CAPITULO 17

ABRAZO MORTAL - DEADLOCK

17.1. – PROBLEMA [9][44]

Dos objetos (Procesos) desean hacer uso de un único recurso no compartible o un número finito de ellos, y cada uno de ellos posee la porción que el otro necesita. Un ejemplo clásico es el de dos procesos P(1) y P(2), cada uno de ellos tiene asignadas 2 unidades de cinta y cada uno de ellos necesita una tercera, pero en su sistema de ejecución existen sólo 4 unidades en total. Obviamente estamos en presencia de un deadlock, ya que P(1) espera recursos de P(2) y P(2) espera recursos de P(1).

Para hacer uso de recursos que puedan llevarnos al estado anterior, debería establecerse algún tipo de mecanismo o protocolo:

1. Comenzar a usar el recurso sólo en el caso de que nadie lo esté usando, pues si ya está siendo usado por otro ocurre el "DEADLOCK" (o cualquier otro desastre). Si no se fija si ya se está usando "puede siempre ocurrir" el "DEADLOCK".
2. Si dos procesos desean usarlo en el mismo instante, la solución de arriba no sirve porque o bien los dos comienzan a utilizarlo (deadlock) o ambos esperan que el otro comience, que es otra forma de "DEADLOCK" (starvation - inanición). Esto se puede arreglar por medio de darle prioridad a uno de ellos.
3. Si un grupo de procesos tiene mayor prioridad sobre otro, puede ocurrir que el segundo grupo nunca tome el recurso, luego, hay que inventar un mecanismo que varíe esa prioridad. Por ejemplo, pasado X tiempo, o cuando haya N procesos en espera, la prioridad se invierte.

17.2. - UTILIZACION DE RECURSOS [9][44]

Dentro de un sistema, con un número de procesos que compiten por recursos no compartibles, se debe tener en cuenta que los recursos son limitados: no se puede pedir más de los que existen y no se puede utilizar un recurso que ya esté siendo utilizado por otro proceso. Luego un proceso puede utilizar un recurso de la siguiente manera:

Pedirlo : (REQUEST) si no puede ser satisfecho esperar. Será incluido en una cola en espera de ese recurso (Tablas).
Usarlo : (USE) el proceso puede operar el recurso.
Liberarlo : (RELEASE) el proceso libera el recurso que fue pedido y asignado.

17.3. - DEFINICION DE DEADLOCK [9][44]

Un conjunto de procesos está en estado de "DEADLOCK" cuando cada proceso del conjunto está esperando por un evento que solo puede ser causado por otro proceso que está dentro de ese conjunto. En el ejemplo de las cintas, el evento que se espera es "liberación de la cinta" (igual tipo de recurso).

También es posible estar frente a casos de "DEADLOCK" de distintos tipos de recursos, por ejemplo si un proceso P(1) tiene asignada una lectora de tarjetas y espera por la impresora, y un proceso P(2) tiene asignada la impresora y espera por la lectora de tarjetas.

17.4. - CONDICIONES NECESARIAS PARA EL "DEADLOCK" [9][44]

Se llega al "DEADLOCK" sii las siguientes 4 condiciones se cumplen **simultáneamente**:

17.4.1 - Exclusión Mutua [9][44]

Al menos un recurso debe ser usado en forma exclusiva (no se puede compartir). Si un proceso lo desea, pero ya lo tiene otro, deberá esperar hasta que sea liberado.

17.4.2 - Espera y Retenido (Hold & Wait) [9][44]

Debe existir un proceso que tiene retenido un recurso y está esperando por otro que a su vez está siendo ocupado por otro proceso.

17.4.3 - Sin Desalojo [9][44]

Los recursos no pueden ser desalojados, es decir que un recurso es liberado en forma voluntaria por un proceso y luego que haya finalizado su tarea.

17.4.4 - Espera circular [9][44]

Debe existir un conjunto de procesos {P(0), P(1),...,P(n)} tal que P(0) espera un recurso ocupado por P(1), P(1) espera un recurso ocupado por P(2), y P(n) espera un recurso ocupado por P(0).

De alguna manera esta última condición implica la de hold & wait.

17.5. - GRAFO DE ASIGNACIÓN DE RECURSOS [9][44]

Estos grafos se utilizan para describir los "DEADLOCKS". Sea G = (V,E), donde V es el conjunto de vértices y E es el conjunto de flechas (arcos orientados).

A su vez V está compuesto por 2 tipos:

P = {p(1),p(2),...,p(n)} conjunto de Procesos

R = {r(1),r(2),.....,r(m)} conjunto de Recursos

Los elementos de E son:

(p(i),r(j)) ======> p(i) está esperando una instancia del recurso r(j).

