Sistemas Operativos

Deadlocks

Curso 2024

Facultad de Ingeniería, UDELAR

Agenda

1. Introducción
2. Caracterización de deadlocks
3. Prevención de deadlocks
4. Evitación de deadlocks
5. Detección y recuperación de deadlocks

Introducción

Introducción

- Una de las principales tareas de un sistema operativo es asignar recursos a procesos
- Muchos recursos se pueden asignar a un solo proceso a la vez
- Un proceso puede necesitar reservar más de un recurso (espacios de memoria para copiar datos, múltiples registros en una base de datos, etc.)
- En general, una vez asignado un recurso, no existe un mecanismo simple para que el sistema pueda recuperarlo. Debe esperar que el proceso lo libere o finalice su ejecución
- Pueden generarse deadlocks: situaciones en las que un conjunto de procesos queda bloqueado, cada proceso esperando que se libere un recurso que está controlado por otro proceso

Acceso mutuo a una sección crítica

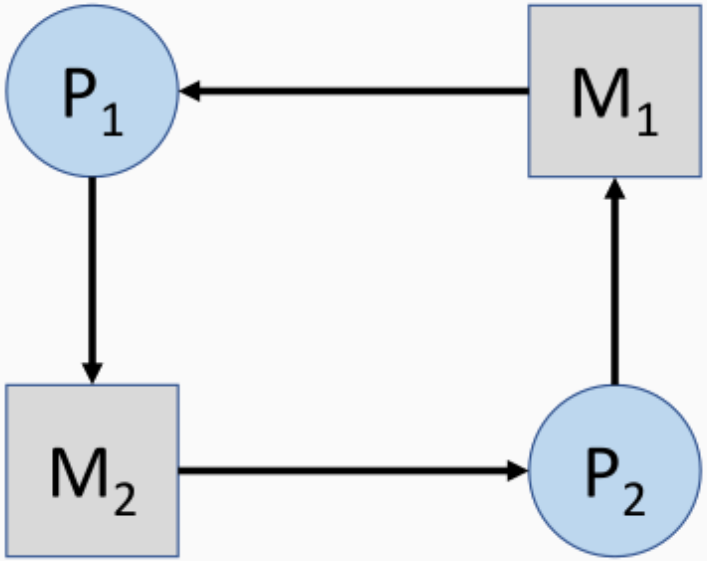
Dos procesos que acceden a una región crítica haciendo uso de dos recursos protegidos por mutex M1 y M2

```
while TRUE do
   NonCriticalSect();
   Mutex_lock(&M1);
   Mutex_lock(&M2);
   CriticalSection();
   Mutex_unlock(&M2);
   Mutex_unlock(&M1);
end while
```

```
while TRUE do
   NonCriticalSect()
   Mutex_lock(&M2);
   Mutex_lock(&M1);
   CriticalSection();
   Mutex_unlock(&M1);
   Mutex_unlock(&M2);
end while
```

Acceso mutuo a una sección crítica

La situación se ejemplifica por el diagrama/grafo



Existen otros patrones más complejos de deadlocks, como las dependencias circulares entre múltiples procesos (e.g., problema de los filósofos)

