

Arquitectura de Sistemas

Gestión de recursos: interbloqueo e inanición

Gustavo Romero López

Updated: 22 de mayo de 2025

Departamento de Ingeniería de Computadores, Automática y Robótica

1. Gestión de recursos

1.1 Tipos de recursos

1.2 Protocolo de utilización

2. Gestión de interbloqueos

2.1 Prevenir

2.2 Evitar

2.3 Detectar y recuperar

2.4 Ignorar

Lecturas recomendadas

J. Bacon Operating Systems (11, 18)

A. Silberschatz Fundamentos de Sistemas Operativos (7)

W. Stallings Sistemas Operativos (6)

A. Tanuenbaum Sistemas Operativos Modernos (3)

- ⊙ Los procesos utilizan los **recursos** de los que dispone la máquina.
- ⊙ Existen recursos de distinta naturaleza: hardware y software.
- ⊙ La forma de utilizar esos recursos puede conducir a situaciones en las que el **progreso** de los procesos involucrados es **imposible** $\implies$ bloqueo irreversible o **interbloqueo**.
 - Hardware:
 - A: adquiere escáner y unidad USB.
 - B: adquiere unidad USB y escáner.
 - Software:
 - A: bloquea registros R1 y R2 de una base de datos.
 - B: bloquea registros R2 y R1 de una base de datos.
 - En el tema anterior hemos visto muchos ejemplos...
- ⊙ El conflicto requiere de al menos **dos agentes activos**.

⊙ Hardware:

- procesador
- memoria: RAM y disco
- red
- monitor
- impresora
- ...

⊙ Software:

- búfer
- ficheros
- directorio
- ...

⊙ También hay interbloqueos en los que no están involucrados recursos: sincronización.

Recursos expropiables y no expropiables

- ⊙ Un recurso se dice **expropiable** si puede serle **retirado** al proceso que lo posee **sin** causarle **daño** alguno, ej: memoria.
- ⊙ Un recurso se dice **no expropiable** si no puede serle **retirado** al propietario sin hacer que **falle**, ej: unidad de cd, impresora.
- ⊙ En los interbloqueos suelen intervenir recursos **no expropiables**.

Protocolo de utilización de recursos (1)

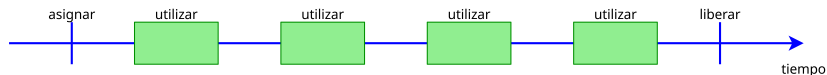
Para utilizar un recurso se sigue este protocolo:

- ⊙ **solicitar** el recurso.
- ⊙ **utilizar** el recurso.
- ⊙ **liberar** el recurso.

Ante una solicitud de recurso que no pueda ser atendida podemos hacer dos cosas:

- ⊙ **bloquear** el proceso hasta que el recurso quede disponible de nuevo.
- ⊙ hacer que la solicitud falle y comunicarlo devolviendo un **código de error**.

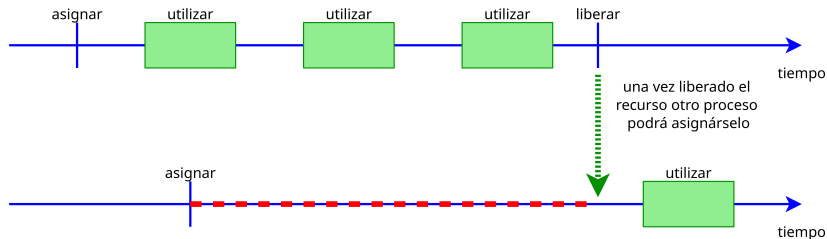
Protocolo de utilización de recursos (2)



La utilización de un recurso es parecida a la gestión de una sección crítica:

- ⊙ asignar ~ pedir permiso para ejecutar la sección crítica
- ⊙ utilizar ~ ejecutar la sección crítica
- ⊙ liberar ~ avisar al salir de la sección crítica

Protocolo de utilización de recursos (3)



- ⊙ El segundo proceso debe esperar a que el primero acabe de utilizar el recurso y lo libere.
- ⊙ Sólo entonces el segundo podrá finalizar la asignación antes de poder utilizarlo.