(r(j),p(i)) ======> r(j) (una instancia de) está asignada al proceso p(i).

Si tenemos (p(1),r(3)) y r(3) es asignado a p(1), instantáneamente esto se transforma en (r(3),p(1)) y cuando el recurso es liberado se elimina la flecha.

Ejemplo :

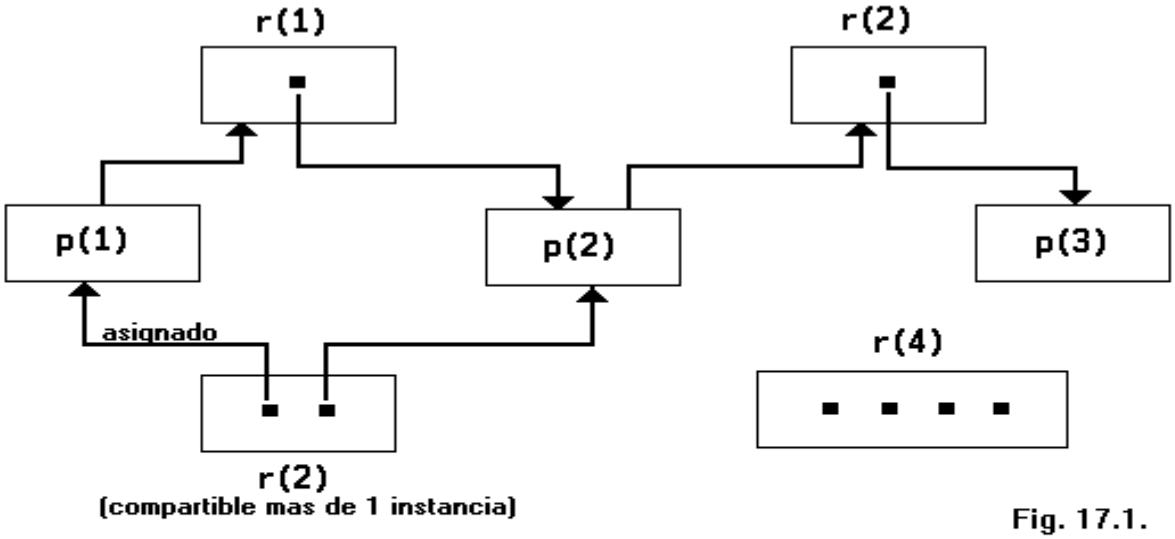

Fig. 17.1.

G
V
P = { p(1), p(2), p(3) }
R = { r(1), r(2), r(3), r(4) }
E = { (p(1),r(1)), (p(2),r(3)), (r(1),p(2)),
(r(2),p(2)), (r(2),p(1)), (r(3),p(3)) }

A partir de aquí es fácil ver que si los grafos no forman un ciclo cerrado no hay procesos en **deadlock**. Si existe ciclo cerrado entonces puede existir un **deadlock**.

Si cada tipo de recurso tiene exactamente una sola instancia y tenemos ciclo cerrado entonces estamos en presencia de un deadlock. En este caso un ciclo cerrado es condición necesaria y suficiente para que exista deadlock.

En caso de tipos de recursos con más de una instancia, la existencia de un ciclo cerrado es condición necesaria, pero no suficiente.

EJEMPLO (Fig. 17.2)

Si a nuestro ejemplo agregamos (p(3),r(2)), tendríamos :
p(1) ---> r(1) ---> p(2) ---> r(3) ---> p(3) - --> r(2) --- > p(1)
p(2) ---> r(3) ---> p(3) ---> r(2) ---> p(2)
en el cual se ve claramente que estos dos ciclos cerrados están señalando 2 deadlocks.
Ahora consideremos el ejemplo de la Fig. 17.2 donde tenemos el ciclo cerrado
p(1) --> r(1) --> p(3) --> r(2) -- > p(1)
pero se ve que si p(4) o p(2) liberan sus recursos se rompe el ciclo cerrado, luego no estamos en presencia de un **deadlock.**

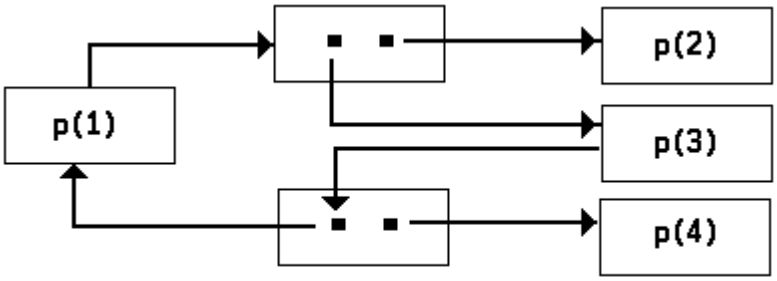

Fig. 17.2.

