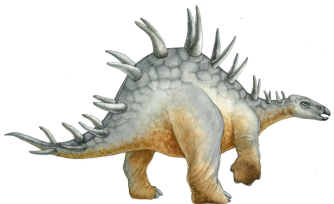


Capítulo 7: Abrazos mortales



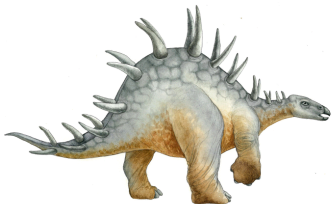
Capítulo 7: Abrazos mortales

- ❑ El problema de abrazo mortal
- ❑ Modelo de sistema
- ❑ Caracterización de abrazo mortal
- ❑ Métodos para manejar abrazos mortales
- ❑ Prevención de abrazos mortales
- ❑ Evitar caer en abrazos mortales
- ❑ Detección de abrazos mortales
- ❑ Recuperación de abrazos mortales



Objetivos del capítulo

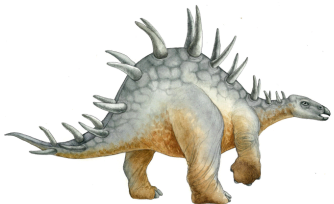
- ❑ Describir los abrazos mortales, que evitan que procesos concurrentes terminen sus tareas.
- ❑ Presentar un número de métodos distintos para prevenir o evitar abrazos mortales en un sistema de cómputo.



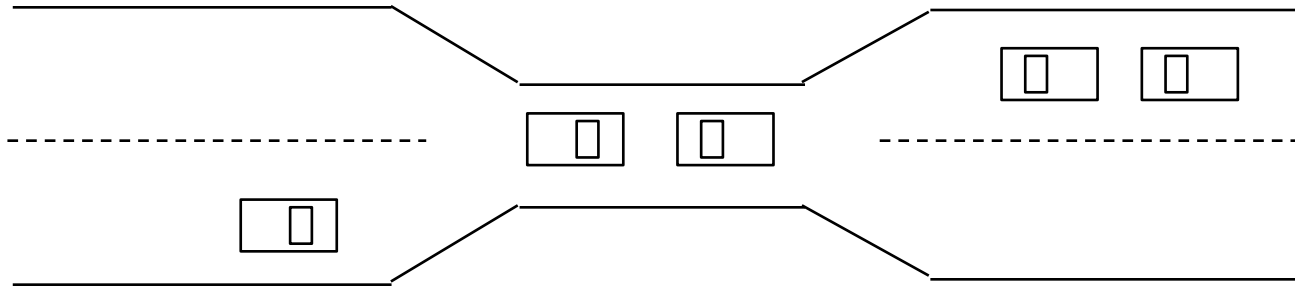
El problema del abrazo mortal

- Un conjunto de procesos bloqueados con un recurso y esperando adquirir otro recurso que tiene otro proceso en el conjunto.
- Ejemplo
 - El sistema tiene 2 discos duros.
 - P_1 y P_2 cada uno tiene un disco y necesita el otro.
- Ejemplo
 - Los semáforos A y B , iniciados en 1

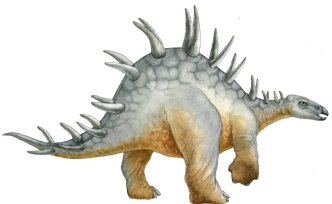
P_0	P_1
wait (A);	wait(B)
wait (B);	wait(A)



Ejemplo: cruzando el puente

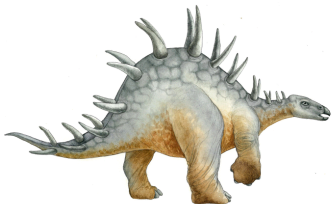


- ❑ Tráfico en una sola dirección.
- ❑ Cada sección del puente puede verse como un recurso.
- ❑ Si un abrazo mortal ocurre, puede resolverse si uno de los autos se regresa en reversa (se le quitan sus recursos -preempt- y se rebobina -rollback-).
- ❑ En un abrazo mortal puede ser necesario mandar en reversa muchos autos.
- ❑ Es posible que ocurra hambruna.



Modelo del sistema

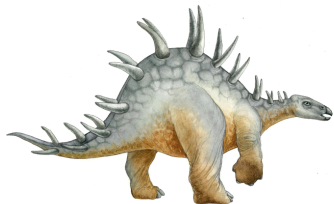
- Tipos de recursos R_1, R_2, \dots, R_m
Ciclos de CPU, memoria, dispositivos E/S
- Cada tipo de recurso R_i tiene W_i instancias.
- Cada proceso utiliza un recurso como sigue:
 - solicitud
 - uso
 - liberación



Caracterización de abrazo mortal

Abrazo mortal puede surgir si ocurren simultáneamente:

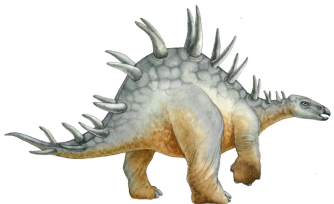
- ❑ **Exclusión mutua:** sólo un proceso puede utilizar un recurso en cada momento.
- ❑ **Mantener y esperar:** un proceso que mantiene al menos un recurso está esperando adquirir recursos que tienen otros procesos.
- ❑ **No preemption:** un recurso puede ser liberado solo de manera voluntaria por el proceso que lo mantiene, una vez que el proceso ha completado su tarea.
- ❑ **Espera circular:** existe un conjunto de procesos en espera $\{P_0, P_1, \dots, P_{n-1}\}$ tal que P_0 está esperando por un recurso que tiene P_1 , P_1 por uno que tiene P_2, \dots, P_{n-1} está esperando por un recurso que tiene P_n , y P_n está esperando por uno que tiene P_0 .



Gráfica de recurso-asignación

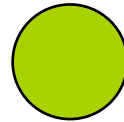
Un conjunto de vértices A y un conjunto de aristas E .

- V está particionado en dos tipos:
 - $P = \{P_1, P_2, \dots, P_n\}$, el conjunto que contiene todos los procesos del sistema.
 - $R = \{R_1, R_2, \dots, R_m\}$, el conjunto de todos los tipos de recursos en el sistema.
- Arista de solicitud – arista dirigida $P_i \rightarrow R_j$
- Arista de asignación – arista dirigida $R_j \rightarrow P_i$



Gráfica de recurso-asignación (Cont.)