Deadlocks: formalización

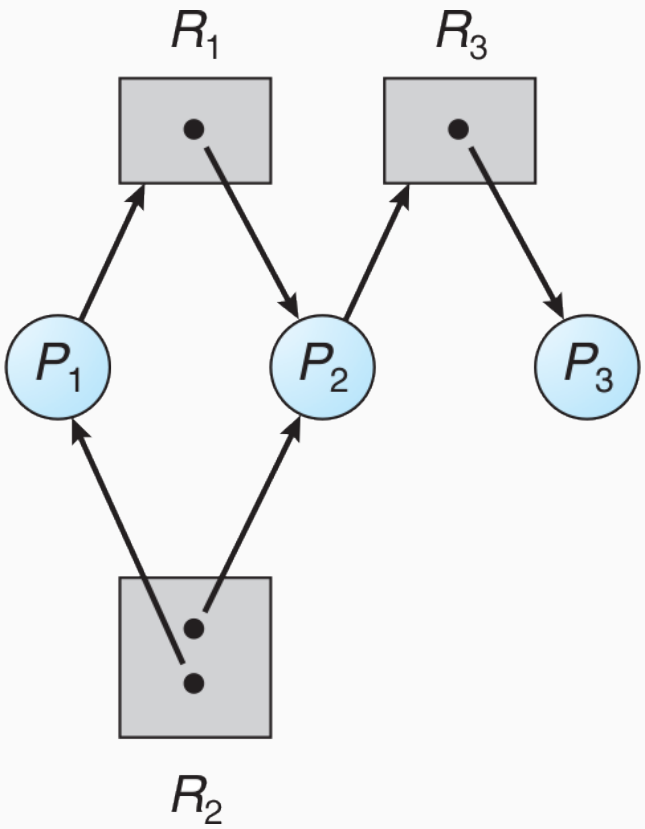
Modelo del sistema computacional:

- Cantidad finita de recursos y procesos
- Varios tipos de recursos con una o más copias idénticas
- Modo de uso: obtener, usar, liberar
- Los procesos esperan en una cola cuando solicitan un recurso ya asignado a otro proceso.
- Los procesos no solicitan más recursos de los existentes
- Un conjunto de procesos está en deadlock cuando cada uno espera por un recurso cuya liberación depende exclusivamente de otro proceso del mismo conjunto.

Deadlocks: formalización

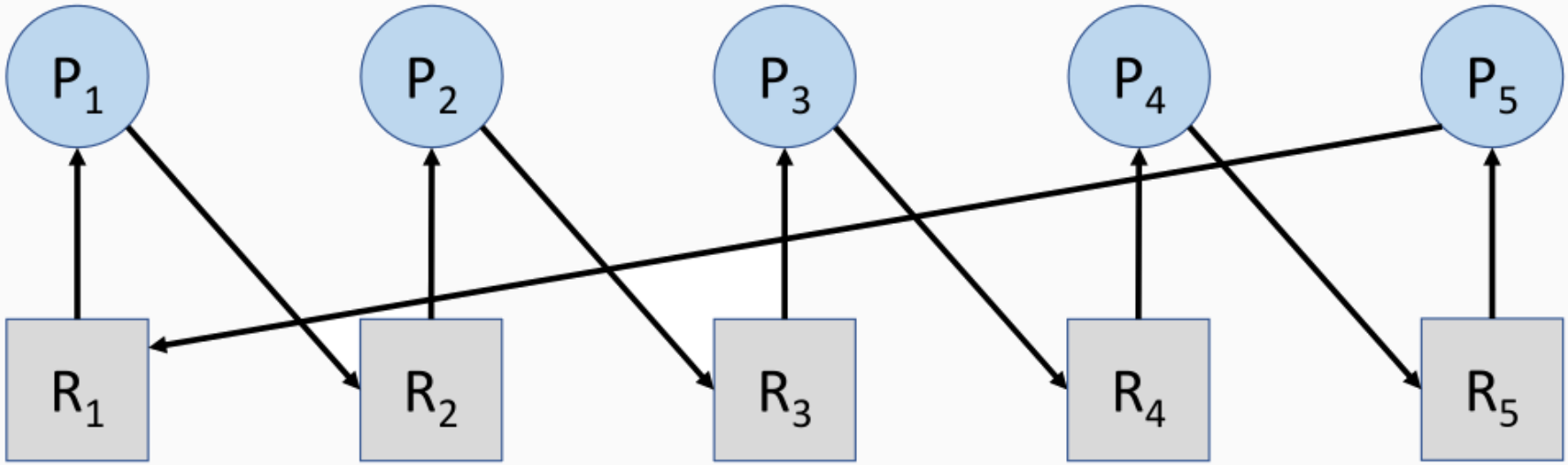
Grafo de asignación de recursos:

- Los nodos representan a los procesos (P_n) y recursos (R_n).
- Una arista $P_i \rightarrow R_j$ indica que el P_i requiere el recurso R_j.
- Una arista $P_j \leftarrow R_i$ indica que el recurso R_j esta asignado al proceso P_i.



