Universidad de Granada

decsai.ugr.es

Tema 5

Gestión y control de concurrencia

DECSAI

Departamento de Ciencias de la Computación e Inteligencia Artificial

Índice

- Problemas producidos por la concurrencia
- Ejecuciones concurrentes sin conflicto
- Algoritmos de control de concurrencia

Introducción

- Ejecución concurrente de transacciones para optimizar la eficiencia.
- Controlador de concurrencia gestiona el uso de recursos.
- Hay que garantizar la consistencia de la BD.

Problemas producidos por la concurrencia

- Anulación de transacción: cuando una transacción quiere acceder a un dato que está siendo modificado por otra o cuando es eliminado por otra.

T1	T2
lee(X,x_i)	
	lee(X,x_i)
	x_i:=x_i+1
	escribe(X,x_i)
x_i:=x_i*3	
escribe(X,x_i)	

Problemas producidos por la concurrencia

- Estado inconsistente de la BD:
  - transacciones concurrentes que violan temporalmente las restricciones de la DB
  - Supongamos que B es clave externa a A, luego no deben tener valores distintos en ningún momento.

T1	T2
lee(A,x_i)	
x_i:=x_i+1	
escribe(A,x_i)	
	lee(A,z_i)
	imprime(z_i)
	lee(B,z_j)
	imprime(z_j)
lee(B,x_j)	
x_j:=x_j+1	
escribe(B,x_j)	

Problemas producidos por la concurrencia

T1	T2
lee(A,xi)	
xi:=xi+1	
escribe(A,xi)	
	lee(A,zi)
	zi := zi * 2
	escribe(A,zi)
	lee(B,zj)
	zj := zj * 2
	escribe(B,zj)
lee(B,xj)	
xj:=xj+1	
escribe(B,xj)	

Problemas producidos por la concurrencia

Operaciones en conflicto:

- pertenecen a distintas transacciones,
- acceden al mismo dato, y
- alguna de ellas ejecuta la orden `escribe()`

Ejecuciones concurrentes sin conflicto

- Un SGBD ejecuta transacciones concurrentes, formadas por sentencias.
- **Plan de ejecución**: secuencia de instrucciones pertenecientes a todas las transacciones concurrentes, ejecutadas en un orden preciso.

Ejecuciones concurrentes sin conflicto

- **Átomo**: fragmento de información en la BD cuyo acceso concurrente debe controlarse.
- ¿Atributo en una tupla?, ¿tupla?, ¿columna?, ¿tabla?
- A menor tamaño, más difícil de controlar (hacen falta más recursos); a mayor tamaño, más transacciones tienen que esperar su turno con el átomo.

Ejecuciones concurrentes sin conflicto

- Pensemos en las dos transacciones anteriores:

T1
lee(A,x_i)
$x_i := x_i + 1$
escribe(A,x_i)
lee(B,x_j)
$x_j := x_j + 1$
escribe(B,x_j)

T2
lee(A,z_i)
$z_i := z_i * 2$
escribe(A,z_i)
lee(B,z_j)
$z_j := z_j * 2$
escribe(B,z_j)

Ejecuciones concurrentes sin conflicto

- Plan de ejecución 1:
  - ¿es correcto?
  - ¿Cuáles serán los valores finales en la BD para A y B?

T1	T2
lee(A,x_i)	
x_i:=x_i+1	
escribe(A,x_i)	
	lee(A,z_i)
	z_i := z_i * 2
	escribe(A,z_i)
lee(B,x_j)	
x_j:=x_j+1	
escribe(B,x_j)	
	lee(B,z_j)
	z_j := z_j * 2
	escribe(B,z_j)

Ejecuciones concurrentes sin conflicto

- Plan de ejecución 1:
  - correcto
  - A = (A+1) * 2

    B = (B+1) * 2
  - Si ejecutamos primero T1 completa y luego T2 completa, nos queda lo mismo: es serializable.

T1	T2
lee(A,x_i)	
x_i:=x_i+1	
escribe(A,x_i)	
	lee(A,z_i)
	z_i := z_i * 2
	escribe(A,z_i)
lee(B,x_j)	
x_j:=x_j+1	
escribe(B,x_j)	
	lee(B,z_j)
	z_j := z_j * 2
	escribe(B,z_j)

Ejecuciones concurrentes sin conflicto

- Plan de ejecución 2:
  - ¿es correcto?
  - ¿Cuáles serán los valores finales en la BD para A y B?

T1	T2
	lee(A,z_i)
	$z_i := z_i * 2$
lee(A,x_i)	
$x_i := x_i + 1$	
	escribe(A,z_i)
	lee(B,z_j)
	$z_j := z_j * 2$
escribe(A,x_i)	
lee(B,x_j)	
$x_j := x_j + 1$	
escribe(B,x_j)	
	escribe(B,z_j)

Ejecuciones concurrentes sin conflicto

- Plan de ejecución 2:
  - incorrecto
  - A = A + 1

    B = B * 2
  - Ejecución no serializable.

T1	T2
	lee(A,z_i)
	z_i := z_i * 2
lee(A,x_i)	
x_i:=x_i+1	
	escribe(A,z_i)
	lee(B,z_j)
	z_j := z_j * 2
escribe(A,x_i)	
lee(B,x_j)	
x_j:=x_j+1	
escribe(B,x_j)	
	escribe(B,z_j)

Ejecuciones concurrentes sin conflicto

- Si el resultado de la ejecución de un conjunto de transacciones concurrentes coincide con la ejecución secuencial de las transacciones, se dice que esa ejecución es **serializable**.
- El número de ejecuciones posibles crece con el número de transacciones y no se pueden explorar todas. Hay ciertas combinaciones que permiten determinar si la ejecución es correcta o no.

Operación

- Conjunto de sentencias que actúan sobre un átomo o variable concretos.
- Dos operaciones que no modifican el átomo y que pertenecen a dos transacciones distintas pueden ejecutarse simultáneamente. Es decir, las operaciones de “sólo lectura” pueden intercarlarse.

Operación compatible

- O_i y O_j son compatibles si toda ejecución simultánea de ambas da el mismo resultado que la ejecución secuencial de ambas en cualquier orden.

Operación compatible

o_1
lee(A,x_1)
$x_1 := x_1 + 1$
imprime(x_1)

o_2
lee(A,x_2)
$x_2 := x_2 * 2$
imprime(x_2)

¿Compatibles o incompatibles?

Operación compatible

o_1
lee(A,x_1)
$x_1 := x_1 + 1$
imprime(x_1)

o_2
lee(A,x_2)
$x_2 := x_2 * 2$
imprime(x_2)

Compatibles

Operación compatible

o_1
lee(A,x_1)
$x_1 := x_1 + 1$
escribe(A,x_1)

o_2
lee(A,x_2)
$x_2 := x_2 * 2$
escribe(A,x_2)

¿Compatibles o incompatibles?