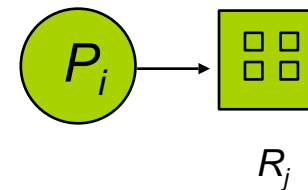
- Proceso



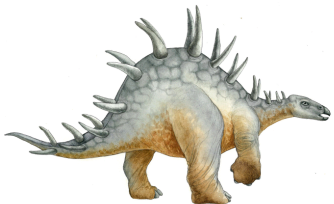
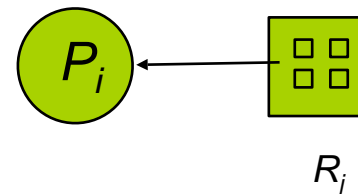
- Tipo de recurso con 4 instancias



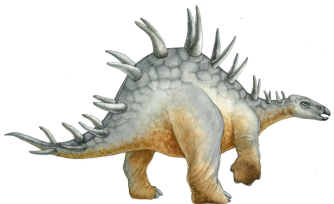
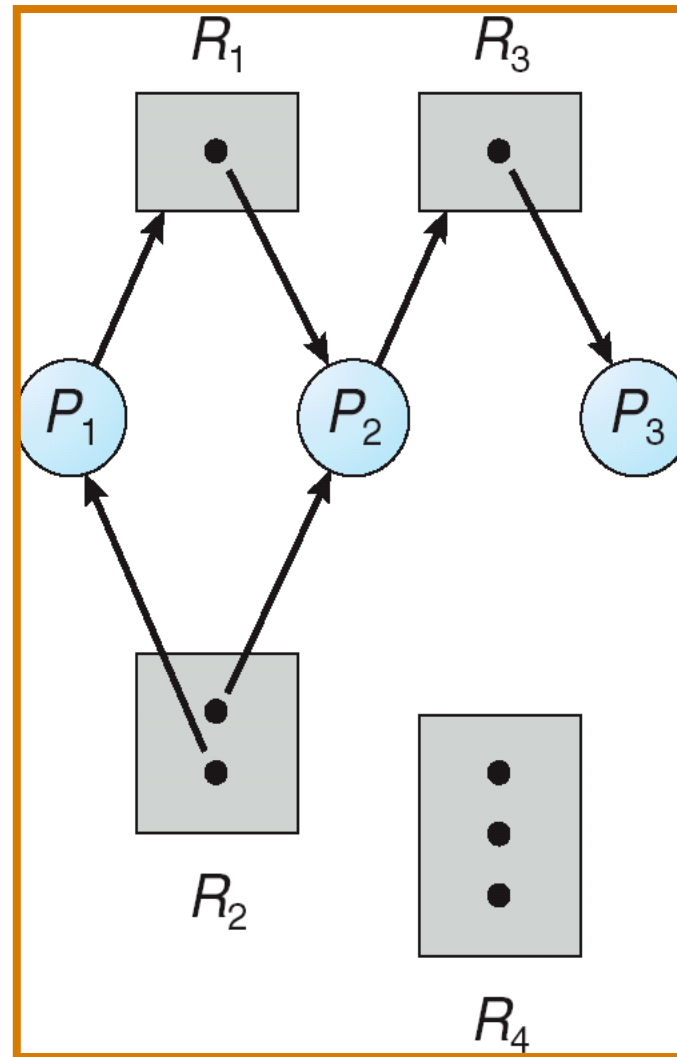
- P_i solicita instancia de recurso R_j



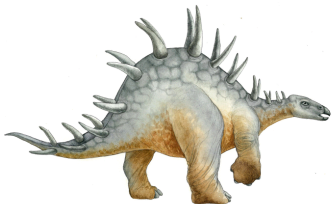
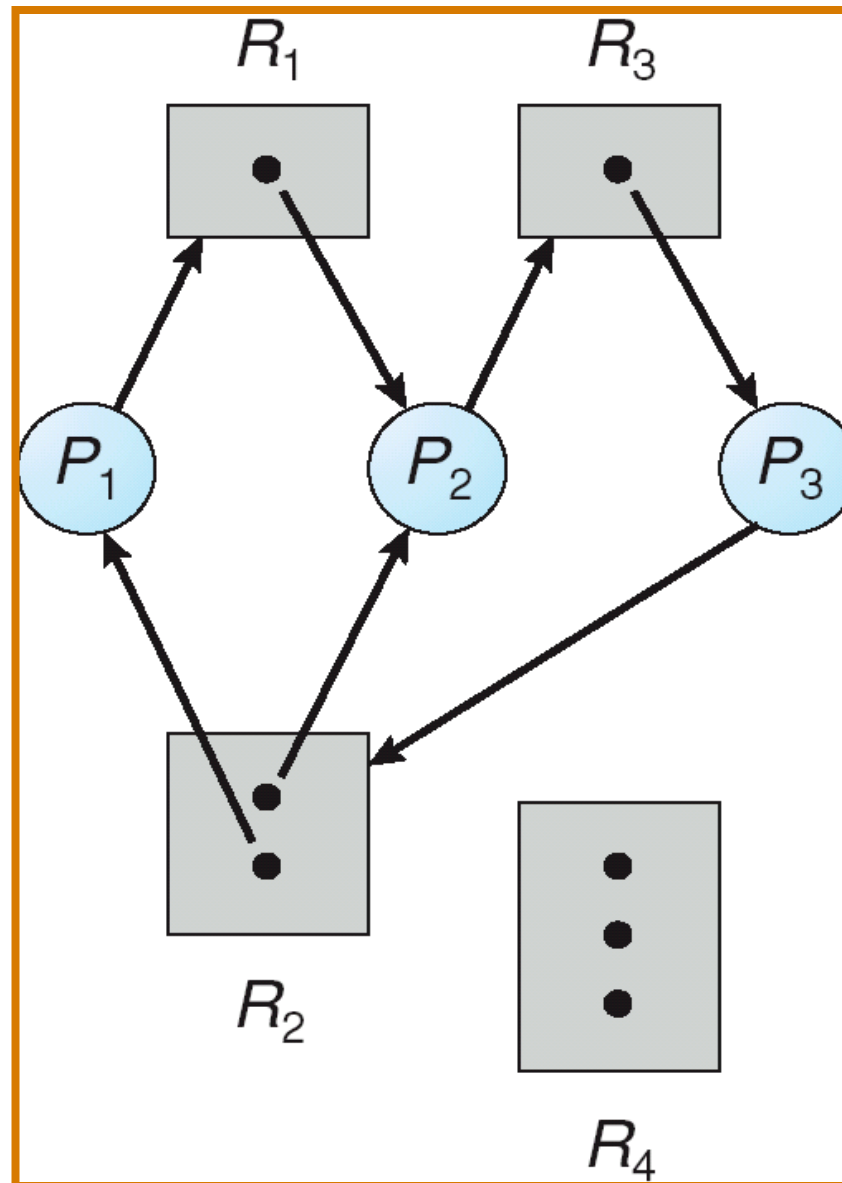
- P_i tiene una instancia de R_j



