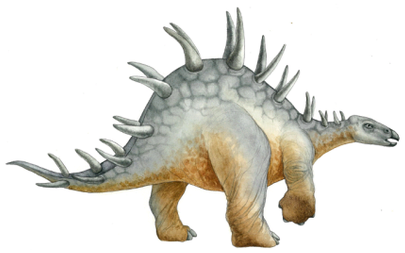
Capítulo 6: Sincronización de procesos

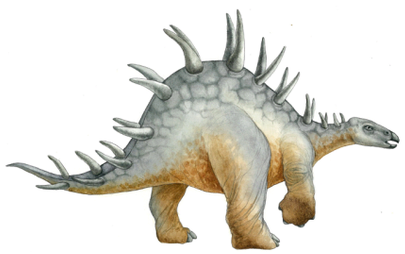
Capítulo 6: Sincronización de procesos

- Antecedentes
- El problema de la sección crítica
- Solución de Peterson
- Hardware de sincronización
- Semáforos
- Problemas clásicos de sincronización
- Monitores
- Ejemplos de sincronización
- Transacciones atómicas



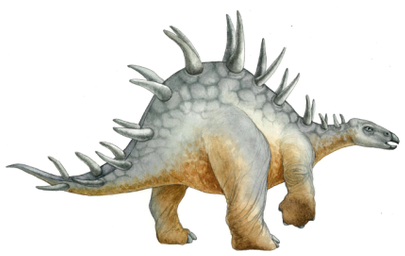
Antecedentes

- Acceso concurrente a datos compartidos puede resultar en inconsistencia de datos
- Consistencia de datos requiere de mecanismos para asegurar la ejecución ordenada de procesos cooperativos
- Buscamos soluciones para el problema del productor-consumidor. Utilizamos contadores.



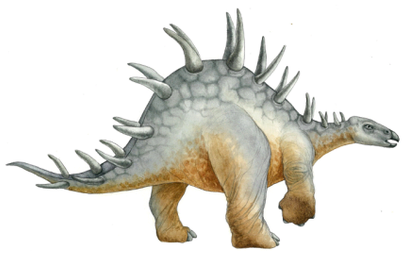
Productor

```
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;
```



Condición de concurso

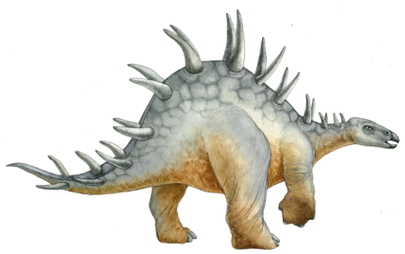
- count++ puede implementarse como

  register1 = count
  register1 = register1 + 1
  count = register1

- count-- puede implementarse como

  register2 = count
  register2 = register2 - 1
  count = register2

- Considera la ejecución alternada, iniciando con “count = 5”:
  - S0: producer execute register1 = count {register1 = 5}
    S1: producer execute register1 = register1 + 1 {register1 = 6}
    S2: consumer execute register2 = count {register2 = 5}
    S3: consumer execute register2 = register2 - 1 {register2 = 4}
    S4: producer execute count = register1 {count = 6 }
    S5: consumer execute count = register2 {count = 4}

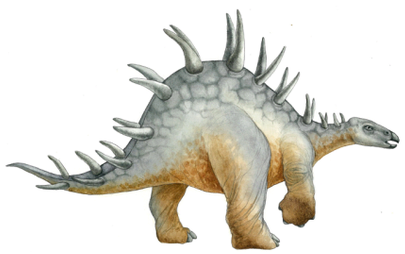


Solución problema de la sección crítica

- Exclusión Mutua - Si el proceso Pi está ejecutando en su sección crítica, ningún otro proceso puede entrar a su sección crítica
- Progreso - Si ningún proceso está ejecutando en su sección crítica y existen procesos que desean entrar a su sección crítica, la selección de los procesos que entrarán a continuación en su sección crítica no puede posponerse indefinidamente
- Espera acotada - Limitar número de veces que otros procesos son permitidos en sus regiones críticas, antes de que otro proceso pueda entrar.

Problema de la sección crítica

- Condición de concurso - Cuando existe acceso concurrente a datos compartidos y el resultado final depende del orden de ejecución.
- Sección crítica - Segmento del código donde se accede a los datos compartidos.
- Sección de entrada - Código que solicita permiso para acceder a la sección crítica.
- Sección de salida - Código que se ejecuta después de salir de la sección crítica



Estructura de un proceso típico

```
while (true) {
```

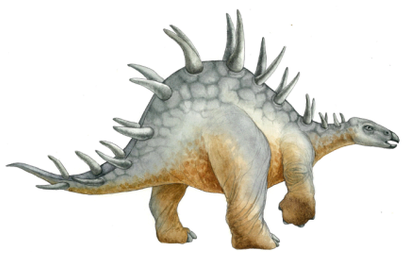
entry section

critical section

exit section

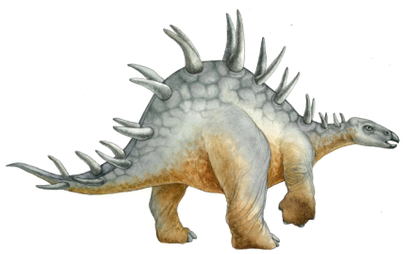
remainder section

```
}
```



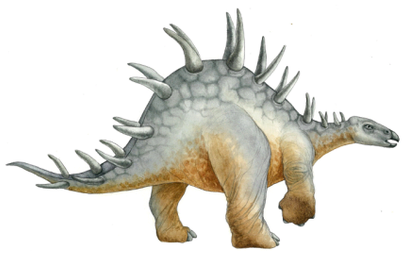
Solución de Peterson

- Solución para dos procesos
- Asume que las instrucciones LOAD y STORE son atómicas; i.e. que no pueden interrumpirse.
- Los dos procesos comparten dos variables:
  - int turn;
  - Boolean flag[2]
- La variable turn indica a quién le toca entrar en su sección crítica.
- El arreglo flag se utiliza para indicar si un proceso está listo para entrar a su sección crítica. flag[i] = true ¡implica que el proceso Pi está listo!



Algoritmo para el proceso Pi

```
while (true) {

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

                critical section

        flag[i] = FALSE;

                remainder section

}
```



Sección crítica utilizando candados

```
while (true) {
```

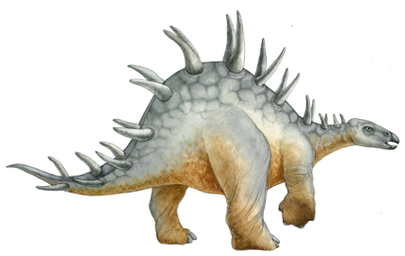
acquire lock

critical section

release lock

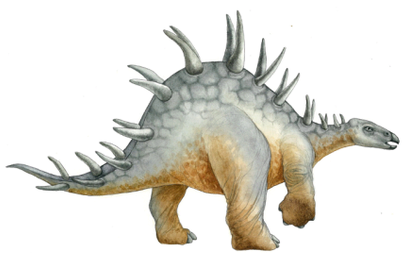
remainder section

```
}
```



Hardware de sincronización

- Soporte de hardware para código de sección crítica
- Uniprocesadores – des-habilitar interrupciones
  - El código en ejecución actual no puede ser sacado del procesador (no-preemption)
  - Generalmente muy ineficiente en sistemas multiprocesador
    - No escalan bien los SO que hacen esto
- Máquinas modernas ofrecen instrucciones atómicas
    - Atómica = no se puede interrumpir
  - Ya sea probar palabra en memoria y asignar valor
  - O intercambiar contenido de dos palabras de memoria



