Sistemas de Computación

Procesos concurrentes

Deadlock e Inanición

Dr. Wenceslao Palma M.
<wenceslao.palma@ucv.cl>

Principios

En el contexto de la mutiprogramación varios procesos pueden competir por un número finito de recursos. Cuando un proceso solicita un recurso y si en ese momento no se encuentra disponible entra en un estado de espera.

Debido a que los recursos son escasos, la solicitud de un recurso considera en forma implícita una competencia.

Un conjunto de procesos se encuentra en deadlock cuando cada uno de ellos espera un evento (adquisición/liberación) que solo puede originar otro proceso del mismo conjunto. Es decir, cada uno de ellos espera por algo que nunca ocurrirá.

El deadlock provoca que los procesos no terminen, lo cual afecta el desempeño y funcionamiento del sistema.

Modelo del sistema

Un proceso debe solicitar un recurso antes de usarlo, y liberarlo al terminar su uso.

Un proceso puede solicitar cuantos recursos requiera para llevar a cabo su tarea.

En condiciones normales, un proceso solo puede utilizar un recurso considerando la siguiente secuencia de eventos:

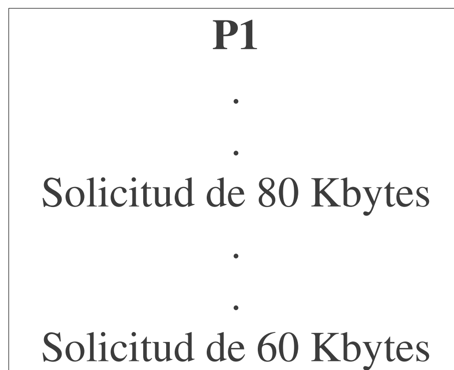
1.- **Solicitud**. si la solicitud no puede ser satisfecha de inmediato, entonces el proceso solicitante debe esperar hasta que pueda adquirir el recurso.
2.- **Utilización**.
3.- **Liberación**. El proceso libera el recurso.

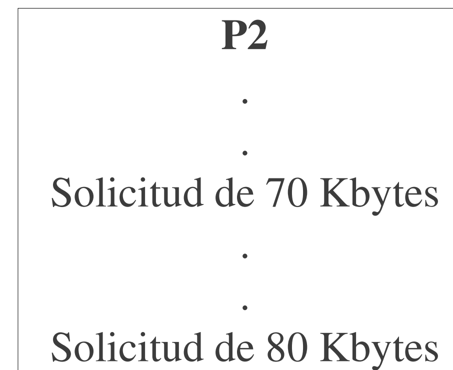
La solicitud y liberación de un recurso son llamadas al sistema.

Recursos reutilizables

Un recurso reutilizable es aquél que puede ser usado con seguridad por un proceso y no se agota con el uso (procesador, canales de E/S, memoria principal y secundaria, dispositivos y estructuras de datos, etc.).

Ejemplos de deadlock con recursos reutilizables:





Si se consideran que la memoria disponible es de 200KB, cuando ambos procesos avancen hasta su segunda petición se producirá deadlock.
Esta situación es difícil de enfrentar cuando no se conoce toda la memoria disponible, es por eso que lo usual es eliminar este tipo de situaciones utilizando memoria virtual.

Considere los recursos D (disco duro), T (cinta) y los procesos P y Q. Además, estos se ejecutan en forma concurrente de la siguiente manera

p0p1q0q1p2q2

Proceso P	
p0	Solicitar (D)
p1	Bloquear (D)
p2	Solicitar (T)
p3	Bloquear (T)
p4	realizar funcion
p5	Desbloquear (D)
p6	Desbloquear (T)

Proceso Q	
q0	Solicitar (T)
q1	Bloquear (T)
q2	Solicitar (D)
q3	Bloquear (D)
q4	realizar funcion
q5	Desbloquear (T)
q6	Desbloquear (D)

Recursos consumibles

Un recurso es consumible cuando puede ser creado y destruido.