Adquisición de recursos (1)

- ⊙ Algunos recursos son administrados por el **usuario**, ej: bases de datos, memoria compartida,...
- ⊙ Un método de administración podría ser asociar un semáforo a cada recurso.
- ⊙ El **orden** de adquisición de recursos es **importante**.

un recurso

```
semáforo_binario recurso;
```

```
recurso.decrementar();
```

```
usar_recurso();
```

```
recurso.incrementar();
```

varios recursos simultáneos

```
semáforo_binario recurso1,  
                    recurso2;
```

```
recurso1.decrementar();
```

```
recurso2.decrementar();
```

```
usar_recursos();
```

```
recurso2.incrementar();
```

```
recurso1.incrementar();
```

Adquisición de recursos (2): 2 recursos por proceso

Programa 1 → ok

```
semáforo_binario recurso1, recurso2;
```

```
void* hebra_a(void*){
```

```
    recurso1.decrementar();
```

```
    recurso2.decrementar();
```

```
    usar_recursos();
```

```
    recurso2.incrementar();
```

```
    recurso1.incrementar();
```

```
}
```

```
void* hebra_b(void*){
```

```
    recurso1.decrementar();
```

```
    recurso2.decrementar();
```

```
    usar_recursos();
```

```
    recurso2.incrementar();
```

```
    recurso1.incrementar();
```

```
}
```

Programa 2 → interbloqueo

```
semáforo_binario recurso1, recurso2;
```

```
void* hebra_a(void*){
```

```
    recurso1.decrementar();
```

```
    recurso2.decrementar();
```

```
    usar_recursos();
```

```
    recurso2.incrementar();
```

```
    recurso1.incrementar();
```

```
}
```

```
void* hebra_b(void*){
```

```
    recurso2.decrementar();
```

```
    recurso1.decrementar();
```

```
    usar_recursos();
```

```
    recurso1.incrementar();
```

```
    recurso2.incrementar();
```

```
}
```

Adquisición de recursos (3): 10 unidades en total

semáforo_contador recurso(10);

Proceso 1

```
for (int i = 0; i < 6; ++i)  
    recurso.decrementar();
```

```
usar_recurso();
```

```
for (int i = 0; i < 6; ++i)  
    recurso.incrementar();
```

Proceso 2

```
for (int i = 0; i < 6; ++i)  
    recurso.decrementar();
```

```
usar_recurso();
```

```
for (int i = 0; i < 6; ++i)  
    recurso.incrementar();
```

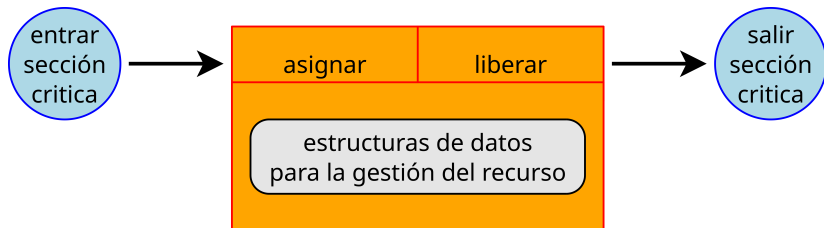
- ⊙ Interbloqueo en caso de reparto 5/5.

Gestión de recursos: recursos exclusivos

- ⊙ Un recurso se dice exclusivo cuando se asigna como una **unidad única e indivisible**.
- ⊙ Un gestor de recursos exclusivos podría implementarse como un **monitor** con dos operaciones:
 - `asignar(recurso)`
 - `liberar(recurso)`
- ⊙ Existe un gran **parecido** entre la gestión de recursos exclusivos y las secciones críticas que deben ejecutarse en exclusión mutua.

Gestión de recursos: recursos **divisibles**

- ⊙ Un gestor de recursos divisibles también podría implementarse como un **monitor**, pero con dos operaciones ligeramente diferentes:
 - `asignar(recurso,cantidad/localización)`
 - `liberar(recurso,cantidad/localización)`
- ⊙ Este tipo de recursos pueden asignarse **por partes**, con lo que deberemos especificar tanto la **cantidad** como la **localización** de cada una de las partes.
- ⊙ Ejemplo: puertos de E/S mapeados en memoria.



