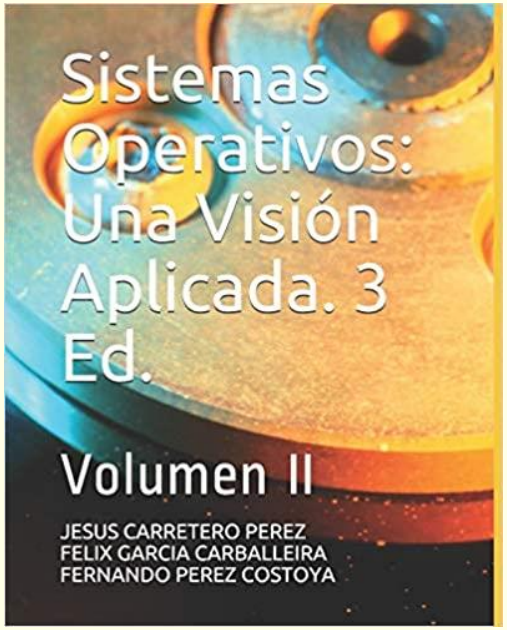
# Sistemas operativos avanzados

## Interbloqueos (extracto)




**Capítulo 7 del libro**

# Contenido

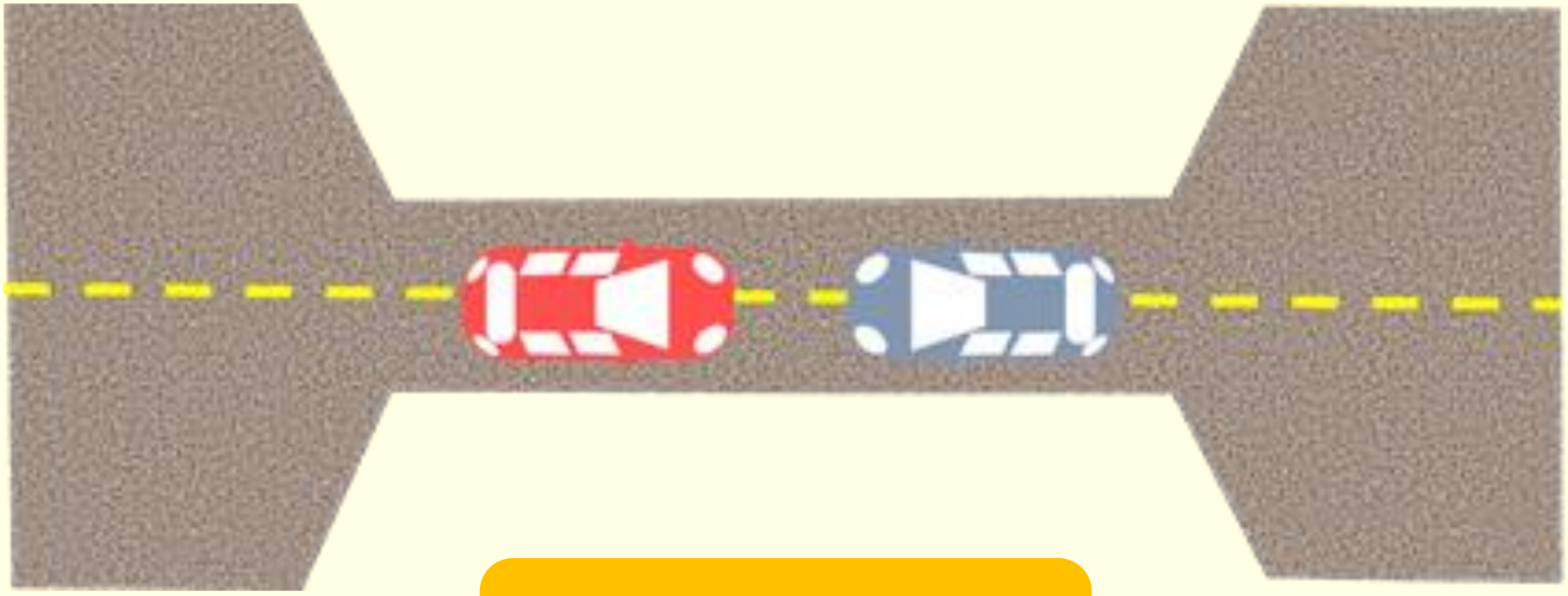
- Introducción
- Escenarios de interbloqueos
- Definición de interbloqueo
- Modelo básico del sistema
- Tratamiento del interbloqueo
- Detección y recuperación del interbloqueo
- Prevención del interbloqueo
- Predicción del interbloqueo
- Tratamiento del interbloqueo en los sistemas operativos
- Diseño de aplicaciones concurrentes libres de interbloqueos

# Introducción

- Procesos compiten por recursos y se comunican
  - SO ofrece servicios para ello
  - Sin embargo, no es suficiente
- Puede haber conflictos que pueden causar bloqueo indefinido

  $\Rightarrow$ **Interbloqueo** (*deadlock*)
- Conocido y estudiado desde hace mucho tiempo
  - Importante desarrollo teórico pero
  - Aplicación limitada en los SSOO
    - SSOO diseñados sin interbloqueos internos pero
    - Ofrecen servicios que no están libres de interbloqueos
- Aparece en otras disciplinas informáticas (p.e. BB.DD.)

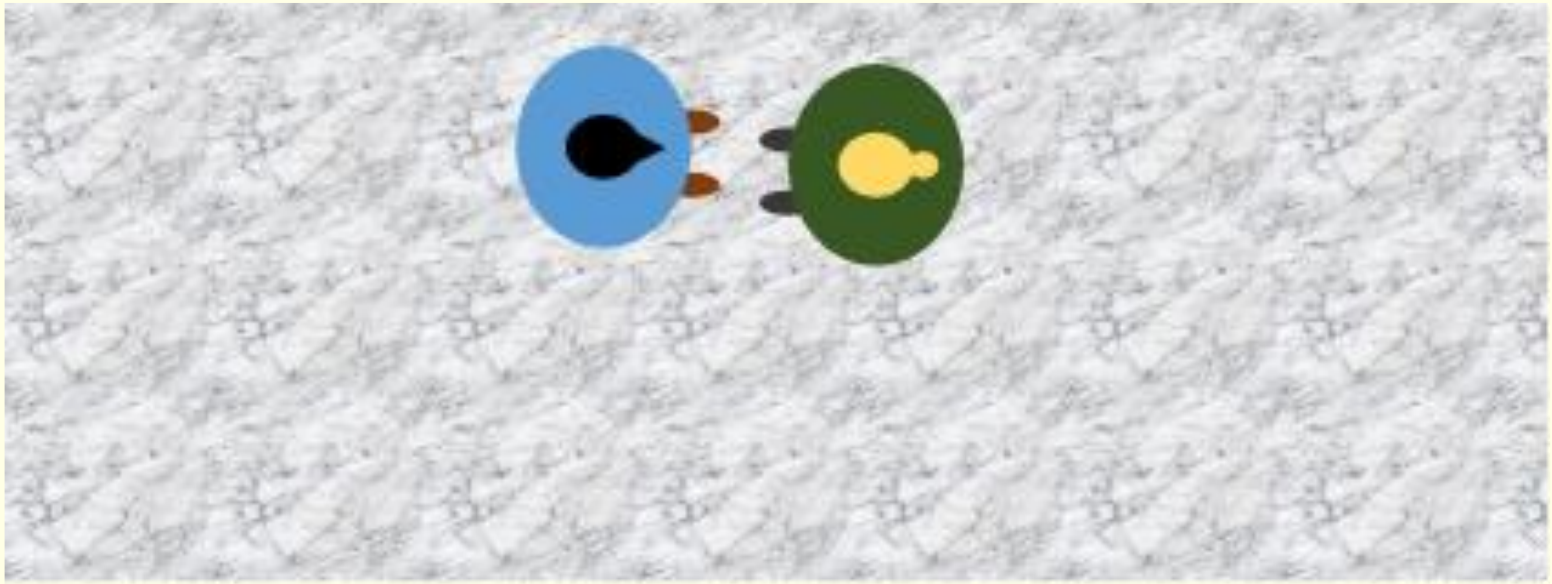
# Problema basado en hechos reales



**Interbloqueo**

**Bloqueo permanente**

# *Deadlock* (interbloqueo) vs *Livelock*



***Livelock***

**Reintentos continuos**

**que podrían no progresar**

# Escenarios de interbloqueos

- Elementos que intervienen:
  - Entidades activas: usamos término “proceso” pero pueden ser:
    - *threads*, activaciones SO (llamadas, interrupciones, excepciones)
  - Recursos de uso exclusivo
  - Acciones de las entidades activas sobre los recursos
- Escenarios:
  - Procesos independientes compitiendo por recursos
    - Responsable de evitar interbloqueos: SO
  - Procesos/*threads* cooperantes (aplicación concurrente)
    - Interbloqueo: error de programación de la aplicación
  - Actividades concurrentes internas del SO
    - Interbloqueo: error de programación del SO

# Procesos independientes

**Proceso $P_1$**

```
Solicita(C)
Uso del recurso C
Solicita(I)
Uso de ambos recursos
Libera(I)
Uso del recurso C
Libera(C)
```

**Proceso $P_2$**

```
Solicita(I)
Uso del recurso I
Solicita(C)
Uso de ambos recursos
Libera(C)
Uso del recurso I
Libera(I)
```