Cuando un proceso adquiere un recurso, éste deja de existir (interrupciones, señales, mensajes e información en buffer de E/S).

Considere los siguientes procesos, los cuales utilizan receive bloqueante:

```
P1
....
receive(P2);
....
send(P2,M1);
```

```
P2
....
receive(P1);
....
send(P1,M2);
```

La causa del deadlock es un error de diseño !

Condiciones para el deadlock

Una situacion de deadlock se presenta ssi en un sistema se presentan simultáneamente las siguientes cuatro condiciones:

1.- **Exclusión mutua**. Por lo menos un recurso debe retenerse en modo compartido. Si otro proceso lo solicita, deberá esperar.

2.- **Retención y espera**. Debe existir un proceso que retenga por lo menos un recurso y espere adquirir otros recurso retenidos por otros procesos.

3.- **No apropiación**. Los recursos no se pueden quitar. Un recurso sólo puede ser liberado voluntariamente, por el proceso que lo retiene, luego que haya cumplido su tarea.

4.- **Espera circular**. Al menos, un proceso P1 espera por un recurso retenido por P2 y P2 espera por un recurso retenido por P1.

Las condiciones 1 a la 3 son necesarias para el deadlock pero no suficientes. La cuarta condición es una consecuencia potencial de las tres primeras.

Prevención del deadlock

A grandes rasgos consiste en impedir la aparición de algunas de las 3 condiciones necesarias (métodos indirectos) o evitar la aparición de la espera circular (métodos directos). Son estrategias muy conservadoras, limitan el acceso a los recurso e imponen restricciones a los procesos.

Exclusión Mutua. Esta condición no puede anularse ya que se pone en peligro la consistencia de los datos.

Retención y espera. Esta condición puede prevenirse exigiendo a todos los procesos que soliciten todos los recursos que necesitarán al mismo tiempo y bloqueando el proceso hasta que todos los recursos puedan concederse simultáneamente. En el contexto de sistemas operativos esto es ineficiente e impracticable.

No apropiación. Se puede prevenir de varias formas. Si a un proceso que retiene ciertos recursos se le deniega una solicitud, deberá liberarlos y solicitarlos nuevamente, cuando sea necesario, junto con el recurso adicional. Por otra parte, si un proceso solicita un recurso retenido por otro, el sistema operativo puede expulsar al segundo proceso y exigirle que libere sus recursos.

Espera circular. Se puede prevenir definiendo un orden lineal sobre los recursos. Para esto se asocia un índice a cada recurso. El recurso R_i antecede a R_j en la ordenación si $i<j$.
Si dos procesos A y B están interbloqueados es porque A adquirió R_i y solicitó R_j, mientras que B ha adquirido R_j y solicitado R_i. Esto es imposible porque implica que $i<j$ y $j<i$.

Predicción del deadlock

Con la predicción del deadlock es posible alcanzar las tres condiciones necesarias (), pero se realizan elecciones acertadas para que nunca se llegue a la situación de deadlock. Con esto el grado de concurrencia es mayor.

Se decide en forma dinámica si la petición actual de asignación, en caso de ser concedida, lleva a una situación de deadlock.

La predicción necesita conocer las peticiones futuras de los procesos.

Existen dos enfoques para la predicción:

- No iniciar un proceso si sus demandas pueden llevar a deadlock.
- No conceder una solicitud de incrementar los recursos de un proceso si esta asignación puede llevar a deadlock.

Negativa de iniciación de procesos.