Estructura de datos para solución de hardware

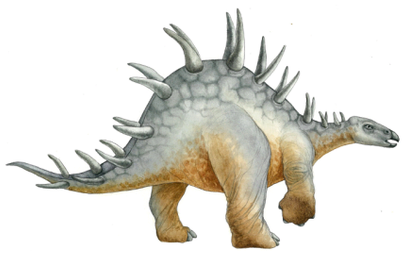
```
public class HardwareData
{
    private boolean value = false;

    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);
    }
}
```

Solución utilizando 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();
}
```

Solución utilizando Swap

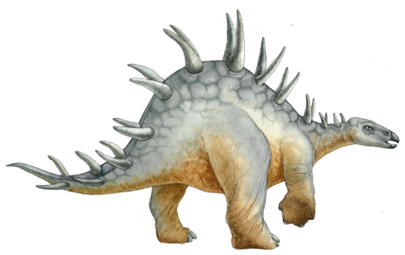
```
// lock is shared by all threads
HardwareData lock = new HardwareData(false);

// 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();
}
```



Semáforo

- Herramienta de sincronización que no requiere espera ocupada (busy waiting)
- Semáforo S – variable entera
- Dos operaciones estándar modifican S: acquire() y release()
  - Originalmente llamadas P() y V()
- Menos complicado
- Sólo puede accederse a través de dos operaciones atómicas

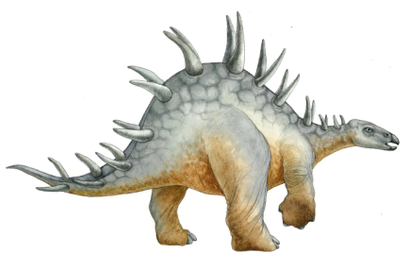
```
acquire() {
    while value <= 0
       ; // no-op
    value--;
}

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



Semáforo como herramienta de sincronización

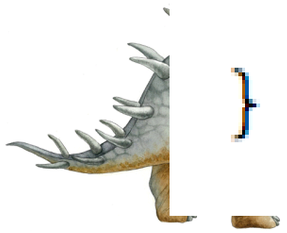
- Semáforo de cuenta – valor entero con rango sobre un dominio no restringido
- Semáforo binario – valor entero con rango entre 0 y 1; más sencillo de implementar
  - También conocido como candado mutex

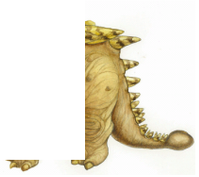
```
Semaphore S = new Semaphore();

S.acquire();

   // critical section

S.release();

   // remainder section
```



Ejemplo Java utilizando semáforos

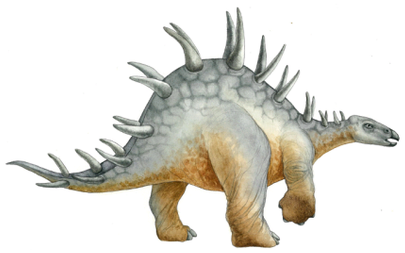
```
public class Worker implements Runnable
{
    private Semaphore sem;
    private String name;

    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);
       }
    }
}
```





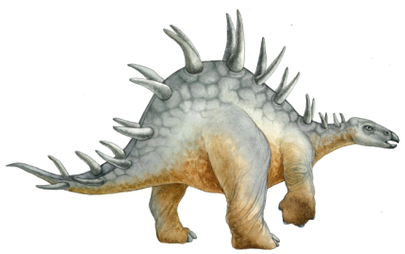
Ejemplo Java utilizando semáforos

```
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();
    }
}
```



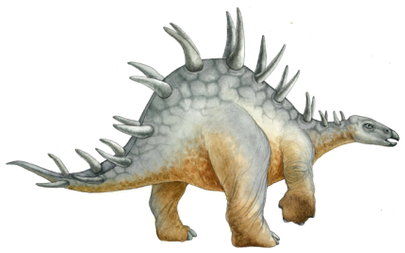
Implementación de semáforos

- Garantizar que nunca dos procesos cualesquiera pueden ejecutar acquire () y release () en el mismo semáforo al mismo tiempo
- Su implementación se convierte en el problema de sección crítica, donde colocamos el código para *wait* y *signal*.
  - Ahora podemos tener espera ocupada en la implementación de la sección crítica
    - Afortunadamente el código es corto
    - Y la espera ocupada es pequeña
- Algunas aplicaciones pasan mucho tiempo en sus secciones críticas y por lo tanto esta NO es una buena solución.



Implementación de semáforos sin espera ocupada

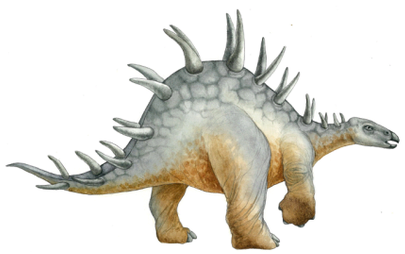
- Asociamos a cada semáforo una cola de espera. Cada entrada en una cola de espera tiene dos datos:
  - valor (de tipo entero)
  - apuntador a la siguiente entrada en la lista
- Dos operaciones:
  - block – coloca al proceso que invoca la operación en la cola de espera apropiada.
  - wakeup – quita uno de los procesos de la cola de espera y lo pasa a la cola de listos.



Implementación de semáforos sin espera ocupada

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

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

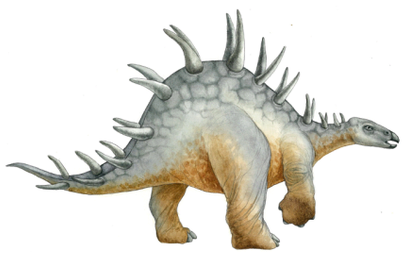


Abrazo mortal y hambruna

- Abrazo mortal (Deadlock) – dos o más procesos esperan indefinidamente por un evento que sólo puede generar uno de los procesos en espera
- Sean S y Q dos semáforos iniciados a 1

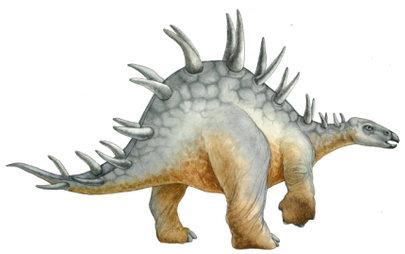
P0	P1
S.acquire();	Q.acquire();
Q.acquire();	S.acquire();
.	.
.	.
.	.
S.release();	Q.release();
Q.release();	S.release();

- Hambruna (Starvation) – bloqueo indefinido. Un proceso nunca puede eliminarse de la cola de un semáforo donde está suspendido.



