

Sincronización entre procesos


Problemas clásicos

Rodolfo Baader

Departamento de Computación, FCEyN,
Universidad de Buenos Aires, Buenos Aires, Argentina

Sistemas Operativos, segundo cuatrimestre de 2014

(2) Siempre los mismos problemas...

- En muchas áreas de la computación hay problemas clásicos.
- Eso significa que aparecen una y otra vez bajo diferentes formas, pero son básicamente el mismo problema.
- En el caso de sincronización entre procesos también sucede que muchos de estos problemas nos suelen inducir a pensar en “soluciones” incorrectas.
- Por eso está bueno estudiarlos y conocer buenas formas de solucionarlos.
- Además, analizarlos suele aportar una mirada bastante profunda sobre problemas fundamentales de la computación, y a veces, del razonamiento humano.
- **OJO:** conocer los problemas clásicos y sus soluciones no debe ser una excusa para la pereza. Los problemas concretos tienen particularidades y contextos que hacen necesario evaluarlos de manera individual y **pensar** en cada caso. 
- Dicho de otra forma, el conocimiento no reemplaza a la inteligencia. Ambos se complementan.

(3) Los turnos

- Problema:
 - Tenemos una serie de procesos $P_i, i \in \{1 \dots N\}$ que se están ejecutando en simultáneo (supongamos que tenemos N CPUs).
 - Cada proceso i tiene una sentencia s_i .
 - Queremos que en el sistema global se ejecuten s_1, s_2, \dots, s_N .
- ¿De qué nos disfrazamos?
- Podemos utilizar semáforos.

(4) Los turnos (cont.)

Listing 1: Inicialización

```
• proc init()
{
    for (i= 0; i<N+1; i++)
        semaforos[i]= new Semáforo(0);

    for (i= 0; i<N; i++)
        fork P(i);

    semaforos[0].signal();
}
```

Listing 2: Proceso P_i

```
• proc P(i)
{
    sem_ant= semaforos[i];
    sem_sig= semaforos[i+1];
    sem_ant.wait();
    s(i);
    sem_sig.signal();
}
```

(5) Barrera - Rendezvous

- Otro problema muy común es el de *rendezvous* (punto de encuentro), o *barrera de sincronización*.
- Dados $P_i = [a(i); b(i)]$, $i \in \{1 \dots N\}$, asegurarse que los $b(i)$ se ejecuten recién después de que se ejecutaron todos los $a(i)$.
- Es decir, queremos *poner una barrera* entre los a y los b .
- Sin embargo, no hay que restringir de más: no hay que imponer ningún orden entre los $a(i)$ ni entre los $b(i)$.

(6) Rendezvous (cont.)

Listing 3: Inicialización

```
• proc init()  
  {  
    en_barrera= 0; // Cant. de procs en la barrera.  
    mutex= new Semáforo(1); // Permiso empezar de inmediato.  
    barrera= new Semáforo(0);  
  
    for (i= 0; i<N; i++) fork P(i);  
  }
```

(7) Rendezvous (cont.)

Listing 4: Proceso P_i

```
• proc P(i)
  {
    a(i);

    mutex.wait(); // Acceso a var. compartida.
    en_barrera++;
    bool todos_en_barrera= (en_barrera==N);
    mutex.signal();


    if (todos_en_barrera)
      // Todos están listos para continuar. Que sigan.
      barrera.signal();

    barrera.wait();

    b(i);
  }
```

- Esta solución no es correcta.
- ¿Qué está mal?

(8) Rendezvous (cont.)

- En el ejemplo anterior había deadlock. 

Listing 5: Proceso P_i (correcto)

- ```
proc P(i)
{
 a(i);

 mutex.wait(); // Acceso a var. compartida.
 en_barrera++;
 bool todos_en_barrera= (en_barrera==N);
 mutex.signal();

 if (todos_en_barrera)
 // Todos están listos para continuar. Que sigan.
 barrera.signal();

 barrera.wait();
 barrera.signal();

 b(i);
}
```

- Notar que hay un signal() de más.



## (9) Rendezvous (cont.)

- ¿Qué pasa si contamos con la primitiva `broadcast()`?

### Listing 6: Proceso $P_i$

- ```
proc P(i)
{
    a(i);

    mutex.wait(); // Acceso a var. compartida.
    en_barrera++;
    bool todos_en_barrera= (en_barrera==N);
    mutex.signal();

    if (todos_en_barrera)
        // Todos están listos para continuar. Que sigan.
        barrera.broadcast();
    else
        barrera.wait();

    b(i);
}
```

- Esta solución no es correcta.
- ¿Qué está mal?

(10) Sección crítica de a k

- Siguiente problema. N procesos, sólo k ejecutan a la vez la sección crítica $sc(i)$.
- Es fácil:

Listing 7: Inicialización

```
proc init()  
{  
    sem= new Semáforo(k);  
  
    for (i= 0; i<N; i++) fork P(i);  
}
```

Listing 8: Proceso P_i

```
proc P(i)  
{  
    sem.wait();  
    sc(i);  
    sem.signal();  
}
```

(11) Lectores/escritores

- Otro problema más. Se da mucho en bases de datos.
- Hay una variable compartida.
- Los escritores necesitan acceso exclusivo.
- Pero los lectores pueden convivir.
- ¿Cómo podría solucionarse?

(12) Lectores/escriptores (cont.)

