# *Deadlocks*
# 8

## Sistemas operativos y distribuidos

Gustavo Distel
gd@cs.uns.edu.ar

DCIC - UNS

## *Deadlocks*

Contenido

- Modelo de sistema.
- *Deadlock* en aplicaciones multi-hiladas.
- Caracterización de los *deadlocks*.
- Métodos para manejar *deadlocks*.
- Prevención de *deadlocks*.
- Evasión de *deadlocks*.
- Detección de *deadlocks*.
- Recuperación de *deadlocks*.



## *Deadlocks*

- En un entorno de multiprogramación, varios hilos pueden competir por un número finito de recursos.
- Un hilo solicita recursos, y si los recursos no están disponibles pasa al estado de espera.
- Un hilo en espera podría nunca cambiar de estado, porque los recursos que ha solicitado están en poder de otros hilos en espera → *deadlock*.
- Como ya vimos, los *deadlocks* son un tipo de *liveness*.
  - *Deadlock*: situación en la que cada proceso en un conjunto de procesos está esperando un evento que solo puede ser causado por otro proceso del conjunto.
- Los SOs generalmente no proveen ningún tipo de prevención de *deadlock*, y sigue siendo responsabilidad de los programadores asegurarse de diseñar programas libres de *deadlocks*.
- Los problemas de *deadlocks*, así como otros *liveness*, se están volviendo más desafiantes a medida que se incrementa la demanda de concurrencia y paralelismo en los sistemas *multicore*.



## Modelo de sistema

- Los recursos por los cuales lo hilos compiten pueden dividirse en varios tipos (o clases), cada uno de los cuales consta de cierto número de instancias idénticas, tales como:
  - Ciclos de *CPU*.
    - *CPU quad core* (el tipo de recurso *CPU* tiene cuatro instancias).
  - Archivos.
  - Dispositivos de E/S (Interfaces de red y unidades de *DVD*).
    - Dos interfaces de red (el tipo de recurso red tiene dos instancias).
- Si un hilo solicita una instancia de un tipo de recurso, la asignación de cualquier instancia debe satisfacer la solicitud. Si no es así, las instancias no son idénticas y las clases de tipo de recurso no se han definido correctamente.

# Modelo de sistema

- Un hilo debe solicitar un recurso antes de usarlo y debe liberar el recurso después de usarlo.
- Obviamente, el número de recursos solicitados no puede exceder el número total de recursos disponibles en el sistema.
- Bajo el modo normal de operación, un hilo puede utilizar un recurso en la siguiente secuencia:
  - 1. Solicitud: el hilo solicita el recurso. Si la solicitud no se puede otorgar de inmediato (por ej., si un *mutex lock* está siendo retenido actualmente por otro hilo), el hilo solicitante debe esperar hasta que pueda adquirir el recurso.
  - 2. Uso: el hilo puede operar en el recurso (por ej., si el recurso es un *mutex lock*, el hilo puede acceder a su sección crítica).
  - 3. Liberación: el hilo libera el recurso.
- La solicitud y liberación de recursos pueden ser llamadas del sistema, por ej.:
  - Archivos: `open(), close()`.
  - Memoria: `allocate(), free()`.



# Modelo de sistema

- Una tabla del sistema registra si cada recurso está libre o asignado a un hilo.
- Si un hilo solicita un recurso que está asignado a otro hilo, se agrega a una cola de hilos que esperan ese recurso.
- Un conjunto de hilos está en un estado de *deadlock* cuando cada hilo del conjunto espera un evento que solo puede ser causado por otro hilo del conjunto.
- Los eventos que nos interesan son la adquisición y liberación de recursos.
- Los recursos suelen ser lógicos (por ej, *mutex locks*, semáforos y archivos); sin embargo, otros tipos de eventos pueden provocar *deadlocks*, incluida la lectura desde una interfaz de red o el *IPC*.
- Recordar el deadlock que se podía producir en el problema de los filósofos.



# *Deadlocks* en aplicaciones multi-hiladas

```
void *routineA (void *t){
  long my_id = (long)t;
    while(true){
      printf("[%ld] Acceso\n", my_id);
      pthread_mutex_lock(&mutexA);
      pthread_mutex_lock(&mutexB);
      printf("[%ld]recurso compartido:
              %d\n", my_id, cont++);
      pthread_mutex_unlock(&mutexB);
      pthread_mutex_unlock(&mutexA);
    }
  pthread_exit (EXIT_SUCCESS);
}
```

```
void *routineB (void *t){
  long my_id = (long)t;
    while(true){
      printf("[%ld] Acceso\n", my_id);
      pthread_mutex_lock(&mutexB);
      pthread_mutex_lock(&mutexA);
      printf("[%ld] recurso compartido:
              %d\n", my_id, cont++);
      pthread_mutex_unlock(&mutexA);
      pthread_mutex_unlock(&mutexB);
    }
  pthread_exit (EXIT_SUCCESS);
}
```



# *Deadlocks* en aplicaciones multi-hiladas

*Livelock*

- Los *livelock* son otra forma de *liveness failure*.
- Es similar a un *deadlock* en que ambos evitan que continúen dos o más hilos.
- Mientras que el *deadlock* ocurre cuando cada hilo de un conjunto está bloqueado esperando un evento que solo puede ser causado por otro hilo del conjunto, el *livelock* ocurre cuando un hilo intenta continuamente una acción que falla.
- Se puede obtener un *livelock* si modificamos el código anterior agregando la función `pthread_mutex_trylock()`, que intenta adquirir un *mutex lock* sin bloquearse.
- Los *Livelock* ocurren generalmente cuando los hilos reintentan operaciones fallidas al mismo tiempo.
- Se puede evitar haciendo que cada hilo vuelva a intentar la operación de falla en momentos aleatorios.