Ejemplo

Deadlock para el problema de los filósofos



Caracterización de deadlocks

Condiciones necesarias

Puede ocurrir un deadlock si se cumplen cuatro condiciones simultáneamente (condiciones de Coffman)

1. **Exclusión mutua**: al menos un recurso debe ser de uso exclusivo (no compartible).
2. **Retención y espera**: un proceso debe mantener un recurso mientras espera para obtener otro.
3. **No expropiable**: los recursos no deben ser expropiables (se liberan voluntariamente).
4. **Espera circular**: dados los procesos $\{P_0, P_1, ..., P_n\}$, P_0 debe esperar por un recurso obtenido por P_1, P_1 por uno de P_2, ..., P_n por uno de P_0.

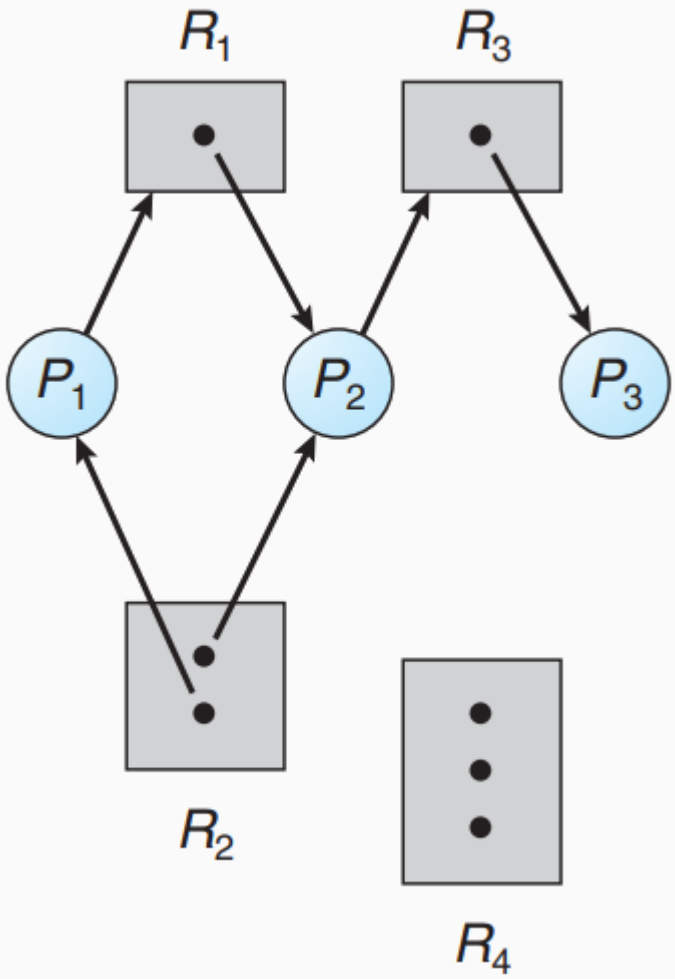
Las cuatro condiciones son necesarias individualmente y suficientes cuando se tiene una sola instancia de cada recurso

Deadlocks y ciclos

- Si el grafo de asignación de recursos no contiene ciclos, entonces los procesos no sufren deadlock
- Si solo existe una instancia de un recurso, entonces un ciclo implica deadlock (condición necesaria y suficiente). Cada proceso involucrado en el ciclo puede sufrir deadlock.
- Si un recurso tiene varias instancias, un ciclo no implica necesariamente que se produzca deadlock. El ciclo es condición necesaria pero no suficiente para que exista deadlock.

