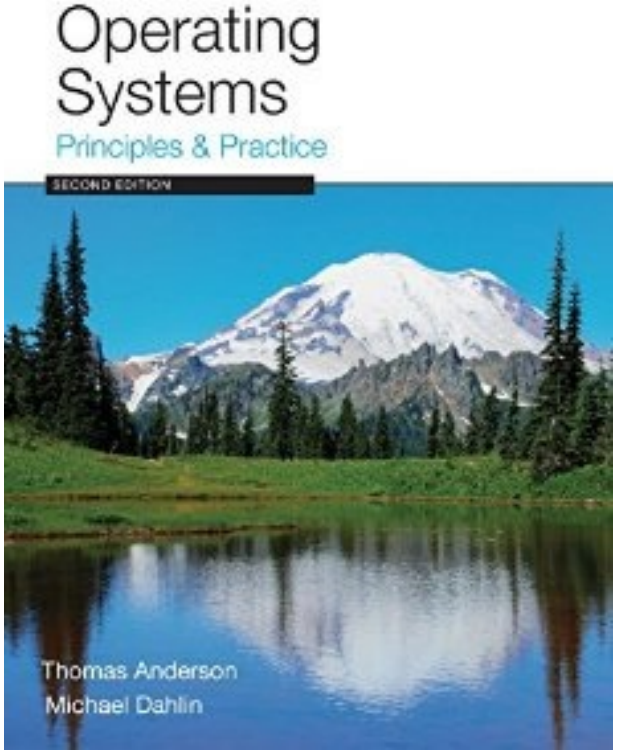
Sistemas Operativos

Capítulo 6 Deadlocks

Prof. Javier Cañas R.

Estos apuntes están tomados en parte del texto: “Operating System: Principles and Practice” de T. Anderson y M. Dahlin



Motivación

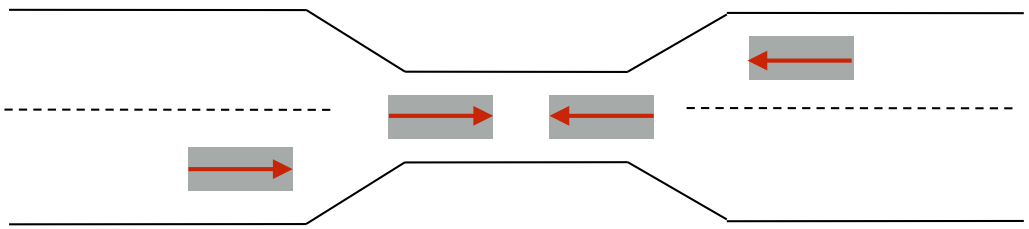
- Un desafío al desarrollo de aplicaciones multithread es la posibilidad de ocurrencia de deadlock.
- Un deadlock es un ciclo de espera entre un conjunto de threads, donde cada thread espera que algún otro thread realice alguna acción, pero esta acción no puede ocurrir.
- En general, un deadlock es conjunto de threads, cada uno reteniendo un recurso y esperando obtener un recurso que está retenido por otro thread dentro del conjunto.

Ejemplos

- Un sistema tiene 2 discos: T1 y T2 tienen asignado un disco y cada uno de ellos necesita otro.

- Un puente angosto:

Un puente angosto



Ejemplos

Thread A

```
lock1.acquire();
lock2.acquire();
lock2.release();
lock1.release();
```

Thread B

```
lock2.acquire();
lock1.acquire();
lock1.release();
lock2.release();
```

¿Bajo qué condiciones ocurre deadlock?