Operación compatible

o_1
lee(A,x_1)
$x_1 := x_1 + 1$
escribe(A,x_1)

o_2
lee(A,x_2)
$x_2 := x_2 * 2$
escribe(A,x_2)

Incompatibles

o_1	o_2
lee(A,x_1)	
	lee(A,x_2)
$x_1 := x_1 + 1$	
	$x_2 := x_2 * 2$
escribe(A,x_1)	
	escribe(A,x_2)

Operaciones permutables

- O_i y O_j son permutables si la ejecución de O_j tras O_i da el mismo resultado que la ejecución de O_i tras O_j.

Operaciones permutables

O_1
lee(A,x_1)
$x_1 := x_1 + 1$
escribe(A,x_1)

O_2
lee(A,x_2)
$x_2 := x_2 + 10$
escribe(A,x_2)

Son permutables

Operaciones permutables

o_1
lee(A,x_1)
$x_1 := x_1 + 1$
escribe(A,x_1)

o_2
lee(A,x_2)
$x_2 := x_2 + 10$
escribe(A,x_2)

Son permutables

Pero ¿son compatibles?

Operaciones permutables

O_1
lee(A,x_1)
$x_1 := x_1 + 1$
escribe(A,x_1)

O_2
lee(A,x_2)
$x_2 := x_2 + 10$
escribe(A,x_2)

O_1	O_2	A
lee(A,x_1)		6
	lee(A,x_2)	6
$x_1 := x_1 + 1$		7
	$x_2 := x_2 + 10$	16
escribe(A,x_1)		7
	escribe(A,x_2)	16

¡Incompatibles!

Pero ¿por qué?

Un ejemplo

- Prueba con la permutabilidad de...

T_1
lee(A,x_1)
lee(B,x_2)
$x_1 := x_1 * 2 + x_2$
escribe(A,x_1)

T_2
lee(A,x_1)
$x_1 := x_1 * x_1$
escribe(B,x_1)

Un ejemplo

- ¿Y la compatibilidad?

T_1
lee(A,x_1)
lee(B,x_2)
$x_1 := x_1 * 2 + x_2$
escribe(A,x_1)

T_2
lee(A,x_1)
$x_1 := x_1 * x_1$
escribe(B,x_1)

Un ejemplo

- Considera la ejecución

T_1
lee(A,x_1)
lee(B,x_2)
$x_1 := x_1 * 2 + x_2$
escribe(A,x_1)

T_2
lee(A,x_1)
$x_1 := x_1 * x_1$
escribe(B,x_1)

T_1	T_2
lee(A,x_1)	
	lee(A,x_1)
lee(B,x_2)	
	$x_1 := x_1 * x_1$
$x_1 := x_1 * 2 + x_2$	
escribe(A,x_1)	
	escribe(B,x_1)

Ejecuciones serializables

- Transformaciones sobre una ejecución:
  - Separación de operaciones compatibles: dadas dos operaciones entrelazadas en transacciones distintas y cambiarlas por una secuencia de operaciones que den el mismo resultado.
  - Re-ordenación de operaciones compatibles: cambiar el orden de ejecución de operaciones permutables.

Ejecuciones serializables

- Teorema:
  - Una condición suficiente para una ejecución serializable es que pueda ser transformada por separación y permutación en una sucesión de transacciones.

Ejecuciones serializables

- Supongamos:

T_1	T_2
lee(A,x_1)	
x_1 := x_1 * 10	
escribe(A,x_1)	
	lee(A,x_3)
lee(B,x_2)	
	x_3 := x_3 * 5
...	...

T_1	T_2
...	...
	escribe(A,x_3)
x_2 := x_2 * 3	
escribe(B,x_2)	
	lee(B,x_4)
	x_4 := x_4 * 3
	escribe(B,x_4)

Ejecuciones serializables

- Separación de operaciones compatibles:

T_1	T_2
lee(A,x_1)	
$x_1 := x_1 * 10$	
escribe(A,x_1)	
	lee(A,x_3)
	$x_3 := x_3 * 5$
	escribe(A,x_3)
...	...

T_1	T_2
...	...
lee(B,x_2)	
$x_2 := x_2 * 3$	
escribe(B,x_2)	
	lee(B,x_4)
	$x_4 := x_4 * 3$
	escribe(B,x_4)

Ejecuciones serializables

- Permutación de operaciones permutables:

T_1	T_2
lee(A,x_1)	
x_1 := x_1 * 10	
escribe(A,x_1)	
lee(B,x_2)	
x_2 := x_2 * 3	
escribe(B,x_2)	
...	...

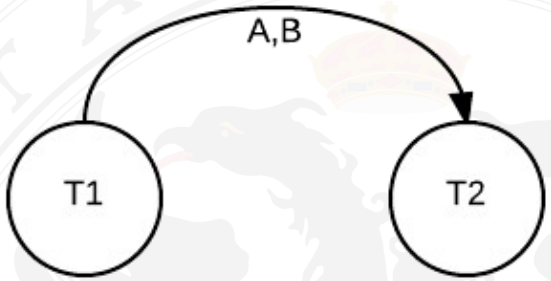
T_1	T_2
...	...
	lee(A,x_3)
	x_3 := x_3 * 5
	escribe(A,x_3)
	lee(B,x_4)
	x_4 := x_4 * 3
	escribe(B,x_4)

Grafo de precedencia

- Nodos: transacciones
- Arcos: restricciones en la ejecución
- T_i precede a T_j si, y sólo si, existen dos operaciones no permutables sobre el mismo átomo en las dos transacciones y la operación de T_i es anterior a la de T_j
- Habrá un arco entre dos transacciones si una precede a otra, y se marca el arco con el nombre del átomo implicado.

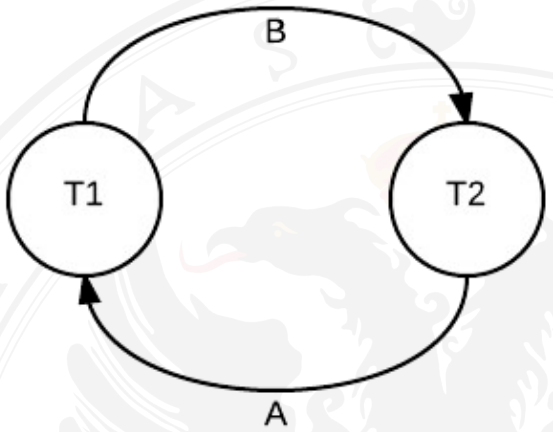
Grafo de precedencia

T1	T2
lee(A,x_i)	
x_i:=x_i+1	
escribe(A,x_i)	
	lee(A,z_i)
	z_i := z_i * 2
	escribe(A,z_i)
lee(B,x_j)	
x_j:=x_j+1	
escribe(B,x_j)	
	lee(B,z_j)
	z_j := z_j * 2
	escribe(B,z_j)