**Recursos físicos:**
**C: cinta**
**I: impresora**

**Traza ejecución con Interbloqueo ABBA**

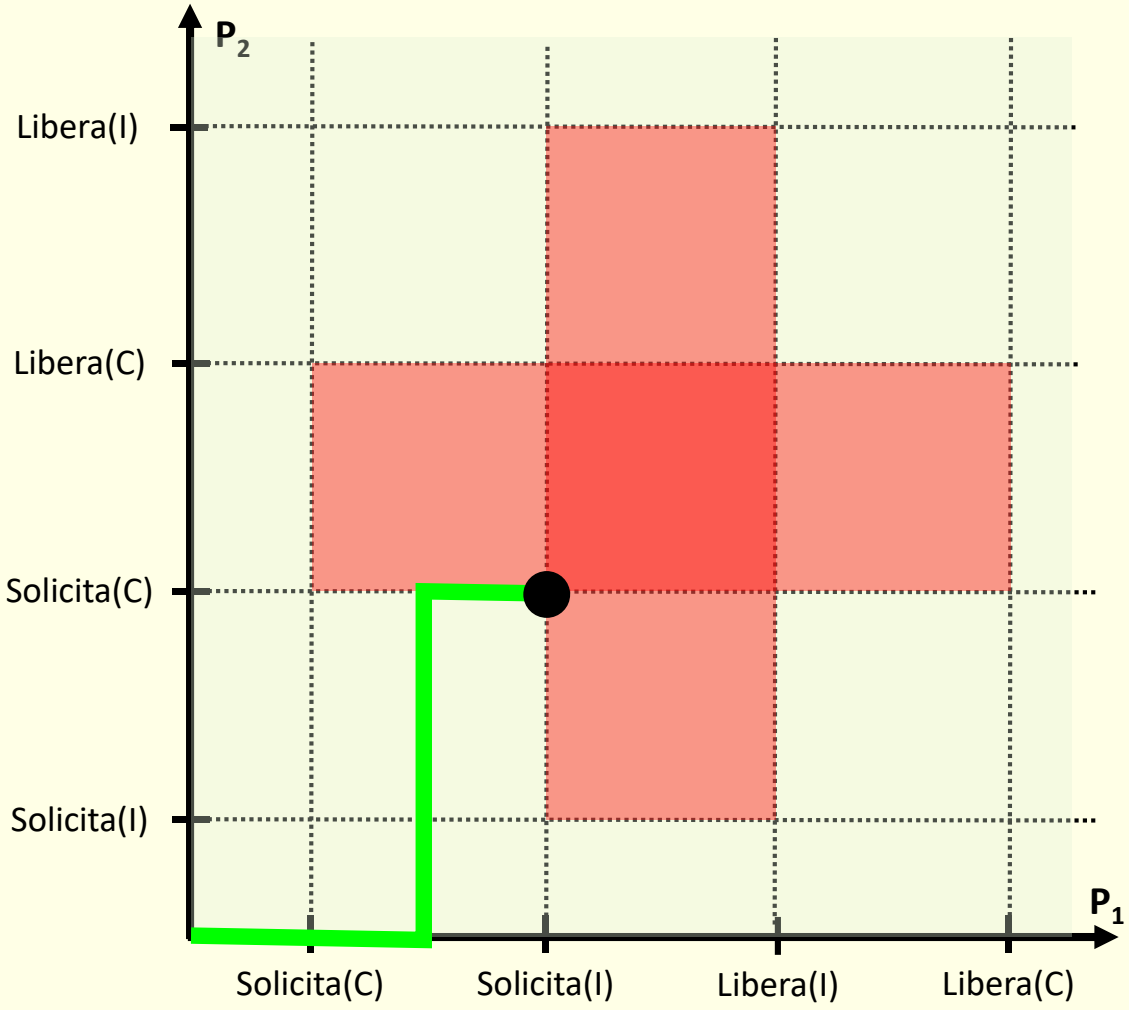
1. $P_1$: Solicita(C)
2. $P_2$: Solicita(I)
3. $P_2$: Solicita(C) → se bloquea puesto que el recurso no está disponible
4. $P_1$: Solicita(I) → se bloquea por recurso no disponible: **hay interbloqueo**

**Traza ejecución sin Interbloqueo**

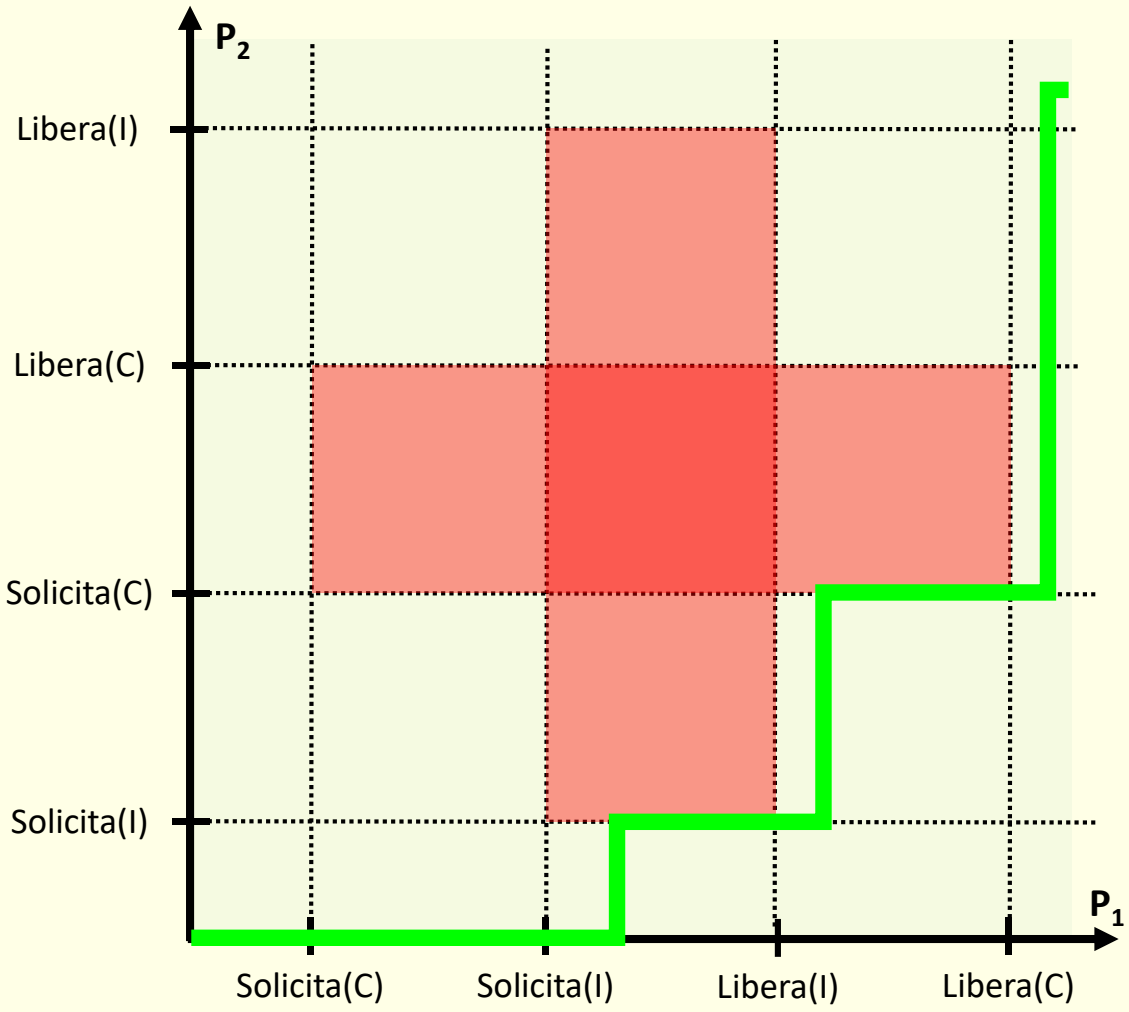
1. $P_1$: Solicita(C)
2. $P_1$: Solicita(I)
3. $P_2$: Solicita(I) → se bloquea puesto que el recurso no está disponible
4. $P_1$: Libera(I) → se desbloquea $P_2$ puesto que el recurso ya está disponible
5. $P_2$: Solicita(C) → se bloquea puesto que el recurso no está disponible
6. $P_1$: Libera(C) → se desbloquea $P_2$ porque el recurso ya está disponible
7. $P_2$: Libera(C)
8. $P_2$: Libera(I)

**Responsable SO**

# Diagrama de trayectoria de procesos: interbloqueo



# Diagrama de trayectoria de procesos: sin interbloqueo



# Escenarios interbloqueos: *threads* misma aplicación

| **Thread $P_1$** | **Thread $P_2$** | **Thread $P_3$** |
|---|---|---|
| **lock**(Ma) | **lock**(Mb) | **lock**(Mc) |
| $tarea_{11}$ | $tarea_{21}$ | $tarea_{31}$ |
| **lock**(Mb) | **lock**(Mc) | **lock**(Ma) |
| $tarea_{12}$ | $tarea_{22}$ | $tarea_{32}$ |
| **unlock**(Ma) | **unlock**(Mb) | **unlock**(Mc) |
| $tarea_{13}$ | $tarea_{23}$ | $tarea_{33}$ |
| **unlock**(Mb) | **unlock**(Mc) | **unlock**(Ma) |

**Recursos lógicos: *mutex***

1. $P_1$: lock(Ma)
2. $P_2$: lock(Mb)
3. $P_3$: lock(Mc)
4. $P_3$: lock(Ma) → se bloquea puesto que el recurso no está disponible
5. $P_1$: lock(Mb) → se bloquea puesto que el recurso no está disponible
6. $P_2$: lock(Mc) → se bloquea por recurso no disponible: **hay interbloqueo**