# Caracterización de los *deadlocks*

## Condiciones necesarias

- Una situación de *deadlock* puede surgir si se dan simultáneamente en un sistema las siguientes cuatro condiciones:
  - 1. **Exclusión mutua**: al menos un recurso debe mantenerse en un modo no compartido, es decir, solo un hilo puede usarlo a la vez. Si otro hilo solicita ese recurso, el hilo solicitante tendrá que esperar hasta que el recurso sea liberado.
  - 2. **Retención y espera**: un hilo debe estar reteniendo al menos un recurso y esperando para adquirir otros recursos adicionales que actualmente están retenidos por otros hilos.
  - 3. **Sin desalojo:** los recursos no pueden ser desalojados; es decir, un recurso solo puede ser liberado voluntariamente por el hilo que lo retiene, después de que dicho hilo haya completado su tarea.
  - 4. **Espera circular**: Debe existir un conjunto $\{T_0, T_1, ..., T_n\}$ de hilos en espera, tal que $T_0$ esté esperando un recurso retenido por $T_1$, $T_1$ esté esperando a un recurso retenido por $T2, ..., T_{n-1}$ está esperando un recurso retenido por $T_n$, y $T_n$ está esperando un recurso retenido por $T_0$.



# Caracterización de los *deadlocks*

## Grafo de asignación de recursos

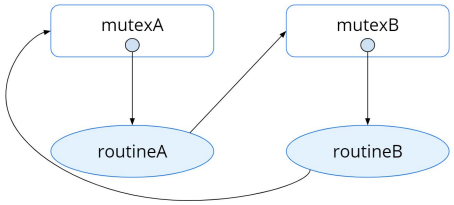
- Los *deadlocks* se pueden representar con un grafo de asignación de recursos.
- Consta de un conjunto de vértices **V** y un conjunto de aristas **E**.
- El conjunto de vértices **V** se divide en dos tipos diferentes de nodos:
  - $T = \{T_1, T_2, ..., T_n\}$: todos los hilos activos en el sistema, y
  - $R = \{R_1, R_2, ..., R_m\}$: todos los tipos de recursos en el sistema.
- **Arista de solicitud**: arista dirigida desde el hilo $T_i$ al tipo de recurso $R_j$ $(T_i \rightarrow R_j)$.
  - significa que el hilo $T_i$ ha solicitado una instancia de tipo de recurso $R_j$ y actualmente espera por el recurso.
- **Arista de asignación**: arista dirigida del tipo de recurso $R_j$ al hilo $T_i$ $(R_j \rightarrow T_i)$.
  - Significa que se ha asignado una instancia del tipo de recurso $R_j$ al hilo $T_i$.



# Caracterización de los *deadlocks*

## Grafo de asignación de recursos

- Los hilos $T_i$ se representan como círculos.
- Los recursos $R_j$ de cada tipo como rectángulos.
- Dado que el tipo de recurso $R_j$ puede tener más de una instancia, éstas se representan como un punto dentro del rectángulo.



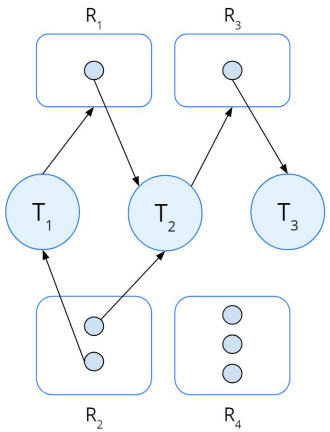

- Cuando el hilo $T_i$ solicita una instancia del tipo de recurso $R_j$, se inserta una arista de solicitud en el grafo de asignación de recursos.
- Cuando esta solicitud se puede cumplir, la arista de solicitud se transforma instantáneamente en una arista de asignación.
- Cuando el hilo ya no necesita acceso al recurso, lo libera. Como resultado, la arista de asignación se elimina.



# Caracterización de los *deadlocks*

## Grafo de asignación de recursos

- Los conjuntos **T**, **R**, y **E**:
  - $T=\{T_1, T_2, T_3\}$, $R=\{R_1, R_2, R_3, R_4\}$
  - $E=\{T_1 \rightarrow R_1, T_2 \rightarrow R_3, R_1 \rightarrow T_2, R_2 \rightarrow T_2, R_2 \rightarrow T_1, R_3 \rightarrow T_3\}$
- Instancias de recursos:
  - Una instancia del tipo de recurso $R_1$.
  - Dos instancias de tipo de recurso $R_2$.
  - Una instancia del tipo de recurso $R_3$.
  - Tres instancias de tipo de recurso $R_4$.
- Estados de los hilos:
  - El hilo $T_1$ contiene una instancia del tipo de recurso $R_2$ y espera una instancia del tipo de recurso $R_1$.
  - El hilo $T_2$ contiene una instancia de $R_1$ y una instancia de $R_2$ y está esperando una instancia de $R_3$.
  - El hilo $T_3$ contiene una instancia de $R_3$.



# Caracterización de los *deadlocks*

Grafo de asignación de recursos

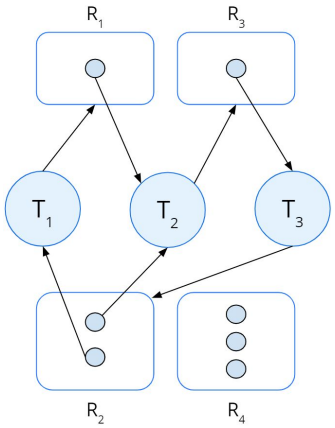
- A partir del grafo de asignación de recursos se puede demostrar que, si el grafo no contiene ciclos, entonces ningún hilo en el sistema está en *deadlock*.
- Si el grafo contiene un ciclo, puede existir un *deadlock*.
- Si cada tipo de recurso tiene exactamente una instancia, entonces un ciclo implica que se ha producido un *deadlock*.
  - Cada hilo involucrado en el ciclo está en *deadlock*. En este caso, un ciclo en el grafo es una condición necesaria y suficiente para la existencia de un *deadlock*.
- Si cada tipo de recurso tiene varias instancias, entonces un ciclo no implica necesariamente que se haya producido un *deadlock*.
  - En este caso, un ciclo en el grafo es una condición necesaria pero no suficiente para la existencia de un *deadlock*.