Al liberar p(2) o p(4) queda liberada una instancia que permite la asignación a alguno de los procesos en espera.

CONCLUSION: *Cuando no existe un ciclo no hay deadlock. Ahora si existe un ciclo puede o no haber deadlock.*

17.6. - MANEJO DE DEADLOCK [9][44]

Existen cuatro maneras de manejar los **Deadlocks** :
- Ignorar su ocurrencia
- No permitir que ocurran (Evitarlo).
- Prevenir que ocurran (Prevención)
- Permitirlo y luego recuperar (es caro y dificultoso).

17.6.1 - PREVENCION. [9][44]

Antes vimos cuáles eran las **condiciones necesarias** para que ocurra un "Deadlock", luego si aseguramos que una de ellas no ocurre, estamos seguros de que no tendremos un deadlock.

17.6.1.1 - EXCLUSION MUTUA [9][44]

Para evitarlo se deberían utilizar siempre recursos que pudiesen ser compartidos, ya que los procesos no deben esperar la exclusividad para usar recursos compartibles, por ejemplo los archivos en modo lectura. Pero esto no es real, pues siempre se necesitarán por ejemplo archivos en modo escritura, cintas, etc. (algunos recursos son intrínsecamente no compartibles)

17.6.1.2 - ESPERA Y RETENIDO (Hold & Wait) [9][44]

Existen dos formas de evitarlo :
1. Un proceso debe tener asignados todos sus recursos antes de comenzar su ejecución.
Si necesita 3 unidades de cinta, no se le asignan las 3 hasta que estén todas disponibles, y el proceso no comienza hasta tanto no se tengan las 3, y las tendrá todo el tiempo en que ese proceso ejecute, aún si las usa solo al final.
2. Si un proceso tiene asignado un recurso no puede asignársele otro hasta que haya liberado al anterior.
Si un proceso que imprime un listado además utiliza un periférico de entrada dedicado y un archivo, cuando termina la utilización del periférico dedicado y el archivo se le desasignan ambos y se le asigna nuevamente el archivo y la impresora hasta su finalización.

PROBLEMAS : Un recurso que es asignado al comienzo del proceso pero solo es utilizado al final de ese proceso es inutilizado mucho tiempo (bajo uso de recurso). O un proceso puede esperar indefinidamente un recurso muy popular. (*Starvation- Inanición*).

17.6.1.3 - SIN DESALOJO [9][44]

Una forma de evitarlo es si un proceso tiene recursos y pide otro no disponible, pierde todos los recursos y solo recomenzará cuando los tenga todos a su disposición.

Otra forma sería si un proceso pide algún recurso, si este está disponible se lo asigna, si no, se controla si ese recurso está asignado a un proceso en espera de algún otro recurso y si es así se lo quita y asigna al solicitante; si no está disponible de ninguna manera, espera.

Esto es lo que se usa en recursos cuyos estados son fácilmente salvables, como memoria y procesador (donde es necesario previamente salvar PSW, registros, etc.).

Piense Ud. cuán engorroso sería en el caso de una impresora (las líneas saldrían mezcladas).

17.6.1.4 - ESPERA CIRCULAR [9][44]

Para asegurar que esta condición no se cumpla consideremos lo siguiente:

A cada "tipo de recurso" le asignamos un número Natural único

R = {r(1),r(2),....,r(n)} F:R --> N

F(lectora)=1 , F(disco)=5 , F(cinta)=7 , F(impresora)=12

y consideremos el siguiente protocolo de asignación:

Un proceso solo puede pedir recursos (tipo) en orden creciente, si necesita uno de orden más bajo, debe liberar todos los recursos de orden más alto. Es decir:

Un proceso puede pedir un r(j) si y solo si F(r(j)) > F(r(i)) para todo i de los recursos que ya tiene. Si no cumple esta condición, debe liberar todos los r(i) que cumplen $F(r(i)) \geq F(r(j))$.

Dadas estas condiciones se puede demostrar que no tendremos nunca una espera circular.

DEMOSTRACION

Supongamos que ésta existe en el conjunto (p(0),p(1), ...,p(n)) o sea que p(i) está esperando r(i), quien está detenido por p(i+1) y por supuesto p(n) está esperando por un recurso retenido por p(0) (módulo n+1).

Como p(i+1) está reteniendo r(i) y esperando r(i+1) entonces F(r(i)) < F(r(i+1)), esto implica :

F(r(0)) < F(r(1)) < ..... < F(r(n)) < F(r(0)) ==> por transitividad F(r(0)) < F(r(0)) que es absurdo, luego es absurda la existencia de una espera circular.