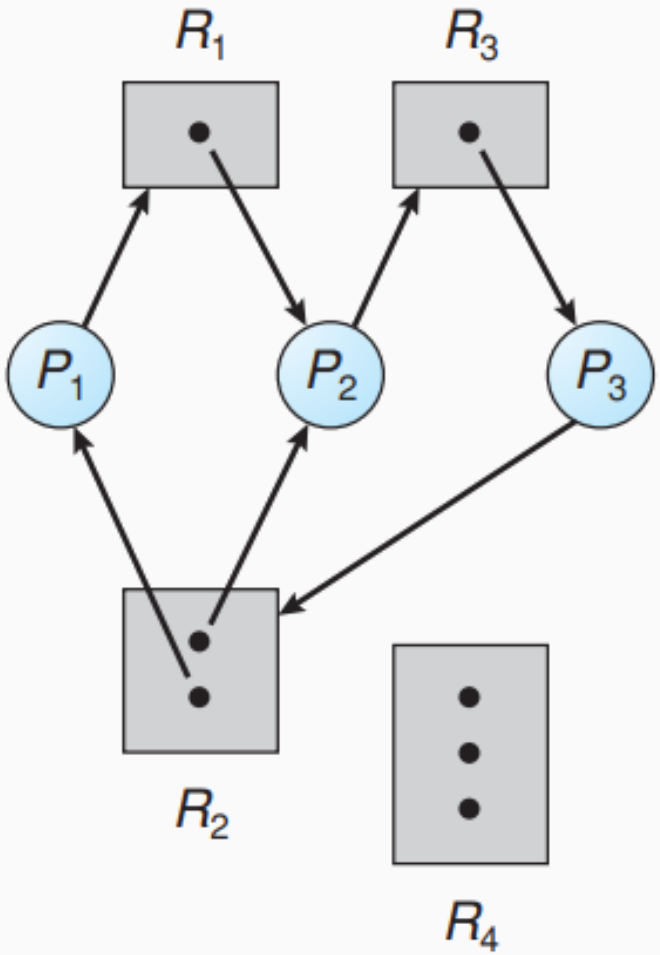
Deadlocks y ciclos

¿Puede existir deadlock en el siguiente grafo de asignación de recursos?



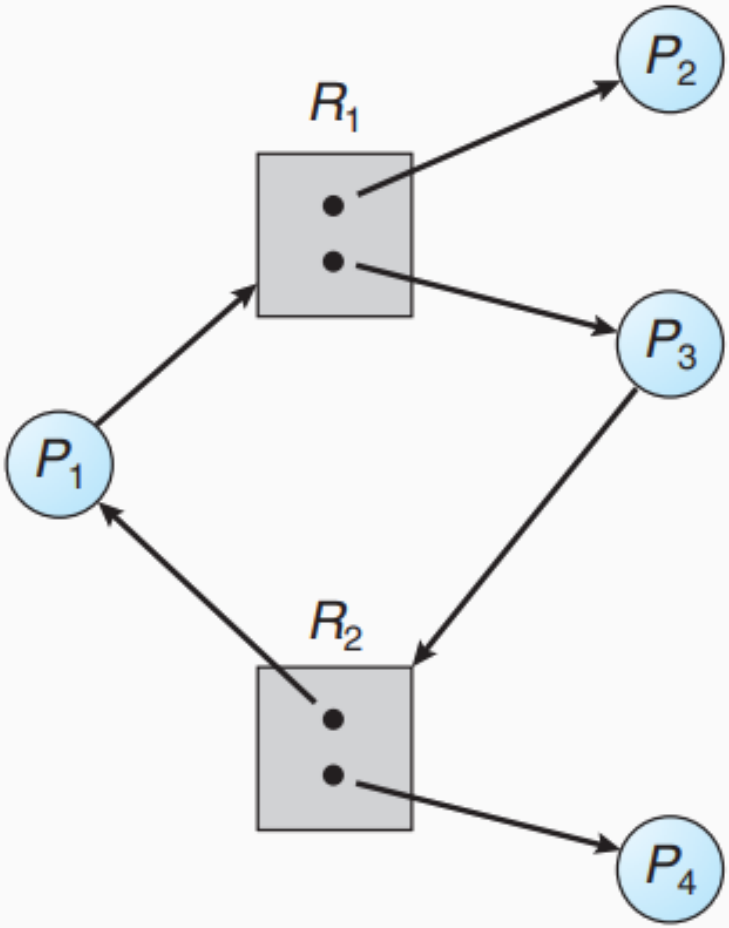
Deadlocks y ciclos

¿Puede existir deadlock en el siguiente grafo de asignación de recursos?