# Caracterización de los *deadlocks*

Grafo de asignación de recursos

- Para el grafo anterior supongamos que el hilo $T_3$ solicita una instancia del tipo de recurso $R_2$. Como no hay instancia de recurso disponible, agregamos una solicitud.
- Existirán dos ciclos mínimos en el sistema:
  - $T_1 \rightarrow R_1 \rightarrow T_2 \rightarrow R_3 \rightarrow T_3 \rightarrow R_2 \rightarrow T_1$
  - $T_2 \rightarrow R_3 \rightarrow T_3 \rightarrow R_2 \rightarrow T_2$
- Los hilos $T_1$, $T_2$ y $T_3$ están bloqueados.
- El hilo $T_2$ está esperando al recurso $R_3$, que está retenido por el hilo $T_3$.
- El hilo $T_3$ está esperando al hilo $T_1$ o al hilo $T_2$ que libere el recurso $R_2$.
- Además, el hilo $T_1$ está esperando que el hilo $T_2$ libere el recurso $R_1$.

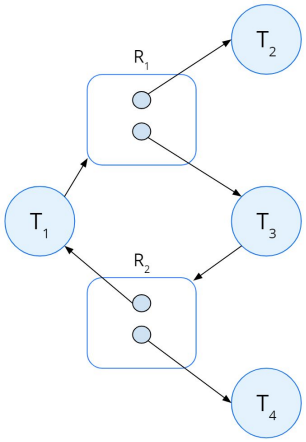





# Caracterización de los *deadlocks*

Grafo de asignación de recursos

- En el siguiente grafo de asignación de recursos, también tenemos un ciclo:
  - $T_1 \rightarrow R_1 \rightarrow T_3 \rightarrow R_2 \rightarrow T_1$
- Sin embargo, no hay *deadlock*; el hilo $T_4$ puede liberar su instancia de tipo de recurso $R_2$ y ese recurso puede asignarse a $T_3$, rompiendo el ciclo.
- Por lo tanto, si un grafo de asignación de recursos no tiene un ciclo, entonces el sistema no está en *deadlock*.
- Si hay un ciclo, entonces el sistema puede o no estar en *deadlock*.






# Métodos para manejar *deadlocks*

- Se pueden tratar de tres formas:
  - Podemos ignorar el problema por completo y pretender que nunca se producen *deadlocks* en el sistema.
  - Podemos usar un protocolo para prevenir o evitar bloqueos, asegurando que el sistema nunca ingrese en un estado de *deadlock*.
  - Podemos permitir que el sistema entre en *deadlock*, detectarlo y recuperarse.
- La primera solución es la utilizada por la mayoría de los sistemas operativos, incluidos **Linux** y **Windows**.
- Depende de los desarrolladores del *kernel* y de aplicaciones escribir programas que manejen *deadlocks*.
- Algunos sistemas, como las bases de datos, adoptan la tercera solución, permitiendo que se produzcan *deadlocks* y luego gestionan la recuperación.

# Prevención de *deadlocks*

- Para que se produzca un *deadlock*, se deben cumplir cada una de las cuatro condiciones necesarias.
- Al asegurarnos de que al menos una de estas condiciones no se cumpla, podemos evitar que se produzca un *deadlock*.
- Desarrollamos este enfoque examinando cada una de las cuatro condiciones necesarias por separado.



# Prevención de *deadlocks*

## Exclusión mutua

- La condición de exclusión mutua debe mantenerse.
  - Es decir, al menos un recurso debe ser no compartible.
- Los recursos compartibles no requieren acceso mutuamente exclusivo y, por lo tanto, no pueden participar en un *deadlock*.
  - Los archivos de solo lectura son un buen ejemplo de un recurso compartible. Si varios hilos intentan abrir un archivo de solo lectura al mismo tiempo, se les puede otorgar acceso simultáneo al archivo.
  - Un hilo nunca necesita esperar un recurso compartible.
- En general, no se puede evitar los *deadlocks* al negar la condición de exclusión mutua, porque algunos recursos son intrínsecamente no compartibles.
  - Por ejemplo, un *lock mutex* no puede ser compartido simultáneamente por varios hilos.



# Prevención de *deadlocks*

## Retención y espera

- Se debe garantizar que, cada vez que un hilo solicite un recurso, no retenga otro.
- **Protocolo**: requerir que cada hilo solicite y se le asignen todos sus recursos antes de que comience la ejecución.
  - Es impráctico para la mayoría de las aplicaciones debido a la naturaleza dinámica de la solicitud de recursos.
- **Protocolo alternativo**: que un hilo solicite recursos solo cuando no tiene ninguno.
  - Un hilo puede solicitar algunos recursos y usarlos y antes de que pueda solicitar recursos adicionales, debe liberar todos los recursos que tiene asignados actualmente.
- **Desventajas principales**.
  - Baja utilización de recursos, ya que los recursos pueden asignarse pero no utilizarse durante un largo período (*mutex lock*).
  - ***Starvation***: un hilo que necesita varios recursos puede tener que esperar indefinidamente, porque al menos uno de los recursos que necesita siempre se asigna a otro hilo.