17.6.2 - Formas de EVITAR EL DEADLOCK [9][44]

Para evitar DEADLOCKS, en los casos en que se deba coexistir con las 4 condiciones necesarias, de debe conocer de qué manera los procesos requerirán sus recursos.

El modelo más sencillo es conocer de antemano la cantidad máxima de cada tipo de recurso a necesitar. Conocido esto es posible construir un algoritmo que permita que nunca se entre en "estado de deadlock". Consiste en examinar periódicamente el número de recursos disponibles y asignados para evitar una "espera circular".

Un estado "seguro" (**SAFE**) es cuando se pueden asignar recursos a cada proceso en algún orden y evitar el deadlock.

Formalmente: Un sistema está en estado "**seguro**" si existe una "**secuencia segura de procesos**". Una secuencia <p(1),p(2), ...,p(n)> es segura si por cada p(i) los recursos que necesitará pueden ser satisfechos por los recursos disponibles más todos los recursos retenidos por todos los p(j) tal que j < i. En este caso p(i) puede esperar que todos los p(j) terminen, obtener todos sus recursos y luego que los libera, el p(i+1) a su vez, puede obtener todos los recursos necesarios. Si esta secuencia no existe el sistema está en estado inseguro "**UNSAFE**".

Veamos un ejemplo:

Los procesos necesitarían cintas, existiendo un máximo de 12 cintas disponibles.

P(0) ----> 10 P(1) ----> 4 P(2) ----> 9

En un instante T(0) dado tienen asignado:

T(0) P(0) <---- 5 P(1) <---- 2 P(2) <---- 2 3 Libres

Existe una secuencia segura <P(1),P(0),P(2)>.

	Necesita	Tiene
P(1)	4	2
P(0)	10	5
P(2)	9	2

P(1) toma 2 más. Cuando termina libera 4, más la instancia que quedaba libre son 5, luego

P(0) toma 5. Cuando termina libera las 10 retenidas y luego

P(2) toma 7 y luego termina.

Luego estamos en un estado seguro, con lo cual no podemos pasar a un estado de "deadlock" si se asignan los recursos en el orden dado por la secuencia dada.

De un estado "seguro" podemos pasar a uno "inseguro" (UNSAFE). Supongamos que en el instante T(1), se da un recurso más a P(2) :

T1) P(0) <---- 5 P(1) <---- 2 P(2) <---- 3 2 Libres

Solo P(1) puede satisfacer sus requerimientos, aquí no existe una secuencia segura, pues ni P(0), ni P(2) pueden obtener la totalidad de recursos que requieren, entonces podemos desembocar en un estado de "deadlock". Si se hubiera respetado la secuencia "segura", esto no se habría producido.

Luego se pueden definir algoritmos que aseguren no entrar en un estado "inseguro", evitando en conclusión el estado de "deadlock".

Esto está indicando que cuando un recurso es pedido, aún estando disponible, es necesario verificar si no desembocará en un estado "inseguro".

CONCLUSIONES: ** Un estado de "deadlock" es un estado "inseguro".*
** Un estado "inseguro" puede desembocar en un "deadlock", pero no necesariamente lo es.*
** Si aún pidiendo un recurso, y estando este libre, se debe esperar, esto reduce la utilización del recurso.*

17.6.2.1 - ALGORITMOS PARA EVITAR EL DEADLOCK (Varias instancias por recurso) [9][44]

El **Algoritmo del Banquero.** (The Banker's Algorithm - Dijkstra (1965) Habermann (1969)) se basa en determinar si los recursos que están libres pueden ser adjudicados a procesos sin abandonar el estado "seguro".

Supondremos N procesos y M clases de recursos y ciertas estructuras de datos

- **Disponible** : Vector de longitud M. Disponible(j) = k, indica que existen k instancias disponibles del recurso r(j).
- **MAX** : Matriz de NxM. Define la máxima demanda de cada proceso. MAX(i,j) = k, dice que p(i) requiere a lo sumo k instancias del recurso r(j).
- **Asignación** : Matriz de NxM. Define el número de recursos de cada tipo actualmente tomados por cada proceso. Asignación(i,j,) = k, dice que el proceso p(i) tiene k instancias del recurso r(j).
- **Necesidad** : Matriz de NxM. Define el resto de necesidad de recursos de cada proceso. Necesidad(i,j) = k significa que el proceso p(i) necesita k más del recurso r(j).
- **Requerimiento** : Es el vector de requerimientos de p(i). Requerimiento (i)(j) = k, indica que p(i) requiere k instancias del recurso r(j).

De lo cual se desprende que:

Necesidad(i,j) = MAX(i,j) - Asignación(i,j)

Para simplificar utilizaremos la notación siguiente:

Si X e Y son dos vectores:

$X \leq Y$ sii $X(i) \leq Y(i)$ para todo i y

$X < Y$ sii $X \leq Y$, y $X \neq Y$.