Definición de interbloqueo

- ⊙ Definición:

Un conjunto hebras se encuentran interbloqueadas si cada uno de ellas está esperando un suceso que sólo otra hebra del conjunto puede causar.

- ⊙ Habitualmente el suceso esperado es la liberación de algún tipo de recurso o sección crítica.
- ⊙ Ninguno de los procesos puede...
 - continuar la ejecución
 - liberar un recurso o sección crítica
 - ser despertado (desbloqueado)

Condiciones **necesarias** para un interbloqueo

Exclusividad: Cada recurso, o es asignado en exclusiva a un proceso, o está disponible.

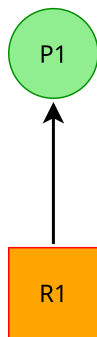
Retención y espera: Los procesos que tiene recursos asignados pueden solicitar nuevos recursos.

No expropiación: Los recursos ya asignados no pueden ser arrebatados al proceso sino que este debe liberarlos de forma voluntaria.

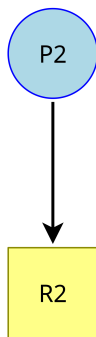
Espera circular: Debe existir una cadena circular de dos o más procesos cada uno de los cuales espera por un recurso en manos de otro proceso.

Condiciones **necesarias** pero **no suficientes** (Coffman, 1971).

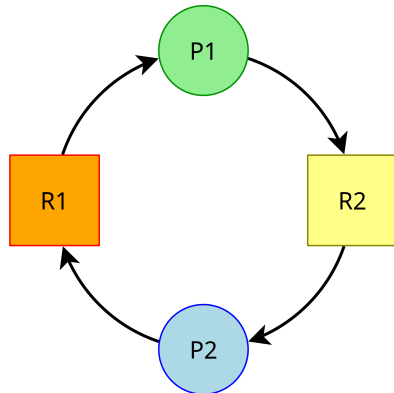
Grafo de asignación de recursos



(a)



(b)



(c)

- a) El recurso R1 ha sido asignado al proceso P1.
- b) El proceso P2 espera a que el recurso R2 le sea asignado.
- c) Ejemplo de interbloqueo.

Resolución de interbloqueos

Prevenir: anular una de las cuatro condiciones necesarias para la aparición de un interbloqueo.

Evitar: asignar recursos de forma cuidadosa para que no pueda aparecer el interbloqueo.

Detectar y recuperar: intentar detectar interbloqueos y tomar medidas para resolverlos en caso de que aparezcan.

Ignorar: no hacer nada con la esperanza de que no aparezca un interbloqueo.

Prevenir: atacar la condición de exclusividad

- ⊙ Algunos recursos necesitan ser utilizados de forma **exclusiva**, ej: impresora, teclado, pila,...
- ⊙ Con algunos recursos podemos hacer “**spooling**”:
 - todo proceso cree utilizar de forma inmediata y exclusiva el recurso.
 - sólo el demonio del recurso lo utiliza realmente.
 - no aplicable a todos los recursos, ej: tabla de procesos.
- ⊙ Otros dispositivos pueden ser **virtualizados**:
 - multiplexación en el tiempo del uso del procesador.
- ⊙ Tanto el spooling como la virtualización introducen **nuevos problemas** en el sistema, ej: competición por espacio en memoria y/o disco.

Prevenir: atacar la condición de retención y espera

- ⊙ El proceso debe **solicitar por adelantado** todos los recursos que va a necesitar.
- ⊙ Si consigue todos los recursos podrá finalizar y devolverlos.
- ⊙ Si no puede conseguirlos todos simultáneamente deberá esperar.
- ⊙ Inconvenientes:
 - Todo proceso debe conocer por adelantado que recursos va a necesitar → entre poco práctico e imposible.
 - No hace un uso óptimo de los recursos.
 - La espera para poder disponer de todos los recursos puede ser larga.

Prevenir: atacar la condición de no expropiación

- ⊙ ¿A quién expropiar?
 - ¿Al proceso que **solicita** un recurso ya asignado?
 - ¿Al proceso **propietario** del recurso solicitado?
- ⊙ La expropiación requiere la capacidad de **almacenar** el estado del recurso expropiado para su posterior **devolución** → no siempre posible.
- ⊙ La expropiación de alguno recursos puede producir resultados **erróneos**, ej: impresora.

