Problemas clásicos de concurrencia

Roberto Gómez Cárdenas
rogomez@campus.cem.itesm.mx

October 4, 1999

En los sesentas Edsger Dijkstra y cinco de sus colegas desarrollaron uno de los primeros sistemas operativos que soportaban la multiprogramación. El sistema fue diseñado en la Universidad Tecnológica de Eindhoven en Holanda, y fue llamado sistema multiprogramación THE, (debido al nombre de la Universidad en alema'n.). Una de las principales aportaciones de este sistema fue la incorporación del concepto de semáforos, como herramienta para implementación de la exclusión mutua.

Por ese mismo tiempo Dijkstra [Dij65] escribió un artículo sobre la copperación de procesos. Este papel mostraba como utilizar los semáforos para resolver una variedad de problemas de sincronizaci'on, como el de los productores/consumidores, los filósofos que cenan y el del barbero dormilón.

En su papel sobre monitores Tony Hoare [Hoa74] introdujo el concepto de *semáforos binarios separados* y muestra como usarlos para implementar monitores. Sin embargo Dikstra, ([Dij79, Dij80]) fue el que tiempo después le dio un nombre a la técnica e ilustró su uso general. Más concretamente Dijkstra, (ver [Dij79]), mostró como implementar semáforos generales usando solo semáforos binarios.

Es posible encontrar diferentes soluciones para el problema de los filósofos que cenan. Una de las soluciones utiliza asimetría para evitar el interbloqueo (i.e. que dos filósofos se bloqueen entre ellos). Es decir un filósofo toma el tenedor en orden diferente que los otros. También se podría definir un orden en el que los filósofos de número impar tomen un tenedor en ese orden, y que los filósofos de número par lo toman en otro orden. Una segunda forma de evitar el interbloqueo es permitir que a lo más cuatro filófosos puedan tomar tenedores, (i.e. permitir que a lo más cuatro filósofos se sienten en la mesa.). Una tercera solución es tener a los filósofos tomando tenedores al mismo tiempo, dentro de una sección crítica, sin embargo esto puede llevar a un problema de inanición (i.e. algún filósofo podría quedarse sin cenar). Mani Chandy y Jay Misra, ([ChMi84]), proponen una cuarta solución, basada en el paso de fichas entre los filósofos, (esta solución es apropiada para sistemas distribuidos).

Las anteriores soluciones son determinísticas, (i.e. cada proceso ejecuta un conjunto predecible de acciones). Lehman y Rabin,([LeRa81]), presentan una interesante solución probabilística que es perfectamente simétrica. La idea base es que cada filósofo usa lanzamientos de monedas para determinar el orden en el cual trataran de tomar los tenedores, y si dos filósofos desean utilizar el mismo tenedor, el filósofo que mas recientemente utilizo el tenedor le da preferencia al otro.

Courtois, Heymans y Parnas,([Co+71]), introducieron el problema de los lectores escritores y presentaron dos soluciones usando semáforos. En una de ellas se le da preferencia a los lectores y en la otra a los escritores.

1 Los *clásicos* de la concurrencia

Estos problemas se encuentran en cualquier libro de sistemas operativos, dentro del capítulo de procesos y/o sincronización de procesos. A continuación se el planteamiento de estos.

1. *El del fumador de cigarros*, ([Pat71]). Considere un sistema con tres procesos *fumadores* y un proceso *agente*. Cada fumador esta continuamente enrrollando y fumando cigarrillos. Sin embargo, para enrrollar y fumar un cigarrillo, el fumador necesita tres ingredientes: *tabaco*, *papel*, y *fosforos*. Uno de los procesos fumadores tiene papel, otro tiene el tabaco y el tercero

los fosforos. El agente tiene una cantidad infinita de los tres materiales. El agente coloca dos de los ingredientes sobre la mesa. El fumador que tiene el ingrediente restante enrrolla un cigarillo y se lo fuma, avisando al agente cuando termina. Entonces, el agente coloca dos de los tres ingredientes y se repite el ciclo. Escriba un programa en *pseadoucódigo*, utilizando sémaforos, para sincronizar al agente y a los fumadores.

2. *La panadería de Lamport*, ([Lam74]). En este problema una panadería tiene una variedad de panes y pasteles vendidos por n vendedores. Cada uno de los cuales toma un número al entrar. El cliente espera hasta oír su número. Cuando el vendedor se desocupa, llama al siguiente número. Escriba un procedimiento para los vendedores y otro para los clientes.