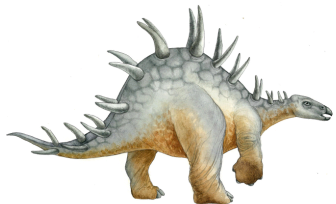
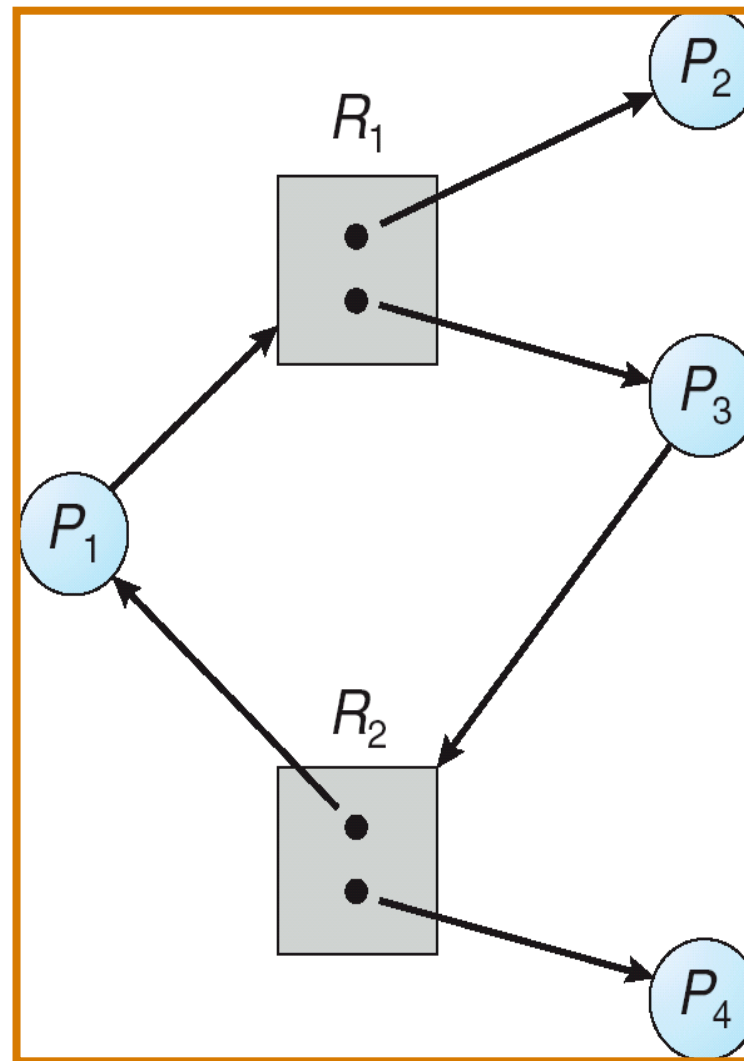
Ejemplo de gráfica de recurso-asignación



Gráfica recurso-asignación con abrazo mortal

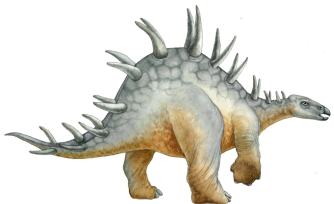


Gráfica con ciclo pero SIN abrazo mortal



Hechos básicos

- Si una gráfica no contiene ciclos \Rightarrow no abrazo mortal.
- Si una gráfica contiene un ciclo \Rightarrow
 - si sólo hay una instancia por tipo de recurso, entonces abrazo mortal.
 - si hay muchas instancia por tipo de recurso, *posibilidad* de abrazo mortal.



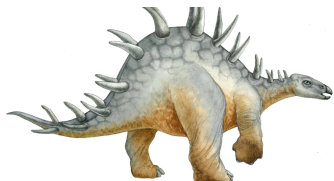
Ejemplo de abrazo mortal en Java

```
class A implements Runnable
{
    private Lock first, second;

    public A(Lock first, Lock second) {
        this.first = first;
        this.second = second;
    }

    public void run() {
        try {
            first.lock();
            // do something
            second.lock();
            // do something else
        }
        finally {
            first.unlock();
            second.unlock();
        }
    }
}
```

Thread
A



```
class B implements Runnable
{
    private Lock first, second;

    public A(Lock first, Lock second) {
        this.first = first;
        this.second = second;
    }

    public void run() {
        try {
            second.lock();
            // do something
            first.lock();
            // do something else
        }
        finally {
            second.unlock();
            first.unlock();
        }
    }
}
```

Thread
B

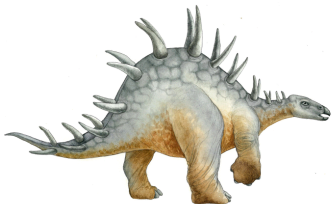


Ejemplo de abrazo mortal en Java

```
public static void main(String arg[]) {  
    Lock lockX = new ReentrantLock();  
    Lock lockY = new ReentrantLock();  
  
    Thread threadA = new Thread(new A(lockX,lockY));  
    Thread threadB = new Thread(new B(lockX,lockY));  
  
    threadA.start();  
    threadB.start();  
}
```