¿Se le ocurre alguna forma de hacer una impresora expropiable?

Prevenir: atacar la condición de espera circular

- ⦿ 1ª solución: permitir a cada proceso la asignación de **un único recurso** → demasiado restrictiva.
- ⦿ 2ª solución: imponer un **orden estricto** entre recursos para su asignación:
 - Una vez asignado un recurso no puede solicitar otro recurso si su número de orden es menor.
 - Aplicable a múltiples procesos: el que tenga el recurso con el mayor número de orden siempre podrá finalizar.
 - Es imposible encontrar una ordenación que satisfaga a todo el mundo.
- ⦿ En el grafo de asignación de recursos nunca podrá aparecer un ciclo con lo que nunca se producirá un interbloqueo.

Viabilidad de la prevención

condición	estrategia
exclusividad	multiplexación
retener y esperar	solicitar recursos por adelantado
no expropiación	expropiar recursos
espera circular	ordenación de recursos

- ⊙ Solución demasiado **restrictiva**, ej: asignación en orden.
- ⊙ **Utilización** de recursos demasiado **baja**, ej: solicitar al principio y no poder liberar hasta la finalización.
- ⊙ Demasiada **sobrecarga**, ej: la expropiación requiere tiempo y espacio para almacenar y restaurar.
- ⊙ Consecuencia $\Rightarrow$ ninguna de estas soluciones se emplea en sistemas comerciales.

Evitar interbloqueos (1)

- ⊙ Actuar sobre las decisiones que el sistema toma en tiempo de ejecución con respecto a si una **nueva solicitud** de recurso podría conducir a un interbloqueo.
- ⊙ Dos políticas son posibles:
 - todo/nada: **no iniciar** una **nueva hebra** si la suma de todos los recursos que necesita más los recursos ya asignados a otras hebras puede conducir a un interbloqueo.
 - por partes: **no conceder** solicitudes de **nuevos recursos** si su asignación puede llevar el sistema a un interbloqueo.
- ⊙ Siempre hemos de tener en cuenta el **peor caso** posible.
- ⊙ La mayor restricción es **conocer por adelantado** los recursos que va a necesitar un proceso.

Modelo teórico para evitar interbloqueos (1)

n hebras: $H_1, H_2, \dots, H_n$

m tipos de recursos: $R_1, R_2, \dots, R_m$

cantidad total de cada recurso: $R = (r_1, r_2, \dots, r_m)$

cantidad de recursos disponibles $D = (d_1, d_2, \dots, d_m)$

matriz de
solicitudes:

$$S = \begin{pmatrix} S_{11} & S_{12} & \cdots & S_{1m} \\ S_{21} & S_{22} & \cdots & S_{2m} \\ \cdot & \cdot & \cdots & \cdot \\ S_{n1} & S_{n2} & \cdots & S_{nm} \end{pmatrix}$$

matriz de
asignación:

$$A = \begin{pmatrix} A_{11} & A_{12} & \cdots & A_{1m} \\ A_{21} & A_{22} & \cdots & A_{2m} \\ \cdot & \cdot & \cdots & \cdot \\ A_{n1} & A_{n2} & \cdots & A_{nm} \end{pmatrix}$$

Invariantes y restricciones:

- Los recursos o bien están disponibles o bien han sido asignados:

$$R_i = D_i + \sum_{k=1}^n A_{ki} \quad \forall i$$

- Ninguna hebra solicita más recursos de los que posee el sistema:

$$S_{ki} \leq R_i \quad \forall k, i$$

- Ninguna hebra recibe más recursos de los que solicita:

$$A_{ki} \leq S_{ki} \quad \forall k, i$$

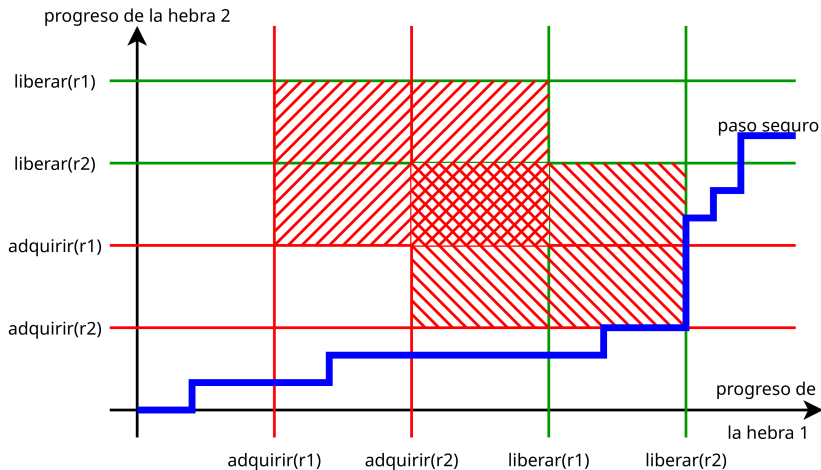
Modelo teórico para evitar interbloqueos (3)

- ⊙ Iniciar una nueva hebra o asignar un nuevo recurso si y sólo si:

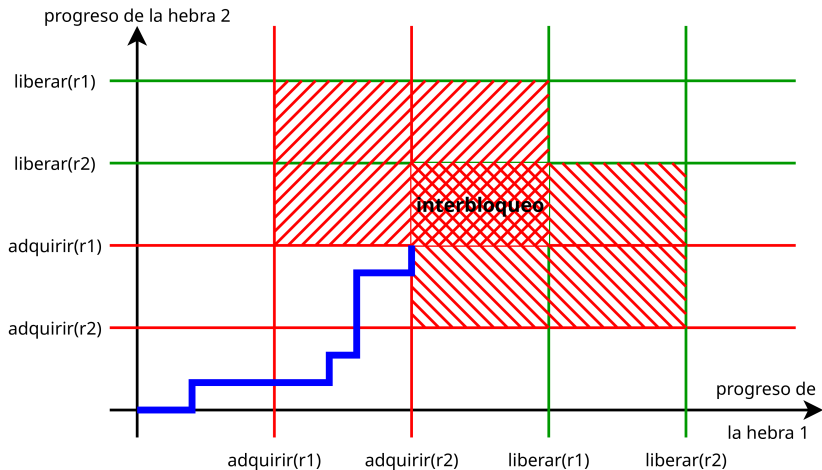
$$R_i \geq S_{n+1,i} + \sum_{k=1}^n S_{ki} \quad \forall i$$

- ⊙ Esta política es muy **pesimista** porque asume que todas las hebras solicitarán sus recursos al **mismo tiempo**.
- ⊙ En la práctica sin embargo suele pasar que...
 - Algunos de los recursos nunca son solicitados.
 - Pocas hebras necesitarán todos sus recursos a la vez.
- ⊙ Es necesario un **algoritmo más optimista y mejor** $\implies$
Algoritmo del banquero

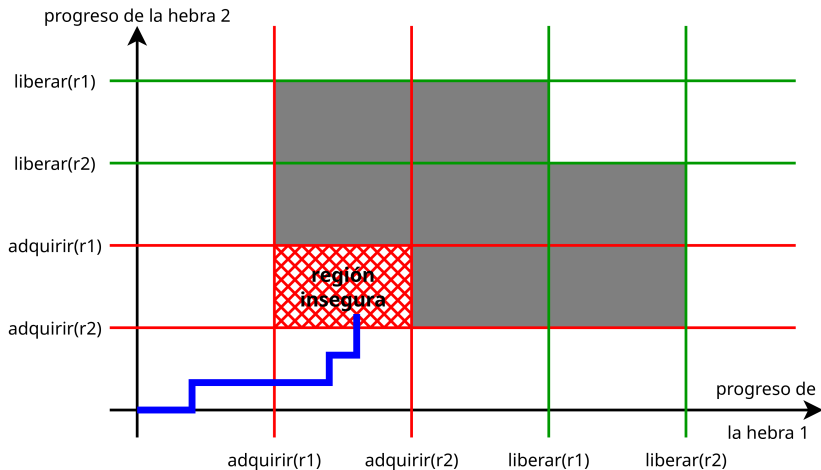
Evitar interbloqueos: estados seguros e inseguros (1)



