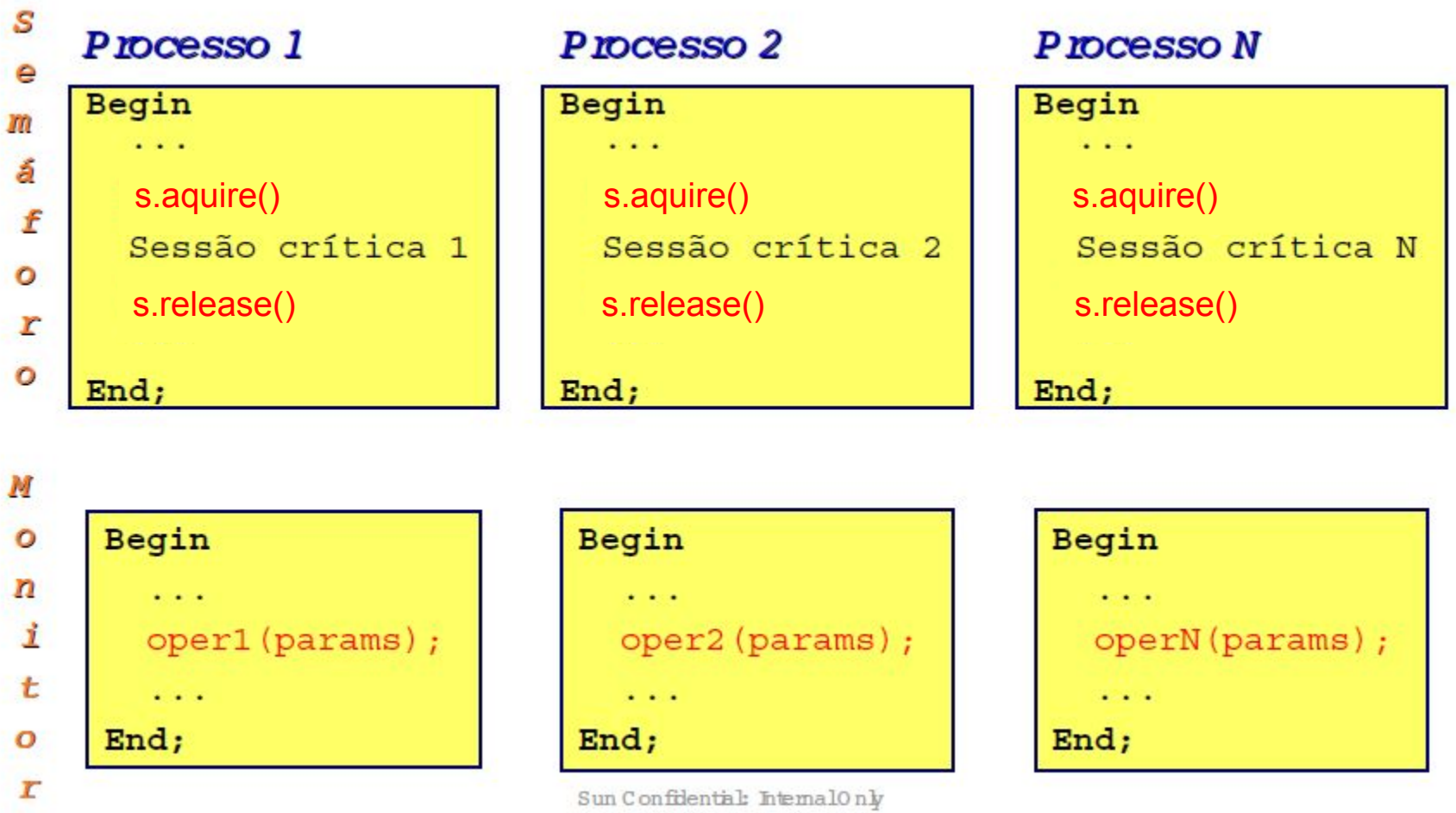
- Monitores

```
monitor: nomemonitor;

declaração de variáveis globais;

procedure operação1(parâmetros);
  declaração de variáveis locais;
  begin
    código que implementa a operação
  end;
...
procedure operaçãoN(parâmetros);
  declaração de variáveis locais;
  begin
    código que implementa a operação
  end;

begin
  código de inicialização das variáveis globais
end
```