Ejemplos

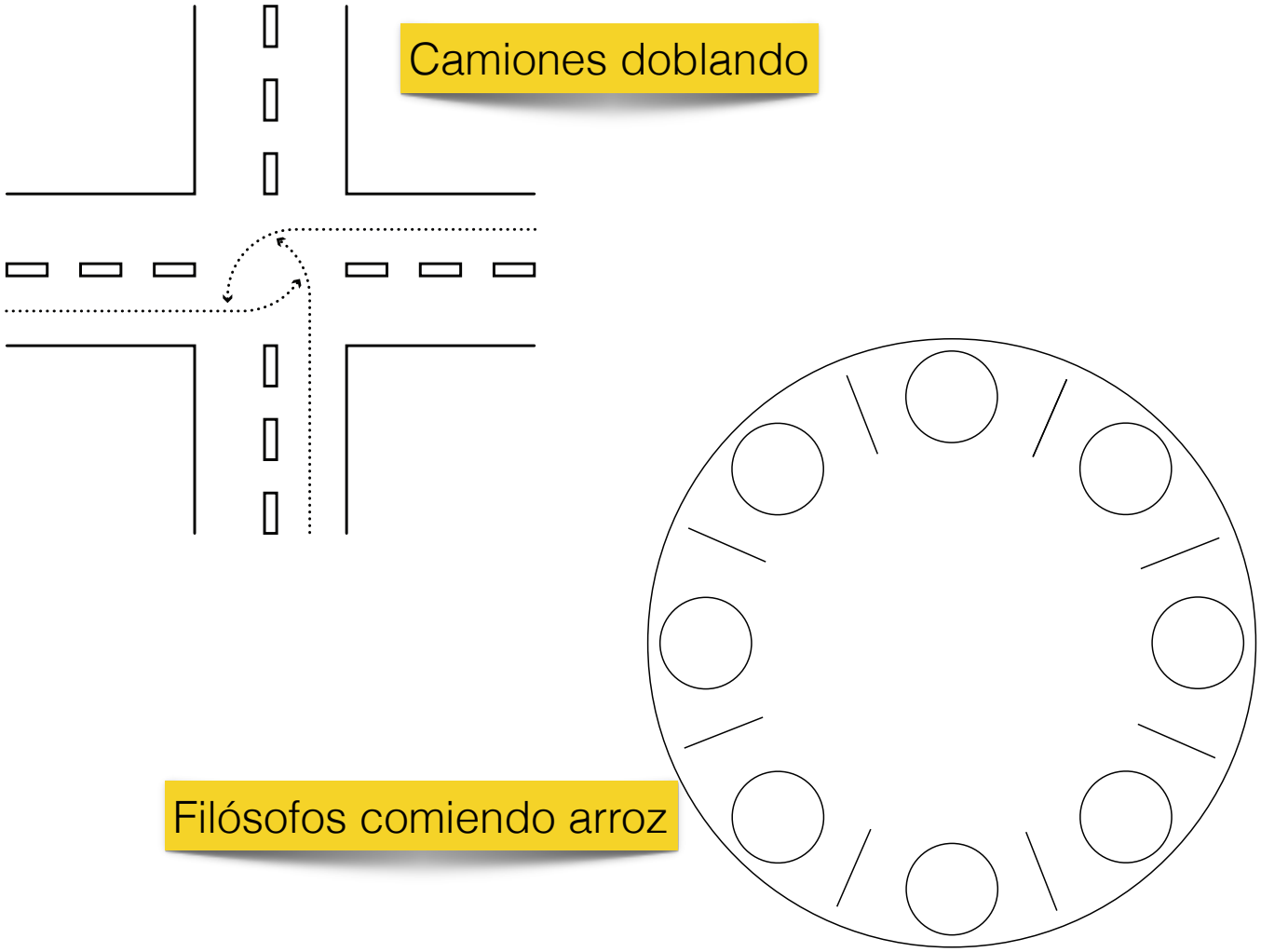
Thread A

lock1.acquire();
…
lock2.acquire();
while (necesito_esperar) {
    cv.wait(&lock2);
}
lock2.release();
…….
lock1.release();

Thread B

lock1.acquire();
……..
lock2.acquire();
……..
cv.signal()
lock2.release();
….
lock1.release();