**Responsable desarrollador de la aplicación**

# Escenarios interbloqueos: con espera activa

**Proceso $P_1$**

```
while (Solicita(C)==false);
Uso del recurso C
while (Solicita(I)==false);
Uso de ambos recursos
Libera(I)
Uso del recurso C
Libera(C)
```

**Proceso $P_2$**

```
while (Solicita(I)==false);
Uso del recurso I
while (Solicita(C)==false);
Uso de ambos recursos
Libera(C)
Uso del recurso I
Libera(I)
```

---

1. $P_1$: Solicita(C) → verdadero puesto que el recurso está disponible
2. $P_2$: Solicita(I) → verdadero puesto que el recurso está disponible
3. $P_2$: Solicita(C) → falso ya que el recurso no está disponible
4. $P_1$: Solicita(I) → falso ya que el recurso no está disponible: **interbloqueo**

**El bloqueo no es un requisito para el interbloqueo**

# Escenarios interbloqueos: auto-interbloqueo

```
Proceso P
lock(M)
........
lock(M)
```

**Error trivial**

---

**Generalmente, más sutil**

```
F() {
   ..........
   lock(M)
   ..........
   unlock(M)
   ..........
}
```

**Proceso llama a F estando en posesión de M**

**Uso de *mutex* recursivos recursivos o no recursivos**

# Escenarios de interbloqueos: eventos asíncronos

**Con señales**

**Proceso**

```
..........
f();
..........

f() {
  lock(M)
  tarea <------ señal
  unlock(M)
  ........}
```

**Rutina de tratamiento de la señal**

```
..........
f();
..........
```

**También dentro del SO con las interrupciones**

# Escenarios interbloqueos: operación interna del SO

```
renombrar(rutaPrevia, rutaNueva) {
  dirOrg = directorio padre de rutaPrevia
  dirDst = directorio padre de rutaNueva
  if (dirOrg != dirDst) {
    Bloquea acceso a dirOrg
    Bloquea acceso a dirDst
    Elimina entrada rutaPrevia de dirOrg
    Añade entrada rutaNueva en dirDst
    Desbloquea acceso a dirOrg
    Desbloquea acceso a dirDst
  }
  else .........
}
```

---

```
P1: renombrar("/dir1/fA", "/dir2/fB");
P2: renombrar("/dir2/fC", "/dir1/fD");
```

---

1. Llamada de $P_1$: Bloquea acceso a "/dir1"
2. Llamada de $P_2$: Bloquea acceso a "/dir2"
3. Llamada de $P_2$: Bloquea acceso a "/dir1" → se bloquea
4. Llamada de $P_1$: Bloquea acceso a "/dir2" → interbloqueo

# Escenarios interbloqueos: competencia uso recurso

**Recurso con *N* unidades**

**Cada proceso puede requerir hasta *M* unidades**

**Proceso $P_1$**

```
Solicita(100K)
Solicita(100K)
Solicita(100K)
```

**Proceso $P_2$**

```
Solicita(200K)
Solicita(100K)
```

**Memoria disponible inicialmente 450K**

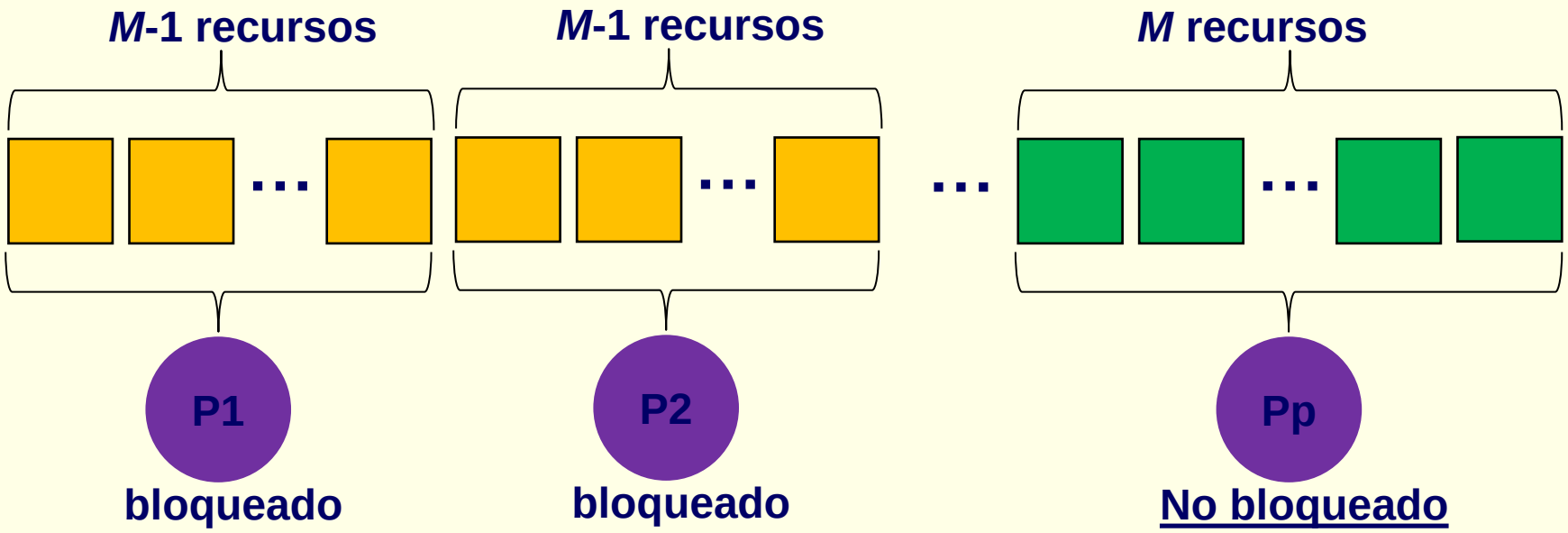
**(no hay memoria virtual; podría corresponder con uso interno del SO)**

# Escenarios interbloqueos: competencia uso recurso

**Recurso con *N* unidades**

**P procesos cada uno requiere hasta *M* unidades**

**¿Valor de *N* mínimo para que no haya interbloqueos?**




**Solución: P*(M-1)+1**

# Escenarios interbloqueos: comunicación de procesos

**Proceso $P_1$**
**Envía**($C_1$,A)
**Envía**($C_1$,B)
**Recibe**($C_2$,C)
**Envía**($C_1$,D)
**Envía**($C_1$,E)
**Recibe**($C_2$,F)

**Proceso $P_2$**
**Recibe**($C_1$,G)
**Recibe**($C_1$,H)
Procesa mensajes
**Envía**($C_2$,I)

**Proceso $P_3$**
**Recibe**($C_1$,J)
**Recibe**($C_1$,K)
Procesa mensajes
**Envía**($C_2$,L)

**C1 y C2 son colas de mensajes**

1. $P_1$: Envía($C_1$,A)
2. $P_1$: Envía($C_1$,B)
3. $P_1$: Recibe($C_2$,C) → se bloquea puesto que el recurso no está disponible
4. $P_2$: Recibe($C_1$,G)
5. $P_3$: Recibe($C_1$,J)
6. $P_3$: Recibe($C_1$,K) → se bloquea puesto que el recurso no está disponible
7. $P_2$: Recibe($C_1$,H) → se bloquea puesto que el recurso no está disponible

# Escenarios interbloqueos: *livelock*

**Procesos lo reintentan pero puede que nunca progresen**

**Proceso $P_1$**

```
while (Solicita(C)==false);
Uso del recurso C
if (Solicita(I)==false)
    Libera(C)
    Deshace trabajo hecho
    Vuelve al inicio
Uso de ambos recursos
Libera(I)
Uso del recurso C
Libera(C)
```

**Proceso $P_2$**

```
while (Solicita(I)==false);
Uso del recurso I
if (Solicita(C)==false)
    Libera(I)
    Deshace trabajo hecho
    Vuelve al inicio
Uso de ambos recursos
Libera(C)
Uso del recurso I
Libera(I)
```

**No siempre posible "deshacer trabajo"**

# Definición de interbloqueo

- Conjunto de procesos tal que cada uno está esperando un recurso que solo puede liberar otro proceso del conjunto
- Condiciones de Coffman:
  - **Exclusión mutua**
    - Interbloqueo solo afecta a recursos de uso exclusivo
  - **Retención y espera**
    - Proceso mantiene recursos asignados mientras espera por otros
  - **Sin expropiación**
    - No pueden expropiarse recursos asignados a un proceso
    - Recursos expropiables (p.e. UCP) no afectados por interbloqueo
  - **Espera circular**
    - lista circular: proceso espera recursos asignados a siguiente proceso
    - Condición dinámica que identifica el interbloqueo

