

Sincronización entre procesos 3: Razonamiento y problemas clásicos (segunda parte)

Fernando Schapachnik

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

Sistemas Operativos, primer cuatrimestre de 2021

(2) Créditos

Basado fuertemente en trabajo de Sergio Yovine.

(3) La clase anterior

- Vimos:
 - Algunos problemas clásicos.
 - Fairness
 - Exclusión mutua
 - Progreso del sistema (o “de alguno”, *LOCK-FREEDOM*).
 - Progreso global dependiente (*STARVATION-FREEDOM*).
 - Progreso global absoluto (*WAIT-FREEDOM*).
- Hoy vamos a ver:
 - Más problemas clásicos.
 - Una jerarquía de mecanismos de sincronización.

(4) Livelock

- El primo olvidado del deadlock: *livelock*.
- Un conjunto de procesos está en livelock si estos continuamente cambian su estado en respuesta a los cambios de estado de los otros.
- Ejemplo: sistema automatizado para terapia intensiva.
- Ejemplo: queda poco espacio en disco. Proceso A detecta la situación y notifica a proceso B (bitácora del sistema). B registra el evento en disco, disminuyendo el espacio libre, lo que hace que A detecte la situación y ...

(5) Sección crítica de a $M \leq N$

- Propiedad *SCM* a garantizar:

$\forall \tau. \forall k.$

$$1) \quad \#\{i \mid \tau_k(i) = CRIT\} \leq M$$

$$2) \quad \forall i. \tau_k(i) = TRY \wedge \#\{j \mid \tau_k(j) = CRIT\} < M \\ \implies \exists k' > k. \tau_{k'}(i) = CRIT\}$$

```
1  semaphore sem = M;
2  proc P(i) {
3    // TRY
4    sem.wait();
5    // CRIT
6    sc(i);
7    // EXIT
8    sem.signal();
9  }
```

(6) Lectores/escritores

- Se da mucho en bases de datos.
- Hay una variable compartida.
- Los escritores necesitan acceso exclusivo.
- Pero los lectores pueden leer simultáneamente.
- Propiedad **SWMR** (Single-Writer/Multiple-Readers):

$$\forall \tau. \forall k. \exists i. \text{writer}(i) \wedge \tau_k(i) = \text{CRIT} \implies \forall j \neq i. \tau_k(j) \neq \text{CRIT}$$

$$\forall \tau. \forall k. \exists i. \text{reader}(i) \wedge \tau_k(i) = \text{CRIT} \implies$$

$$\forall j \neq i. \tau_k(j) = \text{CRIT} \implies \text{reader}(j)$$

(implicada en la anterior)

(7) Lectores/escritores (cont.)

- Dos semáforos y un contador

```
semaphore wr = 1;
semaphore rd = 1;
int readers = 0;

proc writer(i) {
    // TRY
    wr.wait();
    // CRIT
    write();
    // EXIT
    wr.signal();
}

proc reader(i) {
    // TRY
    rd.wait();
    readers++;
    if (readers == 1)
        wr.wait();
    rd.signal();
    // CRIT
    read();
    // EXIT
    rd.wait();
    readers--;
    if (readers == 0)
        wr.signal();
    rd.signal();
}
```

- ¿Esta solución es correcta?

(8) Lectores/escritores (cont.)

- Puede haber inanición de escritores.
- ¿Por qué? Puede ser que haya siempre (al menos) un lector.
- Se viola la propiedad de *progreso global dependiente* (*STARVATION-FREEDOM*).
- Esto es, si todo proceso sale de *CRIT* entonces todo proceso que está en *TRY* entra inevitablemente a *CRIT*.

$\forall \tau.$

$$\forall k. \forall i. \tau_k(i) = CRIT \implies \exists k' > k. \tau_{k'}(i) = REM$$

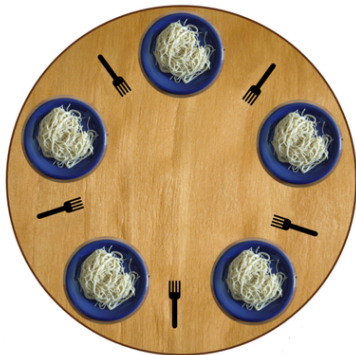
\implies

$$\forall k. \forall i. \tau_k(i) = TRY \implies \exists k' > k. \tau_{k'}(i) = CRIT$$

- Tarea: pensar cómo garantizar *STARVATION-FREEDOM* en este caso.

(9) Filósofos que cenan

- Dining Quintuple/Philosophers, Dijkstra, 1965.
- 5 filósofos en una mesa circular.
- 5 platos de fideos y 1 tenedor entre cada plato.
- Para comer necesitan dos tenedores.



(10) Filósofos que cenan (cont.)

- Código de los filósofos

```
proc Filósofo(i) {  
  while (true) {  
    pensar();           // REM  
    tomar_tenedores(i); // TRY  
    comer();            // CRIT  
    soltar_tenedores(i); // EXIT  
  }  
}
```

- Problema: programar `tomar_tenedores()` y `soltar_tenedores()` satisfaciendo:

EXCL-FORK

Los tenedores son de uso exclusivo.

WAIT-FREEDOM

No haya deadlock.

STARVATION-FREEDOM

No haya inanición.

EAT

Más de un filósofo esté comiendo a la vez (variante de *SCM*).