En este caso, Thread A llama lock2 reteniendo lock1 y espera sobre cv la liberación de lock2. Ocurre deadlock si Thread B necesita lock1


Camiones doblando
Filósofos comiendo arroz

1 Deadlock y Starvation (Inanición)

- Ambos conceptos tienen relación con la vida de un proceso o thread.
- En starvation (inanición) un thread queda detenido por un tiempo indefinido.
- Un deadlock es también starvation, pero tiene una condición más fuerte: no hay como continuar. Deadlock implica starvation.
- Por ejemplo escritores y lectores: un escritor puede esperar un tiempo indefinido, si hay un lector y llegan nuevos lectores, o si hay un escritor y llegan más escritores.

… Deadlock y Starvation (Inanición)

- El problema es que un sistema sujeto a deadlock o starvation, puede comportarse normalmente, excepto para una poco probable combinación de circunstancias.
- Como el testing no descubre deadlock ni starvation, es necesario un esfuerzo de diseño.

2 Condiciones necesarias para deadlock

- Existen 4 condiciones necesarias (pero no suficientes) para que ocurra deadlock.
- Conocer estas condiciones es útil para diseñar soluciones. Prevenir que ocurra cualquiera de ellas, elimina el deadlock.

Las 4 condiciones

1. **Recursos limitados**: Hay un número finito de threads que simultáneamente pueden utilizar un recursos.

2. **No se puede quitar**: Una vez que el thread adquiere un recurso, éste no se puede quitar externamente.

3. **Espera y retención**: Un thread mantiene un recuso mientras espera por otro.

4. **Espera circular**: Existe un conjunto de threads que esperan, y cada thread espera por un recuso que tiene otro thread.

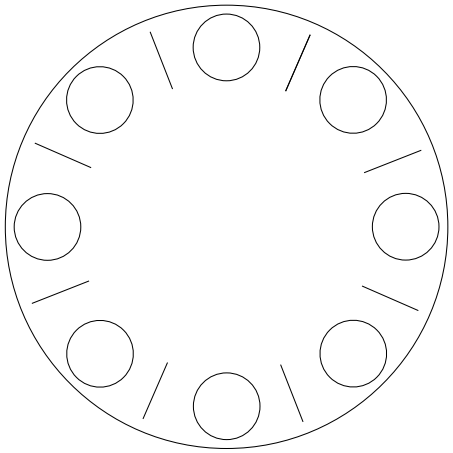
Ejemplo: filósofos

1. **Recursos limitados**: Cada palillo puede ser retenido sólo por un filósofo a la vez.

2. **No se puede quitar**: Una vez que el filósofo toma un palillo, no se le puede quitar

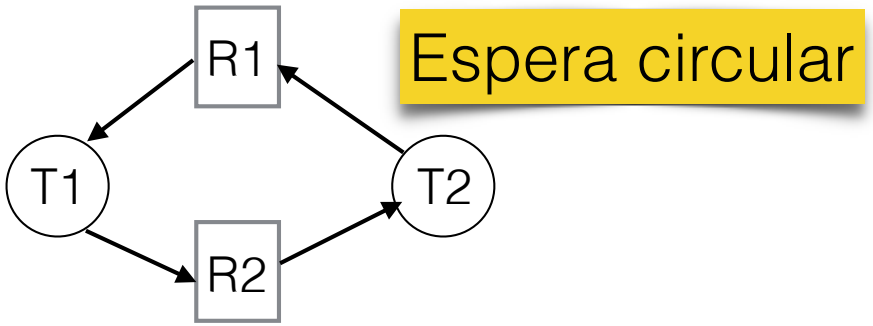
3. **Espera y retención**: Si espera por un palillo, no suelta el que ya tiene.

4. **Espera circular**: Existe un conjunto de threads que esperan donde cada thread espera por un recuso que tiene otro thread.