# Prevención de *deadlocks*

## Sin desalojo

- Si un hilo retiene algunos recursos y solicita otro recurso que no se le puede asignar de inmediato (es decir, tiene que esperar), entonces todos los recursos que el hilo está reteniendo actualmente se desalojan.
  - Los recursos desalojados se agregan a la lista de recursos que el hilo está esperando.
  - El hilo se reiniciará solo cuando pueda recuperar sus recursos antiguos, así como los nuevos que está solicitando.
- Alternativamente, si un hilo solicita recursos, primero verificamos si están disponibles:
  - Si lo están, los asignamos.
  - Si no lo están, verificamos si están asignados a algún otro hilo que esté esperando recursos. Si es así, se apropian los recursos deseados al hilo de espera y se los asignamos al hilo solicitante.
  - Si no, debe esperar y algunos de sus recursos pueden ser apropiados, pero solo si otro hilo los solicita. Se reinicia cuando puede recuperar los recursos.

# Prevención de *deadlocks*

Sin desalojo

- Este protocolo a menudo se aplica a recursos cuyo estado se puede guardar y restaurar fácilmente más tarde:
  - Registros de *CPU* o transacciones de bases de datos.
- En general, no se puede aplicar a recursos tales como *mutex locks* y semáforos, precisamente el tipo de recursos donde los *deadlocks* ocurren con mayor frecuencia.



# Prevención de *deadlocks*

Espera circular

- Una forma de garantizar que esta condición nunca se cumpla es imponer un orden en todos los tipos de recursos y exigir que cada hilo solicite recursos en orden creciente.
- Sea $R = \{R_1, R_2, ..., R_m\}$ el conjunto de tipos de recursos.
  - Asignamos a cada tipo de recurso un número entero único, que nos permite comparar dos recursos y determinar si uno precede a otro en nuestro pedido.
- Se define una función uno a uno $F: R \rightarrow N$, donde $N$ es el conjunto de números naturales.
- Por ejemplo, el orden de bloqueo en el programa **Pthread** anterior podría ser:
  - `F(mutexA) = 1`
  - `F(mutexB) = 5`



# Prevención de *deadlocks*

Espera circular

- **Protocolo para evitar *deadlocks*:**
  - Cada hilo puede solicitar recursos solo en un orden creciente de enumeración.
  - Un hilo puede solicitar inicialmente una instancia de un recurso, por ejemplo, $R_i$. Después de eso, el hilo puede solicitar una instancia del recurso $R_j$ si y solo si $F(R_j) > F(R_i)$.
  - Por ej., usando la función definida anteriormente, un hilo que quiera usar tanto el `mutexA` como el `mutexB` al mismo tiempo primero debe solicitar el `mutexA` y luego el `mutexB`.
- Alternativamente, podemos exigir que un hilo que solicite una instancia del recurso $R_j$ debe haber liberado cualquier recurso $R_i$ tal que $F(R_i) \geq F(R_j)$.
- Si se usan estos dos protocolos, entonces la condición de espera circular no puede mantenerse.
- Dar un orden o una jerarquía no evita en sí mismo un *deadlock*. Depende de los desarrolladores de aplicaciones escribir programas que sigan el orden.



# Evasión de *deadlocks*

- Un método alternativo para evitar *deadlocks* es requerir información adicional sobre cómo se van a solicitar los recursos.
- Por ej., en un sistema con recursos $R_1$ y $R_2$, el sistema puede necesitar saber que el hilo $P$ solicitará primero $R_1$ y luego $R_2$ antes de liberar ambos recursos, mientras que el hilo $Q$ solicitará $R_2$ y luego $R_1$.
- Con este conocimiento de la secuencia completa de solicitudes y liberaciones de cada hilo, el sistema puede decidir para cada solicitud si el hilo debe esperar o no para evitar un posible *deadlock* futuro.
- Cada solicitud requiere que, al tomar esta decisión, el sistema considere los recursos disponibles actualmente, los recursos asignados a cada hilo y las futuras solicitudes y liberaciones de cada hilo.

# Evasión de *deadlocks*

- El modelo más simple y útil requiere que cada hilo declare el número máximo de recursos de cada tipo que pueda necesitar.
- Dada esta información *a priori*, es posible construir un algoritmo que garantice que el sistema nunca entrará en un estado de bloqueo.
- Un algoritmo de evasión de *deadlock* examina dinámicamente el estado de asignación de recursos para garantizar que nunca pueda existir una condición de espera circular.
- El estado de asignación de recursos se define por el número de recursos disponibles y asignados y las demandas máximas de los hilos.



# Evasión de *deadlocks*

## Estado seguro

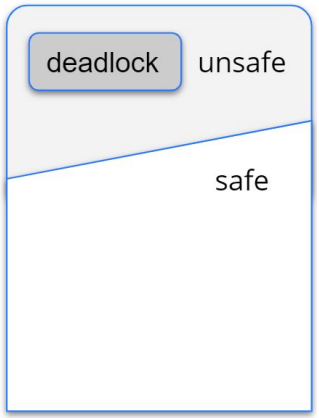
- Un estado es seguro si el sistema puede asignar recursos a cada hilo (hasta su máximo) en algún orden y aún así evitar un *deadlock*.
- Un sistema está en un estado seguro solo si existe una secuencia segura.
  - Una secuencia de hilos $<T_1, T_2, ..., T_n>$ es una secuencia segura para el estado de asignación actual si, para cada $T_i$, las solicitudes de recursos que $T_i$ aún puede hacer pueden ser satisfechas por los recursos disponibles actualmente más los recursos retenidos por todos los $T_j$, con $j < i$.
- En esta situación, si los recursos que necesita no están disponibles de inmediato, entonces puede esperar hasta que todos hayan terminado.
- Cuando hayan terminado, $T_i$ puede obtener todos sus recursos necesarios, completar su tarea designada, devolver los recursos asignados y finalizar.
- Cuando $T_i$ termina, $T_{i+1}$ puede obtener los recursos necesarios, y así sucesivamente.
- Si no existe tal secuencia, se dice que el estado del sistema no es seguro.