Grafo de precedencia

T1	T2
	lee(A,z_i)
	$z_i := z_i * 2$
lee(A,x_i)	
$x_i := x_i + 1$	
	escribe(A,z_i)
	lee(B,z_j)
	$z_j := z_j * 2$
escribe(A,x_i)	
lee(B,x_j)	
$x_j := x_j + 1$	
escribe(B,x_j)	
	escribe(B,z_j)



Grafo de precedencia

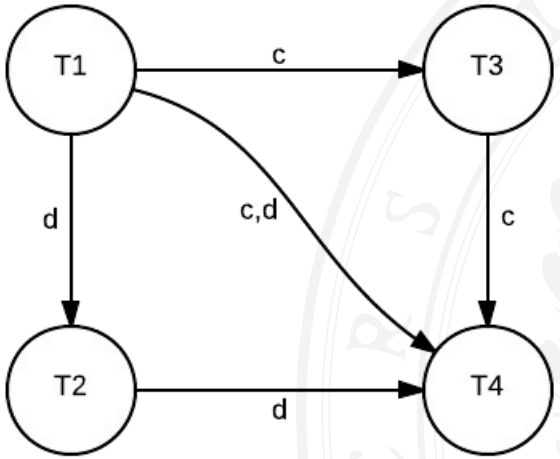
- Teorema:
  - Condición suficiente para que una ejecución sea serializable es que el grafo de dependencias no presente ciclos dirigidos.

Grafo de precedencia

- Un ejemplo:
  - lee(T_1,c), lee(T_2,b), lee(T_1,d), escribe(T_1,c), lee(T_3,c), escribe(T_3,c), lee(T_2,d), lee(T_3,a), lee(T_4,c), escribe(T_2,d), escribe(T_2,b), lee(T_4,d), escribe(T_4,d)

Grafo de precedencia

- Un ejemplo:
  - lee(T_1,c), lee(T_2,b), lee(T_1,d), escribe(T_1,c), lee(T_3,c), escribe(T_3,c), lee(T_2,d), lee(T_3,a), lee(T_4,c), escribe(T_2,d), escribe(T_2,b), lee(T_4,d), escribe(T_4,d)



Algoritmos de control de concurrencia

- Técnicas para garantizar el aislamiento de las transacciones:
  - Permitir la ejecución y deshacer las que produzcan conflicto: *técnicas de ordenación por marcas de tiempo*.
  - Evitar ciclos mediantes esperas: *técnicas de bloqueo*.

Técnicas de ordenación por marcas de tiempo

- Cada transacción recibe una marca de tiempo única cuando comienza.
- A cada átomo *x*, se asocia una referencia a la última transacción que opera sobre él: *R(x)*
- Algunos algoritmos de este tipo son:
  - Ordenación total
  - Ordenación parcial
  - Ordenación parcial multi-versión
  - Control de concurrencia mediante validación

Algoritmo de ordenación total

- Sean dos transacciones T_i y T_j con i y j marcas de tiempo tales que i<j, el algoritmo garantiza que T_i accede antes que T_j.
- Si una operación falla (ABORT), se deshace la transacción y se re-lanza más tarde, asignándole a la retrasada una referencia mayor que la de todas las actuales.
- **Problema**: las lecturas concurrentes también se ordenan, sin ser necesario por no ser conflictivas.

Algoritmo de ordenación total

```
Procedimiento lee (Ti, a);
   INICIO
      Si (R(a) <= i) entonces
        {Ejecutar la lectura}
        R(a) := i;
      si-no
        {ABORT: abortar la ejecucion}
      Fin-si
   FIN

Procedimiento escribe (Ti, a);
   INICIO
      Si (R(a) <= i) entonces
        {Ejecutar la escritura}
        R(a) := i;
      si-no
        {ABORT: abortar la ejecucion}
      Fin-si
   FIN
```

Algoritmo de ordenación parcial

- Sólo se ordenan las parejas de operaciones lee/escribe, escribe/lee y escribe/escribe.
- Cada átomo *x* tiene dos referencias: una para determinar la última transferencia que lo leyó *RR(x)* y otra para la que lo actualizó *WR(x)*.
- Si una operación falla (ABORT), se deshace la transacción y se re-lanza más tarde, asignándole a la retrasada una referencia mayor que la de todas las actuales.

Algoritmo de ordenación parcial

```
Procedimiento lee (Ti, a); {* escribe/lee *}
   INICIO
      Si (WR(a) <= i) entonces
        {Ejecutar la lectura}
        RR(a) := Max(RR(a),i);
      si-no
        {ABORT: abortar la ejecucion}
      Fin-si
   FIN

Procedimiento escribe (Ti, a); {* lee/escribe y escribe/escribe *}
   INICIO
      Si (RR(a) <= i) y (WR(a) <= i) entonces
        {Ejecutar la escritura}
        WR(a) := i;
      si-no
        {ABORT: abortar la ejecucion}
      Fin-si
   FIN
```

Algoritmo de ordenación parcial

Problema: hace trabajo en vano cuando *RR(a)* <= *i* < *WR(a)*, porque escribe información obsoleta.

Alternativa:

```
Procedimiento escribe (Ti, a); {* lee/escribe y escribe/escribe *}
   INICIO
      Si (RR(a) <= i) entonces
        Si (WR(a) <= i) entonces
           {Ejecutar la escritura}
           WR(a) := i;
        Fin-si
      si-no
        {ABORT: abortar la ejecucion}
      Fin-si
   FIN
```

Si la última que escribió es posterior, se ignora la operación de escritura.

Algoritmo de ordenación parcial multi-versión

- Persigue que las lecturas no aborten una transacción, pero implica que haya distintas versiones de cada átomo.
- Sólo habrá que buscar la última versión escrita con referencia menor que la de la transacción en curso.

Algoritmo de ordenación parcial multi-versión

```
Procedimiento lee (Ti, a);
   INICIO
      j := {ultima version de a};
      Repite mientras (WRj(a) > i)
        j := j - 1;
      Fin-repite
      {Ejecutar la lectura de la version j}
      RRj(a) := Max(RR(a),i);
   FIN
```

Algoritmo de ordenación parcial multi-versión

```
Procedimiento escribe (Ti, a);
   INICIO
      j := {ultima version de a};
      Repite mientras (RRj(a) > i)
        j := j - 1;
      Fin-repite
      Si (WRj(a) > i) entonces --Solo si 0 en lectura
        {ABORT: Abortar ejecucion}
      si-no
        {Ejecutar la escritura insertando una versión (j+1) de a}
        WRj+1(a) := i;
      Fin-si
   FIN
```

Control de concurrencia mediante validación

- Las transacciones se dividen en tres fases:
  - *Lectura*: donde se leen átomos, realizan cálculos y actualizan variables,
  - *Validación*: se comprueba la validez de los datos,
  - *Escritura*: se vuelcan los átomos al buffer.
- Para cada transacción, se guardan las marcas de tiempo para inicio, validación y fin.