Deadlocks y ciclos

¿Puede existir deadlock en el siguiente grafo de asignación de recursos?



Cómo abordar el problema de deadlocks

En general, los sistemas operativos no pueden evitar deadlocks sin incluir lógica específica con ese cometido

Las principales estrategias para abordar el problema de deadlocks se enfocan en:

- **Ignorar**: bajo el supuesto que un deadlock es un evento de probabilidad muy baja
- **Prevenir o evitar**: utilizar un protocolo para prevenir o evitar que ocurra alguna de las condiciones de Coffman (especialmente, la condición #4, espera circular)
- **Detectar y recuperar**: permitir que ocurran deadlocks, pero detectarlos y recuperar el sistema

Ignorar deadlocks

- Algoritmo del avestruz: se espera que el deadlock no suceda
- Se basa en la baja probabilidad de un evento de deadlock
- Evita las complejidades y la reducción de desempeño al incluir un sistema de detección o prevención de deadlocks
- Si se produce un deadlock, puede ser necesario desactivar el sistema o eliminar manualmente un conjunto de procesos
  (puede ser una solución aceptable para muchos sistemas).

Prevención de deadlocks

Prevención de deadlocks

Estrategia genérica

Asegurar que al menos una de las cuatro condiciones necesarias nunca se cumpla.

Prevención de deadlocks: exclusión mutua

- La exclusión mutua es muy difícil de prevenir
- Algunos recursos son compartibles (no requieren acceso en exclusión mutua) y no están involucrados en un deadlock
  (e.g., memoria o archivos de solo lectura)
- Otros recursos son intrínsecamente de uso exclusivo (archivos de escritura, mutex, etc.) y no puede negarse la condición de exclusión mutua

Prevención deadlocks: retención y espera

- Un proceso puede solicitar recursos solamente cuando no tiene ningún recurso asignado
- Los recursos son asignados en bloque (todos a la vez) o no son asignados
- Un proceso debe liberar todos sus recursos antes de solicitar recursos adicionales
- Desventajas:
  - Bajo aprovechamiento de recursos
  - Puede generar posposición indefinida

Prevención de deadlocks: recursos no expropiables

- Se expropian todos los recursos de un proceso si solicita un nuevo recurso que no le puede ser asignado inmediatamente.
- El proceso continuará su ejecución cuando pueda volver a obtener los recursos que tenía asignados, junto con el nuevo recurso solicitado.
- Aplicable a recursos cuyo estado puede ser almacenado y recuperado de modo sencillo (registros, direcciones de memoria, etc.)
- No aplicable a semáforos, mutex, etc.

Prevención de deadlocks: espera circular

Se define un orden total en el conjunto de recursos y se impone a cada proceso a obtener los recursos en orden. Dado un conjunto de recursos $R = \{R_1, R_2, \ldots, R_m\}$, se define una función $F : R \to N$ que indica el orden.

Un protocolo para evitar deadlocks se basa en que los procesos solo pueden requerir recursos en orden creciente de enumeración.

El protocolo se basa en dos condiciones:

1. inicialmente, un proceso puede obtener cualquier recurso R_i, pero luego solo puede requerir recursos R_j, tal que $F(R_j) \geq F(R_i)$
2. para obtener un recurso R_j se deben liberar todos los recursos R_i tal que $F(R_i) \geq F(R_j)$

Evitación de deadlocks

Evitación de deadlocks

Los algoritmos de prevención se basan en limitar las solicitudes, para asegurar que alguna condición de Coffman no suceda.

Efectos secundarios: baja utilización de recursos y reducción del rendimiento del sistema.

Método alternativo: usar información adicional sobre cómo se solicitarán y liberarán los recursos.

El sistema puede decidir para cada solicitud si el proceso debe esperar o no, para evitar un posible deadlock.