Grafo de asignación

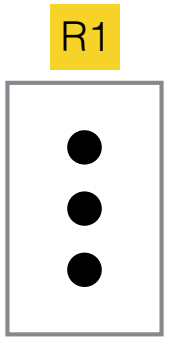
- T={T1, T2, ..., Tn}, todos los Thread
- R={R1, R2, ..., Rm}, todos recursos
- Arco de solicitud (request): Ti → Rj
- Arco de asignación: Rj → Ti



Espera circular

…. Grafo de asignación

- Un recurso con varias instancias se representa por:

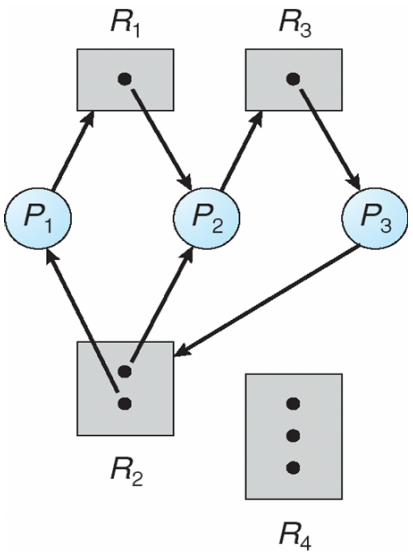


R1 tiene 3 instancias

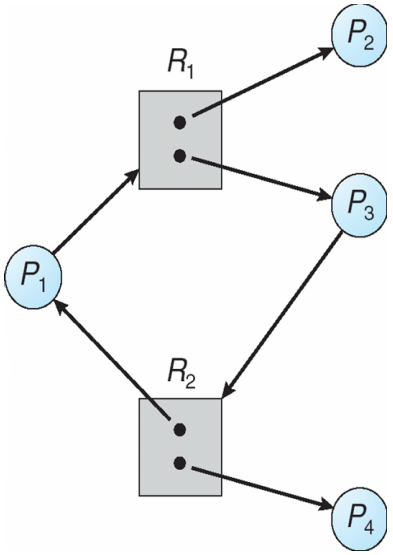
- Cuando hay más de una instancia por recurso, pueden existir ciclos, pero no necesariamente deadlocks.

- Los siguientes ejemplos muestras ciclos con y sin deadlocks.

Múltiples instancias de un recurso



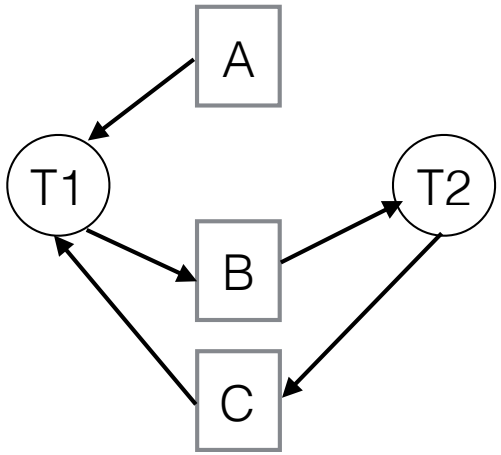
Con deadlock



Sin deadlock

3 Prevención

- ¿Cómo evitar?



	Thread1	**Thread2**
1	Adquiere A	
2		Adquiere B
3	Adquiere C	
4		Espera C
5	Espera B	

- La espera circular ocurre en el paso 5, pero la suerte ya está sellada cuando en el paso 2 el thread 2 adquiere B.

Mecanismos de prevención

A. **Limitar** el comportamiento del programa para prevenir una de las 4 condiciones necesarias.

B. **Predecir el futuro**: Si se sabe lo que harán los threads se pueden evitar deadlocks haciendo que los threads esperen.