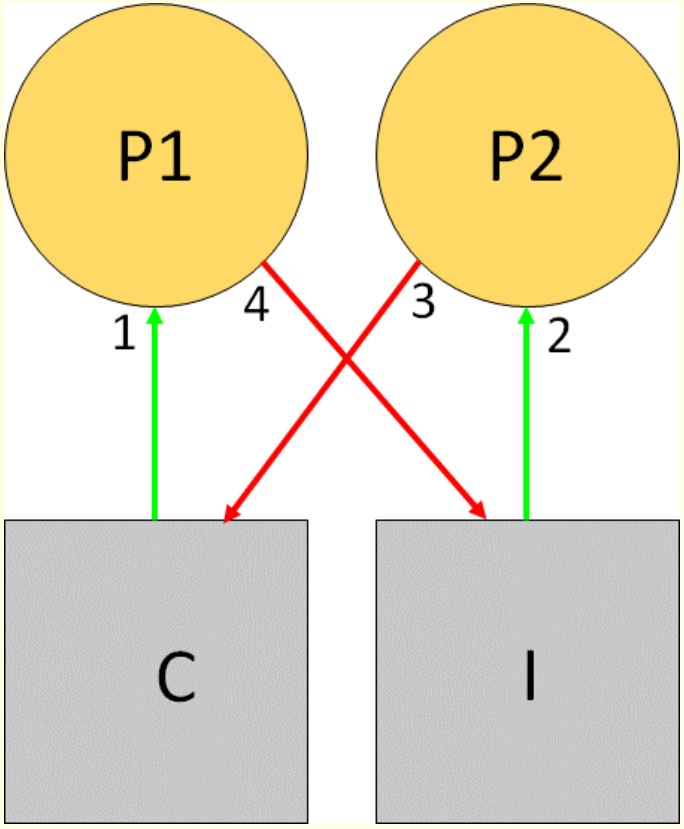
# Modelo básico del sistema

- Conjunto de procesos o *threads*
- Conjunto de recursos de uso exclusivo (N unidades/recurso)
- Relaciones entre procesos y recursos:
  - Asignación: nº unidades asignadas a cada proceso
  - Pendientes: nº unidades pedidas pero no asignadas
- Primitivas genéricas: *Solicitud* ($R_i$) y *Liberación* ($R_i$)
- Carácter dinámico del sistema:
  - Procesos y recursos aparecen y desaparecen
- Representación habitual grafo (en algunos casos matriz)
- Condiciones Coffman **necesarias y suficientes** para interbloqueo
  - Si ciclo en el grafo hay interbloqueo
- Nos centramos en este modelo básico
  - Recoge las características principales de los sistemas reales

**Libro estudia modelo extendido**

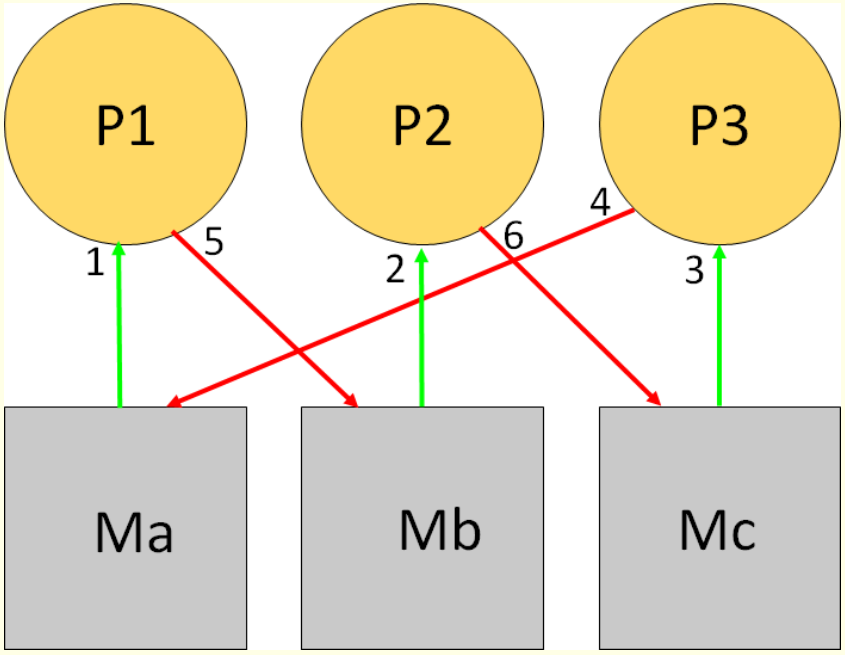
# Grafo de asignación de recursos

1. `P1: Solicita(C)`
2. `P2: Solicita(I)`
3. `P2: Solicita(C)` → se bloquea puesto que el recurso no está disponible
4. `P1: Solicita(I)` → se bloquea por recurso no disponible: **hay interbloqueo**



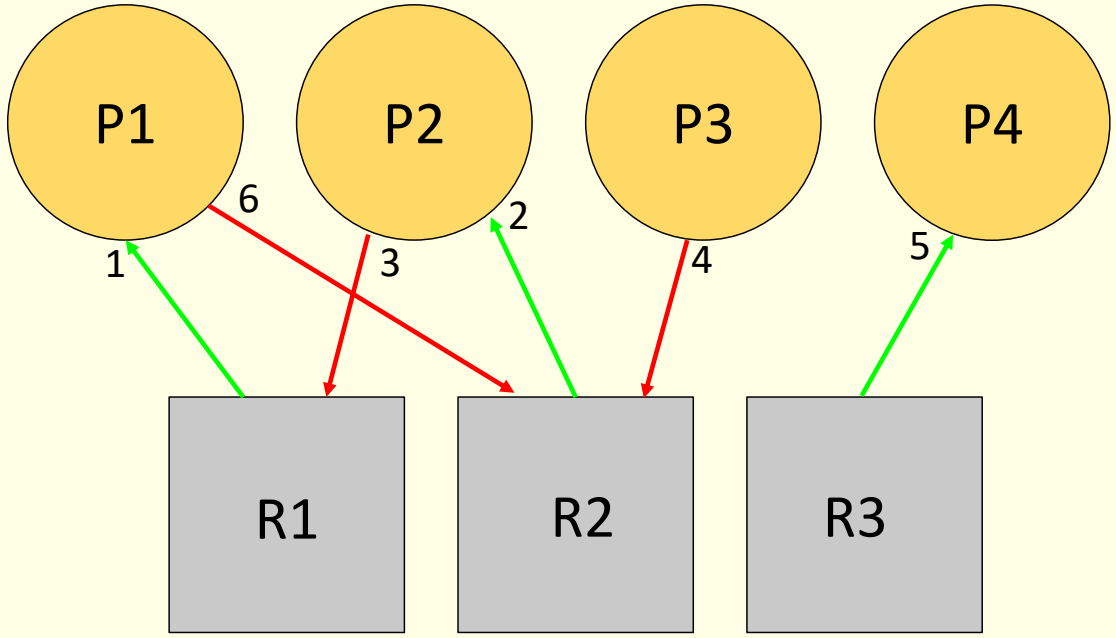
# Grafo de asignación de recursos

1. $P_1$: `lock(Ma)`
2. $P_2$: `lock(Mb)`
3. $P_3$: `lock(Mc)`
4. $P_3$: `lock(Ma)` → se bloquea puesto que el recurso no está disponible
5. $P_1$: `lock(Mb)` → se bloquea puesto que el recurso no está disponible
6. $P_2$: `lock(Mc)` → se bloquea por recurso no disponible: **hay interbloqueo**

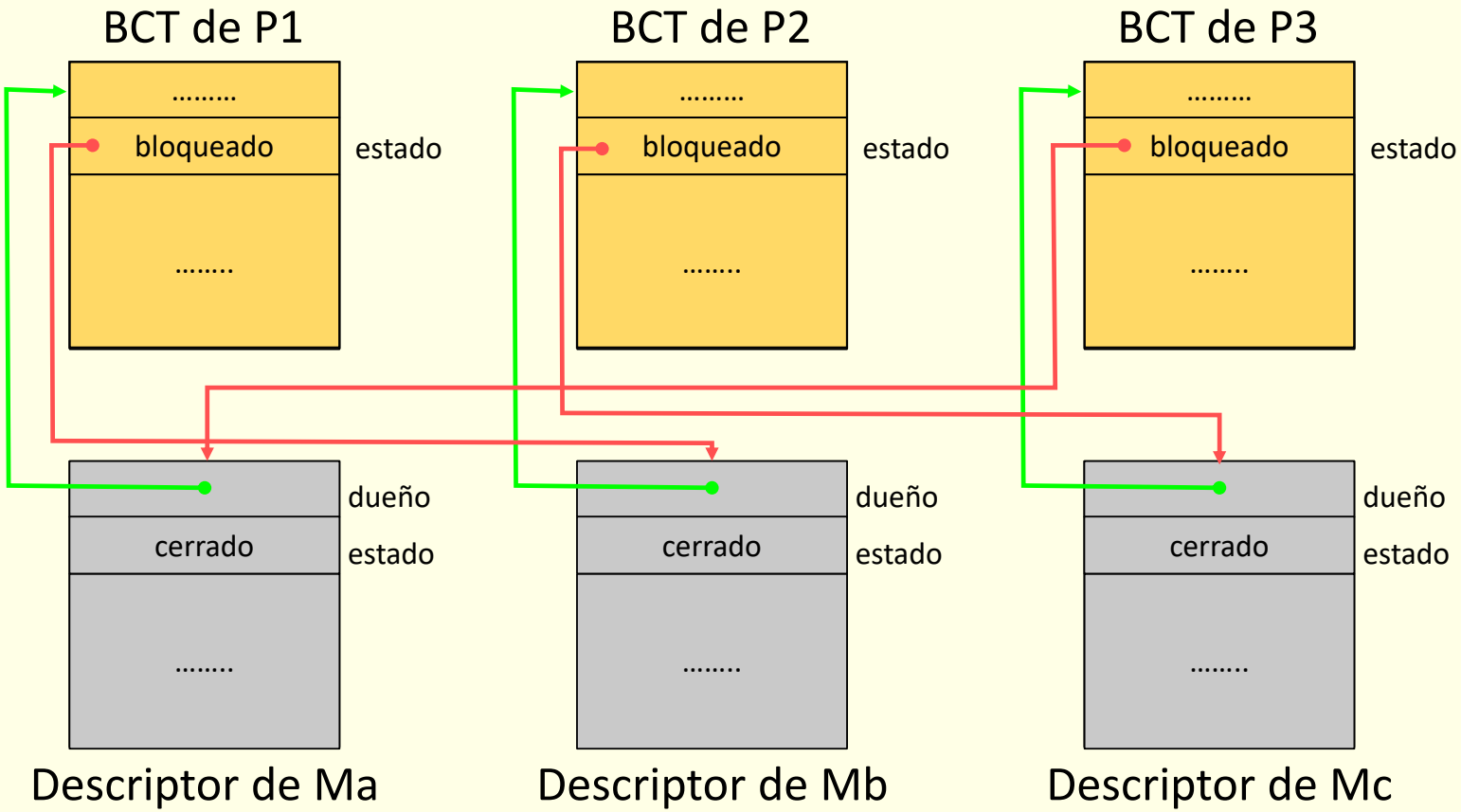


# Grafo de asignación de recursos

1. $P_1$: solicita($R_1$)
2. $P_2$: solicita($R_2$)
3. $P_2$: solicita($R_1$) → se bloquea, puesto que el recurso no está disponible
4. $P_3$: solicita($R_2$) → se bloquea, puesto que el recurso no está disponible
5. $P_4$: solicita($R_3$)
6. $P_1$: solicita($R_2$) → se bloquea, puesto que el recurso no está disponible