Control de concurrencia mediante validación

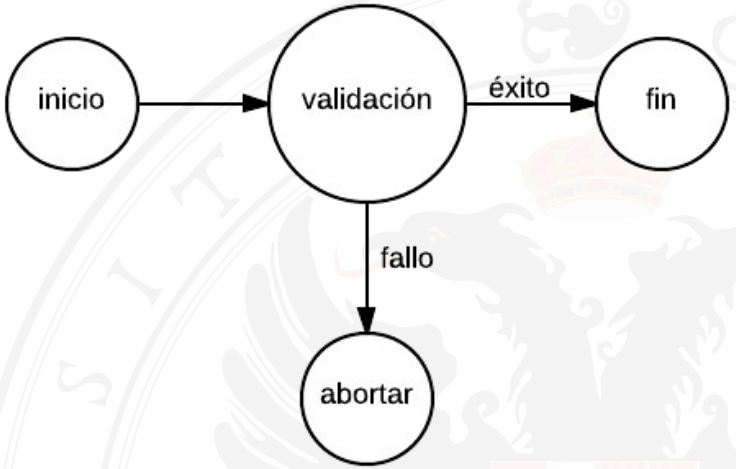
T_1
lee(A,x1)
lee(B,x2)
x1 := x1 * 2 + x2
escribe(A,x1)

Momentos

inicio (T_1)

valida (T_1)

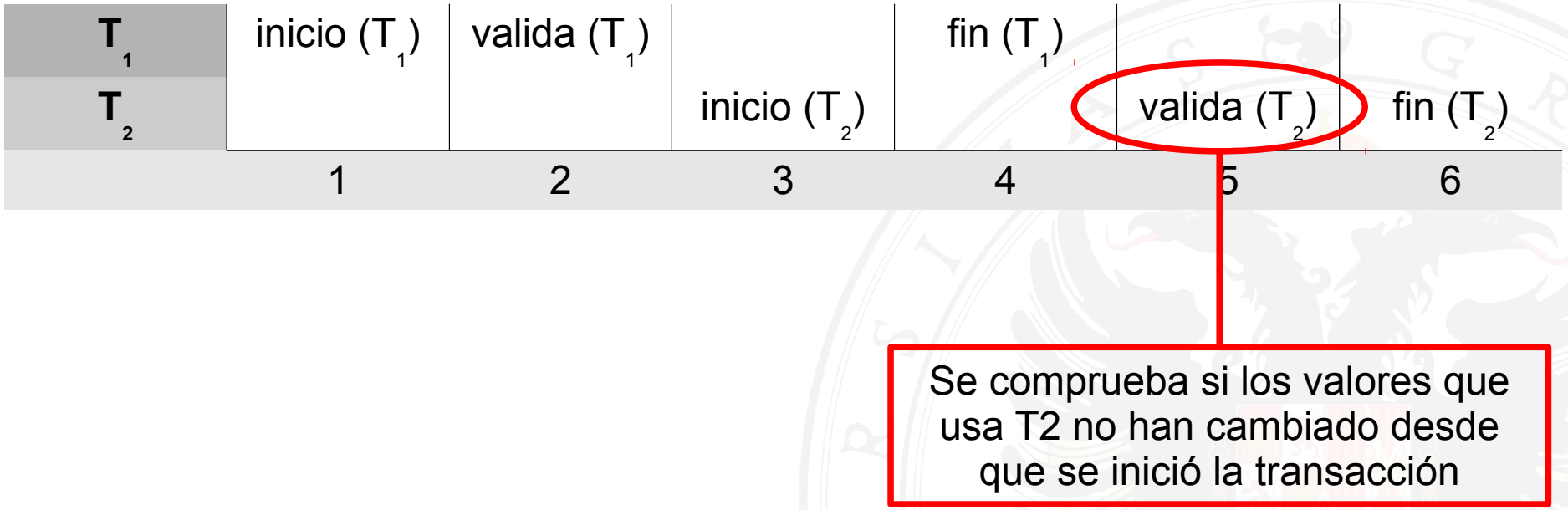
fin (T_1)



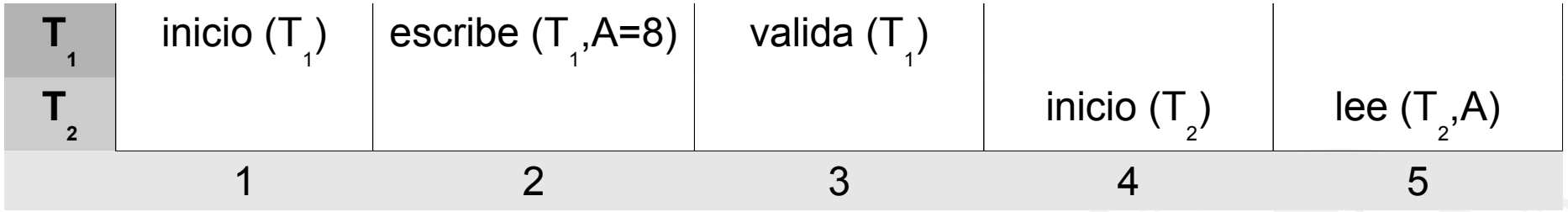
Ejemplo de entrelazado

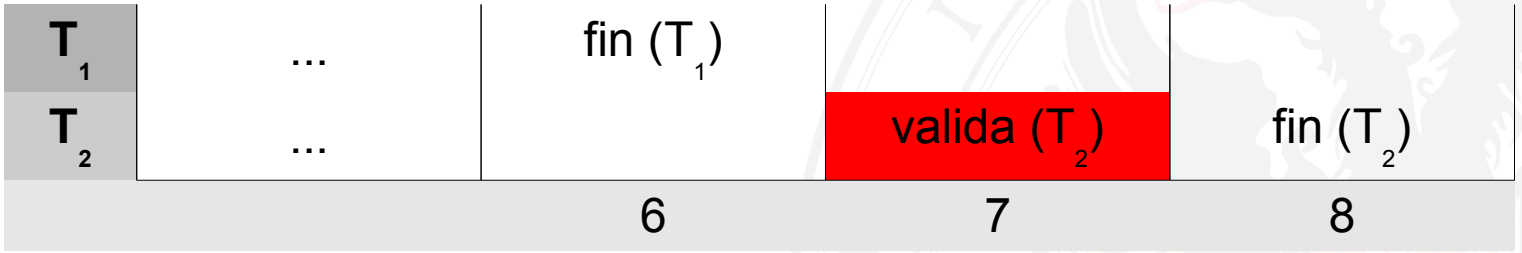
T_1	inicio (T_1)	valida (T_1)		fin (T_1)		
T_2			inicio (T_2)		valida (T_2)	fin (T_2)
	1	2	3	4	5	6