# Evasión de *deadlocks*

## Estado seguro

- Un estado seguro no es un estado en *deadlock*.
- Por el contrario, un estado en *deadlock* es un estado inseguro.
- Sin embargo, no todos los estados inseguros son *deadlocks*. Un estado inseguro puede o no conducir a un *deadlock*.
- Mientras el estado sea seguro, el SO puede evitar estados inseguros (y *deadlocks*).
- En un estado inseguro, el sistema operativo no puede evitar que los hilos soliciten recursos de tal manera que se produzca un *deadlock*.






# Evasión de *deadlocks*

## Estado seguro

- Un algoritmo de evasión garantiza que el sistema permanezca siempre en un estado seguro.
  - Inicialmente, el sistema está en un estado seguro.
  - Cada vez que un hilo solicita un recurso que está disponible actualmente, el sistema debe decidir si el recurso se puede asignar de inmediato o si el hilo debe esperar.
  - La solicitud se otorga solo si la asignación deja el sistema en un estado seguro.
  - En este esquema, si un hilo solicita un recurso que está disponible, puede ocurrir que tenga que esperar.
    - Por lo tanto, la utilización de recursos puede ser menor de lo que sería sin este algoritmo de evasión.

# Evasión de *deadlocks*

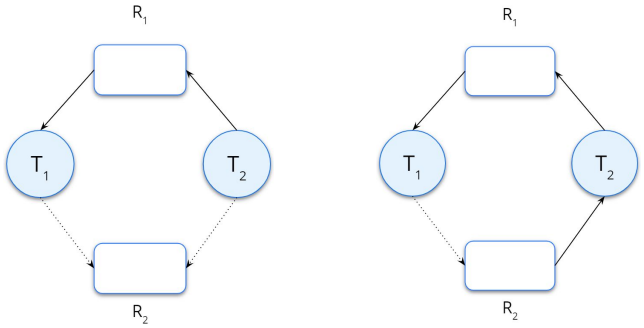
## Algoritmo de grafo de asignación de recursos

- Si tenemos una sola instancia de cada tipo de recurso, se puede usar una variante del grafo de asignación de recursos.
- **Arista de reclamo**: $P_i \rightarrow R_j$ indica que el proceso $P_i$ podría solicitar el recurso $R_j$ (se representa por una línea discontinua).
- La arista de reclamo se convierte en arista de solicitud cuando un proceso solicita un recurso, y la de solicitud se convierte en una de asignación cuando el recurso se asigna al proceso.
- Cuando un recurso es liberado por un proceso, la arista de asignación se reconvierte en una arista de reclamo.
- Los recursos deben ser reclamados *a priori* en el sistema.
- Supongamos que el proceso $P_i$ solicita un recurso $R_j$:
  - La solicitud puede otorgarse solo si la conversión de la arista de solicitud en una arista de asignación no da como resultado un ciclo en el grafo de asignación de recursos.
- Si no existe un ciclo, la asignación del recurso dejará el sistema en un estado seguro. Si se encuentra un ciclo, la asignación pondrá el sistema en un estado inseguro.



# Evasión de *deadlocks*

## Algoritmo de grafo de asignación de recursos




- Supongamos que $T_2$ solicita $R_2$. Aunque $R_2$ está libre, no podemos asignarlo a $T_2$, ya que esto creará un ciclo en el grafo.



# Evasión de *deadlocks*

## Algoritmo del banquero

- El algoritmo del banquero es aplicable a sistemas de asignación de recursos con múltiples instancias de cada tipo de recurso.
- Cuando ingresa un nuevo hilo al sistema, debe declarar el número máximo de instancias de cada tipo de recurso que pueda necesitar, el cual no puede exceder el número total de recursos en el sistema.
- Cuando se solicita un conjunto de recursos, el sistema debe determinar si la asignación de estos recursos lo dejará en un estado seguro.
  - Si lo hace, los recursos se asignan.
  - De lo contrario, el hilo debe esperar hasta que algún otro hilo libere suficientes recursos.
- Se deben mantener varias estructuras de datos para implementar el algoritmo del banquero.
- Estas estructuras mantienen el estado de asignación de recursos del sistema, donde:
  - `n` es el número de hilos en el sistema y
  - `m` es el número de tipos de recursos.



# Evasión de *deadlocks*

## Algoritmo del banquero

`n`: nro. de hilos.
`m`: nro. recursos.

- `Disponible` (*`Available`*):
  - Vector de longitud `m` que indica el número de recursos disponibles de cada tipo.
  - Si `Disponible[j]` = `k`, entonces se dispone de `k` instancias del tipo de recurso $R_j$.
- `Máximo` (*`Max`*):
  - Matriz `n × m` que define la demanda máxima de cada hilo.
  - Si `Máximo[i][j]` = `k`, entonces el hilo $T_i$ puede solicitar como máximo `k` instancias del tipo de recurso $R_j$.
- `Asignación` (*`Allocation`*):
  - Matriz `n × m` que define el número de recursos de cada tipo actualmente asignado a cada hilo.
  - Si `Asignación[i][j]` = `k`, entonces el hilo $T_i$ tiene asignado actualmente `k` instancias del tipo de recurso $R_j$.

# Evasión de *deadlocks*

## Algoritmo del banquero

- `Necesidad` (*`Need`*):
  - Matriz `n × m` que indica los recursos aún necesarios de cada hilo.
  - Si `Necesidad[i][j]` = `k`, entonces el hilo $T_i$ puede necesitar `k` instancias más del tipo de recurso $R_j$ para completar su tarea.
    - `Necesidad[i][j] = Máximo[i][j] - Asignación[i][j]`.

`n`: nro. de hilos.
`m`: nro. recursos.