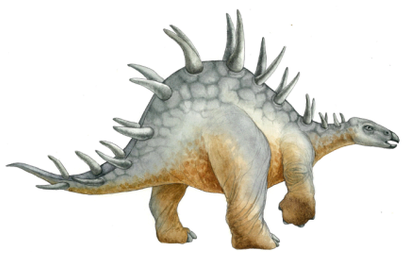
Problemas clásicos de sincronización

- Problema del buffer acotado
- Problema de lectores y escritores
- Problema de los filósofos comensales



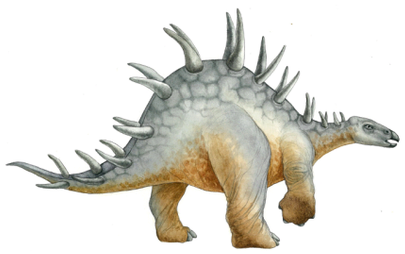
Problema de buffer acotado

- N buffers, cada uno puede contener un elemento
- Semáforo mutex iniciado en 1
- Semáforo full iniciado en 0
- Semáforo empty iniciado al valor N



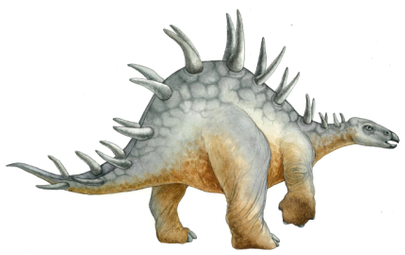
Problema de buffer acotado

```
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
    }
}
```



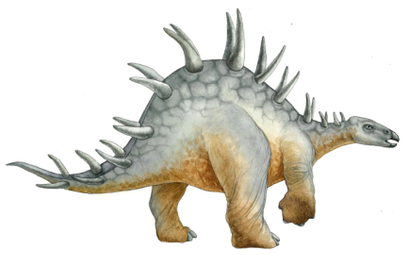
Problema de buffer acotado: insert

```
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 acotado: 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;
}
```



Problema de buffer acotado

- La estructura del proceso productor

```
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 acotado

- Estructura del proceso 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 acotado

- La fabrica

```
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 lectores y escritores

- Un conjunto de datos está compartido entre un número de procesos concurrentes
  - Lectores – solamente leen el conjunto de datos; ellos no realizan actualizaciones
  - Escritores – pueden leer y escribir.

- Problema – Múltiples lectores lean simultáneamente. Sólo un escritor puede acceder a los datos compartidos al mismo tiempo.

- Datos compartidos
  - Conjunto de datos
  - Semáforo mutex iniciado en 1
  - Semáforo db iniciado en 1
  - Entero readerCount iniciado en 0

Problema de lectores y escritores

Interfaz para candados de lectura-escritura

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

Problema de lectores y escritores

Métodos llamados por escritores

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

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

Problema de lectores y escritores

- La estructura del proceso escritor

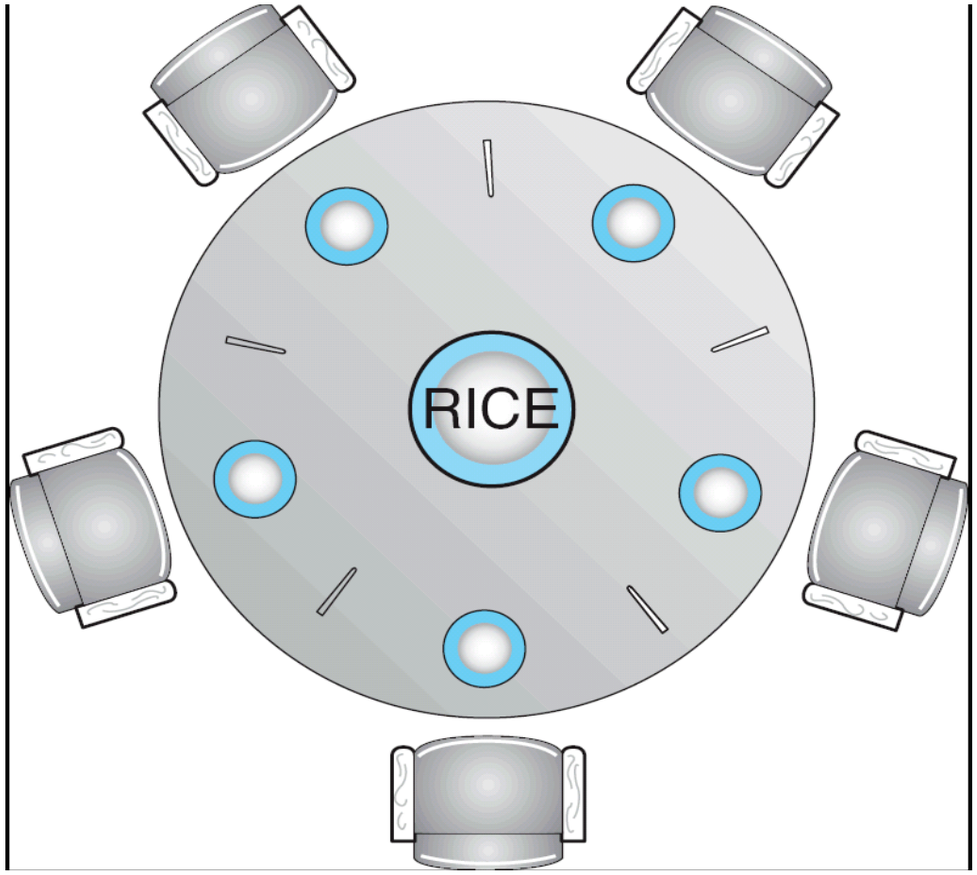
```
public class Writer implements Runnable
{
    private RWLock db;

    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 de lectores y escritores

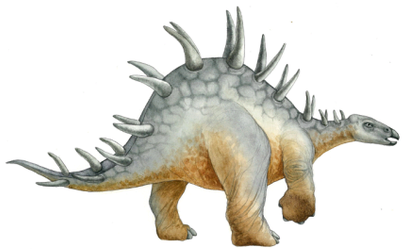
- La estructura del proceso lector

```
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 de los filósofos comensales



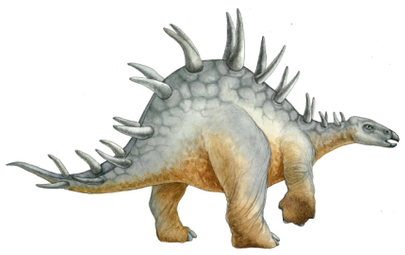
- Datos compartidos
  - Cacerola de arroz (conjunto de datos)
  - Semáforo chopStick [5] iniciado en 1



Problema de los filósofos comensales

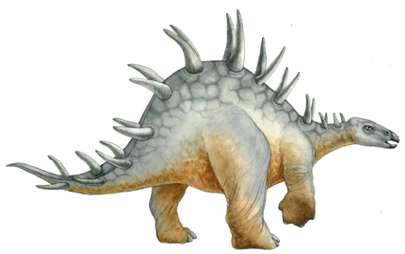
- La estructura del filósofo *i*:

```
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();
}
```



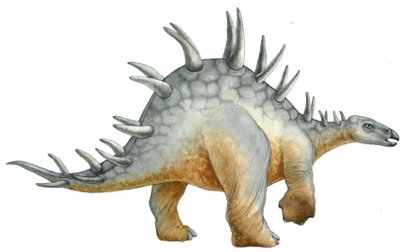
Problemas con los semáforos

- Uso correcto de las operaciones de semáforos:
  - mutex.acquire() …. mutex.release()
  - mutex.wait() … mutex.wait()
  - Omitir mutex.wait () o mutex.release() (o ambos)