Ejemplo de entrelazado

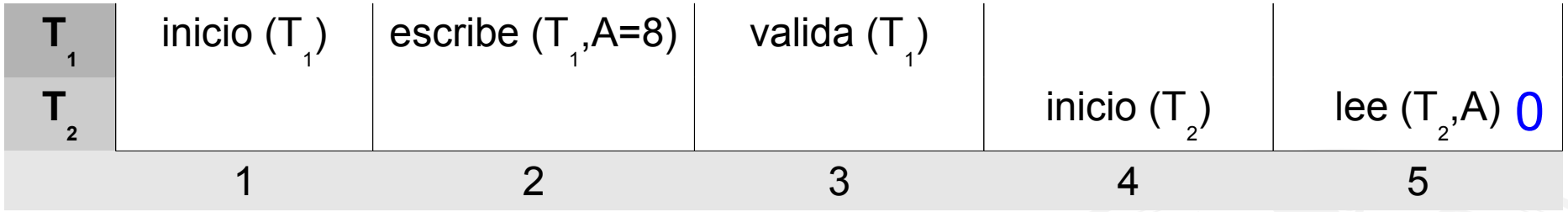


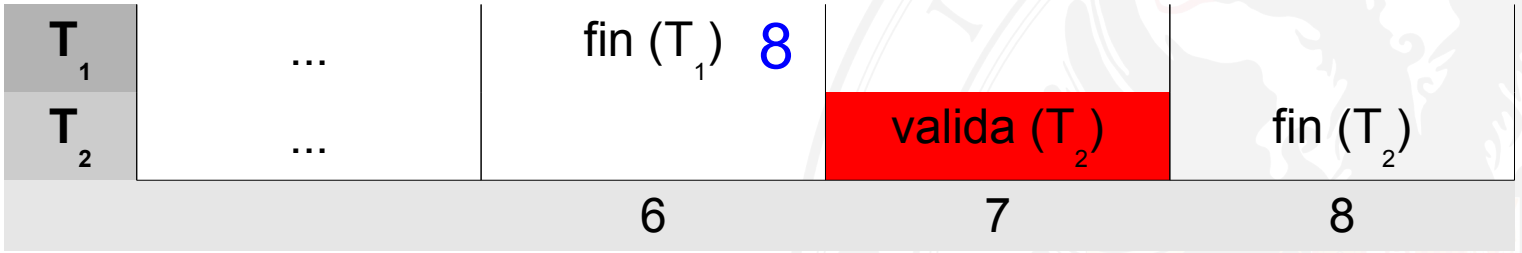
Situación conflictiva 1



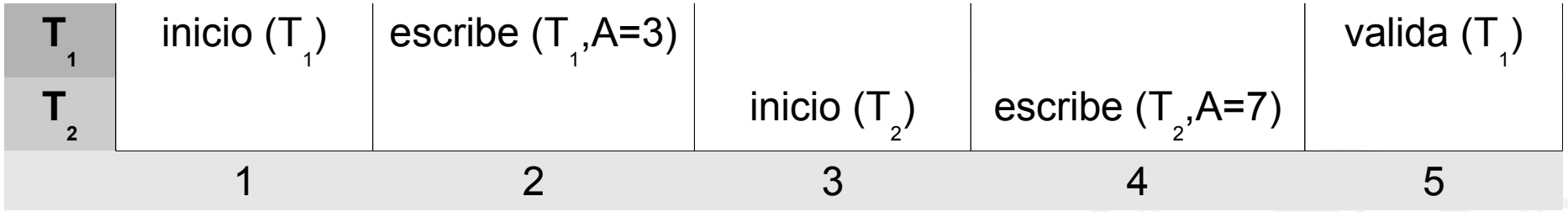


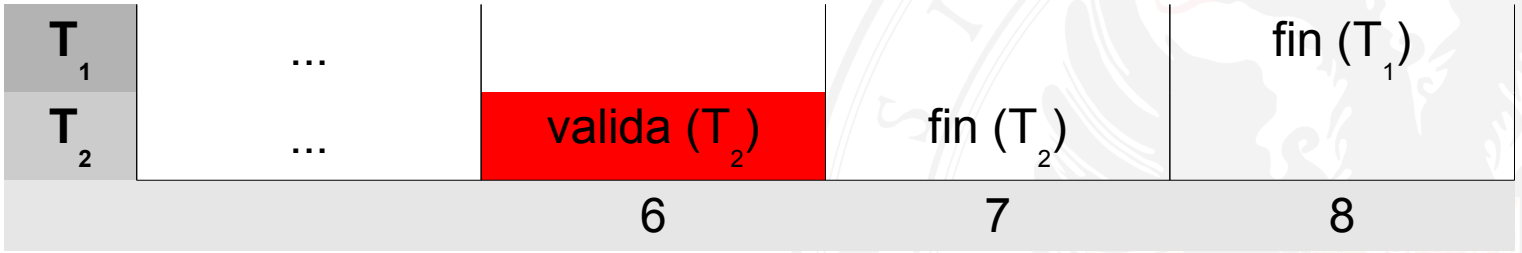
Situación conflictiva 1



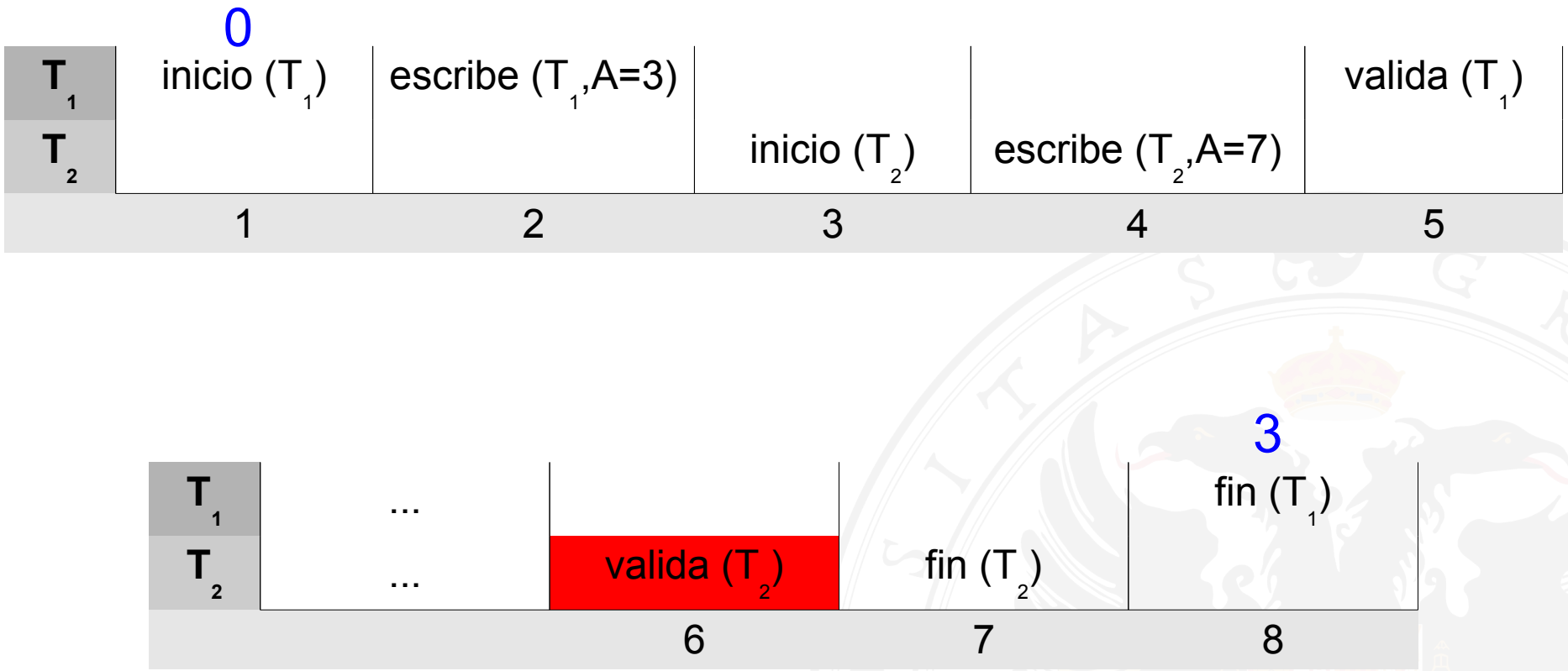


Situación conflictiva 2





Situación conflictiva 2



Comprobación de validez

```
Procedimiento COMPROBAR (i);
INICIO
   Para cada j tal que (inicio(i) < fin(j))
      /* Tj activas con marcas inicio o válida /
      Si (AR(i) ∩ AW(j) ≠ ∅) entonces
        {ABORT} /* Abortar la ejecucion */
      Fin-si
   Fin-para
   Para cada j tal que (fin(j) < valida(i))
      /* Tj activas con marca válida /
      Si (AW(i) ∩ AW(j) ≠ ∅) entonces
        {ABORT} /* Abortar la ejecucion de Ti */
      Fin-si
   Fin-para
FIN
```

Comprobación de validez

```
Procedimiento COMPROBAR (i);
INICIO
   Para cada j tal que (inicio(i) < fin(j))
      /* Tj activas con marcas inicio o válida /
      Si (AR(i) ∩ AW(j) ≠ ∅) entonces
        {ABORT} /* Abortar la ejecucion */
      Fin-si
   Fin-para
   Para cada j tal que (fin(j) < valida(i))
      /* Tj activas con marca válida /
      Si (AW(i) ∩ AW(j) ≠ ∅) entonces
        {ABORT} /* Abortar la ejecucion de Ti */
      Fin-si
   Fin-para
FIN
```

Transacciones

Átomos leídos

Átomos escritos

Un ejemplo

Plan	Etapas
lee (T_1, A)	inicio (T_1)
escribe (T_1, A)	
lee (T_2, B)	inicio (T_2)
escribe (T_1, C)	
	valida (T_1)
	fin (T_1)
lee (T_3, B)	inicio (T_3)
escribe (T_3, A)	valida (T_2)
	valida (T_3)
	fin (T_2)

Hasta ahora...

- Lo que hemos visto, planifica el retraso de transacciones que ejecutan accesos conflictivos y la repetición de operaciones ya ejecutadas.
- Estas técnicas detectan ejecuciones no serializables.

Técnicas de bloqueo

- Evitan ejecuciones incorrectas manteniendo en espera las transacciones con operaciones conflictivas sobre el mismo átomo.
- Permiten la ejecución simultánea de operaciones compatibles en base a su *modo*.
- **Modo de operación**:
  - Modos clásicos: lectura, actualización/escritura
  - Carácter: exclusivo, protegido