Ante cada solicitud, el sistema considera los recursos actualmente disponibles, los recursos actualmente asignados a cada proceso y las futuras solicitudes y liberaciones de cada proceso.

Evitación de deadlocks

Los algoritmos que utilizan este enfoque difieren en la cantidad y el tipo de información requerida.

El modelo más simple y útil requiere que cada proceso declare la cantidad máxima de recursos de cada tipo que puede necesitar.

Un algoritmo para evitar deadlocks examina dinámicamente la asignación de recursos para garantizar que nunca pueda existir una condición de espera circular.

Se define el estado de asignación de recursos por la cantidad de recursos disponibles, los recursos asignados y las demandas máximas de los procesos.

Evitación de deadlocks

Estado seguro: un estado E es seguro si el sistema puede satisfacer los pedidos de recursos en algún orden evitando caer en deadlock (i.e. existe al menos una secuencia segura).

Secuencia segura: Una secuencia de procesos $< P_0, ..., P_n >$ es segura para un estado E si los recursos pedidos por cada proceso P_i pueden ser satisfechos por los recursos disponibles más los recursos retenidos por todo P_j con $j < i$.

Un estado es inseguro si no existe una secuencia segura.

Un estado inseguro puede llevar a un deadlock, dependiendo del orden en que los procesos soliciten y liberen los recursos.

Evitación de deadlocks

Un algoritmo de evitación asegura que el sistema se encuentra siempre en un estado seguro.

Diferentes algoritmos según la cantidad de instancias por recurso.

- **Única instancia**: Algoritmo de detección de ciclos en el grafo de asignación de recursos
- **Múltiples instancias**: Algoritmo del Banquero

Ejemplo con múltiples instancias de un recurso

Dado un sistema con 12 unidades de un recurso.

Se tiene el siguiente estado en t_0:

	Máximo	Actual	Diferencia
P_0	10	5	5
P_1	4	2	2
P_2	9	2	7

Hay 3 unidades libres.

Secuencia segura: $< P_1, P_0, P_2 >$

Ejemplo con múltiples instancias de un recurso

Dado un sistema con 12 unidades de un recurso.

Se tiene el siguiente estado en t_0:

	Máximo	Actual	Diferencia
P_0	10	5	5
P_1	4	2	2
P_2	9	2	7

Hay 3 unidades libres.

Secuencia segura: $< P_1, P_0, P_2 >$

¿Qué sucede si en t_1 el proceso P_2 solicita y se le asigna un recurso adicional?

Ejemplo con múltiples instancias de un recurso

Estado en t_1 (2 unidades libres):

	Máximo	Actual	Diferencia
P_0	10	5	5
P_1	4	2	2
P_2	9	3	6

El sistema ya no está en un estado seguro.

Solo se le pueden asignar todos sus recursos al proceso P_1. Cuando los devuelva, el sistema tendrá 4 unidades disponibles. P_0 puede solicitar 5 unidades adicionales y tendrá que esperar, porque no están disponibles. P_2 puede solicitar 6 unidades adicionales y debe esperar, generando un deadlock.

Ejemplo con múltiples instancias de un recurso

El error fue conceder la solicitud del proceso P_2 para obtener una unidad adicional

Si se hubiera forzado a P_2 a esperar que alguno de los otros procesos terminara y liberara sus recursos, se hubiera evitado el deadlock

Detección de ciclos en el grafo de asignación de recursos

Si existe solo una instancia de cada recurso, es posible trabajar sobre el grafo de asignación de recursos para evitar deadlocks.

Además de las aristas de solicitud y asignación, se agregan aristas de reclamo: $P_i \dashrightarrow R_j$ indica que el proceso P_i puede solicitar el recurso R_j en algún momento (en el futuro)

La arista de reclamo se convierte a una arista de solicitud cuando P_i solicita el recurso R_j

Cuando P_i libera un recurso R_j, la arista de asignación $R_j \rightarrow P_i$ se reconvierte en una arista de reclamo $P_i \dashrightarrow R_j$.