C. **Detectar y corregir**: permitir que los threads se recuperen del deadlock haciendo un undo.

D. **Lo que** (casi) **todos hacen**

A) Limitar comportamiento de programas

- Proporcionar suficientes recursos. Por ejemplo entregar más palillos a los filósofos.
- Eliminar la retención y espera: Soltar el lock cuando se está llamando a otros módulos. Esto significa reestructurar el módulo.
- Eliminar espera circular: Siempre adquirir locks en un orden fijo

B) Predecir el futuro

- Se utiliza el algoritmo del banquero
  - Establecer anticipadamente los máximos recursos que se necesitarán.
  - Asignar recursos dinámicamente. Esperar si al satisfacer un pedido, puede llevar a deadlock.
  - El requerimiento puede ser satisfecho si es posible garantizar algún orden de threads que sea libre de deadlock.

C) Detección y Reparación

- En vez de prevenir, algunos sistemas permiten que ocurran, los detectan y reparan.
- Se pueden detectar deadlocks analizando grafos, cuando existe sólo una instancia de cada recurso.
- Cuando existen recursos que tienen más de una instancia se utiliza una variación del algoritmo del banquero.

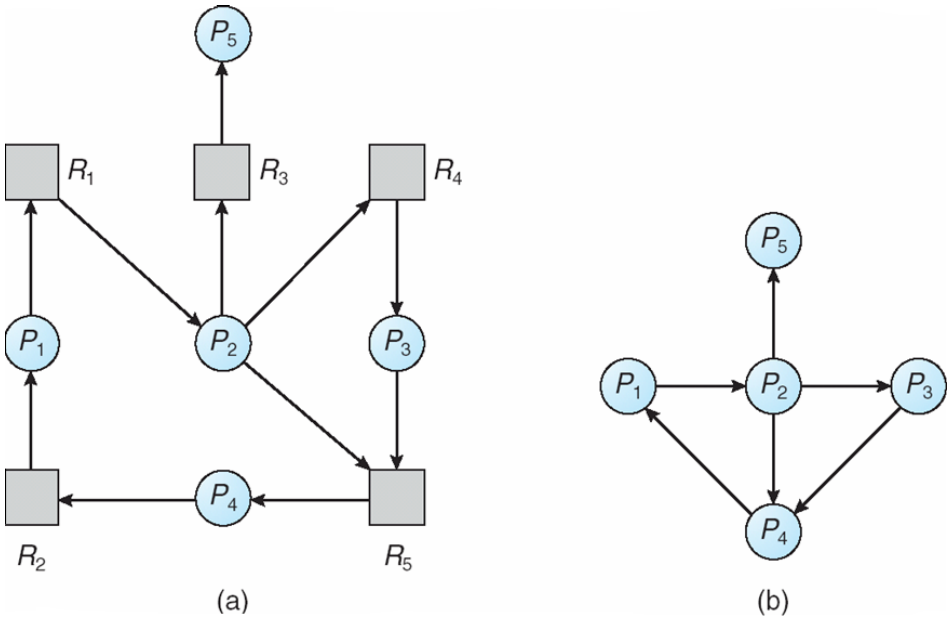
D) Suponer que nunca van a ocurrir

Linux, Windows

Detectar y Recuperar: Grafo wait-for

- Mantenga el grafo “espera” (wait-for)
  - Nodos son threads
  - Ti → Tj si Ti espera por Tj
- Periódicamente invocar un algoritmo que busca ciclos en un grafo. Si hay ciclos, existe deadlock.
- Un algoritmo para detectar un ciclo en un grafo requiere una complejidad O(E+V).

Grafo de asignación de recursos y grafo “Wait-for”



Grafo Resource-Allocation

Grafo wait-for correspondiente

4 Algoritmo del banquero para evitar deadlocks

- Una técnica de evitar deadlocks es esperar hasta que todas las necesidades de recursos sean satisfechas, sin retener recursos. Una vez que esto se cumple, atómicamente se asignan.