3. *Los filósofos que cenan*, ([Dij71]). Hay cinco filósofos chinos que se pasan sus vidas pensando y comiendo. Comparten una mesa circular, alrededor de la cual se sientan. En su centro se encuentra con una provisión infinita de arroz, y sobre ella hay cinco palitos, uno de cada lado de los filósofos. Cuando un filósofo piensa, no interactua con sus colegas. De vez en cuando, un filósofo tiene hambre y trata de levantar los dos palitos más cercanos a él. Un filófoso puede levantar un palito a la vez, y no puede tomar un palito que ya está en la mano de un vecino. Cuando un filósofo tiene ambos palitos, puede comer. Cuando terminó de hacerlo, deja sus dos palitos y comienza a pensar de nuevo. Escriba un programa, en base a semáforos, que permita que lo anterior pueda llevarse a cabo.

4. *El barbero dormilón*, ([Dij65]). Una peluquería tiene un barbero, una silla de peluquero y n sillas para que se sienten los clientes en espera, si es que los hay. Si no hay clientes presentes, el barbero se sienta en su silla de peluquero y se duerme. Cuando llega un cliente, éste debe despertar al barbero dormilón. Si llegan más clientes mientras el barbero corta el cabello de un cliente, estos deben esperar sentados (si hay sillas desocupadas) o salirse de la peluquería (si todas las sillas están ocupadas). El problema consiste en programar al barbero y los clientes sin entrar en condición de competencia.

5. *Lectores y escritores*, ([Co+71]). Imaginemos una enorme base de datos, como por ejemplo un sistema de reservaciones de en una línea aérea, con muchos procesos en competencia, que intentan leer y escribir en ella. Se puede aceptar que varios procesos lean la base de datos al mismo tiempo, pero si uno de los procesos está escribiendo, (es decir modificando) la base de datos, ninguno de los demás procesos debería tener acceso a ésta, ni siquiera los lectores. El problema es como programar a los lectores y escritores.

6. *Productor/Consumidor*[1]., ([Dij65]). Dos procesos comparten un almacén (buffer) de tamaño fijo. Uno de ellos, el productor, coloca información en el almacén (buffer) mientras que el otro, el consumidor, la obtiene de él. Si el productor desea colocar un nuevo elemento, y el almacén se encuentra lleno, este debera irse a “dormir”. El consumidor despertará al productor cuando elimine un elemento del almacén. De forma análoga, si el almacén esta vacío y el consumidor desea eliminar un elemento del almacén, este debe “dormirse” hasta que el productor coloque algo en el almacén.

2 Soluciónes a algunos problemas

A continuación se presentan soluciones algunos de los problemas discutifos anteriormente. Dichas soluciones fueron tomadas de [Tan91]. Como toda solución, no es la única, es posible que el lector encuentre otra.

[1]También conocido como *bounded buffer problem* o problemna del buffer limitado

2.1 Solución al problema productor consumidor con semaforos

La solución utiliza tres semáforos. La solución es presentada en un lenguaje cercano a C.

```
semaforo mutex = 1;
semaforo vacio = N;
semaforo lleno = 0;

productor()
{
  while (TRUE) {
    produce_item();
    P(vacio);
    P(mutex);
     introducir_item();
    V(mutex);
    V(lleno);
}

consumidor()
{
  while (TRUE) {
    produce_item();
    P(lleno);
    P(mutex);
     retirar_item();
    V(mutex);
    V(vacio);
    consume_item();
}
```

2.2 Solución productor consumidor con monitores

Un ejercicio interesante para el lector es el de comparar esta solución con la presentada en la sección anterior. La solución es presentada en un lenguaje muy cercano a Pascal.

```
monitor ProductorConsumidor
   condition  vacio, lleno;
   integer count;

   procedure producir;
   begin
     if (count = N) then
       wait(lleno);
     introducir_item;
     count=count+1;
     if (count = 1) then
       signal(vacio);
   end
```

```
   procedure consumir;
   begin
     if (count = 0) then
       wait(vacio);
     retirar_item;
     count=count-1;
     if (count = N-1) then
       signal(lleno);
   end;
   count=0;
end monitor;

procedure productor
begin
  while (true) do
  begin
    producir_item;
    ProductorConsumidor.producir
  end
end

procedure consumidor
begin
  while (true) do
  begin
    ProductorConsumidor.consumir;
    consumir_item;
  end
end
```