Considere un sistema de n procesos y m tipos diferentes de recursos. Se definen los vectores y matrices siguientes:

$Recursos=(R_1, R_2, R_3, \cdots, R_m)$ cantidad total de recursos en el sistema

$Disponible=(D_1, D_2, D_3, \cdots, D_m)$ cantidad total de cada recurso sin asignar a los procesos

$$Demanda=\begin{bmatrix} C_{11} & C_{12} & \cdots & C_{1m} \\ C_{21} & C_{22} & \cdots & C_{2m} \\ \vdots & \vdots & \vdots & \vdots \\ C_{n1} & C_{n2} & \cdots & Cnm \end{bmatrix}$$

exigencias de recursos para cada proceso

C_{ij} = demanda del recurso j por parte del proceso i

$$Asignacion=\begin{bmatrix} A_{11} & A_{12} & \cdots & A_{1m} \\ A_{21} & A_{22} & \cdots & A_{2m} \\ \vdots & \vdots & \vdots & \vdots \\ A_{n1} & A_{n2} & \cdots & Anm \end{bmatrix}$$

asignación actual

A_{ij} = asignación actual del recurso j al proceso i

Se puede ver que se cumplen las siguientes relaciones:

1.-Para todo $i, R_i=D_i+\sum_{k=1}^{n} A_{ki}$, todos los recursos están asignados o disponibles.

2.-Para todo $k, i; C_{ki} \leq R_i$, ningún proceso puede demandarmás recursos que la cantidad total de recursos del sistema.

3.-Para todo $k, i; A_{ki} \leq C_{ki}$, ningún proceso tiene asignados más recursos de cualquier tipo que los que declaró necesitar.

Con los tres puntos anteriores es posible definir una política de predicción de deadlock que rechace iniciar un nuevo proceso si sus exigencias de recursos pueden conducir a un deadlock.

Un nuevo proceso P_{n+1} comenzará solo si para todo i: $R_i \geq C_{(n+1)i} + \sum_{k=1}^{n} C_{ki}$

Un nuevo proceso comenzará sólo si puede ser satisfecha la demanda máxima de todos los procesos actuales más la del nuevo proceso.

Negativa de asignación de recursos

Fue propuesta por Dijkstra, se denomina **algoritmo del banquero**.

Consideremos un sistema con número fijo de procesos y un número fijo de recursos. En un instante dado, un proceso tendrá asignado cero o más recursos. El estado del sistema es, simplemente, la asignación actual de recursos a los procesos.

Un estado seguro es un estado en el cual existe al menos una secuencia que no lleva a deadlock, es decir, todos los procesos pueden ejecutarse hasta el final.

	R1	R2	R3
P1	3	2	2
P2	6	1	3
P3	3	1	4
P4	4	2	2

Claim Matrix

	R1	R2	R3
P1	1	0	0
P2	6	1	2
P3	2	1	1
P4	0	0	2

Allocation Matrix

R1	R2	R3
9	3	6

Resource Vector

R1	R2	R3
0	1	1

Available Vector

(a) Initial state

	R1	R2	R3
P1	3	2	2
P2	0	0	0
P3	3	1	4
P4	4	2	2

Claim Matrix

	R1	R2	R3
P1	1	0	0
P2	0	0	0
P3	2	1	1
P4	0	0	2

Allocation Matrix

R1	R2	R3
6	2	3

Available Vector

(b) P2 runs to completion

	R1	R2	R3
P1	0	0	0
P2	0	0	0
P3	3	1	4
P4	4	2	2

Claim Matrix

	R1	R2	R3
P1	0	0	0
P2	0	0	0
P3	2	1	1
P4	0	0	2

Allocation Matrix

R1	R2	R3
7	2	3

Available Vector

(c) P1 runs to completion

	R1	R2	R3
P1	0	0	0
P2	0	0	0
P3	0	0	0
P4	4	2	2

Claim Matrix

	R1	R2	R3
P1	0	0	0
P2	0	0	0
P3	0	0	0
P4	0	0	2

Allocation Matrix

R1	R2	R3
9	3	4

Available Vector

(d) P3 runs to completion

La estrategia de predicción de deadlock no lo predice con certeza, sólo anticipa la posibilidad de deadlock y asegura que nunca exista tal posibilidad.

La predicción tiene la ventaja de que no es necesario expulsar y hacer retroceder procesos, como en la detección, y es menos restrictiva que la prevención.

Cómo se determina un estado inseguro?