Los recursos deben reclamarse a priori en el sistema, antes de iniciar la ejecución

Detección de ciclos en el grafo de asignación de recursos

Un proceso P_i solicita el recurso R_j: solo se le concede si convertir la arista de reclamo $P_i \dashrightarrow R_j$ en una de asignación $R_j \rightarrow P_i$ no forma de un ciclo en el grafo de asignación de recursos

Se aplica un algoritmo de detección de ciclos (ejecuta en $O(n^2)$ operaciones, n es el número de procesos en el sistema)

Si no existe ciclo, la asignación del recurso dejará el sistema en un estado seguro

Si se encuentra un ciclo, la asignación dejaría al sistema en un estado inseguro. El proceso P_i debe esperar que se satisfagan sus solicitudes

Caso general: múltiples instancias de un recurso

Cuando existen múltiples instancias de un recurso, no es aplicable el algoritmo de detección de ciclos en el grafo de asignación de recursos

Deben aplicarse métodos más complejos, ej., el Algoritmo del Banquero

Premisas:

- Los procesos deben declarar la máxima cantidad a utilizar de cada recurso.
- Se satisfacen pedidos de recursos solamente si el sistema se mantiene en un estado seguro. En caso contrario, el proceso solicitante debe esperar.

Algoritmo del Banquero

Se requieren estructuras en memoria necesarias para implementar el algoritmo para n procesos y m tipos de recursos.

- `Disponible`: vector de largo m con cantidad de recursos disponibles.
- `Max`: matriz $n \times m$ indicando la demanda máxima de cada recurso de cada proceso.
- `Asignado`: matriz $n \times m$ indicando la asignación actual de cada recurso a cada proceso.
- `Resta`: matriz $n \times m$ indicando los restantes recursos necesitados por cada proceso (`Max` – `Asignado`).

Algoritmo del Banquero

Parte 1: Verificar que un estado sea seguro

1. Definir `Trabajo` vector de largo m con `Trabajo = Disponible` y `Fin` vector de largo n con `Fin = {false,...,false}`
2. Encontrar un índice i tal que $\texttt{Fin}_i$ `== false` y $\texttt{Resta}_i \leq$ `Trabajo`. Si no se encuentra ir al paso 4.
3. Setear `Trabajo = Trabajo +` $\texttt{Asignado}_i$ y $\texttt{Fin}_i$ `= true`, e ir al paso 2.
4. Si $\texttt{Fin}_i$ `== true` para todo i, el estado es seguro.

Algoritmo del Banquero

Parte 2: Determinar si un pedido de recursos del proceso i ($\texttt{Pedido}_i$) puede ser satisfecho.

1. Si $\texttt{Pedido}_i \leq \texttt{Disponible}$ ir al paso 2.
   Sino, el proceso i debe esperar.
2. Actualizar:
   $\texttt{Disponible = Disponible - Pedido}_i$
   $\texttt{Asignado}_i \texttt{ = Asignado}_i \texttt{ + Pedido}_i$
   $\texttt{Resta}_i \texttt{ = Resta}_i \texttt{ - Pedido}_i$
   Verificar que el estado sea seguro (Parte 1).
   - Si el estado es seguro, se asignan los recursos.
   - Sino, se deshacen los cambios y el proceso i debe esperar.

Ejemplo del Algoritmo del Banquero

Sea un sistema con con cinco procesos y tres recursos:
A (10 instancias), B (5 instancias) y C (7 instancias).

En tiempo t_0 se tiene:

	Asignado			*Resta*			*Max*		
	A	B	C	A	B	C	A	B	C
P_0	0	1	0	7	4	3	7	5	3
P_1	2	0	0	1	2	2	3	2	2
P_2	3	0	2	6	0	0	9	0	2
P_3	2	1	1	0	1	1	2	2	2
P_4	0	0	2	4	3	1	4	3	3

Disponible		
A	B	C
3	3	2

El sistema está en un estado seguro
Secuencia segura: $< P_1, P_3, P_4, P_2, P_0 >$