  - Propuestas CODASYL: *consulta no protegida M1* y *consulta protegida M2*, *actualización no protegida M3* y *actualización protegida M4*, *consulta exclusiva M5* y *actualización exclusiva M6*

Técnicas de bloqueo: compatibilidad

		Consulta		Actualización		Exclusiva	
		Prot.	No Prot.	Prot.	No Prot.	Consulta	Actualiz.
Consulta	Prot.	1	1	1	1	0	0
	No Prot.	1	1	0	0	0	0
Actualiz.	Prot.	1	0	1	0	0	0
	No Prot.	1	0	0	0	0	0
Exclus.	Consulta	0	0	0	0	0	0
	Actualiz.	0	0	0	0	0	0

Técnicas de bloqueo: compatibilidad

- El controlador sólo permite la ejecución de operaciones compatibles.
- Los *protocolos de bloqueo* (mecanismos) se basan en dos operaciones:
  - LOCK (a, M): bloqueo del átomo *a* en modo *M*, y
  - UNLOCK (a): desbloqueo del átomo *a*.

Técnicas de bloqueo: compatibilidad

- Se suele usar:
  - Un controlador por cada átomo
  - Un vector de 6 bits para cada átomo y transacción *A(a,i)* que almacena el acceso de la transacción *i* al átomo *a*.
  - Un vector de 6 bits por cada modo (seis vectores en total), con un 1 en la correspondiente posición.

Técnicas de bloqueo: compatibilidad

- Una operación de bloqueo *LOCK (a,M)* es compatible con los modos actuales de ejecución sii

$$M \subset \neg\left(\neg C \times \left(\cup_{i \neq p} A(a, i)\right)\right)$$

Técnicas de bloqueo: compatibilidad

- Una operación de bloqueo *LOCK (a,M)* es compatible con los modos actuales de ejecución sii

Unión de los vectores actuales sobre el átomo a

$$M \subset \neg\left(\neg C \times \left(\cup_{i \neq p} A(a, i)\right)\right)$$

Matriz de **incompatibilidades**

Técnicas de bloqueo: compatibilidad

- Una operación de bloqueo *LOCK (a,M)* es compatible con los modos actuales de ejecución sii

Vector de modos **incompatibles** con las con las ejecuciones actuales

$$M \subset \neg\left(\neg C \times \left(\cup_{i \neq p} A(a, i)\right)\right)$$

Técnicas de bloqueo: compatibilidad

- Una operación de bloqueo *LOCK (a,M)* es compatible con los modos actuales de ejecución sii

$$M \subset \neg\left(\neg C \times \left(\cup_{i \neq p} A(a, i)\right)\right)$$