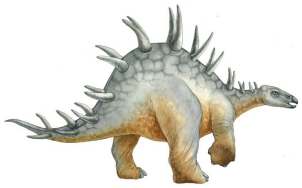
Sincronização

- Monitores x Semáforos



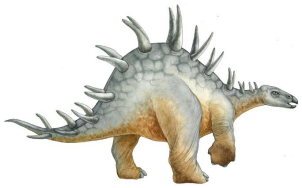
Sincronização

- Monitores
  - Podem ser implementados como uma classe em linguagens orientadas a objetos
    - Nome da classe ou instância do objeto representa o nome do monitor
    - Atributos representam as variáveis globais compartilhadas
    - Métodos representam os procedimentos de entrada
    - Construtor representa o código de inicialização das variáveis compartilhadas



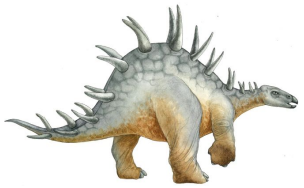
Sincronização

- Monitores em Java
  - Todo objeto Java possui um monitor associado
  - Primitiva `synchronized` permite acessar o monitor de um objeto
    - Primitiva pode ser usada em métodos ou trechos de código (statements)
    - Assegura que, em um dado instante, apenas uma única thread pode executar métodos do objeto
      - Thread possui o bloqueio (lock) do monitor do objeto
      - Thread está dentro do monitor do objeto