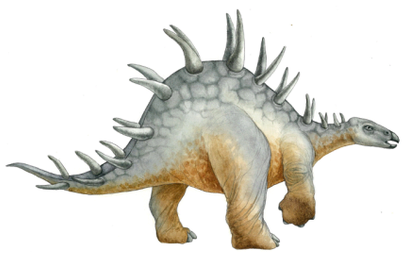
Monitores

- Una abstracción de alto nivel que provee un mecanismo conveniente y efectivo para sincronización de procesos
- Solamente un proceso puede estar activo dentro de un monitor en un momento dado

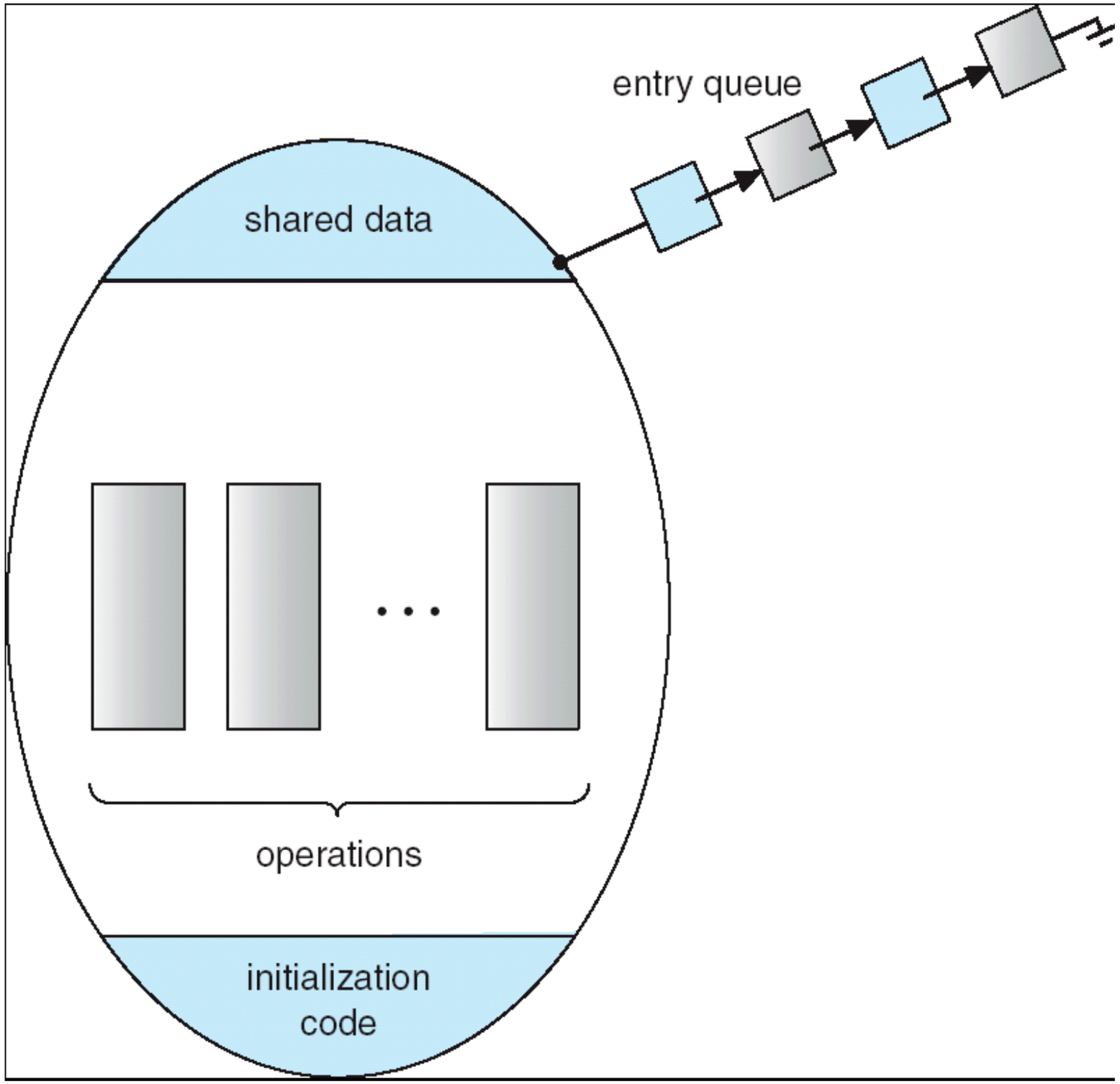


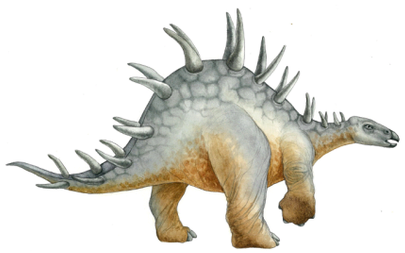
Sintaxis de un monitor

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

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

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

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



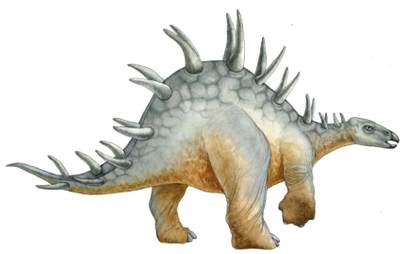
Vista esquemática de un monitor



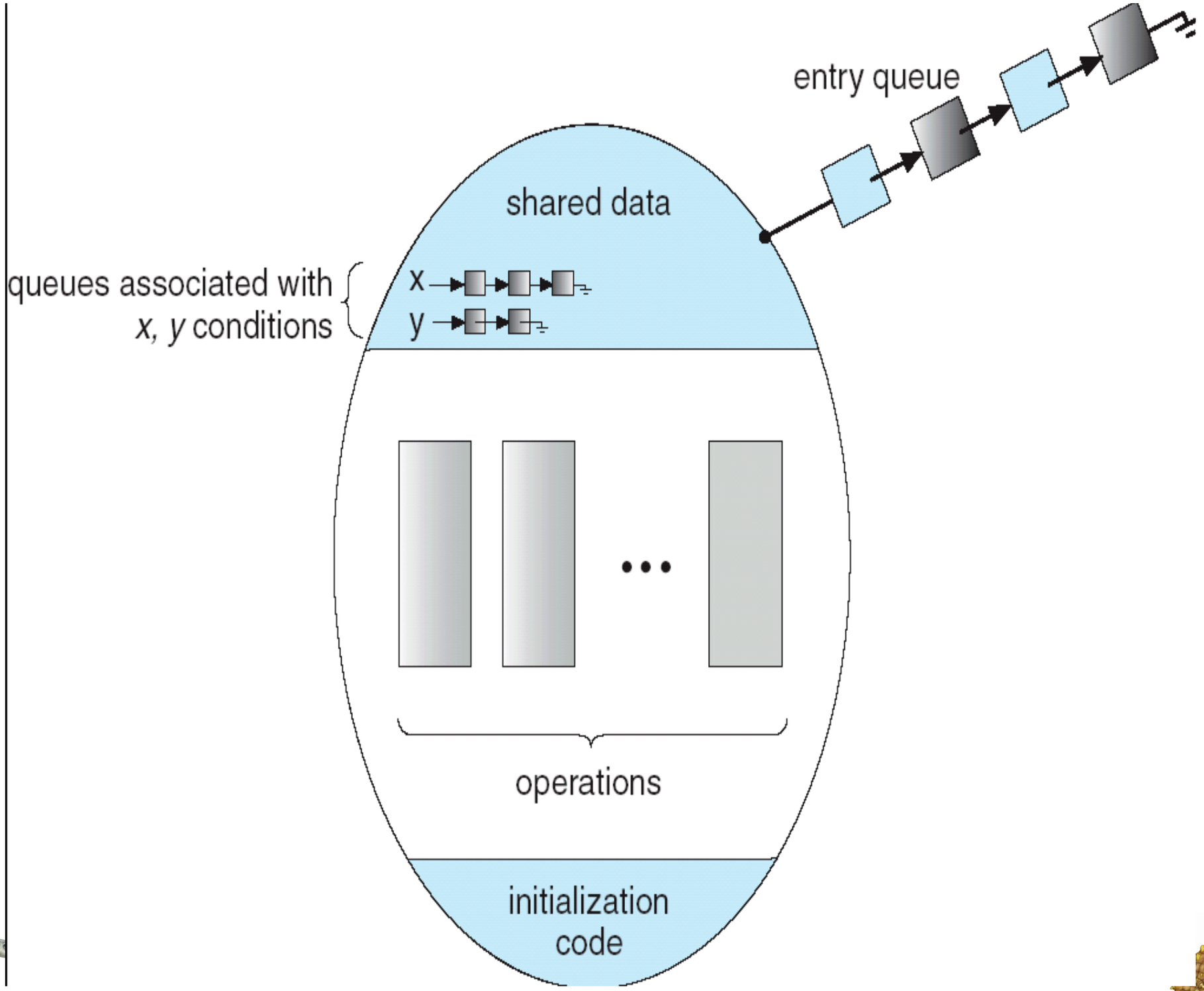


Variables de condición

- Condition x, y;
- Dos operaciones en una variable de condición:
  - x.wait () – el proceso que invoca esta operación es suspendido.
  - x.signal () – re-inicia uno de los procesos (si existe) que invocó x.wait ()