Abrazo mortal posible si:

threadA -> lockY -> threadB -> lockX -> threadA



Manejo de abrazos mortales en Java

```
public class ClockApplet extends Applet implements Runnable
{
    private Thread clockThread;
    private boolean ok = false;
    private Object mutex = new Object();

    public void run() {
        while (true) {
            try {
                // sleep for 1 second
                Thread.sleep(1000);

                // repaint the date and time
                repaint();

                // see if we need to suspend ourself
                synchronized (mutex) {
                    while (ok == false)
                        mutex.wait();
                }
            }
            catch (InterruptedException e) { }
        }
    }

    public void start() {
        // Figure 7.7
    }

    public void stop() {
        // Figure 7.7
    }

    public void paint(Graphics g) {
        g.drawString(new java.util.Date().toString(), 10, 30);
    }
}
```

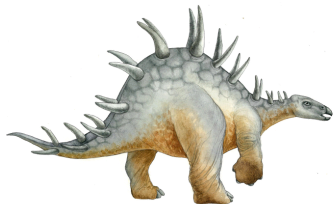


Manejo de abrazo mortal en Java

```
// this method is called when the applet is
// started or we return to the applet
public void start() {
    ok = true;

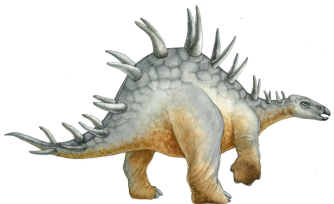
    if (clockThread == null) {
        clockThread = new Thread(this);
        clockThread.start();
    }
    else {
        synchronized(mutex) {
            mutex.notify();
        }
    }
}

// this method is called when we
// leave the page the applet is on
public void stop() {
    synchronized(mutex) {
        ok = false;
    }
}
```



Métodos para manejar abrazos mortales

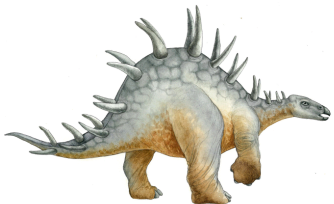
- ❑ Asegurar que el sistema **nunca** entrará en un abrazo mortal.
- ❑ Permitir que el sistema entre en estado de abrazo mortal y luego recuperar.
- ❑ Ignorar el problema y pretender que los abrazos mortales nunca ocurren en el sistema
 - Utilizado por la mayoría de los sistemas operativos, incluyend UNIX.



Prevención de abrazos mortales

Restringir la forma en que se hacen las solicitudes.

- ❑ **Exclusión mutua** – no se requiere para recursos que pueden compartirse; debe forzarse para aquellos que no pueden compartirse.
- ❑ **Mantener y esperar** – debe garantizar que cada vez que un proceso solicita un recurso, no tiene ningún otro recurso.
 - Forzar procesos a solicitar (y se les asignen) todos sus recursos antes de iniciar ejecución, o permitir la solicitud cuando el proceso no tiene ningún recurso.
 - Baja utilización de recursos; hambruna posible.

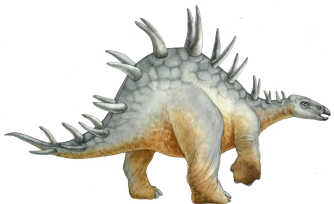


Prevención de abrazos mortales (Cont.)

□ No Preemption –

- Si un proceso que mantiene varios recursos solicita otro recurso que no le puede ser asignado inmediatamente; entonces todos los recursos que ya tenía se liberan.
- Los recursos liberados son añadidos a la lista de recursos por los cuáles está esperando el proceso.
- El proceso será reiniciado hasta que pueda obtener sus viejos recursos y los nuevos que está solicitando.

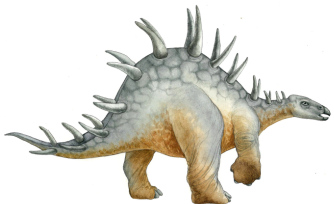
□ Espera circular – imponer un orden total de todos los tipos de recursos y requerir que cada proceso solicite recursos en estricto orden de numeración.



Evitar abrazos mortales

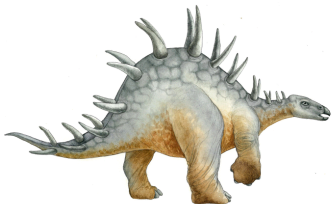
Requiere que el sistema tenga información adicional *a priori*.

- El modelo más sencillo y útil requiere que cada proceso declare el *número máximo* de recursos de cada tipo que requerirá.
- El algoritmo para evitar abrazos mortales examina dinámicamente el estado recurso-asignación para asegurar que nunca exista la condición de espera circular.
- Es *estado* recurso-asignación está definido por el número de recursos disponibles y asignados y la demanda máxima de los procesos.



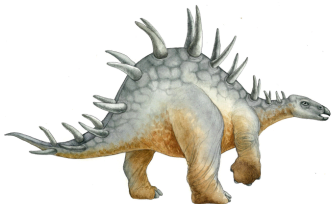
Estado seguro

- Cuando un proceso solicita un recurso disponible, el sistema debe decidir si la asignación inmediata deja el sistema en un estado seguro.
- El sistema está en **estado seguro** si existe una secuencia $\langle P_1, P_2, \dots, P_n \rangle$ de TODOS los procesos en el sistema, tal que para cada P_i , los recursos que P_i solicitará pueden satisfacerse con recursos disponibles + recursos que tienen todos los P_j , con $j < i$.
- Esto es:
 - Si las necesidades de recursos de P_i no están disponibles inmediatamente, entonces P_i puede esperar hasta que todos los P_j *hayan terminado*.
 - Cuando P_j termina, P_i puede obtener los recursos que necesita, ejecutar, regresar los recursos asignados y terminar.
 - Cuando P_i termina, P_{i+1} puede obtener los recursos que necesita y así sucesivamente.