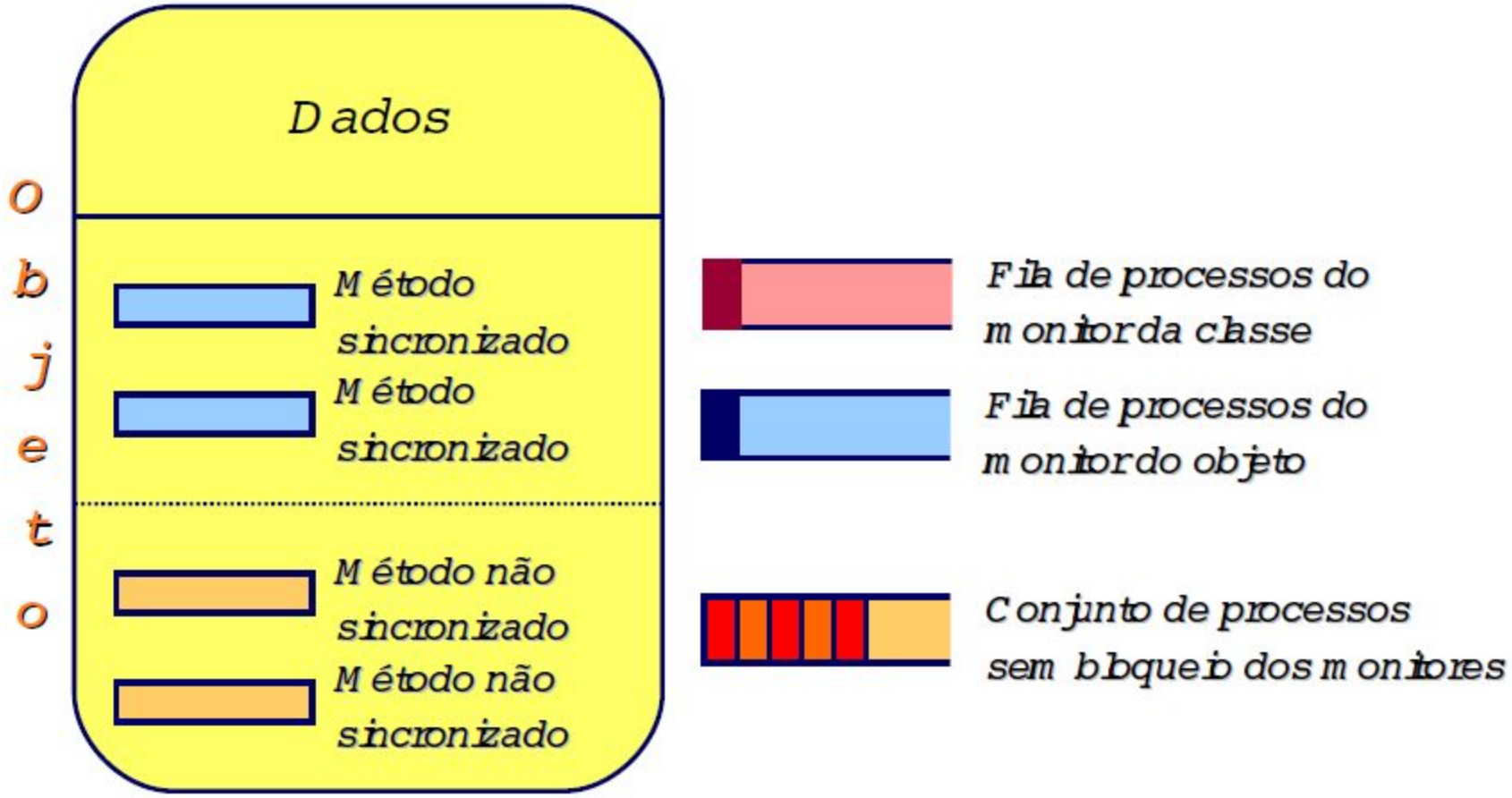
Sincronização

- Monitores em Java
  - Primitiva `synchronized`
    - Métodos Sincronizados
      - Métodos estáticos também podem ser sincronizados
      - Instâncias e classe possuem monitores (locks) independentes
      - Cada monitor (objeto / classe), em um dado instante, permite a execução de uma única thread
    - Trechos de Código Sincronizados
      - Pode incrementar a concorrência de threads
      - Permite execução simultânea de diversos métodos



Sincronização

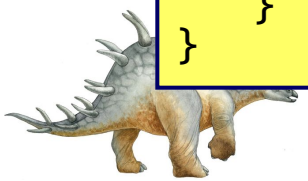
- Monitores em Java



Sincronização

- Monitores em Java
  - Métodos Sincronizados

```
public class SynchClassName {
   private String globalVar;

   public SynchClassName() {
   }

   public synchronized void synchMethod() {
      String localVar;
   }

   public void nonSynchMethod() {
   }

   public static synchronized void synchStaticMethod() {
   }

   public static void nonSynchStaticMethod() {
   }
}
```



Sincronização

- Monitores em Java
  - Métodos Sincronizados

```
public class SynchClass {

   public synchronized void synchMethod(int i) {
      while (true) System.out.println(i);
   }

   public void nonSynchMethod(int i) {
      while (true) System.out.println(i);
   }

   public static synchronized void synchStaticMethod(int i) {
      while (true) System.out.println(i);
   }

   public static void nonSynchStaticMethod(int i) {
      while (true) System.out.println(i);
   }
}
```

Sincronização

- Monitores em Java
  - Métodos Sincronizados

```
public class SynchImpl extends Thread {
  int id;
  SynchClass sc;
  ...
  public void run() {
    switch (id) {
      case 0:
      case 1: sc.synchMethod(id); break;
      case 2:
      case 3: sc.nonSynchMethod(id); break;
      case 4:
      case 5: SynchClass.synchStaticMethod(id); break;
      case 6:
      case 7: SynchClass.nonSynchStaticMethod(id); break;
    }
  }
  ...
}
```

```
public SynchImpl(int id, SynchClass sc) {
  this.id = id;
  this.sc = sc;
}
```

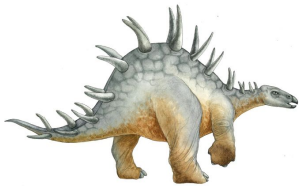
```
public static void main (String[] args) {
  SynchClass sc = new SynchClass();
  for (int i=0; i < 8; i++)
    (new SynchImpl(i, sc)).start();
}
```

Sincronização

- Monitores em Java
  - Trechos de Código Sincronizados

```
public class SynchClassName {
   private String globalVar;
   private Object lock1 = new Object();
   private Object lock2 = new Object();

   public void nonsyncMethod() {
      String localVar;
      ...
      synchronized (this) {
         ...
      }
   ...
}
```

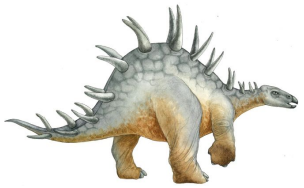
```
   public void nonsyncMethod1() {
      synchronized (lock1) {
         ...
      }
   }
   public void nonsyncMethod2() {
      synchronized (lock2) {
         ...
      }
   }
```



Sincronização

- Monitores em Java
  - Trechos de Código Sincronizados

```
public class SynchClass {
  private Object lock1 = new Object();
  private Object lock2 = new Object();

  public void synchThisMethod(int i) {
     synchronized (this) {
       while (true) System.out.println(i);
     }
  }

  public void synchLock1Method(int i) {
     synchronized (lock1) {
       while (true) System.out.println(i);
     }
  }

  public void synchLock2Method(int i) {
     synchronized (lock2) {
       while (true) System.out.println(i);
     }
  }
}
```