	R1	R2	R3
P1	3	2	2
P2	6	1	3
P3	3	1	4
P4	4	2	2

Claim Matrix

	R1	R2	R3
P1	1	0	0
P2	5	1	1
P3	2	1	1
P4	0	0	2

Allocation Matrix

R1	R2	R3
9	3	6

Resource Vector

R1	R2	R3
1	1	2

Available Vector

(a) Initial state

	R1	R2	R3
P1	3	2	2
P2	6	1	3
P3	3	1	4
P4	4	2	2

Claim Matrix

	R1	R2	R3
P1	2	0	1
P2	5	1	1
P3	2	1	1
P4	0	0	2

Allocation Matrix

R1	R2	R3
0	1	1

Available Vector

(b) P1 requests one unit each of R1 and R3

La predicción supone una serie de restricciones:

Se debe presentar la máxima demanda de recursos por anticipado.

Los procesos a considerar deben ser independientes, es decir, el orden de ejecución no debe estar forzado por condiciones de sincronización.

El número de recursos y de procesos debe ser fijo.

Los procesos no deben finalizar mientras retengan recursos.

Detección de deadlock

Por el contrario a las estrategias de prevención, en la detección de deadlock no limitan el acceso a los recursos ni restringen las acciones de los procesos.

Los recursos se concederán siempre que sea posible. Periódicamente el sistema operativo ejecuta un algoritmo que permite detectar la condición de espera circular.

Algoritmo de detección de deadlock

Utilizaremos el propuesto en el paper “Systems Deadlocks”. Coffman, Elphick & Shoshani. Computing Surveys, June 1971. Sin embargo, usar WFG es más fácil :-)

Utiliza la matriz Asignacion y el vector Disponible descritos anteriormente. Además emplea una matriz Q, de forma que q_{ij} representa la cantidad de recurso de tipo j solicitados por el proceso i. El algoritmo funciona marcando los procesos que no están en deadlock. Inicialmente todos están sin marcar.

La estrategia consiste en encontrar un proceso cuyas solicitudes de recursos puedan ser satisfechas con los recursos disponibles y suponer que esos recurso se conceden, el proceso se ejecuta hasta terminar y libera los recursos. Luego, se busca otro proceso que completar.

A continuación se llevan a cabo los siguientes pasos:

1.- Marcar cada proceso que tiene una fila de la matriz Asignación sólo con ceros.

2.- Iniciar el vector temporal W con el vector Disponible.

3.- Buscar un índice i tal que el proceso i no esté actualmente marcado y la columna i-ésima de Q sea menor o igual que W. Es decir $Q_{ik} <= W_k$. Si no se encuentra el algoritmo ha terminado.

4.- Si se encuentra la columna, marcar el proceso i y sumar la columna correspondiente de la matriz Asignacion a W. Es decir, $W_k = W_k + A_{ik}$. Volver al paso 3.

Existe deadlock ssi hay procesos no marcados al terminar el algoritmo. Cada proceso no marcado está en deadlock.

	R1	R2	R3	R4	R5
P1	0	1	0	0	1
P2	0	0	1	0	1
P3	0	0	0	0	1
P4	1	0	1	0	1

Request Matrix Q

	R1	R2	R3	R4	R5
P1	1	0	1	1	0
P2	1	1	0	0	0
P3	0	0	0	1	0
P4	0	0	0	0	0

Allocation Matrix A

R1	R2	R3	R4	R5
2	1	1	2	1

Resource Vector

R1	R2	R3	R4	R5
0	0	0	0	1

Available Vector

Figure 6.9 Example for Deadlock Detection

Recuperación de deadlock

Una vez detectado, el deadlock se debe recuperar. Existen varias formas de realizar una recuperación:

1.- Abortar todos los procesos que participan del deadlock.

2.- Retoceder cada proceso que está en deadlock hasta algún punto de control definido previamente y volver a ejecutarlos. Es necesaria la disponibilidad de mecanismos de retroceso y reinicio. El deadlock se podría repetir, sin embargo, se asume que con el no determinismo del procesamiento concurrente esto no ocurrirá.