# Grafo de asignación de recursos gestionado por SO



# Tratamiento de interbloqueo

- Detección y recuperación. Se detecta y se recupera
  - Coste de algoritmo de detección
  - Pérdida del trabajo realizado en la recuperación
- Prevención. Asegura que no ocurre fijando reglas
  - Infrautilización de rec.: se deben pedir antes de necesitarlos
- Predicción (*avoidance*). Asegura que no ocurre basándose en conocimiento a priori de necesidades de los procesos
  - Conocer a priori no factible en sistema de propósito general
  - Coste de algoritmo de supervisión
  - Infrautilización de recursos
- Ignorar el problema
  - SO debería asegurar que está libre de interbloqueos internamente
  - Pero con frecuencia no asegura que lo están sus servicios

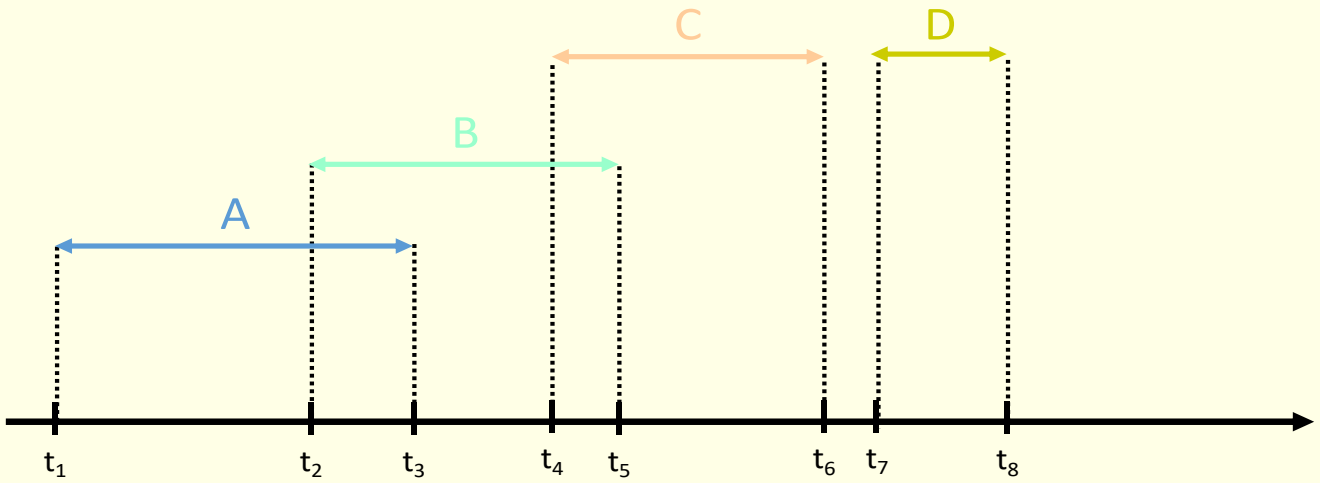
# Detección y recuperación

- Activación del algoritmo de detección
  - Supervisión en cada petición que no puede satisfacerse
    - Puede tener coste demasiado alto
  - Supervisión periódica (o por detección de síntomas)
    - Menos sobrecarga pero detección y recuperación más complejas
- Recuperación del interbloqueo
  - Si supervisión en cada petición
    - Devolver error en la petición o abortar proceso solicitante
  - Si supervisión periódica
    - Quitar recursos a procesos hasta eliminar interbloqueo
    - "Retroceder en el tiempo" no factible en SO de propósito general
      - Abortar procesos perdiendo todo su trabajo realizado
      - Selección de procesos basada en prioridad, nº de recursos asignados...
      - Estrategia adecuada para bases de datos: reiniciar transacción

# Prevención del interbloqueo

- Que no se cumpla una condición necesaria
- "Exclusión mutua" y "sin expropiación" no se pueden relajar
  - Dependen de carácter intrínseco del recurso
- Las otras dos condiciones son más prometedoras

# Prevención evitando "retención y espera"




**solo se puede pedir un recurso si no se tiene ninguno**

**Infrautilización**

$t_1$: solicita(A,B,C)
($t_1$,$t_2$): solo utiliza A
($t_2$,$t_3$): utiliza A y B
$t_3$: Libera(A)
($t_3$,$t_4$): solo utiliza B
($t_4$,$t_5$): utiliza B y C
$t_5$: Liber(B)
($t_5$,$t_6$): solo utiliza C
$t_6$: Libera(C)
$t_7$: solicita(D)
($t_7$,$t_8$): solo utiliza D
$t_8$: Libera(D)

# Prevención evitando "espera circular"

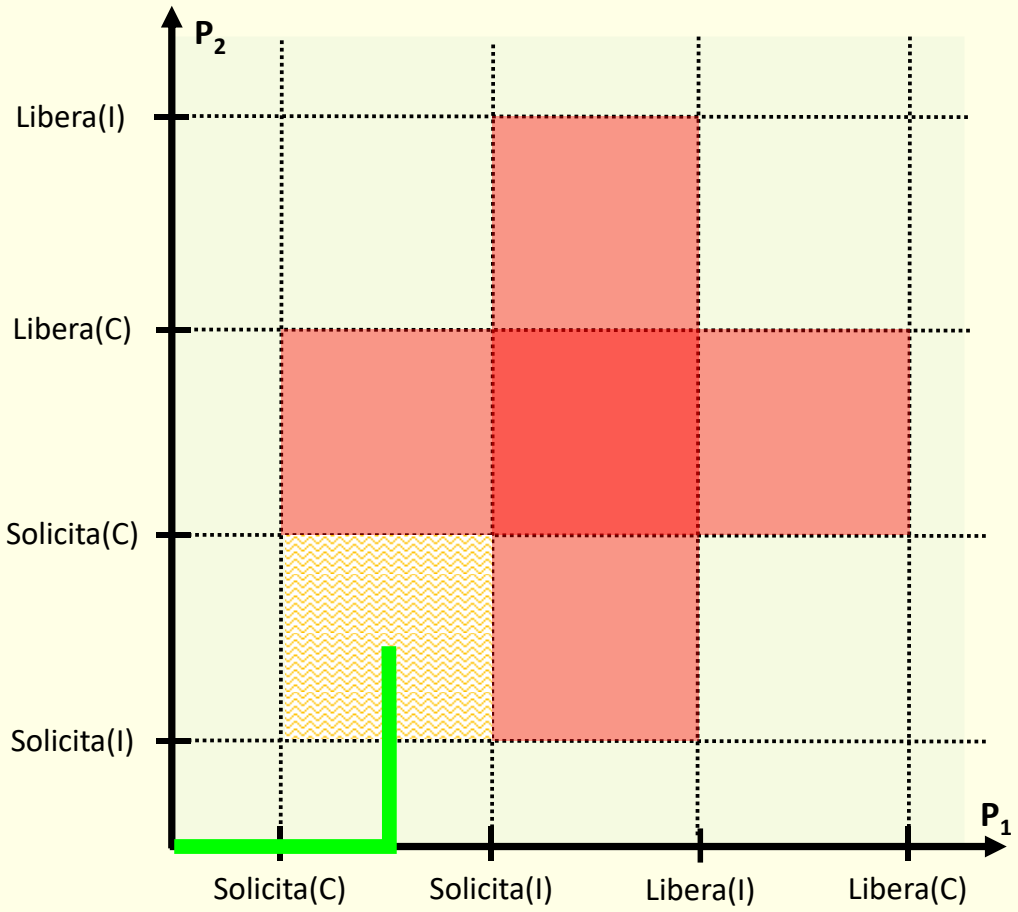
- Método de la ordenación de peticiones:
  - Establece orden de recursos del sistema
    - Según forma de uso más frecuente
  - Restricción: Proceso solo puede pedir en orden
- Conlleva infrautilización:
  - Si A<B<C<D $\rightarrow$ Proceso pide justo cuando necesita
  - Si A>B>C>D $\rightarrow$ Proceso pide todo en $t_1$