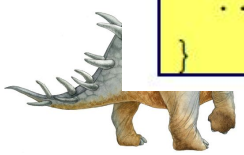


Sincronização

- Monitores em Java
  - Trechos de Código Sincronizados

```
public class SynchImpl extends Thread {
  int id;
  SynchClass sc;

  public SynchImpl(int id, SynchClass sc) {
    this.id = id;
    this.sc = sc;
  }

  public void run() {
    switch (id) {
      case 0:
      case 1: sc.synchThisMethod(id); break;
      case 2:
      case 3: sc.synchLock1Method(id); break;
      case 4:
      case 5: sc.synchLock2Method(id); break;
    }
  }
  ...
}
```

```
public static void main (String[] args) {
  SynchClass sc = new SynchClass();
  for (int i=0; i < 6; i++)
    (new SynchImpl(i, sc)).start();
}
```

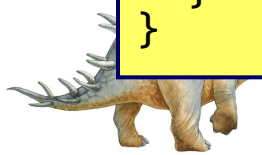


Sincronização

- Monitores em Java
  - Perigo de Deadlocks
    - Deve-se tomar cuidado para evitar deadlock

```
public class BadSynchClass {
  private int value;

  public synchronized int get() {
    return value;
  }

  public synchronized void set(int i) {
    value = i;
  }

  public synchronized void swap(BadSynchClass bsc) {
    int tmp = get();
    set(bsc.get());
    bsc.set(tmp);
  }
}
```

```
public BadSynchClass(int v) {
  value = v;
}
```

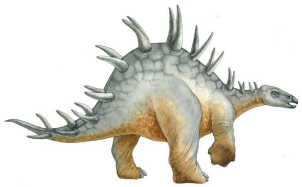


Sincronização

- Monitores em Java
  - Perigo de Deadlocks

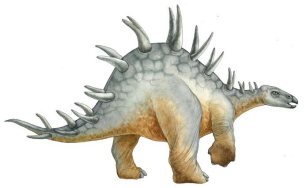
```
public class BadSynchImpl extends Thread {
  BadSynchClass a, b;

  public BadSynchImpl(BadSynchClass a, BadSynchClass b) {
    this.a = a;
    this.b = b;
  }

  public void run() {
    a.swap(b);
    System.out.println("A: " + a.get() + " B: " + b.get());

  }

  public static void main (String[] args) {
    BadSynchClass a = new BadSynchClass(1);
    BadSynchClass b = new BadSynchClass(2);

    (new BadSynchImpl(a, b)).start();
    (new BadSynchImpl(b, a)).start();
  }
}
```

SINALIZAÇÃO ENTRE THREADS



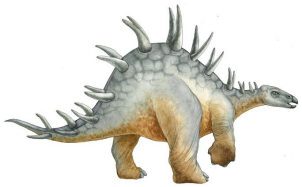
Sincronização

- Sincronização com Monitor
  - Monitor suporta o conceito de sincronização condicional (conditional synchronization)
    - Processos esperam que uma dada condição torne-se verdadeira
  - Implementada com variáveis de condição (condition variables)
    - São usadas para suspender e reativar processos
    - São associadas a condições que causam a suspensão ou reativação de um processo
    - Possuem duas operações (wait / signal)



Sincronização

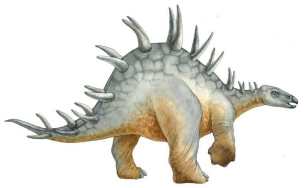
- Sincronização com Monitor
  - x.wait()
    - Suspende processo chamador e libera monitor
    - Monitor insere processo suspenso na fila de espera associada à variável de condição
  - x.signal()
    - Reativa processo que está aguardando na fila associada à variável de condição
    - Processo reativado retoma a execução do ponto logo após a chamada do wait
    - Processo chamador pode ou não liberar monitor (depende da implementação do monitor)



Sincronização

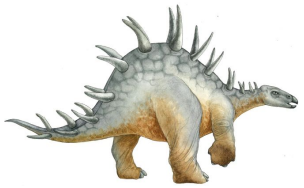
- Sincronização com Monitor
  - Reativação de Processos
    - Monitor Hoare
      - Processo chamador libera o monitor
      - Processo reativado inicia execução
    - Monitor Estilo Java
      - Processo chamador continua a execução
      - Processo reativado apenas aguarda a oportunidade de entrar no monitor

Qual processo deve executar após a chamada da operação signal?