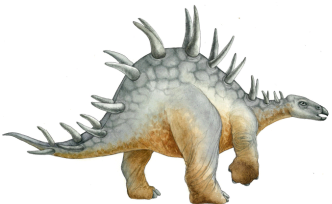
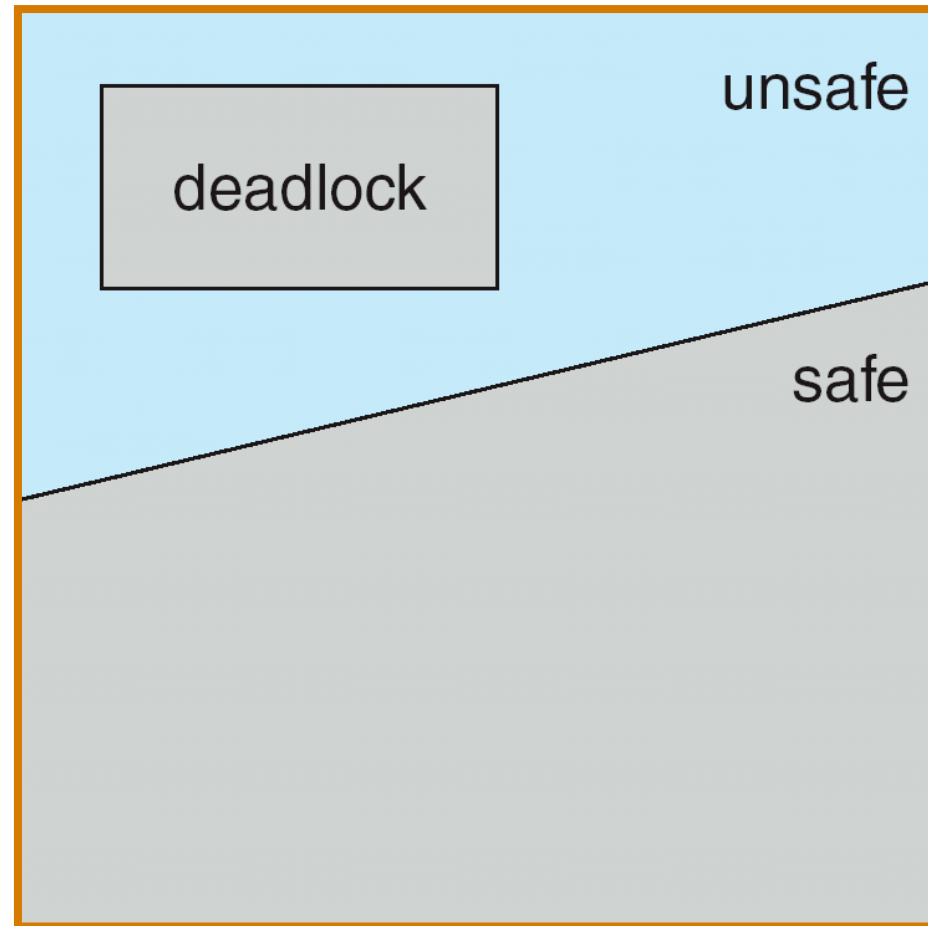


Hechos básicos

- ❑ Si el sistema está en estado seguro \Rightarrow no abrazos mortales.
- ❑ Si el sistema está en estado inseguro \Rightarrow posibilidad de abrazos mortales.
- ❑ Evitar \Rightarrow asegurar que el sistema nunca entra en un estado inseguro.

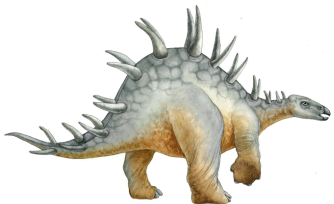


Estados seguro, inseguro y abrazo mortal



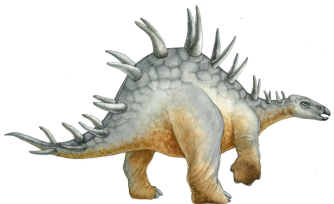
Algoritmos para evitar

- Una sola instancia de cada tipo de recurso.
Utilizar una gráfica de asignación de recursos.
- Varias instancias de un tipo de recurso.
Utilizar el algoritmo del banquero.

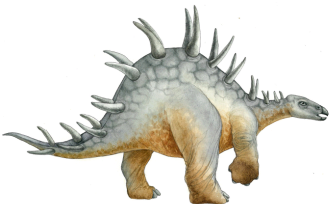
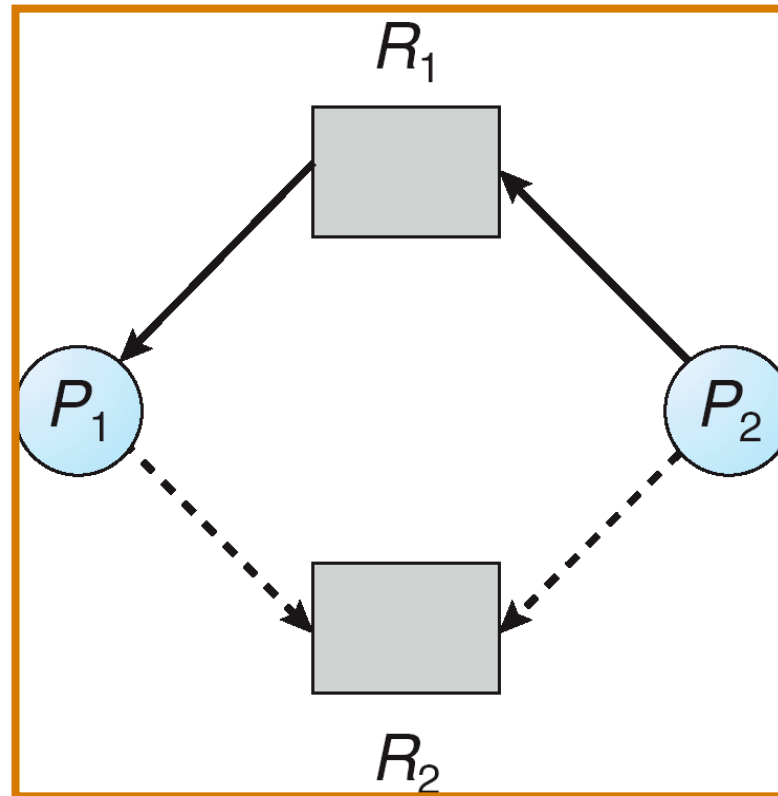