Ejemplo del Algoritmo del Banquero

En el tiempo t_1 el proceso P_0 solicita (A=3, B=2, C=1) recursos

Se actualiza el estado del sistema:

	Asignado			*Resta*			*Max*		
	A	B	C	A	B	C	A	B	C
P_0	(0+3)	(1+2)	(0+1)	(7-3)	(4-2)	(3-1)	7	5	3
P_1	2	0	0	1	2	2	3	2	2
P_2	3	0	2	6	0	0	9	0	2
P_3	2	1	1	0	1	1	2	2	2
P_4	0	0	2	4	3	1	4	3	3

Disponible		
A	B	C
(3-3)	(3-2)	(2-1)

El nuevo estado es seguro

Secuencia segura: $< P_3, P_1, P_0, P_2, P_4 >$

Ejemplo del Algoritmo del Banquero

En tiempo t_1 el proceso P_0 solicita (A=3, B=2, C=2) recursos.

Se actualiza el estado del sistema:

	Asignado			*Resta*			*Max*		
	A	B	C	A	B	C	A	B	C
P_0	(0+3)	(1+2)	(0+2)	(7-3)	(4-2)	(3-2)	7	5	3
P_1	2	0	0	1	2	2	3	2	2
P_2	3	0	2	6	0	0	9	0	2
P_3	2	1	1	0	1	1	2	2	2
P_4	0	0	2	4	3	1	4	3	3

Disponible		
A	B	C
(3-3)	(3-2)	(2-2)

El nuevo estado no es seguro

No hay una secuencia segura

Detección y recuperación de deadlocks

Detección y recuperación de deadlocks

Si un sistema no emplea un algoritmo de prevención o de evitación, un deadlock puede ocurrir.

En esta situación, el sistema debe permitir detectar el deadlock
y proporcionar un mecanismo de recuperación.

Algoritmo de detección de deadlocks

Un algoritmo de detección examina el sistema y determina si existe un deadlock.

Si existe una única instancia de cada recurso: detección de ciclos en el grafo de asignación de recursos.

Si existen múltiples instancias de recursos: variante del Algoritmo del Banquero.

Aspectos clave:

- ¿Con que frecuencia ejecutar el algoritmo de detección? Es difícil determinar un balance apropiado (eficacia/overhead).
- ¿Cuántos procesos serán afectados por el deadlock cuando ocurre?

Algoritmo de recuperación de deadlocks

Para recuperar el sistema luego de detectar un estado de deadlock, existen varias alternativas

No automática: Notificación a un operador y recuperación manual

Automática: Finalización de procesos:

- Terminar todos los procesos involucrados en el deadlock.
  Alto costo, se pierde el cómputo realizado por todos los procesos
- Terminar de a uno los procesos, hasta eliminar el deadlock.
  Alto overhead, se debe invocar un algoritmo de detección luego de cancelar cada proceso

Algoritmo de recuperación de deadlocks

Finalizar un proceso puede no ser sencillo

Debe prestarse atención a los recursos que utiliza (e.g., si el proceso está escribiendo un archivo, el archivo quedará en un estado incorrecto)

La terminación parcial requiere una política para determinar cuántos y cuales procesos deben terminarse

Debe considerarse la prioridad del proceso, cuánto ha computado y cuánto le resta por computar, cuántos y qué tipos de recursos ha utilizado el proceso, cuántos recursos más necesita para completarse, el tipo del proceso (interactivo o batch), etc.

Algoritmo de recuperación de deadlocks

Automática: Expropiación de recursos, para asignarlos a otros procesos hasta que se resuelve el deadlock

Aspectos clave:

- Debe seleccionarse los procesos víctima y los recursos a expropiar. Criterios basados en cantidad de recursos asignados y cuánto ha procesado hasta el momento
- Requiere roll back del proceso víctima a un estado seguro, que no es sencillo determinar (requiere almacenar información y checkpoints). Alternativa: rollback total, cancelar el proceso y reiniciarlo
- Puede generar posposición indefinida de procesos. Políticas para evitar expropiar recursos a los mismos procesos: considerar el número de rollbacks ya realizados