Listing 9: Inicialización

- ```
proc init()
{
 variable compartida; // Ésta es la que todos se disputan.
 mutex= new Semáforo(1);
 acceso_exclusivo= new Semáforo(1);
 int lectores= 0;

 for (i= 0; i<N; i++) fork Lector(i);
 for (i= 0; i<M; i++) fork Escriitor(i);
}
```

- Los escritores necesitan acceso exclusivo, sin vueltas:

### Listing 10: Escritores

```
proc Escriitor(i)
{
 acceso_exclusivo.wait();
 Escribir(compartida);
 acceso_exclusivo.signal();
}
```

## (13) Lectores/escriptores (cont.)

### Listing 11: Lectores

```
• proc Lector(i)
{
 mutex.wait();
 lectores++;
 if (lectores==1)
 // Soy el primero, reclamo acceso "exclusivo" para
 // todos los lectores.
 acceso_exclusivo.wait();
 mutex.signal();

 Leer(compartida);

 mutex.wait();
 lectores--;
 if (lectores==0)
 // Soy el último, dejo que entren los escriptores.
 acceso_exclusivo.signal();
 mutex.signal();
}
```

## (14) Lectores/escritores (cont.)

- ¿Puede haber deadlock?
- No, pero puede haber inanición de escritores.
- Tarea: pensar cómo evitarlo.

## (15) Filósofos que cenan (Dining Quintuple/Philosophers)

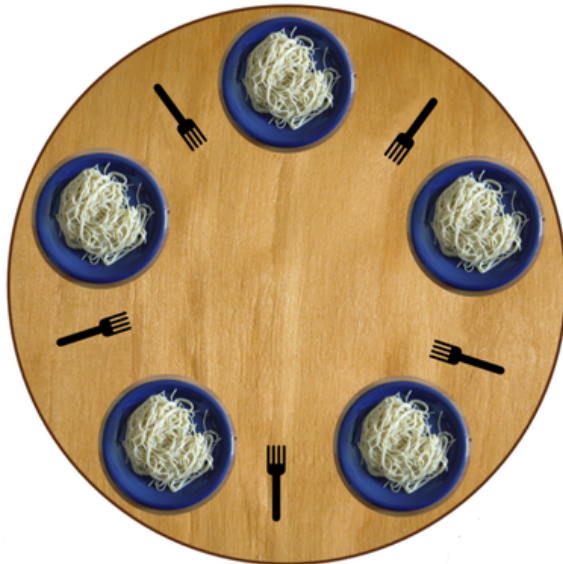
- Este problema lo inventó Dijkstra en 1965.
- Tenemos 5 filósofos en una mesa circular, con 5 platos de fideos y un tenedor entre cada plato. Para comer necesitan dos tenedores.
- Los filósofos hacen:

```
proc Filósofo(i)
 while (true)
 {
 pensar();
 conseguir_tenedores(i);
 comer();
 dejar_tenedores(i);
 }
```

- Problema: programar conseguir\_tenedores() y dejar\_tenedores() de manera tal que:

**EXCL)** Sólo un filósofo tenga cada tenedor en cada momento.  
**DEAD)** No haya deadlock.  
**INAN)** No haya inanición.  
**CONC)** Más de un filósofo esté comiendo a la vez.

## (16) Gráficamente





## (17) Filósofos que cenar

- Empecemos por algunas definiciones básicas:
- Macros cómodas para los tenedores:
  - $\text{izq}(i) = i$
  - $\text{der}(i) = (i + 1) \bmod 5$
- ¿Qué son los tenedores?
- Seguro que hay que garantizar acceso exclusivo a ellos:
- Un arreglo: `tenedores[i] = new Semáforo(1)`

## (18) Solución ingénuo

- Empecemos por la idea más elemental:

```
func conseguir_tenedores(i)
{
 tenedores[izq(i)].wait();
 tenedores[der(i)].wait();
}
```

```
func dejar_tenedores(i)
{
 tenedores[izq(i)].signal();
 tenedores[der(i)].signal();
}
```

- ¿Está bien?
- Esta solución garantiza EXCL, pero falla con DEAD.
- ¿Cómo rompemos el deadlock? Pensemos en las condiciones.
- Tarea: pensar en soluciones para este problema. Buscar las ya existentes.

- Otro problema clásico:
- En una peluquería hay dos salas, una de espera, con  $n$  sillas y otra donde está la única silla donde el único peluquero corta el pelo.
- Si no hay clientes, el peluquero se duerme una siesta.
- Si entra un cliente, y no se puede sentar a esperar, se va.
- Si el peluquero está dormido, lo despierta.

- Vamos a usar tres semáforos:

```
algun_cliente= new Semáforo(0);
pasar= new Semáforo(0);
mutex= new Semáforo(1);
clientes= 0;
```

- El peluquero es sencillo:

```
proc Peluquero
 while (true)
 {
 algun_cliente.wait();
 pasar.signal();
 cortar_pelo();
 }
```

## (21) Barbero (cont.)

- Veamos a los clientes:

```
proc Cliente()
 // ¿Hay lugar? Capacidad = n + 1 (al que le cortan)
 mutex.wait();
 if (clientes==n+1)
 // No, me voy.
 mutex.signal();
 retirarse(); // No vuelve
 // Sí hay lugar, entro.
 clientes++;
 mutex.signal();

 // "Aviso" que llegué.
 algun_cliente.signal();
 // Espero a que me dejen pasar.
 pasar.wait();

 entrar_a_cortarse_el_pelo(); // ¡Todo para esto!

 // Salgo.
 mutex.wait();
 clientes --;
 mutex.signal();
```

- “The Little Book of Semaphores”, Second Edition. Allen B. Downey. <http://greenteapress.com/semaphores/>
- “Cooperating sequential processes”. Edgar W. Dijkstra. Technical Report 123, Univ. Texas. <http://www.cs.utexas.edu/users/EWD/transcriptions/EWD01xx/EWD123.html>.