- Sea `X` e `Y` dos vectores de longitud `n`. Diremos que `X ≤ Y` si y solo si `X[i] ≤ Y[i]` para todo `i = 1, 2, ..., n`.
  - Ej.: Si `X = (1,7,3,2)` e `Y = (0,3,2,1)`, entonces `Y ≤ X`.
- Cada fila de las matrices se trata como vectores de la siguiente manera:
  - $\texttt{Asignación}_i$: recursos actualmente asignados al hilo $T_i$.
  - $\texttt{Necesidad}_i$: recursos adicionales que el hilo $T_i$ aún puede solicitar para completar su tarea.



# Evasión de *deadlocks*

## Algoritmo del banquero: Algoritmo de seguridad

1. Sean `Trabajo` (*`Work`*) y `Finalizar` (*`Finish`*) vectores de longitud `m` y `n`, inicializar:
   - `Trabajo = Disponible` y `Finalizar[i] = false` para `i = 0, 1, ..., n - 1`.
2. Encontrar un índice `i` tal que:
   - a. `Finalizar[i] == false`
   - b. $\texttt{Necesidad}_i \leq \texttt{Trabajo}$
   - Si tal `i` no existe, ir al paso **4**.
3. $\texttt{Trabajo} = \texttt{Trabajo} + \texttt{Asignación}_i$
   - `Finalizar[i] = true`
   - Ir al paso **2**.
4. Si `Finalizar[i] == true` para todo `i`, entonces el sistema se encuentra en estado seguro.

`n`: nro. de hilos.
`m`: nro. recursos.

- Este alg. puede requerir un orden de $m \times n^2$ operaciones para determinar si un estado es seguro.



# Evasión de *deadlocks*

## Algoritmo del banquero: Algoritmo de solicitud de recursos

- Determina si las solicitudes pueden otorgarse de manera segura.
  - $\texttt{Solicitud}_i$ (*`Request`*): vector de solicitud para el hilo $T_i$.
  - Si $\texttt{Solicitud}_i\texttt{[j] == k}$, entonces el hilo $T_i$ necesita `k` instancias del tipo de recurso $R_j$.
- Cuando el hilo $T_i$ realiza una solicitud de recursos, se toman las siguientes acciones:
  - 1. Si $\texttt{Solicitud}_i \leq \texttt{Necesidad}_i$, ir al paso **2**. Sino → `error`: ha excedido su reclamo máx.
  - 2. Si $\texttt{Solicitud}_i \leq \texttt{Disponible}$, ir al paso **3**. Sino → `esperar`: recursos no disponibles.
  - 3. Probar de asignar los recursos solicitados al hilo $T_i$ modificando el estado:
    - $\texttt{Disponible} = \texttt{Disponible} - \texttt{Solicitud}_i$
    - $\texttt{Asignación}_i = \texttt{Asignación}_i + \texttt{Solicitud}_i$
    - $\texttt{Necesidad}_i = \texttt{Necesidad}_i - \texttt{Solicitud}_i$



# Evasión de *deadlocks*

## Algoritmo del banquero: Algoritmo de solicitud de recursos

- Si el estado de asignación de recursos resultante es seguro, la transacción se completa y se le asignan los recursos solicitados al hilo $T_i$.
- Sin embargo, si el nuevo estado no es seguro, $T_i$ debe esperar por la $\texttt{Solicitud}_i$ y se debe restaurar el estado anterior de asignación de recursos.

# Evasión de *deadlocks*

## Algoritmo del banquero: Ejemplo

- Para ilustrar el uso del algoritmo del banquero, considerar un sistema con cinco hilos del $T_0$ al $T_4$ y tres tipos de recursos A, B y C.

| Hilos | Asignación | | | Máximo | | | Disponible | | |
|---|---|---|---|---|---|---|---|---|---|
| | *A* | *B* | *C* | *A* | *B* | *C* | *A* | *B* | *C* |
| $T_0$ | 0 | 1 | 0 | 7 | 5 | 3 | 3 | 3 | 2 |
| $T_1$ | 2 | 0 | 0 | 3 | 2 | 2 | | | |
| $T_2$ | 3 | 0 | 2 | 9 | 0 | 2 | | | |
| $T_3$ | 2 | 1 | 1 | 2 | 2 | 2 | | | |
| $T_4$ | 0 | 0 | 2 | 4 | 3 | 3 | | | |



# Evasión de *deadlocks*

## Algoritmo del banquero: Ejemplo

- `Necesidad[i][j] = Máximo[i][j] - Asignación[i][j].`
- `Total recursos[j] = Disponible[j] + Asignación[0...i-1][j].`

| Necesidad | | |
|---|---|---|
| *A* | *B* | *C* |
| 7 | 4 | 3 |
| 1 | 2 | 2 |
| 6 | 0 | 0 |
| 0 | 1 | 1 |
| 4 | 3 | 1 |

| Total recursos | | |
|---|---|---|
| *A* | *B* | *C* |
| 10 | 5 | 7 |



# Evasión de *deadlocks*

## Algoritmo del banquero: Ejemplo

- Algoritmo de seguridad.
  - 1. Inicializar los vectores `Trabajo` y `Finalizar`
    - `Trabajo = Disponible = (3, 3, 2)`
    - `Finalizar = (F, F, F, F, F)`
  - 2. Buscar un índice `i : Finalizar[i] == F y` $Necesidad_i \leq Trabajo_i$
  - 3.

| i | Trabajo = Trabajo + $Asignación_i$ | Finalizar |
|---|---|---|
| 1 | (3, 3, 2) + (2, 0, 0) = (5, 3, 2) | (F, T, F, F ,F) |
| 3 | (5, 3, 2) + (2, 1, 1) = (7, 4, 3) | (F, T, F, T, F) |
| 4 | (7, 4, 3) + (0, 0, 2) = (7, 4, 5) | (F, T, F, T, T) |
| 2 | (7, 4, 5) + (3, 0, 2) = (10, 4, 7) | (F, T, T, T, T) |
| 0 | (10, 4, 7) + (0, 1, 0) = (10, 5, 7) | (T, T, T, T, T) |

  - 4. Dado que `Finalizar[i] = T ∀ i ⇒` El sistema se encuentra en un estado seguro, la secuencia segura es $<T_1, T_3, T_4, T_2, T_0>$.