# Predicción del interbloqueo

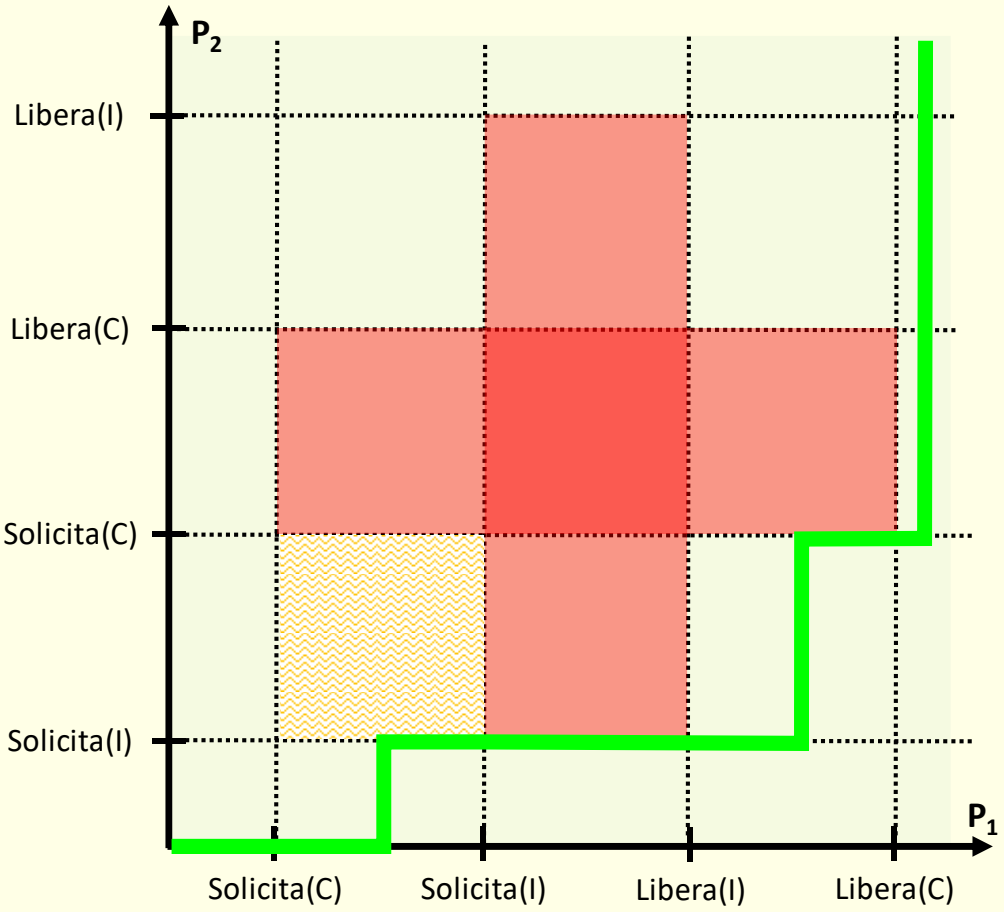
- Punto sin retorno: P1 y P2 obtienen su primer recurso
- No conceder 1 de esas peticiones → sistema en "estado seguro"

| **Proceso $P_1$** | **Proceso $P_2$** |
|---|---|
| Solicita(C) | Solicita(I) |
| Solicita(I) | Solicita(C) |
| Uso de rec. | Uso de rec. |
| Libera(I) | Libera(C) |
| Libera(C) | Libera(I) |

# Diagrama de trayectoria de procesos: estado inseguro



# Diagrama de trayectoria de procesos: estado seguro



# Concepto de estado seguro

- No interbl. aunque procesos pidiesen de golpe necesidades máx.
- Similar a detección pero con necesidades máximas
- Estado seguro:
  - No interbl. usando como solicitudes las necesidades máx.
- Conocimiento a priori no da información sobre uso real
  - Ejemplo mismas necesidades máximas que anterior
    - Pero no es posible interbloqueo aunque sí estado inseguro

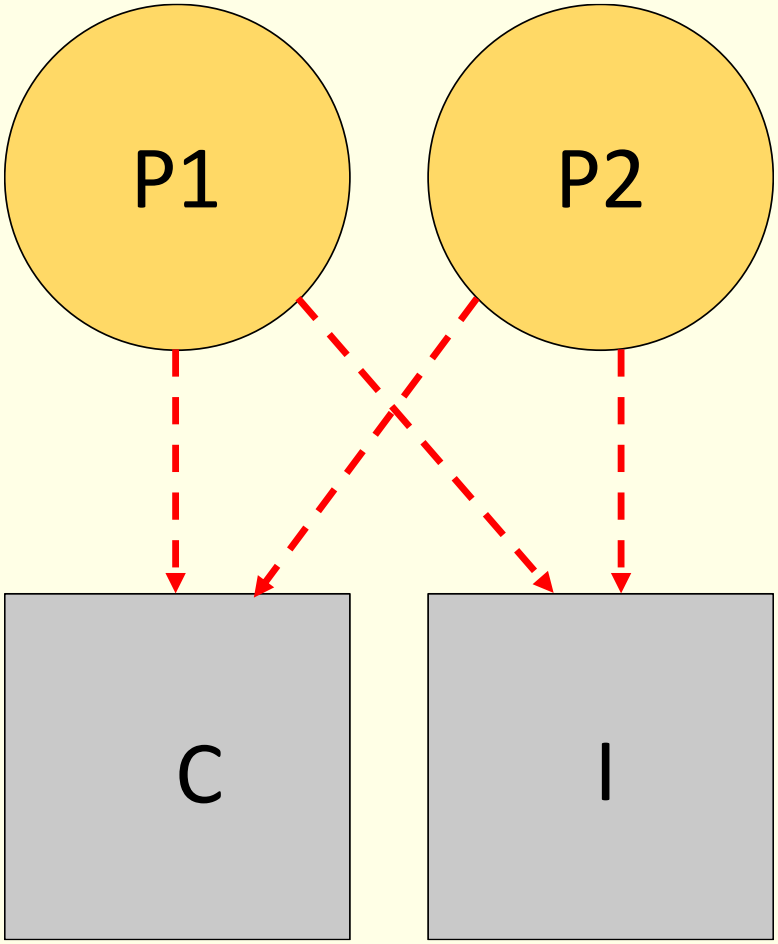
**Proceso $P_1$**

```
Solicita(I)
Uso del recurso I
Libera(I)
Solicita(C)
Solicita(I)
Uso de los recursos
Libera(C)
Libera(I)
```

**Proceso $P_2$**

```
Solicita(C)
Solicita(I)
Uso de los recursos
Libera(C)
Libera(I)
```

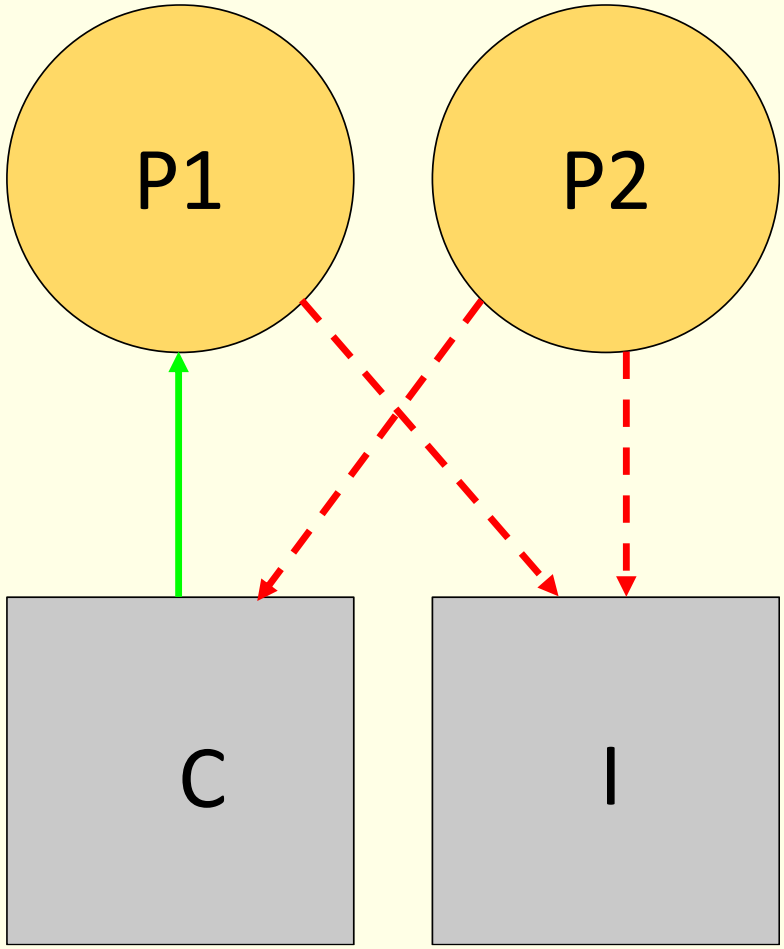
# Grafo de asignación de recursos para estado seguro