Monitor con variables de condición



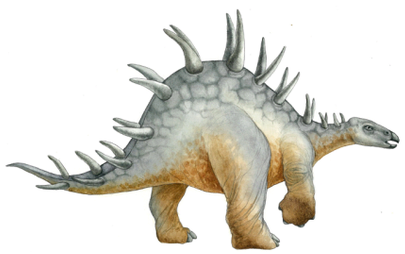
Solución a los filósofos comensales

```
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;
       }
    }
}
```

Solución a los filósofos comensales

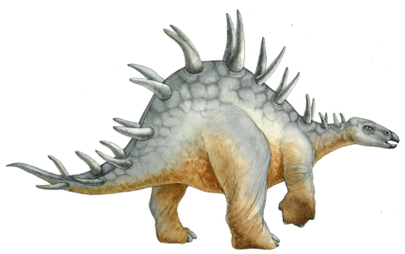
- Cada filósofo invoca las operaciones takeForks(i) y returnForks(i) en esta secuencia:

```
dp.takeForks(i)

COMER

dp.returnForks(i)
```



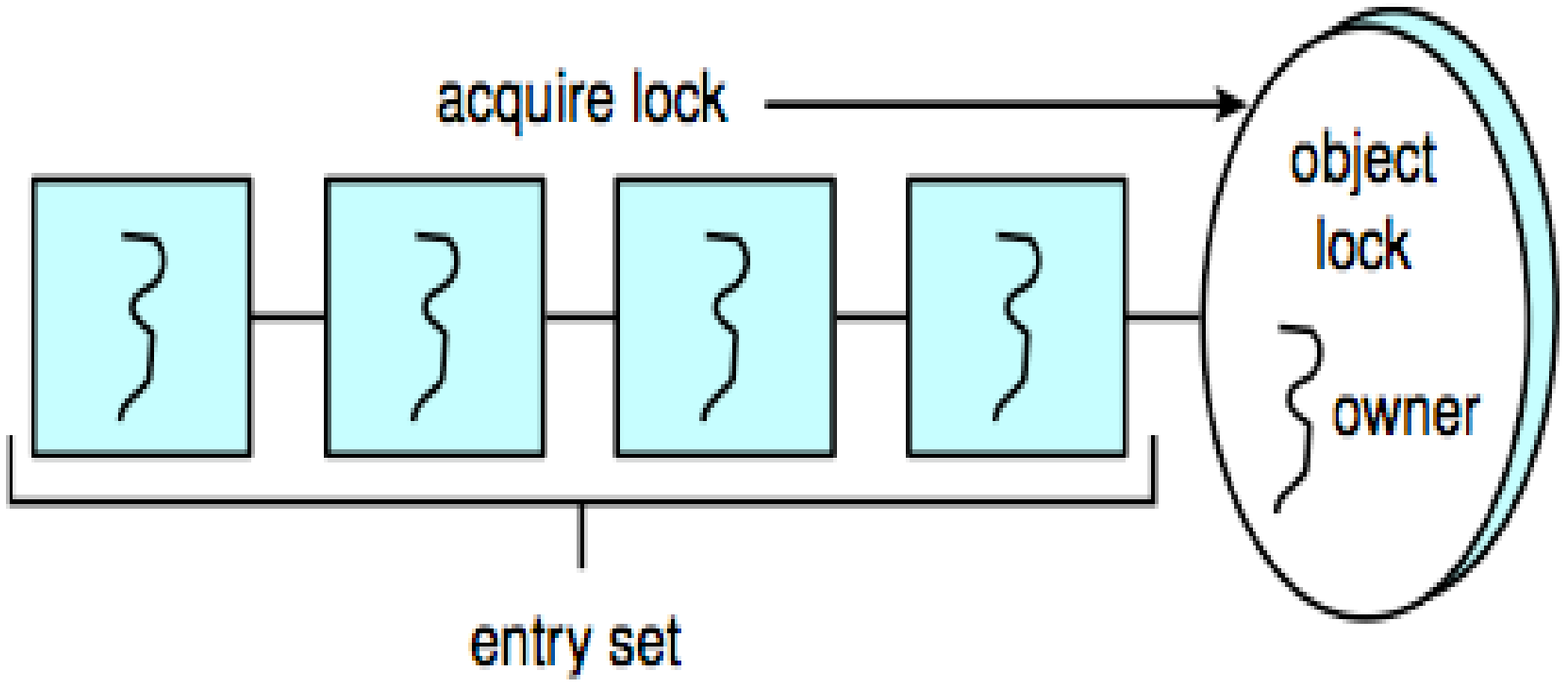
Sincronización en Java

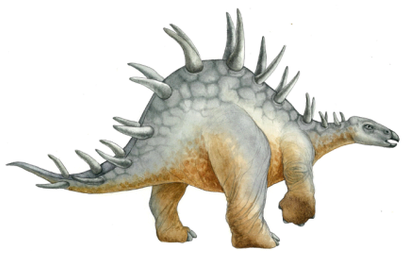
- Java provee sincronización a nivel del lenguaje.
- Cada objeto Java tiene asociado un candado.
- Este candado se adquiere utilizando el método synchronized.
- Este candado es liberado cuando sales del método synchronized.
- Los hilos esperando adquirir el candado del objeto son puestos en el conjunto de entrada del candado.