Además trataremos aparte las matrices por sus filas, por ejemplo, Asignación(i) define todos los recursos asignados por el proceso p(i).

Requerimiento(i) es el vector de requerimientos de p(i).

Requerimiento(i)(j) = k indica que p(i) requiere k instancias del recurso r(j).

ALGORITMO

Cuando se requiere un recurso se toman estas acciones:

1. Si Requerimiento(i) $\leq$ Necesidad(i) seguir, sino ERROR (se pide más que lo declarado en MAX(i)).
2. Si Requerimiento(i) $\leq$ Disponible seguir, si no debe esperar, pues el recurso no está disponible
3. Luego el sistema pretende adjudicar los recursos a p(i), modificando los estados de la siguiente forma:
   Disponible = Disponible - Requerimiento(i)
   Asignación(i) = Asignación(i) + Requerimiento(i)
   Necesidad(i) = Necesidad(i) - Requerimiento(i)

Si esta nueva situación mantiene al sistema en estado "seguro", los recursos son adjudicados. Si el nuevo estado es "inseguro", p(i) debe esperar y, además, se restaura el anterior estado de asignación total de recursos.

17.6.2.2. - ALGORITMO DE SEGURIDAD [9][44]

El algoritmo de seguridad nos permite determinar si un sistema está o no en estado seguro.

P1. Definimos **Work = Disponible** (de M elementos) y Finish = F (de Falso) para todo i con i=1,2,.....,n (Procesos).
P2. Buscamos la punta de la secuencia de 'SAFE' y los subsiguientes. Encontrar el i tal que:

a) Finish(i) = F y

b) Necesidad(i) ≤ Work

si no existe ese i , **ir a Paso P4**.

P3. Work = Work + Asignación(i)

Finish(i) = V (por Verdadero);

ir a Paso P2.

P4. Si Finish(i) = V para todo i, el sistema está en estado "SAFE", o sea, encontramos la secuencia de procesos.

Finish(i) va marcando cuáles son los procesos ya controlados (si están en V) y con Work vamos acumulando a los recursos libres que quedan a medida que terminan. Requiere MxNxN operaciones.

Ejemplo

Sea el conjunto de procesos {p(0),p(1),p(2),p(3),p(4)} y 3 recursos {A,B,C}. A tiene 10 instancias, B tiene 5 instancias y C tiene 7 instancias.

En el tiempo T(0) se tiene lo siguiente:

	Asignación	MAX	Disponible	Necesidad = MAX - Asig
	A B C	A B C	A B C	A B C
p(0)	0 1 0	7 5 3	3 3 2	7 4 3
p(1)	2 0 0	3 2 2		1 2 2
p(2)	3 0 2	9 0 2		6 0 0
p(3)	2 1 1	2 2 2		0 1 1
p(4)	0 0 2	4 3 3		4 3 1

El sistema está en estado seguro ya que la secuencia {p(1) ,p(3), p(4), p(2), p(0)} satisface el criterio de seguridad.

Supongamos ahora que p(1) requiere una instancia de A y 2 instancias de C, entonces: Requerimiento = (1,0,2).

Para decidir que puedo garantizar la satisfacción de este requerimiento:

a) Veo que Req(i) ≤ Disponible ((1,0,2) ≤ (3,2,2))

b) Veo que Req(i) ≤Necesidad ((1 0 2) ≤ (1 2 2))

c) Cumplimos el requerimiento, con lo cual se altera la información reflejando el siguiente estado:

	ASIG	NEC	NUEVO DISP.
p(0)	0 1 0	7 4 3	2 3 0
p(1)	3 0 2	0 2 0	
p(2)	3 0 2	6 0 0	
p(3)	2 1 1	0 1 1	
p(4)	0 0 2	4 3 1	

Y debemos verificar si es un estado seguro, para lo cual ejecutamos nuestro algoritmo de seguridad y hallamos la secuencia {p(1),p(3),p(4),p(0),p(2)}; por lo tanto puedo garantizar el cumplimiento del requerimiento de p(1).

Se puede ver que en este estado un requerimiento de p(4) de (3,3,0) no puede darse pues los recursos no están disponibles. Un requerimiento de p(0) de (0,2,0) no puede darse no porque no haya recursos disponibles, sino porque el estado resultante es inseguro.

Se prueba matemáticamente que si existe una secuencia segura entonces existen infinitas secuencias seguras.

17.6.2.3. - Algoritmo para recursos de una sola instancia [9][44]

Ya que el algoritmo del banquero necesita m x n^2 operaciones, para recursos de una sola instancia usaremos un algoritmo más eficiente.