**Estado inicial**

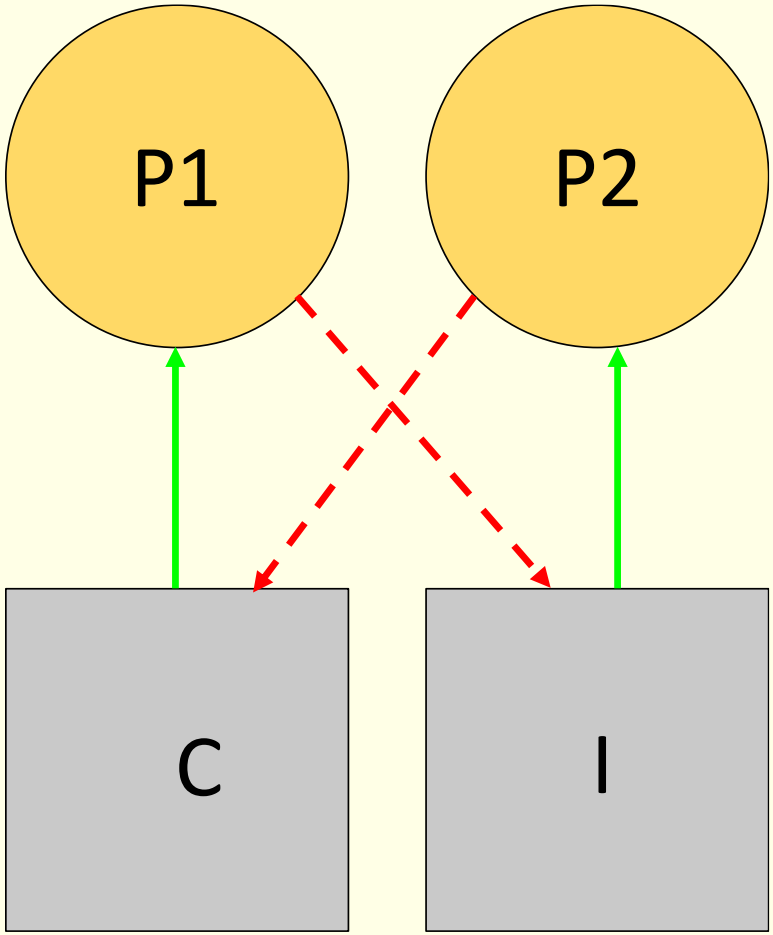
**Refleja necesidades máximas**

# Grafo de asignación de recursos para estado seguro




**Estado después de primera solicitud**

# Grafo de asignación de recursos para estado seguro




**Estado después de dos primeras solicitudes**

**Interbloqueo: estado inseguro**

**No se le concede el recurso a P2 aunque esté libre**

# Limitaciones de estrategias de predicción

- Conocimiento a *priori* de necesidades máximas
  - No factible en SO de propósito general
  - Basado en peor caso posible
- Necesidades máximas no expresan uso real de recursos
- Coste del algoritmo de supervisión
- Infrautilización de recursos
  - Niegan uso de recurso aunque esté libre

# Algoritmo del banquero (Dijkstra 1965)

- Algoritmo de predicción para el modelo extendido
  - Aunque no hemos estudiado ese modelo, es un clásico…
- Estructuras de datos requeridas:
  - Vector D (dim p): $D_i$ cuántas unid de $R_i$ hay disponibles
  - Matriz A (dim pxr): A[i,j] unid de $R_j$ asignadas a $P_i$
  - Matriz N (dim pxr): N[i,j] unid adicionales de $R_j$ puede necesitar $P_i$
    - Es la diferencia entre necesidades máx. y unidades asignadas
    - Inicialmente contiene necesidades máx. de cada proceso
- Solicitud de recursos de Pi: ¿Todos disponibles?
  - Sí. Por cada Rj:
    - A[i,j]= A[i,j]+ Uj y N[i,j]= N[i,j]- Uj (Uj unidades solicitadas de $R_j$)
- Liberación de recursos de Pi:
  - Por cada Rj:
    - A[i,j]= A[i,j]- Uj y N[i,j]= N[i,j]+ Uj (Uj unidades liberadas de $R_j$)

# Algoritmo del banquero

```
S=∅;
Repetir {
    Buscar Pi ∉ en S tal que N[i]≤D;
    Si Encontrado {
        Reducir por Pi: D = D + A[i]
        Añadir Pi a S;
    }
} Mientras (Encontrado)
Si (S==P) // P cjto de todos los procesos
    El estado es seguro
Si no: El estado no es seguro
```

# Estrategia de predicción con alg. del banquero

- Proceso realiza petición de recursos disponibles:
  - Nuevo estado provisional transformando A y N
  - Algoritmo del banquero sobre nuevo estado
  - Si seguro, se asignan recursos
  - Si no, se bloquea al proceso sin asignarle los recursos

# Ejemplo de algoritmo del banquero (1/3)

- Estado actual del sistema (es seguro):

$$A = \begin{pmatrix} 1 & 1 & 0 \\ 0 & 1 & 2 \\ 1 & 0 & 0 \end{pmatrix} \quad N = \begin{pmatrix} 3 & 0 & 2 \\ 2 & 2 & 0 \\ 1 & 1 & 2 \end{pmatrix} \quad D = (2\ 1\ 2)$$

- Secuencia de peticiones:
  - $P_3$ solicita 1 unidad de $R_3$
  - $P_2$ solicita 1 unidad de $R_1$

# Ejemplo de algoritmo del banquero (2/3)

- $P_3$ solicita 1 unidad de $R_3$: Nuevo estado provisional

$$A = \begin{pmatrix} 1 & 1 & 0 \\ 0 & 1 & 2 \\ 1 & 0 & 1 \end{pmatrix} \quad N = \begin{pmatrix} 3 & 0 & 2 \\ 2 & 2 & 0 \\ 1 & 1 & 1 \end{pmatrix} \quad D = (2\ 1\ 1)$$

- ¿Estado seguro?

1. $S=\emptyset$
2. $P_3$: ya que $N[3] \leq D \rightarrow D=D+A[3]=[312] \rightarrow S=\{P_3\}$
3. $P_1$: ya que $N[1] \leq D \rightarrow D=D+A[1]=[422] \rightarrow S=\{P_3, P_1\}$
4. $P_2$: pues $N[2] \leq D \rightarrow D=D+A[2]=[434] \rightarrow S=\{P_3, P_1, P_2\}$

Se acepta petición: estado provisional → estado del sistema

# Ejemplo de algoritmo del banquero (3/3)

- $P_2$ solicita 1 unidad de $R_1$: Nuevo estado provisional

$$A = \begin{pmatrix} 1 & 1 & 0 \\ 1 & 1 & 2 \\ 1 & 0 & 1 \end{pmatrix} \quad N = \begin{pmatrix} 3 & 0 & 2 \\ 1 & 2 & 0 \\ 1 & 1 & 1 \end{pmatrix} \quad D = (1\ 1\ 1)$$

- ¿Estado seguro?

1. $S = \emptyset$
2. $P_3$: ya que $N[3] \leq D \rightarrow D = D + A[3] = [212] \rightarrow S = \{P_3\}$
3. No hay Pi tal que $N[i] \leq D \rightarrow$ Estado inseguro

No se acepta petición:

Se bloquea al proceso y se restaura estado del sistema

# Tratamiento del interbloqueo en los SSOO

- Fundamental la distinción entre dos tipos de recursos:
  - Recursos internos (propios del SO)
    - Uso restringido a una activación del sistema operativo
    - P. ej. *spinlocks* y semáforos internos
    - Interbloqueo puede causar colapso del sistema
  - Recursos de usuario (usados en modo usuario)
    - Uso durante tiempo impredecible
    - P. ej. dispositivo dedicado o semáforo de aplicación
    - Interbloqueo afecta a procesos y recursos involucrados

# Tratamiento interbloqueo en SO: recursos internos

- Interbloqueo interno en SO
  - error de programación de SO
- Uso de estrategias de detección es inadecuado
  - Sobrecarga y pérdida de trabajo
- Uso de estrategias de predicción no es factible
  - A pesar de que se conocen a priori recursos a usar
  - Sobrecarga
- Uso de estrategias de prevención es adecuado
  - Tiempo de uso es breve y acotado
  - Solución habitual: ordenamiento de peticiones

# Tratamiento rec. internos: operaciones bloqueantes

```
renombrar(rutaPrevia, rutaNueva) {
  dirOrg = directorio padre de rutaPrevia
  dirDst = directorio padre de rutaNueva
  if (dirOrg != dirDst) {
    if (dirOrg->descriptor < dirDst->descriptor) {
      Bloquea acceso a dirOrg; Bloquea acceso a dirDst;}
    else {
      Bloquea acceso a dirDst; Bloquea acceso a dirOrg;}
    Elimina entrada rutaPrevia de dirOrg
    Añade entrada rutaNueva en dirDst
    Desbloquea acceso a dirOrg
    Desbloquea acceso a dirDst
  }
  else .........
}
```

---

```
P1: renombrar("/dir1/fA", "/dir2/fB");
P2: renombrar("/dir2/fC", "/dir1/fD");
```

---

1. Llamada de $P_1$: Bloquea acceso a "/dir1" → entra en la rama *if*
2. Llamada de $P_2$: Bloquea acceso a "/dir1" → entra en la rama *else*: bloqueo
3. Llamada de $P_1$: Bloquea acceso a "/dir2"

# Tratamiento rec. internos: operaciones no bloqueantes

```
moverProcesoDeCola(colaOrg, colaDst, proceso) {
  spin_lock(colaOrg->spin);
  spin_lock(colaDst->spin);
  eliminarBCP(colaOrg, proceso);
  insertarBCP(colaDst, proceso);
  spin_unlock(colaOrg->spin);
  spin_unlock(colaDst->spin);
}
```

**Tratamiento del evento1:** `moverProcesoDeCola(colaP1, colaP2, p);`
**Tratamiento del evento2:** `moverProcesoDeCola(colaP2, colaP1, q);`

1. `Tratamiento del evento1: spin_lock(colaP1->spin)`
2. `Tratamiento del evento2: spin_lock(colaP2->spin)`
3. `Tratamiento del evento2: spin_lock(colaP1->spin)` → en espera activa
4. `Tratamiento del evento1: spin_lock(colaP2->spin)` → interbloqueo

```
moverProcesoDeCola(colaOrg, colaDst, proceso) {
    if (colaOrg < colaDst) {
      spin_lock(colaOrg->spin); spin_lock(colaDst->spin);}
    else {
      spin_lock(colaDst->spin); spin_lock(colaOrg->spin);}
    eliminarBCP(colaOrg, proceso);
    insertarBCP(colaDst, proceso);
    spin_unlock(colaOrg->spin);
    spin_unlock(colaDst->spin);
  }
```