Sincronización en Java

Cada objeto tiene asociado un conjunto de entrada

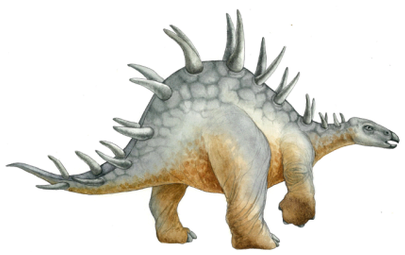




Sincronización en Java

Métodos sincronizados insert() y remove()

```
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;
}
```



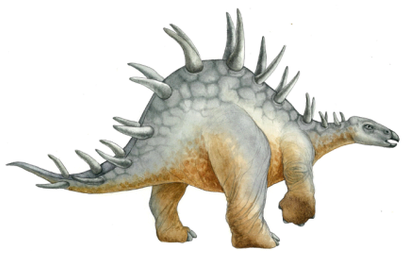
Sincronización Java wait/notify()

- Cuando un hilo ejecuta wait():

  1. El hilo libera el candado del objeto;
  2. El estado del hilo cambia a bloqueado;
  3. El hilo es puesto en el conjunto de espera del objeto.

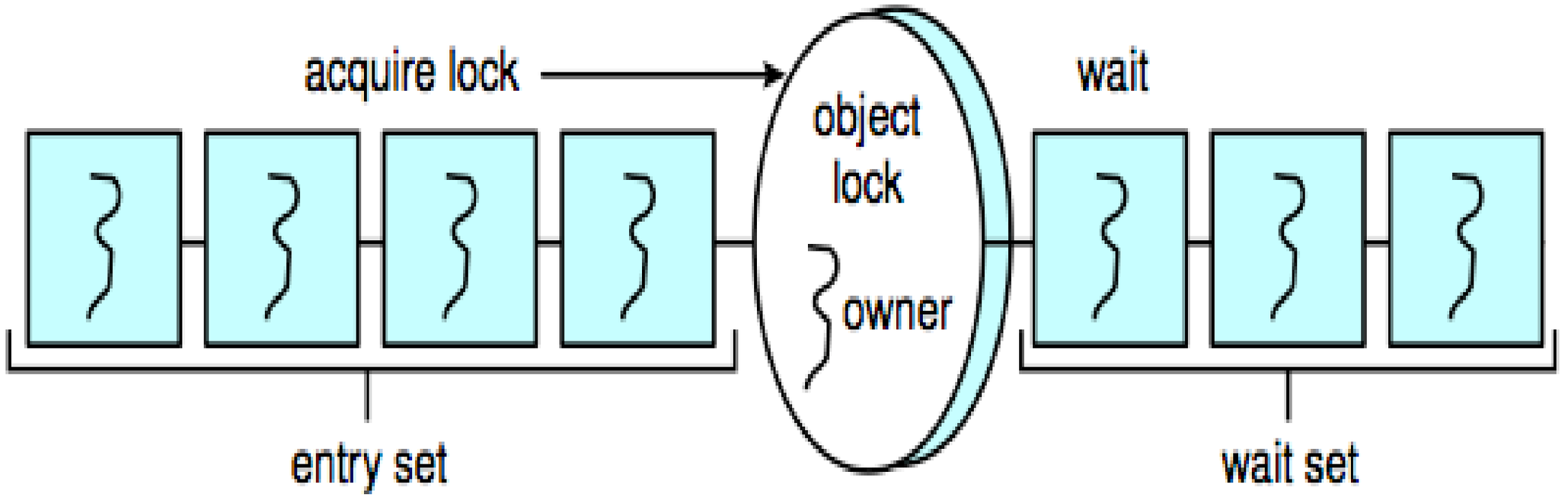
- Cuando un hilo invoca notify():

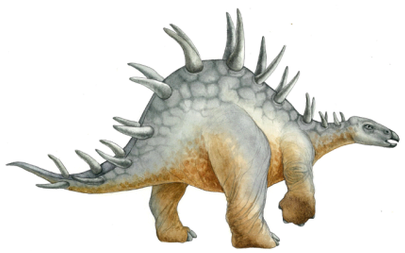
  1. Un hilo arbitrario T del conjunto de espera es seleccionado;
  2. T se mueve del conjunto de espera al de entrada;
  3. El estado de T cambia a Runnable.



Sincronización Java

Conjuntos de entrada y espera





Sincronización Java - Buffer acotado

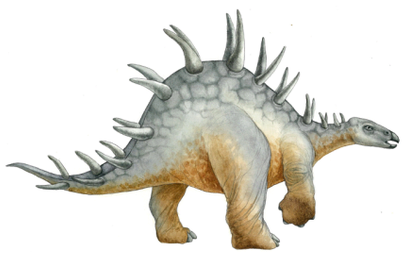
```
public class BoundedBuffer implements Buffer
{
    private static final int BUFFER_SIZE = 5;

    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
    }
}
```

Sincronización Java - Bufer acotado

```
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;
}
```

Sincronización Java

- La llamada a notify() selecciona un hilo arbitrario del conjunto de espera. Es posible que el hilo seleccionado no espera en la condición por la cual fue notificado.
- La llamada notifyAll() selecciona todos los hilos en el conjunto de espera y los mueve al de entrada.
- En general, notifyAll() es una estrategia más conservadora que notify().



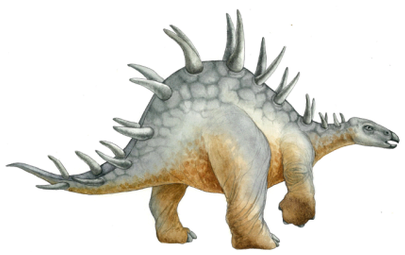
Sincronización Java Lectores-Escritores

```
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
    }
}
```

Sincronización Java Lectores-Escritores

Métodos llamados por lectores

```
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();
}
```



Sincronización Java Lectores-Escritores

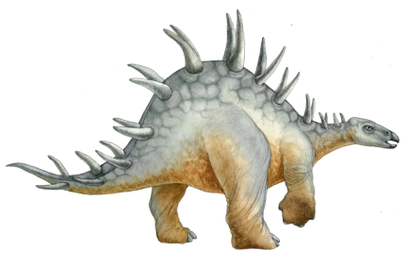
Métodos llamados por los escritores

```
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();
}
```

Sincronización Java

En lugar de sincronizar un método completo,
bloques de código se declaran sincronizados

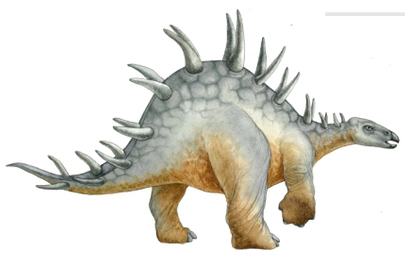
```
Object mutexLock = new Object();
. . .
public void someMethod() {
    nonCriticalSection();

    synchronized(mutexLock) {
       criticalSection();
    }

    remainderSection();
}
```



Sincronización Java

Sincronización de bloque con wait()/notify()