- Un thread no sabe lo que necesitará en forma exacta, pero si puede adquirir todos los recursos que *podría* necesitar.

- El algoritmo del banquero se basa en esta idea optimizándola. Fue desarrollado por Dijkstra.

… Algoritmo del banquero

- En este algoritmo, un thread al comenzar su tarea establece sus requerimientos máximos de recursos, pero los va adquiriendo y liberando en forma incremental.
- La idea general es que el sistema que puede tener deadlock, no necesariamente lo tendrá. Dependerá como se intercalen los requerimientos

Estados de un proceso

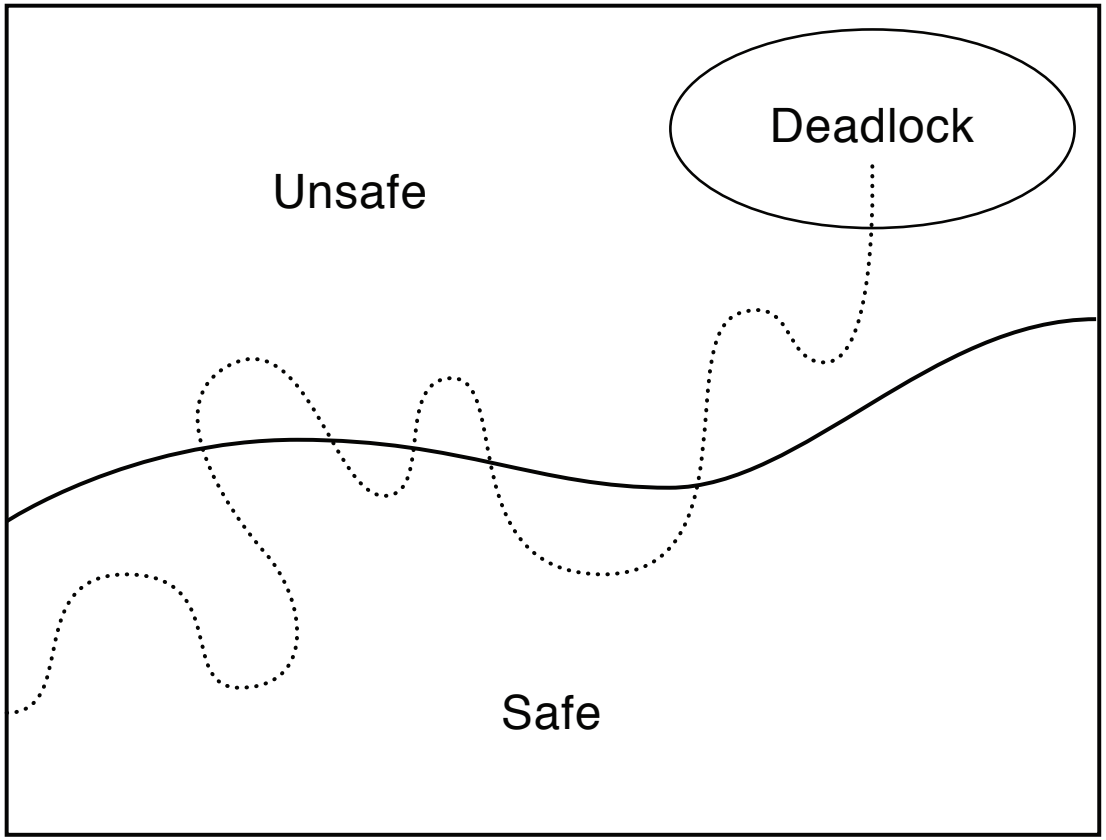
- Un sistema susceptible a deadlocks, puede estar en uno de tres estados:

- **Estado seguro**: para cualquier secuencia posible de requerimientos de recursos, existe al menos una secuencia segura de procesar los requerimientos que eventualmente tenga éxito en satisfacer todos los requerimientos pendientes y futuros. Puede requerir esperar que algunos recursos estén disponibles.