# Tratamiento rec. internos: operaciones asíncrona

**Tratamiento de evento X**

```
.........
spin_lock(s)
.......  <-------- intX
spin_unlock(s)
.........
```

**Rutina de interrupción Y**

```
.........
spin_lock(s)
.......
spin_unlock(s)
.........
```

---

**Tratamiento de evento X**

```
.........
inhibir_local_intY
spin_lock(s)
.......
spin_unlock(s)
habilitar_local_intY
.........
```

**Rutina de interrupción Y**

```
.........
spin_lock(s)
.......
spin_unlock(s)
.........
```

# Tratamiento interbloqueo en SO: recursos de usuario

- Código de procesos que usan recursos es impredecible
- Uso de estrategias de prevención es inadecuado
  - Infrautilización
- Uso de estrategias de predicción no es factible
  - No se conocen a priori recursos a usar
  - Sobrecarga e infrautilización
- Uso de estrategias de detección puede ser adecuado
  - Aplicación a todos los recursos
    - puede llevar demasiada sobrecarga
  - S.O. suele usarla para controlar un tipo de recurso específico
    - No aborta procesos: devuelve error en solicitud conflictiva
  - Aplicaciones que usan distintos tipos de recursos exclusivos
    - O un tipo de recurso no supervisado
    - Pueden sufrir un interbloqueo

# Tratamiento interbloqueo recursos de usuario: Linux

- Pruebas con *mutex*:
  - Auto-interbloqueos
    - Solo los detecta (devuelve error) para *mutex* con control de errores

    **pthread_mutexattr_settype**(&atrib,PTHREAD_MUTEX_ERRORCHECK);
  - Usando dos *mutex*
    - No los detecta aunque tengan activo el control de errores
- Pruebas con cerrojos de ficheros (*fcntl*)
  - Los detecta para cualquiera nº de procesos y cerrojos
  - Devuelve un error en la llamada conflictiva

# Diseño de *apps* concurrentes libres de interbloqueos

- Responsabilidad del desarrollador
  - En algunos casos con apoyo del SO
- Propondremos algunas pautas para evitar los interbloqueos
- Aplicaremos algunas de las estrategias teóricas
  - Principalmente, detección/recuperación y prevención
- Vamos a analizar dos ejemplos probados en Linux
  - Escenario de uso de cerrojos de ficheros
    - Ejemplo de gestión de cuentas bancarias
    - Analizaremos estrategias de detección/recuperación y prevención
    - Linux detecta interbloqueo y devuelve error en la llamada
  - Escenario de uso de *mutex*
    - Ejemplo de gestión de listas
    - Linux no detecta interbloqueo
    - Revisaremos también el tratamiento de eventos asíncronos

# Cerrojos ficheros: detección y terminación ordenada

```
int transferencia_cuentas(int n_cnt_org,int n_cnt_dst,float cantidad){
      int fd; struct cuenta c;

      if ((fd=open(FICHERO_CUENTAS, O_RDWR))<0) return -1;

      // establece cerrojo en cuenta origen
      struct flock fl1 = {.l_whence=SEEK_SET,
            .l_start= n_cnt_org*sizeof(struct cuenta),
            .l_len=sizeof(struct cuenta)};
      fl1.l_type = F_WRLCK;
      if (fcntl(fd, F_SETLKW, &fl1)<0){
            close(fd); return -1;}

      // establece cerrojo en cuenta destino
      struct flock fl2 = {.l_whence=SEEK_SET,
            .l_start=n_cnt_dst*sizeof(struct cuenta),
            .l_len=sizeof(struct cuenta)};
      fl2.l_type = F_WRLCK;
      if (fcntl(fd, F_SETLKW, &fl2)<0){
            close(fd); return -1;} // close libera los cerrojos
      ..........................
```

# Cerrojos ficheros: detección y repetición

```
int transferencia_cuentas(int n_cnt_org,int n_cnt_dst,float cantidad){
      int fd; struct cuenta c;

      if ((fd=open(FICHERO_CUENTAS, O_RDWR))<0) return -1;
      inicio: // punto de reintento
      // establece cerrojo en cuenta origen
      struct flock fl1 = {.l_whence=SEEK_SET,
            .l_start= n_cnt_org*sizeof(struct cuenta),
            .l_len=sizeof(struct cuenta)};
      fl1.l_type = F_WRLCK;
      if (fcntl(fd, F_SETLKW, &fl1)<0){
            close(fd); return -1;}

      // establece cerrojo en cuenta destino
      struct flock fl2 = {.l_whence=SEEK_SET,
            .l_start=n_cnt_dst*sizeof(struct cuenta),
            .l_len=sizeof(struct cuenta)};
      fl2.l_type = F_WRLCK;

      // Único cambio: Nuevo tratamiento de error con reintento
      if (fcntl(fd, F_SETLKW, &fl2)<0){
            if ((errno==EDEADLOCK)&&(reintentos-->0)) {
                  // libera el primer cerrojo y vuelve a intentarlo
                  fl1.l_type = F_UNLCK; fcntl(fd, F_SETLKW, &fl1);
                  // espera plazo aleatorio creciente antes reintento
                  usleep(plazo_aleatorio); plazo_aleatorio <<= 1;
                  goto inicio;
            }
            else {close(fd); return -1;}
      }
      ..........................
```

# Cerrojos ficheros: prevención

```
int transferencia_cuentas(int n_cnt_org,int n_cnt_dst,float cantidad){
      int fd; struct cuenta c;

      // estrategia de prevención
      int cnt1=n_cnt_org, cnt2=n_cnt_dst;
      if (n_cnt_org > n_cnt_dst) {cnt1=n_cnt_dst, cnt2=n_cnt_org;}

      if ((fd=open(FICHERO_CUENTAS, O_RDWR))<0) return -1;

      // establece cerrojo en la primera cuenta
      struct flock fl1 = {.l_whence=SEEK_SET,
            .l_start= cnt1*sizeof(struct cuenta),
            .l_len=sizeof(struct cuenta)};
      fl1.l_type = F_WRLCK;
      if (fcntl(fd, F_SETLKW, &fl1)<0){
            close(fd); return -1;}

      // establece cerrojo en la segunda cuenta
      struct flock fl2 = {.l_whence=SEEK_SET,
            .l_start= cnt2*sizeof(struct cuenta),
            .l_len=sizeof(struct cuenta)};
      fl2.l_type = F_WRLCK;
      if (fcntl(fd, F_SETLKW, &fl2)<0){
            close(fd); return -1;}
      ..........................
```

# *mutex*: gestión de listas con interbloqueo

```
void mover_de_lista(struct lista *origen, struct lista* destino,
      struct nodo *elemento, int posicion_destino) {

      pthread_mutex_lock(&origen->mutex_lista);
      pthread_mutex_lock(&destino->mutex_lista);

      /* elimina el elemento de la lista origen */
      /* añade el elemento a la lista destino en posición dada */

      pthread_mutex_unlock(&origen->mutex_lista);
      pthread_mutex_unlock(&destino->mutex_lista);
}
```

# *mutex*: gestión listas sin interbloqueo por prevención

```
void mover_de_lista(struct lista *origen, struct lista* destino,
      struct nodo *elemento, int posicion_destino) {

      if (origen < destino) {
            pthread_mutex_lock(&origen->mutex_lista);
            pthread_mutex_lock(&destino->mutex_lista);
      }
      else {
            pthread_mutex_lock(&destino->mutex_lista);
            pthread_mutex_lock(&origen->mutex_lista);

      }

      /* elimina el elemento de la lista origen */
      /* añade el elemento a la lista destino en posición dada */

      pthread_mutex_unlock(&origen->mutex_lista);
      pthread_mutex_unlock(&destino->mutex_lista);
}
```

**Asignar orden a *mutex* en aplicación compleja no es trivial.**
**Si funciones de módulo A llaman a las de módulo B**
**mutex de A orden previo a los de B (cuidado con *callbacks*)**

# *mutex*: gestión de listas y señales con interbloqueo

```
tipo_elem *extraer_primero(tipo_lista *lista){
      pthread_mutex_lock(&lista->m);
       ..................
      pthread_mutex_unlock(&lista->m);
      return elem;
}
void insertar_ultimo(tipo_lista *lista, tipo_elem * elem){
      pthread_mutex_lock(&lista->m);
       ..................
      pthread_mutex_unlock(&lista->m);
}
```

```
static void senal(int s) {
       ..................
      insertar_ultimo(&lista_int, el);
}
static void *thread(void *arg) {
       ..................
      el = extraer_primero(&lista_int);
       ..................
}
```

1. $T_1$: llama a extraer_primero y obtiene el cerrojo de la lista
2. Tratamiento de SIGUSR1 (en contexto de $T_1$): invoca insertar_ultimo que solicita el cerrojo → interbloqueo

# *mutex*: gestión de listas y señales sin interbloqueo

```
static void *thread(void *arg) {
        ..............
        sigset_t set, oset;
        sigemptyset(&set);
        sigaddset(&set, SIGUSR1);
        pthread_sigmask(SIG_BLOCK, &set, &oset);
        el = extraer_primero(&lista_int);
        pthread_sigmask(SIG_SETMASK, &oset, NULL);
        ..............
}
```

**Bloquea la entrega de señal al *thread* mientras realiza la extracción.**
**Observe similitud con *spinlock* + prohibir interrupciones dentro del SO.**