# Evasión de *deadlocks*

## Algoritmo del banquero: Ejemplo

- Supongamos que el hilo $T_1$ solicita `(1,0,2)`. ¿puede concederse inmediatamente dicha solicitud?
  - Algoritmo de solicitud de recursos.
  - 1. $Solicitud_1 \leq Necesidad_1$? → (1, 0, 2) ≤ (1, 2, 2) □
  - 2. $Solicitud_1 \leq Disponible$? → (1, 0, 2) ≤ (3, 3, 2) □
  - 3. Intentar modificar el estado del sistema:
    - $Disponible = Disponible - Solicitud_1$ = (3, 3, 2) - (1, 0, 2) = (2, 3, 0)
    - $Necesidad_1 = Necesidad_1 - Solicitud_1$ = (1, 2, 2) - (1, 0, 2) = (0, 2, 0)
    - $Asignación_1 = Asignación_1 + Solicitud_1$ = (2, 0, 0) + (1, 0, 2) = (3, 0, 2)
  - 4. Aplicar algoritmo de seguridad (hacer de ejercicio):
    - El resultado indica que existe la secuencia segura $<T_1, T_3, T_4, T_0, T_2>$.

# Evasión de *deadlocks*

## Algoritmo del banquero: Conclusiones de distintos autores

- ***ANDREW S. TANENBAUM - HERBERT BOS:***
  - Desafortunadamente, pocos autores han tenido la audacia de señalar que, aunque en teoría el algoritmo es maravilloso, en la práctica es esencialmente inútil ya que los procesos raramente saben de antemano cuáles serán sus necesidades máximas de recursos.
  - Además, el número de procesos no es fijo, sino que varía dinámicamente con la entrada y salida de usuarios.
  - Por otra parte, los recursos que se creía que iban a estar disponibles pueden desaparecer súbitamente (las unidades de cinta o impresoras se pueden romper).
  - Por lo tanto, en la práctica pocos sistemas, si es que hay alguno, utilizan el algoritmo del banquero para evitar *deadlocks*.
- ***WILLIAM STALLINGS:***
  - No es realista ya que no se conocen de antemano las demandas máximas.
  - El número de procesos y de recursos puede cambiar con el tiempo (algo puede romperse).
  - La mayoría de los sistemas operativos ignoran el *deadlock*.
- ***ABRAHAM SILBERSCHATZ - PETER BAER GALVIN - GREG GAGNE:***
  - Este algoritmo puede requerir un orden $m \times n^2$ de operaciones para determinar si un estado es seguro.
- En resumen, podemos decir que es un acercamiento teórico para evitar los *deadlocks* en la planificación de recursos.



# Detección de *deadlocks*

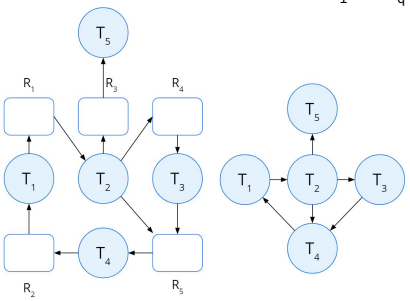
- Si el sistema no emplea un algoritmo de prevención o evasión de *deadlocks*, se puede proporcionar:
  - Un algoritmo que examine el estado del sistema para determinar si se ha producido un *deadlock*.
  - Un algoritmo para recuperarse del *deadlock*.
- Este esquema de detección y recuperación requiere una sobrecarga que incluye, no solo los costos en tiempo de ejecución de mantener la información necesaria y ejecutar el algoritmo de detección, sino también las pérdidas potenciales inherentes a la recuperación de un *deadlock*.



# Detección de *deadlocks*

## Una instancia de cada tipo de recurso

- Grafo de espera: se obtiene del grafo de asignación de recursos eliminando los nodos recurso y contrayendo las aristas apropiadas.
- Una arista de $T_i$ a $T_j$ en un grafo de espera implica que el hilo $T_i$ está esperando que el hilo $T_j$ libere un recurso que $T_i$ necesita.
- Existe una arista $T_i \rightarrow T_j$ en un grafo de espera si y solo si el correspondiente grafo de asignación de recursos contiene dos aristas $T_i \rightarrow R_q$ y $R_q \rightarrow T_j$ para algún recurso $R_q$.




- Existe un *deadlock* en el sistema si y solo si el grafo de espera contiene un ciclo.
- Para detectar *deadlock*, el sistema necesita mantener el grafo de espera e invocar periódicamente un alg. que busque un ciclo en el grafo.
- Un alg. para detectar un ciclo en un grafo requiere operaciones $O(n^2)$, donde $n$ es el número de vértices en el grafo.



# Detección de *deadlocks*

## Múltiples instancias de cada tipo de recurso

- El algoritmo emplea varias estructuras de datos variables en el tiempo, que son similares a las utilizadas en el algoritmo del banquero.

  $n$: nro. de hilos.
  $m$: nro. recursos.

  - `Disponible` (*`Available`*).
    - Vector de longitud $m$ que indica el número de recursos disponibles de cada tipo.
  - `Asignación` (*`Allocation`*).
    - Matriz $n \times m$ que define el número de recursos de cada tipo asignados a cada hilo.
  - `Solicitud` (*`Request`*).
    - Matriz $n \times m$ que indica la solicitud actual de cada hilo.
    - Si `Solicitud[i][j]` = $k$, entonces el hilo $T_i$ está solicitando $k$ instancias adicionales del tipo de recurso $R_j$.
- El algoritmo investiga cada posible secuencia de asignación que resta por completar para determinados hilos.