- **Estado inseguro**: Existe al menos una secuencia de requerimientos futuros de recursos que lleven a deadlock en forma independiente del orden de procesarlos.

- **Deadlock**: el sistema tiene al menos un deadlock.

Estados posibles de un proceso



Ejemplo

- Consideremos un sistema con 12 cintas magnéticas y 3 threads: T0, T1, T2
- Supongamos que en t_0 la situación es:

	Maximum Needs	Current
T0	10	5
T1	4	2
T2	9	2

La secuencia <T1, T0, T2> ¿es segura?

El Algoritmo

- El **algoritmo del banquero** mantiene el sistema en un estado seguro. Lo hace otorgando un recurso si y sólo si resulta en un estado seguro.

- La suma de las máximas necesidades recursos de los threads puede ser mayor que el total de recursos.
  - Siempre que exista una forma en que todos los threads puedan terminar sin caer en deadlock.

- Proceder si y sólo si:
  - Max es el total que el thread necesita para terminar.
  - El total de recursos disponibles - # asignados ≥ Max

Pseudo-código

Estados del Sistema

```
Class ResourceMgr{
    private:
    Lock lock;
    CV cv;
    int r;  //Numero de recursos
    int t;  //Numero de threads
    int avail[]; // avail[i]: instancias del recurso i disponibles
    int max[][]; //max[i][j]: max de recurso i requerido por thread j
    int alloc[][]; //alloc[i][j]: asignación actual de recurso i a thread j
    ....
    }
```

. . . Pseudo-código

Lógica de alto nivel del algoritmo

```
//Invariante: el sistema está en estado seguro
ResourceMgr::Request(int resourceID, int threadID) {
    lock.acquire();
    assert(isSafe());
    while (!wouldBeSafe(resourceID, threadID)) {
        cv.wait(&lock);
    }
    alloc[resourceID][threadID]++;
    avail[resourceID]--;
    assert(isSafe());
    lock.release();
}
```

Función isSafe()

```
/* Un estado es seguro si y sólo si, existe una secuencia segura de
concesiones suficiente para que todos los threads reciban
los máximos recursos que necesitan*/

bool
ResourceMgr::isSafe() {
    int j;
    int toBeAvail[]= copy avail[];
    int need[][]=max[][] - alloc[][];

    bool finish[]=[F, F, F, ....]; //finish[j] es T si thread j está
                                   //garantizado que puede terminar
                                   // F: false, T:true


    while (T) {
        j=cualquier threadID que cumpla:
            (finish[j]==F) && forall i: need[i][j] <= toBeAvail[i];
        if (no existe este j) {
           if (forall j: finish[j]==T)
                return T;
           else
                return F;
           }
         else { // el Thread j termina y devuelve su asignación actual al pool
             finish[j] = T;
             forall i: toBeAvail[i] = toBeAvail[i]+ alloc[i][j];
             }
    }
}
```

Función wouldBeSafe

```
// Hipotéticamente satisface pedido y observa si
// resulta en un estado seguro
bool
ResourceMgr::wouldBeSAfe(int resourceID, int threadID) {
    bool result = F;

    avail[resourceID]--;
    alloc[resourceID][threadID]++;
    if (isSafe()) {
        result = T;
        }
    avail[resourceID]++;
    alloc[resourceID][threadID]--;
    return result;
}
```

Ejercicio

- El estado actual de una aplicación se representa por las siguientes matrices:

threads

$$\mathbf{alloc} = \begin{pmatrix} 0 & 2 & 3 & 2 & 0 \\ 1 & 0 & 0 & 1 & 0 \\ 0 & 0 & 2 & 1 & 2 \end{pmatrix}$$ recursos

$$\mathbf{max} = \begin{pmatrix} 7 & 3 & 9 & 2 & 4 \\ 5 & 2 & 0 & 2 & 3 \\ 3 & 2 & 2 & 2 & 3 \end{pmatrix}$$

$$\mathbf{avail} = \begin{pmatrix} 3 & 3 & 2 \end{pmatrix}$$

- ¿Es seguro el estado actual?
- ¿Puede el thread 1 solicitar un recurso 2?