Sincronização

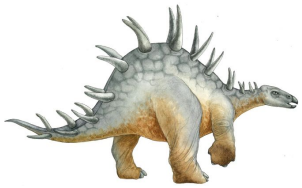
- Monitor Hoare
  - Processo da fila executa após a notificação
  - Permite que o processo notificado inicie sua execução sem intervenção de outros processos
  - Estado no qual o processo inicia a execução é o mesmo de quando a notificação foi realizada
  - Processo pode assumir que a condição é verdadeira após ser reativado
    - Assume que a notificação é ativada apenas quando a condição é verdadeira



```
if (! condition) x.wait();
```

Sincronização

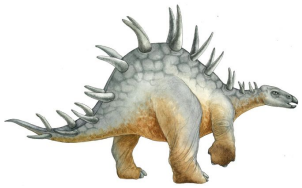
- Monitor Estilo Java
  - Processo notificador continua a execução após realizar a notificação
  - Processo reativado não pode assumir que a condição é verdadeira
    - Notificação é apenas um indicativo de que a condição pode ser verdadeira
    - Processo reativado deve explicitamente reavaliar a condição ao ser reativado
    - Se a condição for falsa, processo deve ser suspenso novamente



```
while (!condition) x.wait();
```

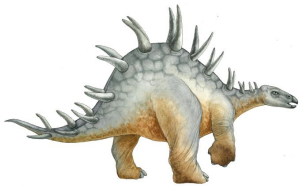
Sincronização

- Monitores Java
  - Java não possui variáveis de condição
  - Todo objeto possui uma única fila de notificação
  - Todo objeto suporta as operações wait / notify / notifyAll
    - Somente podem ser ativadas por thread que possui o monitor do objeto



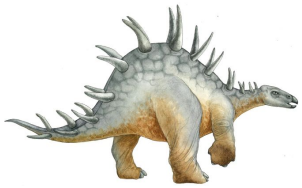
Sincronização

- Monitores Java
  - `wait()`
    - Suspende a thread chamadora e a insere na fila de notificação do objeto
  - `notify()` / `notifyAll()`
    - Reativa uma única thread (`notify`) ou todas as threads (`notifyAll`) da fila de notificação do objeto
    - Thread chamadora continua execução e não libera o monitor
    - Cada thread reativada é colocada na fila do monitor do objeto



Exemplo 1

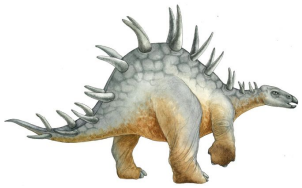
```
public class ThreadA {
       public static void main(String[] args){
        ThreadB b = new ThreadB();
        b.start();
        System.out.println("Total é igual a: " + b.total);
    }
}
public class ThreadB extends Thread {
          int total;
           @Override
           public void run(){
                   for(int i=0; i<200 ; i++){
                       total += i;
                   }
           }
}
```



Solução

```
public class ThreadA {

     public static void main(String[] args){
      ThreadB b = new ThreadB();
      b.start();

      synchronized(b){
          try{
              System.out.println("Aguardando o b completar...");
              b.wait();
          }catch(InterruptedException e){
              e.printStackTrace();
          }

          System.out.println("Total é igual a: " + b.total);
      }
   }
}
```

```
public class ThreadB extends Thread {
        int total;
        @Override
        public void run(){
            synchronized(this){
                for(int i=0; i<200 ; i++){
                    total += i;
                }
                notify();
            }
        }
}
```



Exemplo Monitor Java

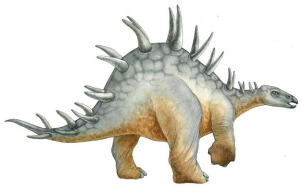
```
public class BoundedBuffer {
   private final String[] buffer;
   private final int capacity;
   private int front, rear, count;
   private final Lock lock = new ReentrantLock();
   private final Condition notFull = lock.newCondition();
   private final Condition notEmpty = lock.newCondition();
...
   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();
      }
   }
...
}
```

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

      this.capacity = capacity;

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

```
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();
      }
   }
```

Algumas explicações do exemplo

- Os dois métodos são protegidos com o bloqueio para garantir a exclusão mútua.
- Em seguida, usamos duas variáveis condicionais. Uma para esperar o buffer não estar vazio e uma outro para aguardar o buffer não estar cheio.
- Você pode ver que a operação de espera está em loop. Isso é para evitar ladrões de signal, problema que pode ocorre quando usando signal & continue
- O BoundedBuffer pode ser facilmente utilizado com várias threads sem problemas.



Fim do Laboratorio 6