# Detección de *deadlocks*

Múltiples instancias de cada tipo de recurso

1. Sean `Trabajo` y `Finalizar` vectores de longitud `m` y `n` respectivamente.
   - Inicializar `Trabajo = Disponible`.
   - Para `i = 0, 1, ..., n−1`, si $\texttt{Asignación}_i \neq 0$, entonces `Finalizar[i] = false`. Sino `Finalizar[i] = true`.
2. Encontrar un índice `i` tal que:
   - `a. Finalizar[i] == false`
   - $\texttt{b. Solicitud}_i \leq \texttt{Trabajo}$
   - Si no existe tal `i`, ir al paso `4`.
3. $\texttt{Trabajo} = \texttt{Trabajo} + \texttt{Asignación}_i$
   - `Finalizar[i] = true`
   - Ir al paso `2`.
4. Si `Finalizar[i] == false` para algún `i, 0 ≤ i <n`, entonces el sistema se encuentra en *deadlock*. Además, si `Finalizar[i] == false`, entonces el hilo $T_i$ se encuentra en *deadlock*.

`n`: nro. de hilos.
`m`: nro. recursos.

- Este algoritmo requiere un orden de operaciones $m \times n^2$ para detectar si el sistema está en *deadlock*.



# Detección de *deadlocks*

Uso del algoritmo de detección

- ¿Cuándo debemos invocar el algoritmo de detección?
  - ¿Con qué frecuencia es probable que ocurra un *deadlock*?
  - ¿Cuántos hilos se verán afectados por un *deadlock* cuando éste ocurra?
- Si ocurren a menudo, entonces debe invocarse con frecuencia.
  - Los recursos asignados a los hilos estarán inactivos hasta que se pueda romper el *deadlock*. Además, el número de hilos en el *deadlock* puede crecer.
- Un caso extremo implica invocar el algoritmo de detección cada vez que no es posible otorgar de inmediato una solicitud de asignación.
  - Se puede identificar no solo el conjunto de hilos en *deadlock* sino también el hilo específico que lo "causó".
- Invocar el algoritmo para cada solicitud de recursos generará una sobrecarga considerable.
- Una alternativa menos costosa es invocar el algoritmo a intervalos definidos, por ejemplo, una vez por hora o cuando la utilización de la *CPU* cae por debajo del 40%.



# Recuperación de *deadlocks*

- Cuando se detecta un *deadlock*, hay varias alternativas:
  - Una posibilidad es informar al administrador del sistema que se ha producido un *deadlock* y dejar que lo maneje manualmente.
  - Otra posibilidad es dejar que el sistema se recupere del *deadlock* automáticamente.
  - Hay dos opciones para romper un *deadlock*.
    - Abortar uno o más hilos para romper la espera circular.
    - Apropiar algunos de los recursos de uno o más hilos bloqueados.



# Recuperación de *deadlocks*

Terminación de procesos e hilos

- Una vez abortado el proceso o hilo, el sistema reclama todos los recursos que tenían asignados.
  - Abortar todos los procesos en *deadlock*:
    - Es un método costoso.
    - Los procesos en *deadlock* pueden haber estado haciendo cálculos durante mucho tiempo, y los resultados de estos cálculos parciales deben descartarse y volver a calcularse.
  - Abortar un proceso a la vez hasta que se elimine el ciclo de deadlock:
    - Este método incurre en una sobrecarga considerable, ya que después de que se aborta cada proceso, se debe invocar un algoritmo de detección para determinar si algún proceso todavía está en *deadlock*.

# Recuperación de *deadlocks*

## Terminación de procesos e hilos

- Muchos factores pueden afectar a cuál de los procesos se elige:
  - ¿Cuál es la prioridad del proceso?
  - ¿Cuánto tiempo ha estado haciendo cálculos y cuánto tiempo más calculará antes de completar su tarea?
  - ¿Cuántos y qué tipos de recursos ha utilizado el proceso? (Por ejemplo, si los recursos son fáciles de apropiar).
  - ¿Cuántos recursos más necesita el proceso para completar su tarea?
  - ¿Cuántos procesos deberán ser terminados?



# Recuperación de *deadlocks*

## Apropiación de recursos

- Se apropian sucesivamente algunos recursos de los procesos y son entregados a otros procesos hasta que se interrumpa el ciclo de *deadlock*. Se deben abordar tres problemas:
  - **Selección de víctima**: ¿qué recursos y a qué procesos van a ser apropiados?
    - Se deben minimizar los costos: los factores de costo pueden ser la cantidad de recursos que tiene un proceso en *deadlock* y la cantidad de tiempo que el proceso ha consumido.
  - ***Rollback***: si se apropia un recurso a un proceso, ¿qué se debe hacer con ese proceso?
    - No puede continuar con su ejecución dado que le falta algún recurso necesario.
    - Se debe hacer *rollback* del proceso a un estado seguro y reiniciarlo desde ese estado, (aunque es difícil saber cuál es un estado seguro).
    - La solución más simple es un total *rollback*: abortar el proceso y luego reiniciarlo.
      - Este método requiere que el sistema mantenga más información sobre el estado de todos los procesos en ejecución.



# Recuperación de *deadlocks*

## Apropiación de recursos

- ***Starvation***: ¿cómo nos aseguramos de que no se produzca inanición? Es decir, ¿cómo podemos garantizar que los recursos no sean siempre apropiados al mismo proceso?
  - En un sistema donde la selección de víctimas se basa en factores de costo, puede ocurrir que el mismo proceso siempre resulte elegido.
  - El proceso nunca completa su tarea, una situación de inanición que se debe abordar.
  - Se debe asegurar que un proceso sea elegido como víctima un número finito de veces.
    - Una solución simple es incluir la cantidad de *rollbacks* en el factor de costo.