Además de los arcos de requerimiento (p(i),r(j)) y de asignación (r,(j),p(i)) agregamos uno de Pedido <p(i),r(j)> que son todos los recursos que en algún momento el proceso p(i) puede llegar a solicitar, y se demarcará por una flecha de tonali-

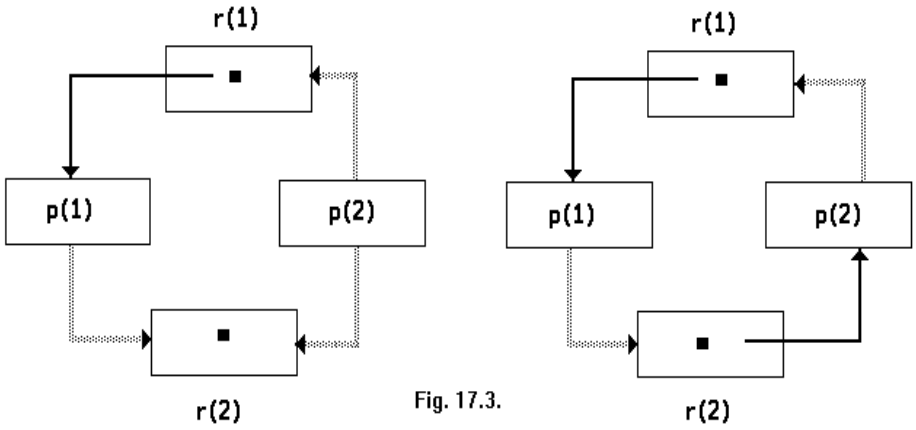

Fig. 17.3.

dad más débil.

Cuando un Pedido es requerido se transforma en Requerimiento.

Cuando un Asignado es liberado se transforma en Pedido.

El algoritmo dice: cuando un proceso p(i) requiere un recurso r(j), solo le será asignado si no se produce un ciclo en el grafo de adjudicación, pues si no hay ciclo, estamos en estado "SAFE". Si hay ciclo, estamos en estado "UNSAFE", luego hay posibilidad de Deadlock.

Por ejemplo si p(2) pide r(2) : hay ciclo, estamos "UNSAFE", si p(1) pidiese r(2) y p(2) pidiese r(1), estaríamos en "Deadlock".

Requiere sólo del orden de NxN operaciones, donde N es el número de procesos (pues es el orden de operaciones de encontrar un ciclo en un grafo : N con N).

17.6.3 - DETECCION DE DEADLOCK. [9][44]

Si no se tienen métodos que aseguren que el deadlock no ocurrirá, será necesario un esquema de detección y recuperación del 'deadlock'.

Se deberá :

a) Mantener información sobre la adjudicación de recursos y pedidos.

b) Tener un algoritmo que utilice esa información y determine si estamos en estado de 'Deadlock'.

Obviamente el algoritmo y el mantenimiento de la información incrementan el overhead que incluye no solo el tiempo de CPU, sino también el costo de mantener la información necesaria para la detección, la ejecución del algoritmo de detección y las pérdidas potenciales en caso de recupero ante deadlock.

EJEMPLO:

5 procesos P(0)....P(4), 3 recursos A, B, C. A tiene 7 instancias, B tiene 2 y C tiene 6.

	Asignación			Requerimiento			Disponible		
	A	B	C	A	B	C	A	B	C
P(0)	0	1	0	0	0	0	0	0	1
P(1)	2	0	0	2	0	2			
P(2)	3	0	2	0	0	0			
P(3)	2	1	1	1	0	0			
P(4)	0	0	2	0	0	2			

Decimos que el sistema no está en estado de deadlock. Si ejecutamos nuestro algoritmo encontramos la secuencia segura p(0), p(2), p(3), p(1), p(4) que nos da Finish(i) = Verdadero para todo i.

Si ahora modificamos que requerimiento de p(2) para el recurso pase a 4 (es decir que p(2) pida dos instancias más de C) el sistema está en deadlock. A pesar de poder utilizar los recursos asignados a p(0) esto no alcanza para satisfacer los requerimientos de los otros procesos. Luego existe un deadlock consistente en los procesos p(1), p(2), p(3) y p(4).

17.6.3.1 - Recursos con varias instancias [9][44]

- **Disponible** : Vector de longitud M que indica los recursos disponibles de cada tipo.
- **Asignación** : Matriz de NxM (N procesos). Define los recursos tomados por cada proceso.
- **Requerimiento o Espera**: Matriz de NxM. Define los recursos requeridos por cada proceso en un instante, es decir, aquellos recursos por los cuales el proceso se encuentra en espera de disponibilidad.