Vector de modos **compatibles** con las con las ejecuciones actuales

Algoritmos de bloqueo

```
Procedimiento LOCK (a, M);
INICIO
   Si  M ⊂ ¬(¬C × ( ∪_{i≠p} A(a,i))) entonces
      A(a,p) := A(a,p) ∪ M;
   si-no
      Insertar (p, M) en Q(a);
      Bloquear p;
   Fin-si
FIN
Procedimiento UNLOCK (a);
INICIO
   A(a,p) := 0;
   Para cada (q, M') de Q(a) hacer
      Si  M' ⊂ ¬(¬C × ( ∪_{i≠p} A(a,i))) entonces
         A(a,q) := A(a,p) ∪ M';
         Extraer (q, M') de Q(a);
         Desbloquear q;
      Fin-si
   Fin-para
FIN
```

Algoritmos de bloqueo

```
Procedimiento LOCK (a, M);
INICIO
   Si  M ⊂ ¬(¬C × ( ∪_{i≠p} A(a,i))) entonces
      A(a,p) := A(a,p) ∪ M;
   si-no
      Insertar (p, M) en Q(a);
      Bloquear p;
   Fin-si
FIN
Procedimiento UNLOCK (a);
INICIO
   A(a,p) := (0);
   Para cada (q, M') de Q(a) hacer
      Si  M' ⊂ ¬(¬C × ( ∪_{i≠p} A(a,i))) entonces
         A(a,q) := A(a,q) ∪ M';
         Extraer (q, M') de Q(a);
         Desbloquear q;
      Fin-si
   Fin-para
FIN
```

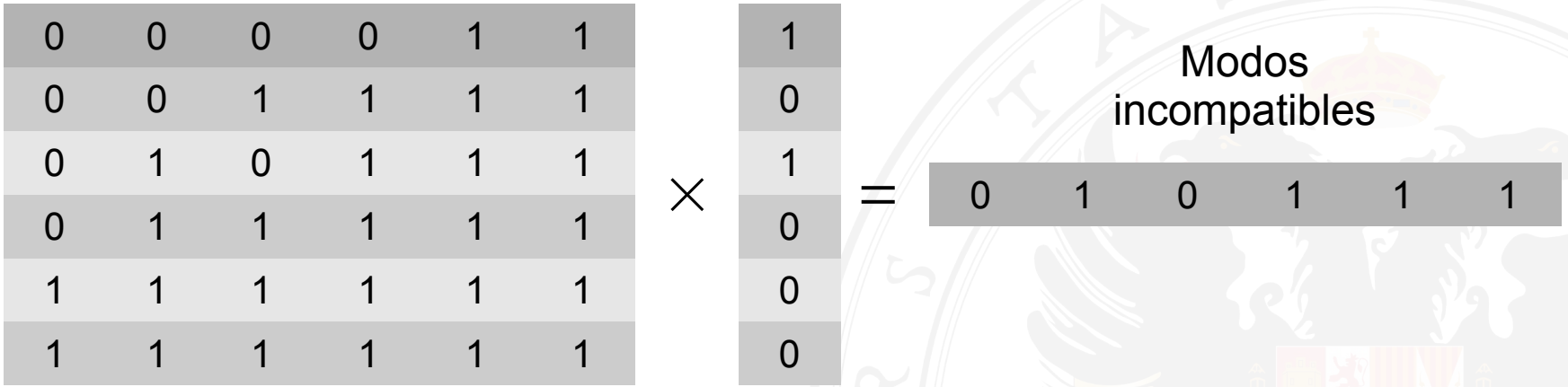
Transacción

Cola de transacciones bloqueadas y modos por operación incompatible sobre el átomo *a*

Transacción y modo de acceso a un átomo

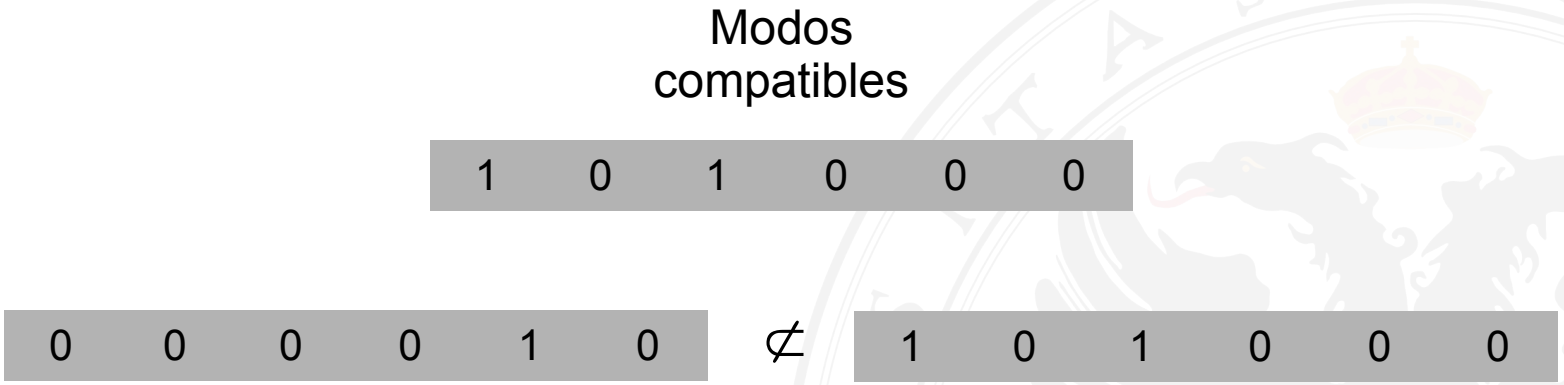
Algoritmos de bloqueo

- ¿Podría ejecutarse una operación con modo M5 si se están ejecutando una con modo M1 y otra con modo M3?



Algoritmos de bloqueo

- ¿Podría ejecutarse una operación con modo M5 si se están ejecutando una con modo M1 y otra con modo M3?



¡No puede!

Algoritmos de bloqueo

- Permiten la ejecución simultánea de operaciones compatibles sobre un átomo.
- Una transacción sólo se ejecuta si consigue el bloqueo de todos sus átomos.

Transacciones en dos fases

- Antes de acceder a un átomo, hay que bloquearlo.
- **Transacción en dos fases**: aquella que no ejecuta LOCK despues de un UNLOCK (primero los LOCKs y luego los UNLOCKs).
- Cuando una transacción alcanza el *bloqueo máximo*, comienza la ejecución de sus operaciones.
- El orden de ejecución de transacciones lo establece el instante en el que alcanzan el estado de bloqueo máximo.

Transacciones en dos fases

Toda ejecución completa de un conjunto de transacciones de dos fases es serializable.