Esquema gráfica asignación de recursos

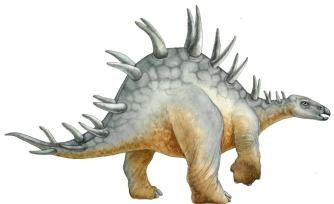
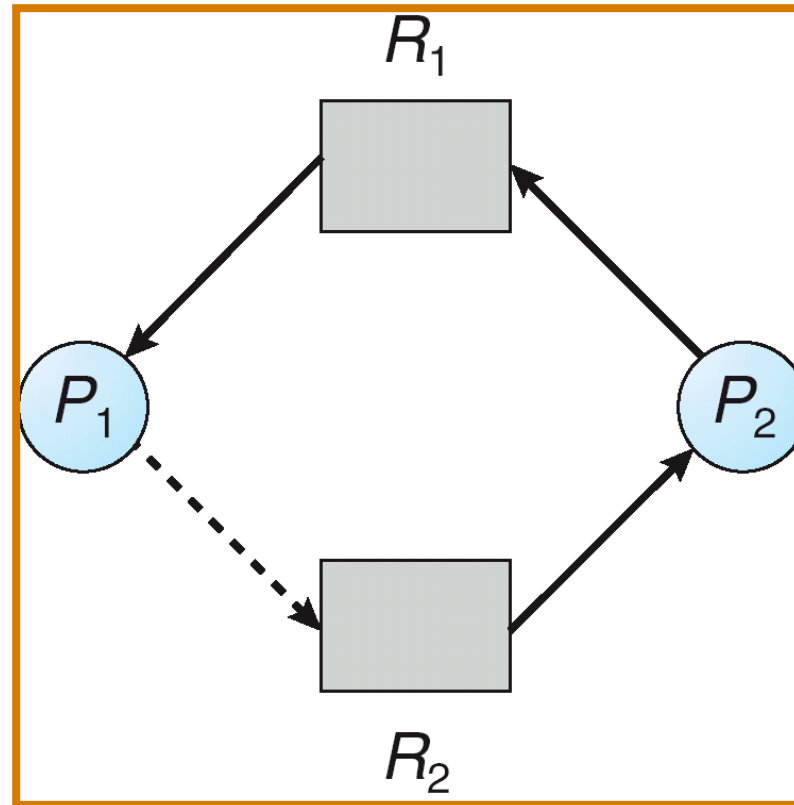
- *Arista de reserva* $P_i \rightarrow R_j$ indica que el proceso P_i puede solicitar el recurso R_j ; representado por una línea punteada.
- Una *arista de reserva* se convierte en una de *solicitud* cuando el proceso solicita el recurso.
- La *arista de solicitud* se convierte en una de *asignación* cuando el recurso es asignado al proceso.
- Cuando el proceso libera el recurso, la *arista de asignación* se convierte nuevamente en una de *reserva*.
- Los recursos deben solicitarse *a priori* en el sistema.



Gráfica de asignación de recursos

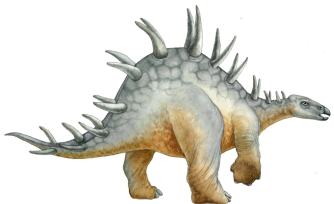


Estado inseguro en gráfica asignación recursos



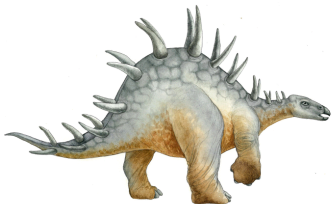
Algoritmo gráfica de asignación recursos

- Suponemos que el proceso P_i solicita un recurso R_j
- La solicitud puede concederse sólo si convirtiendo la arista de solicitud a una de asignación, no forma un ciclo en la gráfica



Algoritmo del banquero

- ❑ Instancias múltiples.
- ❑ Cada proceso debe *a priori* reservar su utilización máxima.
- ❑ Cuando un proceso solicita un recurso, puede tener que esperar.
- ❑ Cuando un proceso obtiene todos sus recursos, debe devolverlos en un tiempo finito de tiempo.

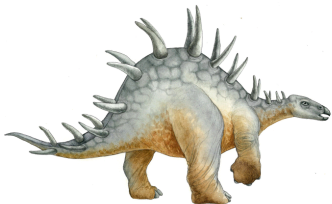


Estructuras de datos para el Alg. Banquero

Sea n = número de procesos, y m = número de tipos de recursos.

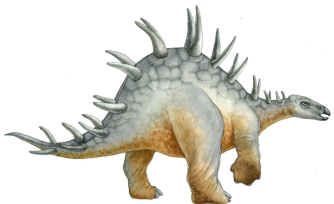
- **Available**: Vector de longitud m . Si $available[j] = k$, existen k instancias disponibles del recurso tipo R_j .
- **Max**: Matriz de $n \times m$. Si $Max[i,j] = k$, el proceso P_i puede solicitar a lo más k instancias del recurso tipo R_j .
- **Allocation**: Matriz de $n \times m$. Si $Allocation[i,j] = k$ entonces P_i tiene asignadas k instancias de R_j .
- **Need**: Matriz de $n \times m$. Si $Need[i,j] = k$, entonces P_i puede requerir k nuevas instancias de R_j para completar su trabajo.

$$Need[i,j] = Max[i,j] - Allocation[i,j].$$



Algoritmo de estado seguro

1. Sean **Work** y **Finish** vectores de longitudes m y n , respectivamente. Iniciamos:
 $Work = Available$
 $Finish[i] = false$ for $i = 0, 1, \dots, n-1$.
2. Find e i tales que ambas:
(a) $Finish[i] = false$
(b) $Need_i \leq Work$
Si no existe tal i , ir al paso 4.
3. $Work = Work + Allocation_i$
 $Finish[i] = true$
ir al paso 2.
4. Si $Finish[i] == true$ para toda i , el sistema está en un *estado seguro*.



Algoritmo de solicitud de recursos para P_i

Request = vector de solicitud del proceso P_i . Si $Request_i[j] = k$ el proceso P_i quiere k del recurso tipo R_j .

1. Si $Request_i \leq Need_i$ ir al paso 2. En otro caso, enviar condición de error, porque el proceso ha excedido su reserva máxima.
2. Si $Request_i \leq Available$, ir al paso 3. En otro caso P_i debe esperar, porque no hay recursos disponibles.
3. Simulamos la asignación de recursos solicitados a P_i modificando el estado como sigue:

$Available = Available - Request_i;$

$Allocation_i = Allocation_i + Request_i;$

$Need_i = Need_i - Request_i;$

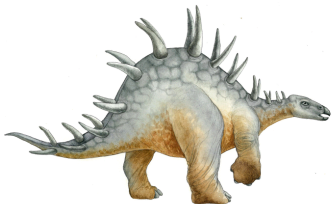
- Si seguro \Rightarrow se asignan los recursos a P_i .
- Si inseguro $\Rightarrow P_i$ debe esperar y se re-instala el viejo estado de recursos-asignación



Ejemplo del algoritmo del banquero

- 5 procesos P_0 a P_4 ;
3 tipos de recursos:
A (10 instancias), B (5 instancias), y C (7 instancias).
- Instantánea en el tiempo T_0 :

	Allocation	Max	Available
	A B C	A B C	A B C
P_0	0 1 0	7 5 3	3 3 2
P_1	2 0 0	3 2 2	
P_2	3 0 2	9 0 2	
P_3	2 1 1	2 2 2	
P_4	0 0 2	4 3 3	



Ejemplo (Cont.)