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

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



Características de concurrencia Java5

Semáforos

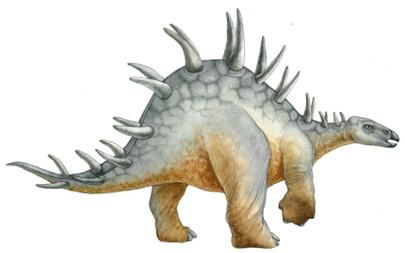
```
Semaphore sem = new Semaphore(1);

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

Características de concurrencia Java5

Una variable de condición se crea primero con ReentrantLock y después invocando su método newCondition()

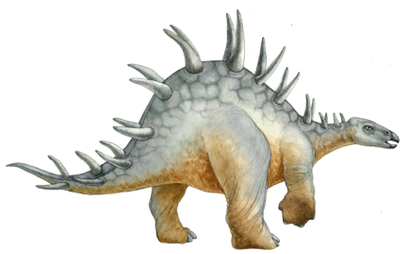
```
Lock key = new ReentrantLock();
Condition condVar = key.newCondition();
```

Una vez hecho esto, es posible invocar los métodos await() y signal().



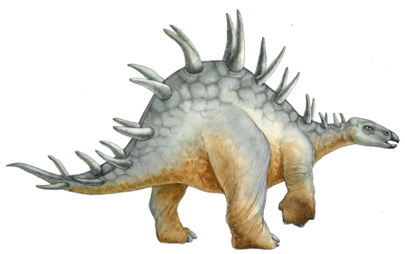
Ejemplos de sincronización

- Solaris
- Windows XP
- Linux
- Pthreads



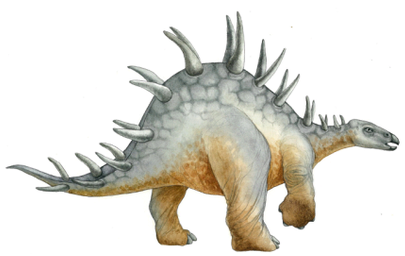
Sincronización en Solaris

- Implementa una variedad de candados para soportar multi-tarea, multi-hilos (incluyendo hilos de tiempo real) y multi-procesamiento
- Utiliza mutexes adaptables por razones de eficiencia para segmentos pequeños de código
- Utiliza variables de condición y candados de lectores-escritores cuando secciones mayores de código requieren acceso a datos
- Utiliza turnstiles para ordenar la lista de hilos esperando para tomar un mutex adaptable o un candado de lector-escritor



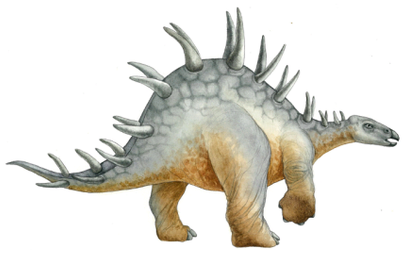
Sincronización en Windows XP

- Utiliza máscaras de interrupción para proteger el acceso a recursos globales en sistemas uni-procesador
- Utiliza spinlocks en sistemas multi-procesador
- También provee objetos despachadores que pueden actuar como mutexes y semáforos
- Los objetos despachadores también proveen eventos
  - Un evento actúa como una variable de condición



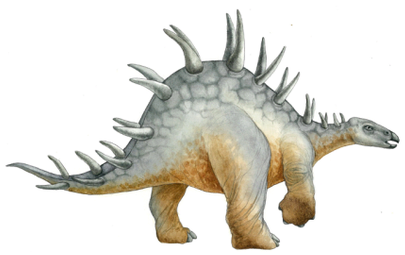
Sincronización Linux

- Linux:
  - Des-habilita las interrupciones para implementar secciones críticas cortas
- Linux provee:
  - semáforos
  - spinlocks



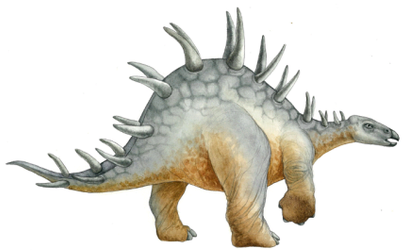
Sincronización en Pthreads

- El API de Pthreads es independiete del SO
- Provee:
  - candados mutex
  - variables de condición
- Extensiones no-portátiles incluyen:
  - candados read-write
  - spinlocks



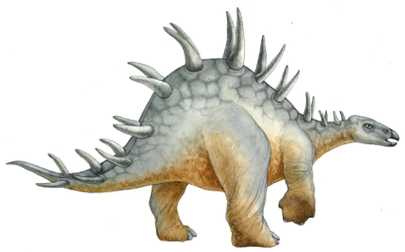
Transacciones atómicas

- Modelo del sistema
- Recuperación basada en bitácoras (logs)
- Puntos de control (Checkpoints)
- Transacciones atómicas concurrentes



Modelo del sistema

- Asegura que las operaciones ocurren como una única unidad de trabajo, todas o ninguna
- Relacionado con el campo de bases de datos
- El reto es asegurar atomicidad a pesar de fallas del sistema de cómputo
- Transacción - colección de instrucciones u operaciones que realizan una única función lógica
  - Nos interesan los cambios en almacenamiento estable – disco
  - Transacción es una serie de operaciones de lectura y escritura
  - Termina con la operación commit (transacción exitosa) o abort (transacción fallida)
  - Transacción abortada debe rebobinar y deshacer los cambios a su estado original



Tipos de medio de almacenamiento

- Almacenamiento volátil – la información almacenada aquí no sobrevive caídas del sistema
  - Ejemplo: memoria principal, cache
- Almacenamiento no-volátil – la información usualmente sobrevive caídas
  - Ejemplo: disco y cinta
- Almacenamiento estable – No se pierde información
  - NO es posible en realidad, aproximación vía replicación o dispositivos RAID con modos independientes de falla

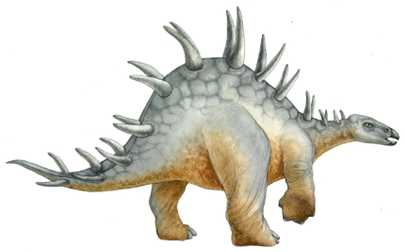
Meta es asegurar la atomicidad de la transacción donde las fallas ocasionan la pérdida de información en almacenamiento volátil

Recuperación basada en bitácoras

- Registro en almacenamiento estable acerca de todas las modificaciones de una transacción
- Más común es bitácora write-ahead
  - Bitácora en almacenamiento estable, cada entrada describe una operación de escritura, incluyendo:
    - Nombre de transacción
    - Nombre de un dato
    - Valor anterior
    - Nuevo valor
  - Se escribe <Ti starts> cuando la transacción Ti inicia
  - Se escribe <Ti commits> cuando Ti commits
  - La entrada en bitácora debe llegar al almacenamiento estable antes de realizar la operación

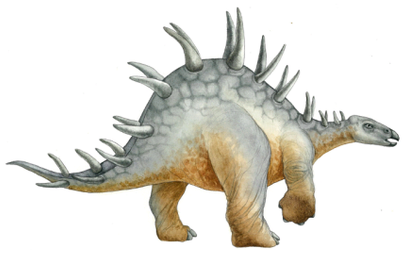
Algoritmo recuperación basado bitácora

- Utilizando la bitácora, el sistema puede manejar cualquier error de memoria volátil
  - Undo(Ti) regresa valores de datos actualizados por Ti
  - Redo(Ti) asigna los valores de todos los datos en la transacción Ti a nuevos valores
- Undo(Ti) y redo(Ti) deben ser idempotentes
  - Múltiples ejecuciones deben tener el mismo resultado como una sóla ejecución