2.3 Solución al problema de la cena de los filósofos

La solución utiliza un arreglo `state` para llevar un registro de la actividad de un filósofo: si esta comiendo, pensando o hambriento. Un filósofo solo puede comer si los vecinos no estan comiendo Los vecinos del i-ésimo filósofo se definen en los macros LEFT y RIGHT. En otras palabras si $i = 2$ entonces $LEFT = 1$ y $RIGHT = 3$. Así mismo el programa utiliza un arreglo de semáforos, uno por cada filósofo, de manera que los filósofos hambrientos pueden bloquearse si los tenedores necesarios están ocupados.

```
#define N         5
#define LEFT      (i-1)%N
#define RIGHT     (i+1)%N
#define THINKING  0
#define HUNGRY    1
#define EATING    2

int state[N];

semaforo mutex = 1;
semaforo s[N];

filosofo()
{
  while (TRUE)  {
```

```
      think();
      toma_tenedor(i);
      come();
      libera_tenedor(i);
  }
}

toma_tenedor(int i)
{
  P(mutex);
  state[i]=HUNGRY;
  test(i);
  V(mutex);
  P(s[i]);
}

libera_tenedor(int i)
{
  P(mutex);
  state[i]=THINKING;
  test(LEFT);
  test(RIGHT);
  V(mutex);
}

test(int i)
{
  if (state[i] == HUNGRY && state[LEFT] != EATING && state[RIGHT] != EATING) {
    state[i] = EATING;
    V(s[i]);
  }
}
```

2.4 Solución problema lectores/escritores

```
semaforo mutex = 1;
semaforo db = 1;
int rc = 0;

lector()
{
   while (TRUE) {
      P(mutex);
      rc = rc+1;
      if (rc == 1)
        P(db);
      V(mutex)
      leer_datos();
      P(mutex);
      rc = rc+1;
      if (rc == 0)
        V(db);
      V(db);
      procesamiento_datos();
}
```

```
escritor()
{
   while (TRUE) {
      analizando_dato();
      P(db);
      escribir_datos();
      V(db);
}
```

2.5 Solución problema barbero dormilón

```
#define CHAIRS 5

semaforo clientes=0;
semaforo barberos=0;
semaforo mutex=1;
int waiting = 0;

barbero()
{
   P(clientes);
   P(mutex)
   waiting = waiting - 1;
   V(barberos)
   V(mutex);
   cortar_pelo();
}


cliente()
{

   P(mutex);
   if (waiting < CHAIRS) {
       waiting = waiting + 1;
       V(clientes);
       V(mutex);
       P(barberos);
       tomar_barbero();
   }
   else
       V(mutex);
}
```

References

[And91] Andrews R. Gregory, *Concurrent Programming, Principles and Practice*, 1991 The Benjamin and Cummings Publishing Company.

[ChMi84] Chandy K.M., and Misra J. *The drinking philosphers problem* ACM Trans. on Prog. Languages and Systems 6, 4(Octoiber), pp. 632-646.

[Co+71] Courtois P.J., Heymans F., y Parnas D.L. *Concurrent Control with Readers and Writers*, Comm. of the ACM, vol. 10, pp. 667-668, Oct. 1971

[De91] Deitel H.M., *Operating Systems*, Addison Wesley, 2do. Edition.

[Dij65] Dijkstra E.W., *Cooperating Sequential Processes* Technological University, Eindhoven, Netherlands, 1965 (Reprinted in F. Genuys (ed.), *Programming Languages*, New York, NY: Academic Press, 1968).

[Dij71] Dijkstra E.W. *Hierarchical Ordering of Sequential Processes* Acta Informatica, Volume 1, Number 2 (1971), pp. 115-138; reprinted in [Hoare and Perrot 1972], pp. 72-93.

[Dij79] Dijkstra E.W. *A tutorial on the split binary semaphore.* EWD 703, Neunen, The Netherlands, March.

[Dij80] Dijkstra E.W. *The superfluity of the general semaphore.* EWD 734, Neunen, The Netherlands, April.

[Her89] Herman, J.S., *A comparison of synchronization mechaniusm for concurrent programming.* Master's thesis, CSE-89-26, University of California at Davis, September 1989.

[Hoa74] Hoare C.A.R. *Monitors: an operating system structuring concept* Comm. of the ACM 17,10 (October), 549-557.

[HoPe72] CAR Hoare and RH Perrot Eds. *Operating Systems Techniques*, Acedemic Press, London 1972

[Lam74] Lamport L., *A New Solution to Dijkstra's Concurrent Programming Problem*, Comm. of the ACM, vol 17, pp. 453-455, Ago. 1974

[LeRa81] Lehmann D., y Rabin M.O., *A symmetric and fully distributed solution to the dining philosophers problem.* Prco. Eighth ACM Symp. on Princ. of Prog, Languages, January 1981, pp. 133-138

[Pat71] Patil S.S. *Limitations and Capabilities of Dijkstra's Semaphore Primitives for Coordination among Processes*, M.I.T. Project MAC Computational Structures Group Memo 57, February 1971.

[Tan91] Tanenbaum A., *Modern Operating Systems*, 1992 Prentice Hall.