- El contenido de la matriz *Need* está definido como $Max - Allocation$.

	Allocation	Max	Need
	A B C	A B C	A B C
P ₀	0 1 0	7 5 3	7 4 3
P ₁	2 0 0	3 2 2	1 2 2
P ₂	3 0 2	9 0 2	6 0 0
P ₃	2 1 1	2 2 2	0 1 1
P ₄	0 0 2	4 3 3	4 3 1

- El sistema está en un estado seguro dado que la secuencia $\langle P_1, P_3, P_4, P_2, P_0 \rangle$ satisface los criterios de seguridad.

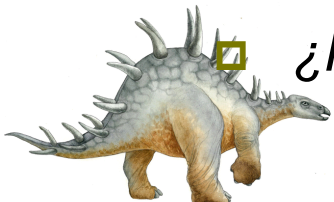


Ejemplo: P_1 Request (1,0,2)

- Checar que Request \leq Available (esto es, $(1,0,2) \leq (3,3,2) \Rightarrow \text{true}$).

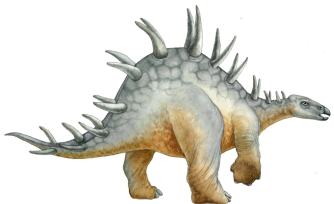
	Allocation	Max	Available
	A B C	A B C	A B C
P_0	0 1 0	7 5 3	2 3 0
P_1	3 0 2	3 2 2	
P_2	3 0 2	9 0 2	
P_3	2 1 1	2 2 2	
P_4	0 0 2	4 3 3	

- Ejecutando el algoritmo de seguridad obtenemos que la secuencia $\langle P_1, P_3, P_4, P_0, P_2 \rangle$ satisface los requerimientos de seguridad.
- ¿Puede autorizarse la solicitud (3,3,0) de P_4 ?
- ¿Puede autorizarse la solicitud (0,2,0) de P_0 ?



Detección de abrazos mortales

- ❑ Permitir al sistema entrar en estado de abrazo mortal
- ❑ Algoritmo de detección
- ❑ Esquema de recuperación

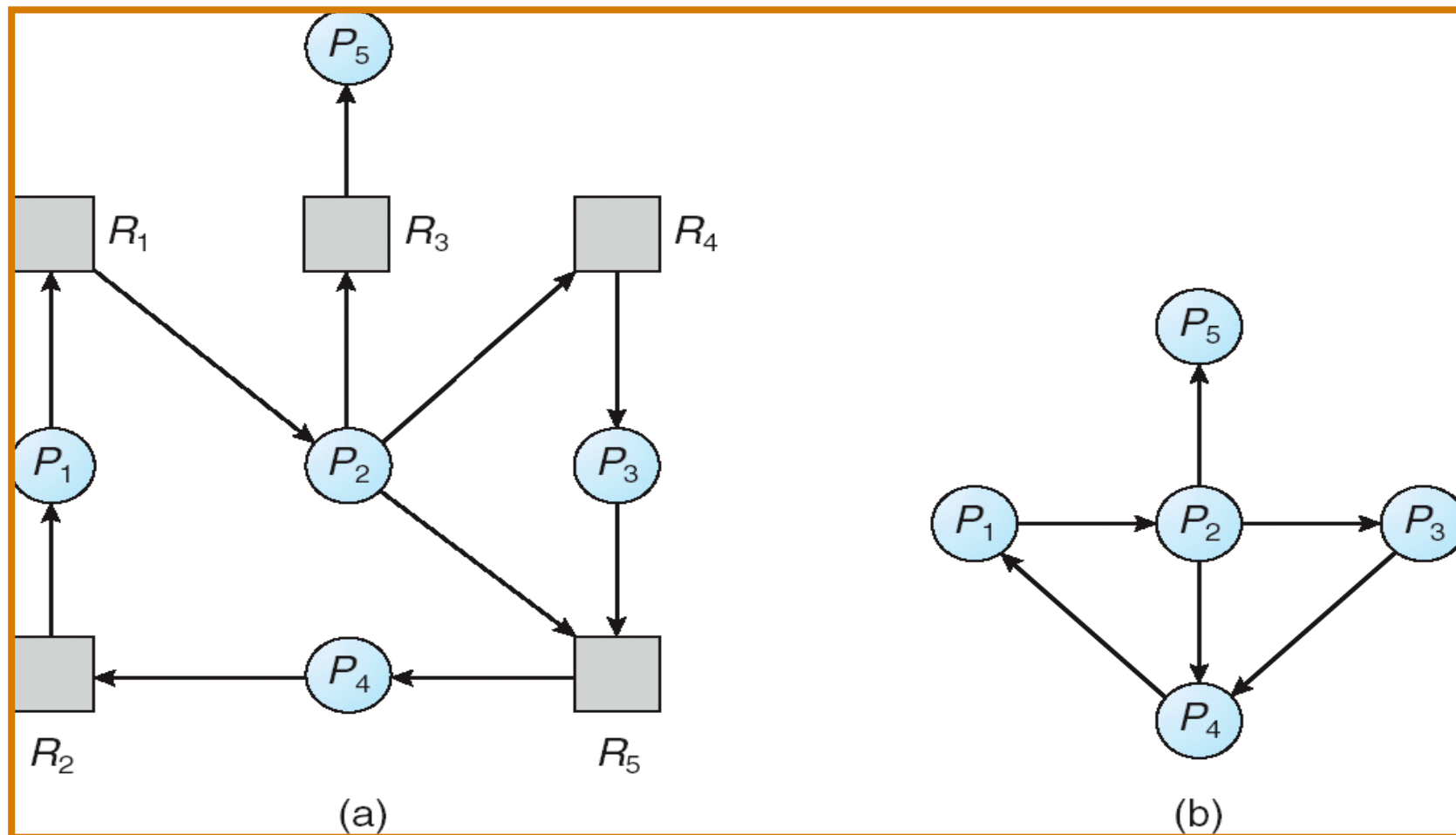


Una instancia por tipo de recurso

- Mantener una gráfica *esperando-a*
 - Los nodos son procesos.
 - $P_i \rightarrow P_j$ si P_i espera a P_j .
- Periódicamente invocar un algoritmo que busca ciclos en la gráfica. **Si hay un ciclo, hay un abrazo mortal.**
- Un algoritmo para determinar si existe un ciclo en una gráfica es orden de n^2 , donde n es el número de vértices.