- Si el sistema falla, regresa a su estado original todos los datos a través de la bitácora
  - Si bitácora contiene <Ti starts> sin <Ti commits>, undo(Ti)
  - Si bitácora contiene <Ti starts> y <Ti commits>, redo(Ti)



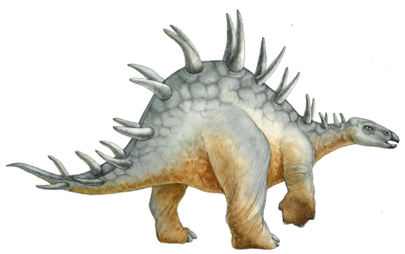
Puntos de control (Checkpoints)

- Bitácora puede ser larga y la recuperación llevar mucho tiempo
- Puntos de control acortan la bitácora y tiempo de recuperación.
- Esquema de puntos de control:
  - Enviar todas las entradas de bitácora en memoria volátil a almacenamiento estable
  - Enviar todos los datos modificados de volátil a estable
  - Enviar una entrada <checkpoint> a la bitácora en almacenamiento estable
- Ahora recuperación solo incluye Ti, tal que Ti inició su ejecución antes del punto de control más reciente y todas las transacciones después de Ti.
- Todas las demás transacciones ya están en almacenamiento estable



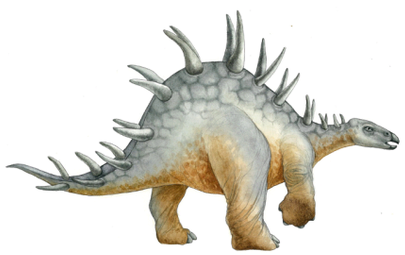
Transacciones concurrentes

- Debe ser equivalente a ejecución en serie - serialización
- Pueda realizar todas las operaciones en la sección crítica
  - Ineficiente, demasiado restrictiva
- Algoritmos de control de concurrencia proveen serialización



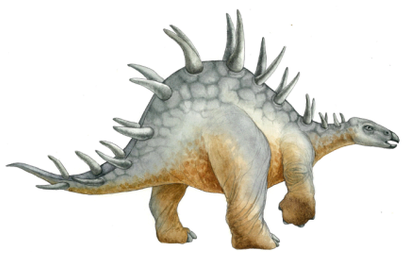
Serialización

- Considera dos elementos de datos A y B
- Considera las transacciones T0 y T1
- Ejecuta T0 y T1 atómicamente
- Ejecutar la secuencia llamada calendarizar
- Atómicamente ejecutar la transacción llamada calendario serial
- Para N transacciones, hay N! calendarios seriales válidos



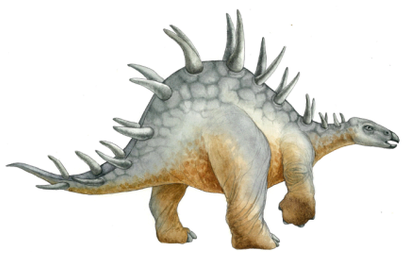
Calendario 1: T0 luego T1

T_0	T_1
read(A)	
write(A)	
read(B)	
write(B)	
	read(A)
	write(A)
	read(B)
	write(B)



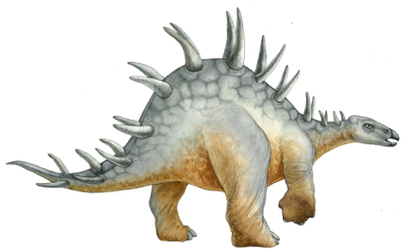
Calendario no-serial

- Calendario no-serial permite ejecución encimada
  - Resultado no necesariamente incorrecto
- Considera el calendario S, operaciones Oi, Oj
  - Conflicto si acceden el mismo dato, con al menos un write
- Si Oi, Oj son consecutivos y operaciones en distintas transacciones & Oi y Oj no se conflictúan
  - Entonces S' con orden invertido Oj Oi equivalente a S
- Si S puede convertirse S' via operaciones de intercambio no-conflicitivas
  - S es conflicto-serializable



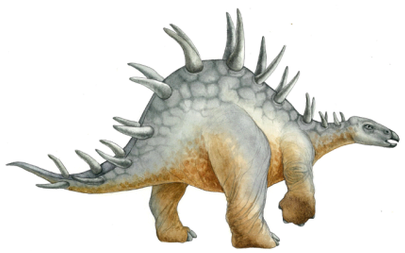
Calendario 2: Serializable Concurrente

T_0	T_1
read(A)	
write(A)	
	read(A)
	write(A)
read(B)	
write(B)	
	read(B)
	write(B)



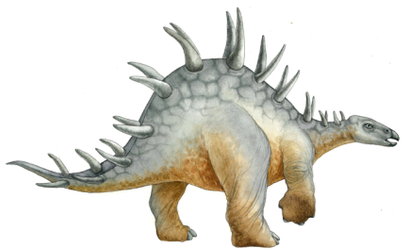
Protocolo de candados

- Asegura la serialización asociando candados con cada elemento en los datos
  - Sigue protocolo de candado para control de acceso
- Candados
  - Compartido – Ti tiene candado modo compartido (S) sobre elemento Q, Ti puede leer Q pero no escribir
  - Exclusivo – Ti tiene candado exclusivo (X) sobre Q, Ti puede leer y escribir Q
- Requiere que cada transacción en elemento Q adquiera el candado apropiado
- Si ya tiene candado, nueva solicitud tiene que esperar
  - Similar al algoritmo de lectores-escritores



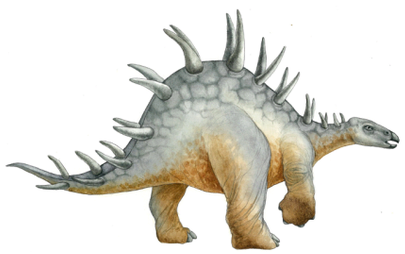
Protocolo de candado de dos fases

- Generalmente asegura conflicto-serializabilidad
- Cada transacción envía solicitudes lock y unlock en dos fases
  - Creciendo – obteniendo candados
  - Encogiendo – liberando candados
- No previene abrazos mortales



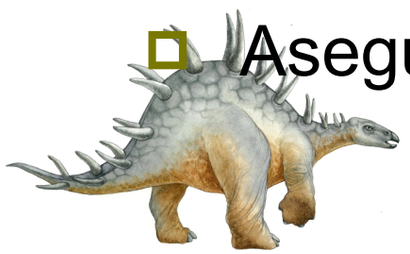
Protocolos basados en timestamps

- Selecciona orden entre transacciones por adelantado – ordenados por timestamp
- Transacción Ti asociado con TS(Ti) antes que Ti inicie
  - TS(Ti) < TS(Tj) si Ti entra al sistema antes que Tj
  - TS puede generarse a partir del reloj del sistema o como un contador lógico incrementado con cada entrada de la transacción
- Timestamps determinan el orden de serialización
  - Si TS(Ti) < TS(Tj), sistema debe garantizar calendario equivalente al calendario serial donde Ti aparece antes que Tj



Protocolo basado en timestamp

- Elemento Q obtiene dos timestamps
  - W-timestamp(Q) – timestamp más grande de cualquier transacción que ejecutó write(Q) con éxito
  - R-timestamp(Q) – timestamp más grande de un read(Q) exitoso
  - Se actualiza cada vez que se ejecutan read(Q) o write(Q)
- Protocolo ordenado por timestamp asegura que posibles read y write se ejecutan en el orden adecuado
- Suponga Ti ejecuta read(Q)
  - Si TS(Ti) < W-timestamp(Q), Ti necesita leer el valor de Q que ya fue sobre-escrito
    - Operación read rechazada y Ti rebobinada
  - Si TS(Ti) ≥ W-timestamp(Q)
    - read ejecutado, R-timestamp(Q) = max(R-timestamp(Q), TS (Ti))
- Asegura conflicto-serializabilidad y libre de abrazos mortales



Final del capítulo 6