3.- Abortar sucesivamente procesos en deadlock hasta que deje de haberlo. El orden en que los procesos abortan debe seguir un criterio de mínimo costo. Después de abortar es necesario ejecutar nuevamente el algoritmo de detección.

4.- Apropiación de recursos en forma sucesiva hasta que el deadlock deje de existir. La selección es basada en costo y es necesario ejecutar nuevemante el algoritmo de detección. Cuando un proceso pierde un recurso por apropiación debe retroceder hasta un “momento anterior” a la adquisición de ese recurso.

Los criterios para 3 y 4 pueden ser:

- Menor cantidad de tiempo de procesador consumido hasta ahora.
- Menor número de líneas de salidas producidas hasta ahora.
- Mayor tiempo restante estimado.
- Menor número toral de recursos asignados hasta ahora.
- Prioridad más baja.

El problema de los filósofos

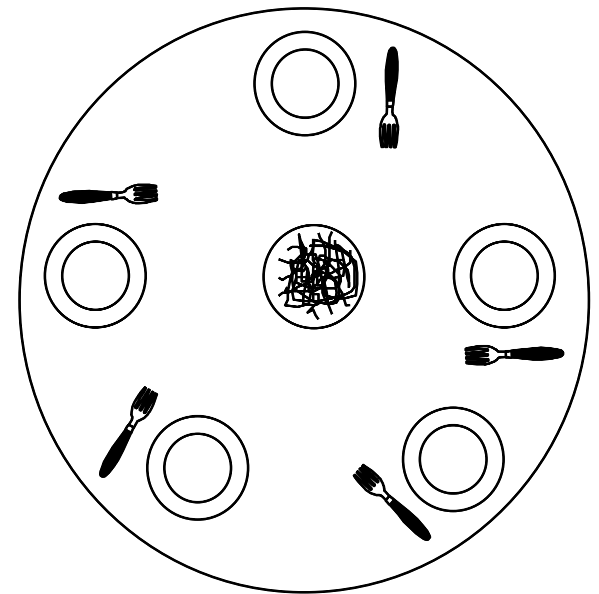
Había una vez cinco filósofos que vivían juntos. La vida de cada uno consistía principalmente en pensar y comer y, tras años de pensar, todos los filósofos se habían puesto de acuerdo en que la única comida que contribuía a sus esfuerzos pensadores eran los espaguetis.

Los preparativos de la comida incluyen: una mesa redonda en donde se ubica una fuente de espaguetis, cinco platos, uno para cada filósofo y cinco tenedores.

Un filósofo que quisiera comer iría a su lugar asignado en la mesa y, usando los dos tenedores de cada lado del plato, tomaría los espaguetis y se los comería. El problema está en inventar un ritual que permite comer a los filósofos. El algoritmo debe satisfacer la exclusión mutua y evitar deadlock e inanición.

```
semaforo tenedor[5] = {1};
int i;
void filosofo(int i){
        while (cierto){
                pensar();
                wait(tenedor[i]);
                wait(tenedor[(i+1) mod 5]);
                comer();
                signal(tenedor[(i+1)mod 5]);
                wait(tenedor[i]);
        }
}

void main(){
        parbegin(filosofo(0), filosofo(1), filosofo(2), filosofo(3),
        filosofo(4));
}
```



Algún problema?

```
semaforo tenedor[5] = {1};
semaforo habitacion = {4};
int i;
void filosofo(int i){
        while (cierto){
                pensar();
                wait(habitacion);
                wait(tenedor[i]);
                wait(tenedor[(i+1) mod 5]);
                comer();
                signal(tenedor[(i+1)mod 5]);
                wait(tenedor[i]);
                signal(habitacion);
        }
}

void main(){
        parbegin(filosofo(0), filosofo(1), filosofo(2), filosofo(3),
        filosofo(4));
}
```