Gráficas de asignación-recurso y esperando-a



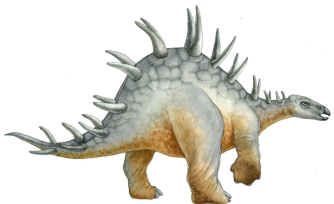
Gráfica asignación-recurso

Gráfica correspondiente esperando-a



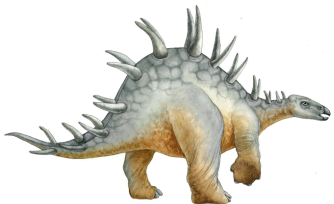
Muchas instancias de un tipo de recurso

- **Available:** Vector de longitud m que indica el número de recursos disponibles de cada tipo.
- **Allocation:** Matriz de $n \times m$ que define el número de recursos de cada tipo asignados a cada proceso.
- **Request:** Matriz de $n \times m$ que indica la solicitud actual de cada proceso. Si $Request[i_j] = k$, el proceso P_i solicita k nuevas instancias de R_j .



Algoritmo de detección

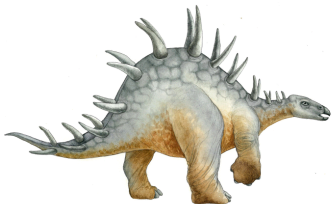
1. Sean *Work* y *Finish* vectores de longitud m y n respectivamente, iniciados así:
 - (a) $Work = Available$
 - (b) For $i = 1, 2, \dots, n$, if $Allocation_i \neq 0$, then
 $Finish[i] = false$; otherwise, $Finish[i] = true$.
2. Encontrar un índice i tal que se cumplan ambas:
 - (a) $Finish[i] == false$
 - (b) $Request_i \leq Work$Si no existe tal i , ir al paso 4.



Algoritmo de detección (Cont.)

3. $Work = Work + Allocation_i$
 $Finish[i] = true$
ir al paso 2.
4. If $Finish[i] == false$, para algunas i , $1 \leq i \leq n$,
entonces el sistema está en estado de abrazo mortal. Más aún, si $Finish[i] == false$, entonces P_i está en abrazo mortal.

El algoritmo requiere un orden de $O(m \times n^2)$ operaciones para determinar si el sistema está en abrazo mortal.



Ejemplo de algoritmo de detección

- Cinco procesos P_0 a P_4 ; tres tipos de recursos A (7 instancias), B (2 instancias), y C (6 instancias).
- Instantánea al tiempo T_0 :

	Allocation	Request	Available
	A B C	A B C	A B C
P_0	0 1 0	0 0 0	0 0 0
P_1	2 0 0	2 0 2	
P_2	3 0 3	0 0 0	
P_3	2 1 1	1 0 0	
P_4	0 0 2	0 0 2	

- La secuencia $\langle P_0, P_2, P_3, P_1, P_4 \rangle$ termina $Finish[i] = \text{true}$ para toda i .



Ejemplo (Cont.)

- P_2 solicita un recurso adicional de tipo C.

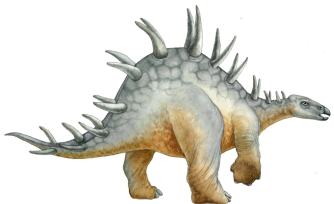
	<u>Request</u>		
	A	B	C
P_0	0	0	0
P_1	2	0	1
P_2	0	0	1
P_3	1	0	0
P_4	0	0	2

- ¿Estado del sistema?
 - Puede recuperar los recursos ocupados por el proceso P_0 , pero son insuficientes para satisfacer las solicitudes de otros procesos.
 - Existe abrazo mortal, formado por los procesos P_1 , P_2 , P_3 , y P_4 .



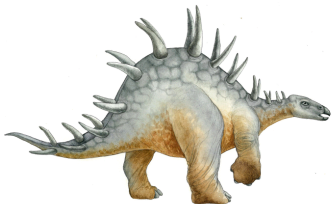
Uso del algoritmo de detección

- Cuándo y con qué frecuencia lo invocamos depende de:
 - Qué tan frecuentemente puede ocurrir un abrazo mortal
 - Cuántos procesos será necesario rebobinar
 - uno por cada ciclo disjunto
- Si el algoritmo de detección se invoca de manera arbitraria, pueden existir muchos ciclos en la gráfica de recursos y no podremos saber cuál de los muchos procesos en abrazo mortal lo "provocó".



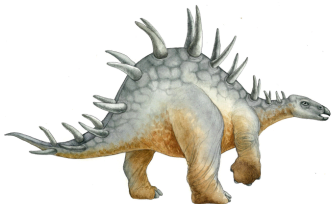
Recuperación abrazo mortal: terminar proceso

- ❑ Abortar todos los procesos en abrazo mortal.
- ❑ Abortar un proceso a la vez hasta que se elimine el ciclo de abrazo mortal.
- ❑ ¿En qué orden decidimos abortar?
 - Prioridad de los procesos.
 - Cuánto tiempo de CPU ha ocupado el proceso y cuánto le falta.
 - Recursos que el proceso ha utilizado.
 - Recursos que el proceso requiere para terminar.
 - Cuántos procesos deberán ser terminados.
 - ¿El proceso es interactivo o por lotes?



Recuperación abrazo mortal: recuperación de recursos

- ❑ Seleccionar una víctima – minimizar el costo.
- ❑ Rebobinar – regresar a un punto seguro, reiniciar el proceso para ese estado.
- ❑ Hambruna – el mismo proceso puede siempre ser seleccionado como víctimas, incluir el número de rebobinado como factor de costo.



Final del Capítulo 7