La matriz de Requerimiento (o Espera) está indicando aquellas instancias por las cuales un determinado proceso se encuentra en espera.

Nótese que tiene sentido analizar para el caso de detección de deadlock aquellas instancias que se encuentran efectivamente asignadas a cada proceso (matriz Asig) y aquellas instancias por las cuales los procesos se encuentran actualmente en espera.

17.6.3.2.- ALGORITMO DE SHOSHANI Y COFFMAN (1970) [9][44]

P1. Work = Disponible

Sí Asignación distinto de 0 ==> Finish(i) = Falso sino Finish(i) = Verdadero

P2. Encontrar un i tal que

a) Finish(i) = F

b) Requerimiento(i) $\leq$ Work sino ir a **Paso P4**.

P3. Work = Work + Asignación

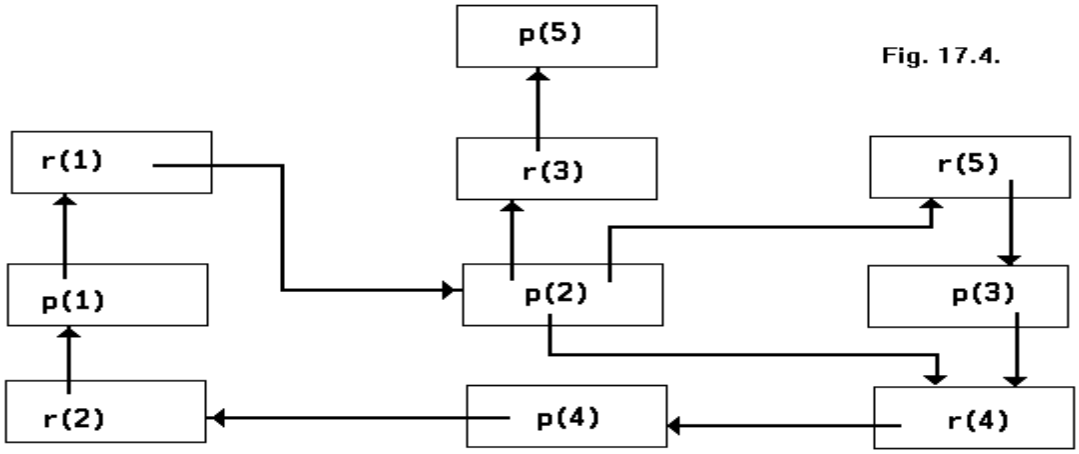

Fig. 17.4.

Finish(i) = V ir a **Paso P2**.
P4. Si Finish(i) = F entonces p(i) está en DEADLOCK.

17.6.3.3 - Algoritmos para recursos de una sola instancia [9][44]

El anterior, como antes requiere MxNxN operaciones. En este caso veamos una variante del grafo de asignación de recursos, llamado GRAFO DE ESPERA, que se obtiene quitando del primero todos los recursos.

Un arco de p(i) a p(j) en un grafo de espera indica que el proceso p(i) está esperando que el proceso p(j) libere algún recurso que el primero necesita.

Un arco p(i),p(j) existe en un grafo de espera si y solo si en el correspondiente grafo de asignación existen dos arcos p(i), r(q) y r(q),p(j) para algún recurso r(q).

Veamos un ejemplo:

En el grafo de espera hay que detectar ciclos, lo que lleva NxN operaciones, donde N es el número de procesos.

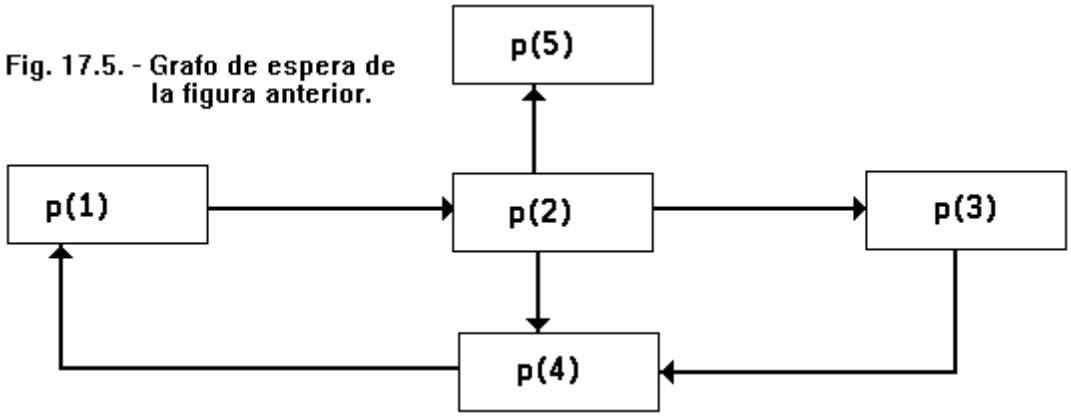

Fig. 17.5. - Grafo de espera de la figura anterior.