Scripts en Octave

```
%sistema.m
%Datos de prueba. Corresponden a ejemplo del
% Anderson 2 edicion,
Allocation= [0, 2, 3, 2, 0; 1, 0, 0, 1, 0; 0, 0, 2, 1, 2];
Max=[7, 3, 9, 2, 4; 5, 2, 0, 2, 3; 3, 2, 2, 2, 3]
Available=[3,3,2];
```

```
%Este es el programa principal que utiliza las demas funciones
%Banker.m
%Javier Canas
clc
sistema; %carga configuracion inicial
disp('Configuracion del Sistema')
Allocation
Max
Available
if is_safe(Allocation, Max, Available)
   disp('Sistema seguro')
else
   disp('Sistema inseguro')
end
```

```
function [code] = is_safe(Alloc, Max, Avail)
% Safety Algorithm
% Bankers Algorithm for deadlock detection
% Javier Canas  12/11/2015
% code =1 safe
% code =0 unsafe
% code = -1 error
code=1;
[n,m]=size(Alloc);
[l,p]=size(Max);
if n ~=l | m ~= p
   disp('Error en tamano de matrices')
   code=-1; return
end
Need=Max - Alloc
Work=Avail'
Finish=false(1,m)
while ~all(Finish)
    i=find_index(Finish, Need, Work)
    if ~i
       break
    end
    Work=Work+Alloc(:,i)
    disp(i)
    Finish(i)=true;
end
code=all(Finish); return
```

```
function [existe] = find_index(A,B,C)
%encuentra indice tal que A(i)==F y B(i) <=C
        existe=0;
        [x,y]=size(B);
        for i=1:y
          if  ~A(i) && all(B(:,i) <= C)
               existe=i;  return
          end
        end
```

```
function code=wouldbesafe(rid, tid,alloc, max, avail)
%Hipoteticamente solicita un recurso
%y ve si resulta en un estado seguro
%El thread tid, necesita un recurso adicional rid
code=0;
avail(rid)=avail(rid)-1
alloc(rid,tid)=alloc(rid,tid)+1
if( is_safe(alloc, max, avail))
      code=1;
end
%como los arreglos son variables locales
%no es necesario restituir su valor original
```

Ejemplo de aplicación

- Se tiene un sistema con 8 páginas de memoria y 3 procesos: A, B y C que necesitan 4, 5 y 5 páginas respectivamente para terminar.

- Al entregar páginas arbitrariamente se llega a deadlock:

Proceso	**Asignación**											
A	0	**1**	1	1	**2**	2	2	**3**	3	3	wait	wait
B	0	0	**1**	1	1	**2**	2	2	**3**	3	3	wait
C	0	0	0	**1**	1	1	**2**	2	2	wait	wait	wait
Total	0	1	2	3	4	5	6	7	8	8	8	8

…Ejemplo de aplicación

- Ahora el sistema sigue el algoritmo del banquero:

Proceso	Asignación																		
A	0	**1**	1	1	**2**	2	2	**3**	3	3	**4**	0	0	0	0	0	0	0	0
B	0	0	**1**	1	1	**2**	2	2	W	W	W	W	**3**	**4**	4	**5**	**0**	0	0
C	0	0	0	**1**	1	1	**2**	2	2	W	W	W	**3**	3	W	W	**4**	**5**	**0**
Total	0	1	2	3	4	5	6	7	7	7	8	4	6	7	7	8	4	5	0

w: wait

- Retardando B y C, A puede terminar, devolver recursos y permitir que B termine.

Sistemas Operativos

Capítulo 6 Deadlocks

Prof. Javier Cañas R.