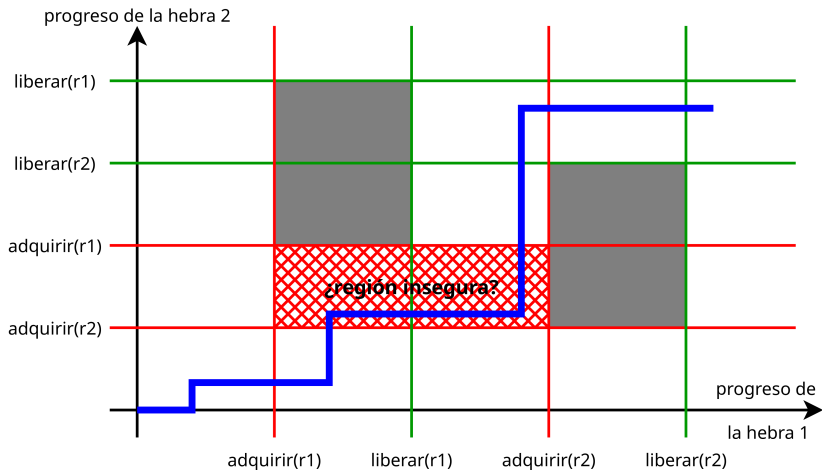
Evitar interbloqueos: estados seguros e inseguros (2)



Evitar interbloqueos: estados seguros e inseguros (3)



Evitar interbloqueos: estados seguros e inseguros (4)



Definición: El estado de un sistema con n hebras es **seguro** mientras exista al menos una **secuencia de ejecución** que permita a **todas finalizar**.

Formalmente:

El estado del sistema es seguro si y sólo si existe una permutación $\langle H_{k_1}, H_{k_2}, \dots, H_{k_n} \rangle$ dentro de $\{H_1, H_2, \dots, H_n\}$

$$\forall i \in \{1, 2, \dots, n\} : \quad S_{k_i} - A_{k_i} \leq D_i + \sum_{s=1}^{i-1} A_{k_s}$$

o

$$\forall i \in \{1, 2, \dots, n\} : \quad S_{k_i} - A_{k_i} \leq R_i - \sum_{s=1}^n A_{k_s}$$

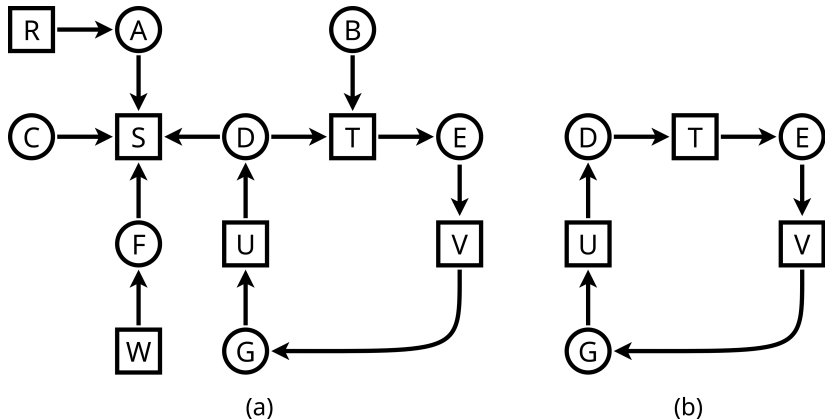
```
const int n;           // número de hebras
const int m;           // numero de recursos
int R[m], D[m];        // recursos totales y disponibles
int S[n][m], A[n][m];  // recursos solicitados y asignados
hebra H[n], HI[n];     // conjunto de hebras/interbloqueadas

respuesta_t banquero()
{
    HI = H; // inicialmente todas interbloqueadas
    for(int i = 0; i < n; ++i) // para cada hebra
        if (H[i] ∈ HI && S[i,*] - A[i,*] < D[*]) // ¿asignable?
        {
            sacar H[i] de HI;           // hebra no interbloqueable
            D[*] = D[*] - A[i,*]; // decrementar disponible
            if (HI vacio)
                return seguro;
        }
    return inseguro;
}
```