17.6.3.4 - CUÁNDO SE APLICAN LOS ALGORITMOS ? [9][44]

Para saber cuándo es necesario aplicar el algoritmo de detección de "deadlock", es necesario saber:

a) La frecuencia de los deadlocks.

b) Cuántos procesos son afectados.

El caso extremo es llamarlo toda vez que un requerimiento no pueda inmediatamente ser satisfecho.

Un caso más simple es invocarlo cada x tiempo, por ejemplo 1 hora, o cuando la utilización del Procesador desciende su actividad por debajo de un determinado porcentaje de utilización, pues un 'deadlock' eventualmente disminuye el uso de procesos en actividad.

Si el algoritmo es invocado en forma arbitraria habrá muchos ciclos en el grafo de recursos, No podremos determinar cuál de los procesos que forman el ciclo es el que “causa” el deadlock.

17.7. - RECUPERACION FRENTE DEADLOCK [9][44]

Frente al problema de deadlock, es necesario romperlo, de acuerdo a:

a) Violar la exclusión mutua y asignar el recurso a varios procesos
b) Cancelar los procesos suficientes para romper la espera circular
c) Desalojar recursos de los procesos en deadlock.

Para eliminar el deadlock matando procesos pueden usarse dos métodos:

- Matar todos los procesos en estado de deadlock. Elimina el deadlock pero a un muy alto costo.
- Matar de a un proceso por vez hasta eliminar el ciclo. Este método requiere considerable overhead ya que por cada proceso que vamos eliminando se debe reejecutar el algoritmo de detección para verificar si el deadlock efectivamente desapareció o no.

Esta eliminación puede no ser tan sencilla (un proceso que está a la mitad de la actualización de un archivo, un proceso que ya imprimió gran parte de un listado, etc.).
La cuestión es básicamente una cuestión de costo. Trataremos de eliminar a aquellos procesos que impliquen un “costo mínimo”. Este costo mínimo lamentablemente depende de varios factores:

- Prioridad de los procesos
- Cuánto hace que el proceso está trabajando y cuánto le falta para terminar.
- Cuántos y qué tipo de recursos usó el proceso.
- Cuántos recursos más necesita el proceso para terminar
- Cuantos procedimientos son necesarios para retrotraer al proceso a cero o a un estado anterior.

Algunas actividades necesarias para realizar estas cancelaciones serían:

17.7.1-- Selección de Víctimas [9][44]

Habría que seleccionar aquellos que rompen el ciclo y tienen mínimo costo. Para ello debemos tener en cuenta:

- Prioridad.
- Tiempo de proceso efectuado, faltante.
- Recursos a ser liberados (cantidad y calidad).
- Cuántos procesos quedan involucrados.
- Si por el tipo de dispositivo es posible volver atrás.

17.7.2.- Rollback [9][44]

- Volver todo el proceso hacia atrás (cancelarlo).
- Volver hacia atrás hasta un punto en el cual se haya guardado toda la información necesaria (CHECKPOINT)

17.7.3.- Inanición (starvation) [9][44]

Si el sistema trabajase con prioridades, podría ser que las víctimas fuesen siempre las mismas, luego habría que llevar una cuenta de las veces que se le hizo **Rollback** y usarlo como factor de decisión.

17.8. - Conclusiones [9][44]

Determinar que se va a usar para evitar el deadlock depende de la clase de recursos (Howard 1973).
Es fácil ver que un sistema que emplee esta estrategia de “clases de recursos” no estará sujeto a deadlocks. Incluso produciéndose un deadlock, no involucrará más de una “clase”, ya que se utiliza la técnica de ordenamiento de recursos.
Dentro de cada clase se utiliza alguna de las técnicas descriptas. Por ejemplo consideremos un sistema con 4 clases de recursos:

- Recursos internos: utilizados por el sistema, por ejemplo BCP
- Memoria Central: memoria usada por el trabajo del usuario
- Recursos del trabajo: dispositivos y archivos
- Espacio de swapping: espacio del trabajo del usuario en almacenamiento secundario

Una solución ideal para el manejo de los deadlocks ordena las clases como sigue:

Bloque de control de Procesos - Numeración de recursos (se demostró que nunca se entra en 'DEADLOCK').
Memoria Central - Desalojo (Memoria Virtual, Swapping)

Cintas - Se puede evitar, ya que la información se puede obtener del lenguaje de comunicación del sistema operativo y asignado en forma total.
Espacio de Swapping (en disco) - Preasignación del espacio total necesario.