Transacciones en dos fases

- Para que la ejecución concurrente sea serializable, toda transacción debe cumplir:
  - que las operaciones LOCK se hagan con el modo correcto y sobre el átomo necesario
  - que la operaciones UNLOCK sobre el átomo se realice una vez terminadas todas las operaciones sobre dicho átomo
  - que no se haga una operación LOCK después de ninguna operación UNLOCK

Técnicas de bloqueo: bloqueo mortal

- Se da un bloqueo mortal (o *deadlock*) cuando:
  - un grupo de transacciones están a la espera de que otras desbloqueen un átomo
  - la ejecución de transacciones no bloqueadas no desbloquea ningún átomo requerido por ninguna de las transacciones bloqueadas
- Posibles tratamientos de esta situación:
  - Prevenir o
  - Detectar

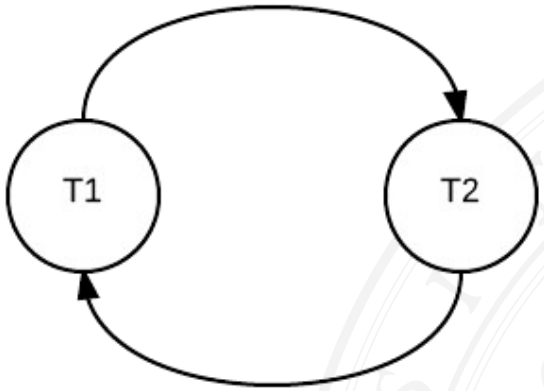
Técnicas de bloqueo: bloqueo mortal

- **Grafos de bloqueo**: grafos en los que los nodos son transacciones que acceden a átomos y los arcos determinan la relación de espera.

Se produce un bloqueo mortal si y sólo si el grafo de bloqueo contiene un ciclo

Técnicas de bloqueo: bloqueo mortal

- Ejemplo de grafo de bloqueo:



T1 espera a T2 (arco de T1 a T2) y T2 espera a T1 (arco de T2 a T1)

Técnicas de bloqueo: bloqueo mortal

- Prevención:
  - Ordenar las transacciones, de modo que no puede haber una transacción *antigua* esperando por una *nueva*
    - *DIE-WAIT*: una transacción sólo espera a otra si esta otra es más joven; si no, se repite desde el principio.
    - *WOUND-WAIT*: una transacción espera a otra más vieja; si no, mata a una transacción más joven

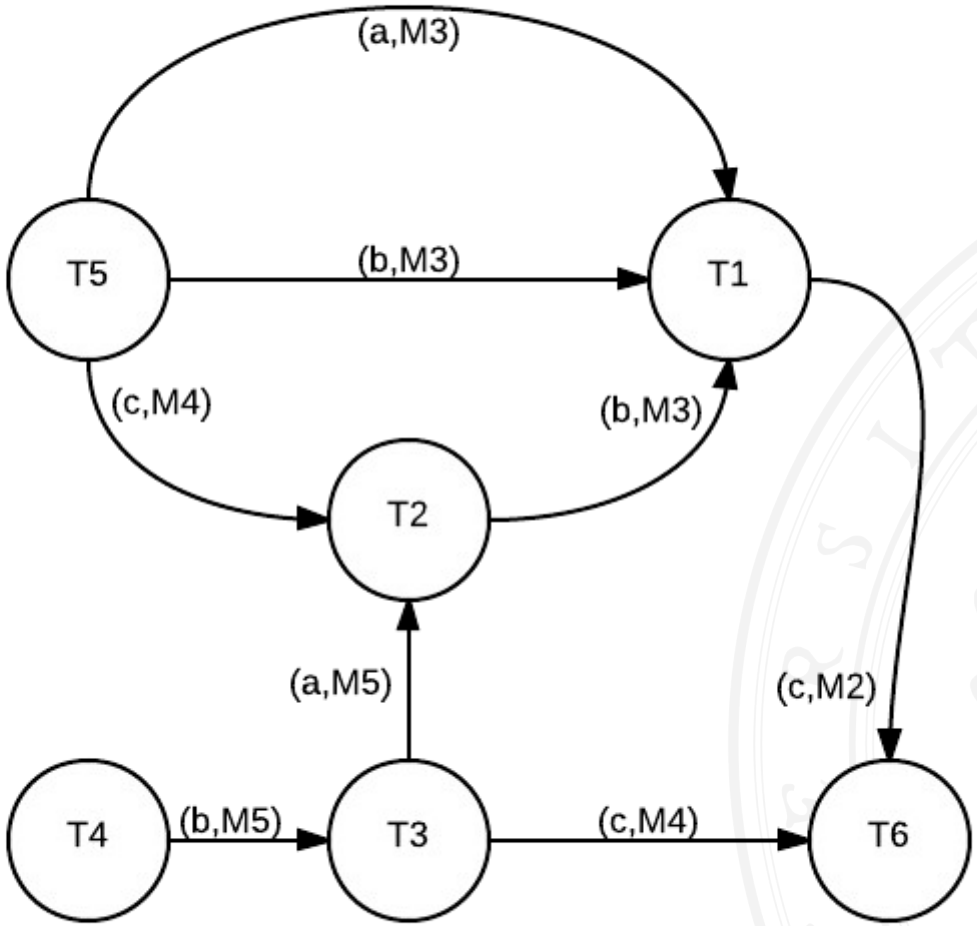
  Problema: aborta muchas transacciones no conflictivas realmente.

Técnicas de bloqueo: bloqueo mortal

- Detección:
  - Explorar el grafo para encontrar ciclos.
  - Si existe un ciclo, matar una transacción del ciclo:
    - la que bloquea el átomo más solicitado, o
    - la que bloquea el mayor número de átomos.

Técnicas de bloqueo: bloqueo mortal

Supongamos seis transacciones y tres átomos



N(1)=1
N(2)=1
N(3)=2
N(4)=1
N(5)=3
N(6)=0

Q(a)={(T5,M3),(T3,M5)}
Q(b)={(T4,M5),(T2,M3),
(T5,M3)}
Q(c)={(T5,M4),(T3,M4),
(T1,M2)}

Técnicas de bloqueo: bloqueo mortal

- N(k) = 0 supone que el nodo está *resuelto*.
- Los nodos resueltos no provocan bloqueo y se eliminan, para re-evaluar los que quedan.
- Para aplicar este método, hay que:
  - re-marcar nodos después de borrar uno,
  - Tener una función *SLOCK (a, k, j)* que determina si la operación *j* puede llevarse a cabo por la transacción *k* sobre el átomo *a* dado el estado actual del grafo.

Técnicas de bloqueo: bloqueo mortal

```
Función DETECTAR;
INICIO
   T = {transacciones j tales que N(j) = 0}
   R = {lista de átomos de las transacciones j incluidas en T}
   Repetir mientras aristas tachadas
      Para cada átomo i de R hacer
         Para cada par (Tk, Mj) en espera del átomo i hacer
            Si (SLOCK (i, k, j)) entonces
               {Tachar arista (ai,Mj) con origen en Tk};
               N(k) := N(k) - 1;
               Si (N(k) = 0) entonces
                  {Quitar Tk de T};
               Fin-si;
            Fin-si;
         Fin-para;
      Fin-para;
   Fin-repetir;
   Si (T = ∅) entonces
      Detectar = falso;
   si-no
      Detectar = verdadero;
   Fin-si;
FIN
```