Detección de interbloqueos (1)

- ⊙ Como parece que prevenir o evitar los interbloqueos no funciona demasiado bien, vamos a intentar detectarlos para poder deshacerlos.
- ⊙ Una solución es modificar el algoritmo del banquero para relajar sus condiciones y hacerlo más eficiente:
 $P[n][m]$: matriz de solicitudes de recursos pendientes de asignar.
 $\text{if}(H[i] \in HI \ \&\& \ P[i,*] \leq D[*]) \dots$
- ⊙ En caso de interbloqueo HI contendrá la lista de hebras causantes.

Detección de interbloqueos (2)



(a) Estudie los recursos asignados y solicitados.

(b) Ciclo descubierto en el grafo de asignación de recursos
⇒ interbloqueo.

¿Con qué frecuencia utilizar el algoritmo de detección?

- ⦿ En el caso extremo de que una solicitud de recurso **no** pueda ser **atendida**
 - ¿Qué coste tendría esto para el sistema?
 - ¿Cuánto incrementaría la complejidad del sistema?
 - ¿Cómo afecta a la temporización?
 - ¿Es efectivo?
- ⦿ **Periódicamente**, pero... ¿con qué periodicidad?
 - ¿Cuánto tiempo estamos dispuestos a esperar tras un interbloqueo para detectarlo?
 - ¿Cuántos recursos del sistema estamos dispuestos a utilizar en el proceso de detección?

Recuperación de un interbloqueo (1)

¿Qué hacer una vez detectado un interbloqueo? → **finalizar procesos**

- ⊙ Finalizar **todos** los procesos interbloqueados.
 - Es rápido pero desperdicia mucho trabajo.
- ⊙ Finalizar procesos **de uno en uno** hasta que desaparezca el interbloqueo.
 - Este método consume más recursos.
 - Es mejor en cuanto a la cantidad de trabajo aprovechado.
 - ¿En qué orden finalizar los procesos?...
- ⊙ **Expropiar recursos.**
 - ¿De qué forma seleccionar el proceso víctima?
 - ¿Permite el sistema volver atrás y recuperarse a un proceso de una expropiación?
 - ¿Cómo protegernos de la inanición?...

Recuperación de un interbloqueo (2)

- ⦿ Recuperación mediante **expropiación**:
 - Tomar el recurso en conflicto de otro proceso.
 - Método aplicable sólo a recursos expropiables.
- ⦿ Recuperación mediante **vuelta atrás** (SO persistente):
 - Almacenar el estado de los procesos periódicamente.
 - Reiniciar el sistema a uno de los estados previos al interbloqueo.
- ⦿ Recuperación mediante **finalización**:
 - Método burdo pero eficaz.
 - Eliminar procesos del ciclo que condujo al interbloqueo hasta permitir a algún otro continuar.
 - Si disponemos de información elegir procesos reiniciables (compilador/base de datos).

Criterios para elegir el proceso a finalizar:

- ⊙ Tamaño de las peticiones pendientes.
- ⊙ Cantidad de recursos asignados.
- ⊙ Prioridad.
- ⊙ Nivel del proceso: usuario/sistema.
- ⊙ Tiempo de ejecución acumulado.
- ⊙ Tiempo de ejecución restante.
- ⊙ Escriba el que más le guste/interese.

El algoritmo del avestruz:



- ⦿ Suponer que no existe el problema del interbloqueo.
- ⦿ Razonable si...
 - los interbloqueos suceden muy raramente.
 - el coste de prevenir, evitar y detectar y recuperar es demasiado alto.
- ⦿ Linux y Windows utilizan este método.
- ⦿ Es un compromiso entre conveniencia y corrección.

Los enfoques anteriores pueden combinarse:

- ⊙ Agrupar los recursos en un cierto número de categorías y ordenarlas, ej:
 - área de intercambio
 - recursos de proceso: dispositivos de E/S, ficheros,...
 - memoria principal
- ⊙ Utilizar prevención de cola circular para evitar interbloqueos entre estas categorías de recursos.
- ⊙ Utilizar el método más apropiado contra interbloqueos dentro de cada categoría de recursos.