- Tarea: escribir formalmente las propiedades.

(11) Filósofos que cenar (cont.)

- Un arreglo de N semáforos

```
#define izq(i)  i
#define der(i) ((i + 1) % N)
semaphore tenedores[N] = 1;

void tomar_tenedores(i) {
    tenedores[izq(i)].wait();
    tenedores[der(i)].wait();
}

void soltar_tenedores(i) {
    tenedores[izq(i)].signal();
    tenedores[der(i)].signal();
}
```

- ¿Esta solución es correcta?

(12) Filósofos que cenan (cont.)

- Propiedades

EXCL-FORK

OK.

WAIT-FREEDOM

NOK.

STARVATION-FREEDOM

NOK.

EAT

NOK.

- Resultado general (N. Lynch, Dist. Algorithms, cap. 11)

NO existe ninguna solución en la que todos los filósofos hacen lo mismo (no existe *solución simétrica*).

- Tarea: pensar en soluciones. Buscar las ya existentes.

(13) Barbero

- En una peluquería hay un único peluquero.
- La peluquería tiene dos salas.
 - una de espera, con N sillas,
 - otra donde está la única silla para cortar el pelo.
- Cuando no hay clientes, el peluquero se duerme una siesta.
- Cuando entra un cliente:
 - Si no hay lugar en la sala de espera, se va.
 - Si el peluquero está dormido, lo despierta.
- Ejercicio:
 - Formalizar las propiedades a garantizar.
 - Probar que la solución propuesta las satisface.

(14) Barbero (cont.)

- Vamos a usar dos semáforos y un objeto atómico:

```
semaphore un_cliente = 0;  
semaphore servido = 0;  
semaphore siguiente = 0;  
atomic<int> clientes = 0;
```

- El peluquero es sencillo:

```
proc Peluquero() {  
    while (true) {  
        // TRY  
        un_cliente.wait();  
        siguiente.signal();  
        // CRIT  
        cortar_pelo();  
        // EXIT  
    }  
}
```

- Veamos a los clientes:

```
proc Cliente() {  
    // TRY  
    // ¿Hay lugar?  
    if (clientes.getAndInc() >= N + 1) { // No  
        clientes.getAndDec();  
        return;  
    }  
    // Sí. Avisarle al peluquero  
    un_cliente.signal();  
    // Espero que me dejen pasar.  
    siguiente.wait();  
    // CRIT  
    hacerse_cortar_el_pelo();  
    // EXIT  
    clientes.getAndDec();  
}
```

(16) ¿Hay vida más allá de TAS?

- ¿Podemos lograr *EXCL* sin TAS?
- Analicemos la jerarquía de objetos atómicos (también llamada de registros atómicos).

(17) Registros RW

Registro básico:

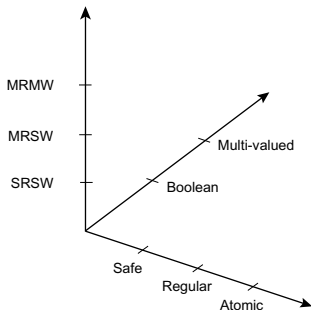
read-write register

Procesos por operación:

single/multiple

Características:

- Si `read()` y `write()` NO se solapan
 `read()` devuelve el **último** valor escrito.
- Si `read()` y `write()` se solapan
 - “Safe”: `read()` devuelve **cualquier** valor.
 - Regular: `read()` devuelve **algún** valor escrito.
 - **Atomic**: `read()` devuelve un valor **consistente** con una serialización.



(18) Registros RW

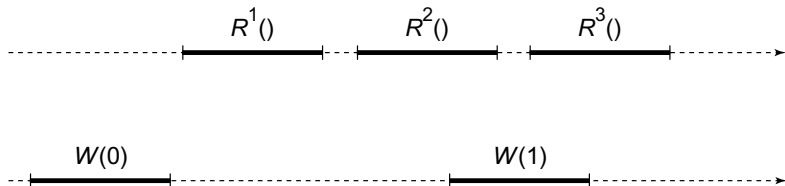


Figure 4.3 A single-reader, single-writer register execution: R^i is the i^{th} read and $W(v)$ is a write of value v . Time flows from left to right. No matter whether the register is *safe*, *regular*, or *atomic*, R^1 must return 0, the most recently written value. If the register is *safe* then because R^2 and R^3 are concurrent with $W(1)$, they can return any value in the range of the register. If the register is *regular*, R^2 and R^3 can each return either 0 or 1. If the register is *atomic* then if R^2 returns 1 then R^3 also returns 1, and if R^2 returns 0 then R^3 could return 0 or 1.

M. Herlihy, N. Shavit. *The Art of Multiprocessor Programming*. 2008.

(19) EXCL con registros RW: Dijkstra

- Registros
 - `flag[i]`: atomic single-writer / multi-reader
 - `turn`: atomic multi-writer / multi-reader

- Proceso i

```
1  /* TRY */
2  L: flag[i] = 1;
3  while (turn  $\neq$  i)
4      if (flag[turn] == 0) turn = i;
5  flag[i] = 2;
6  foreach  $j \neq i$ 
7      if (flag[j] == 2) goto L;
8  /* CRIT */
9  ...
10 /* EXIT */
11 flag[i] = 0;
```

- Garantiza **EXCL**.
- Suponiendo **FAIRNESS**, garantiza **LOCK-FREEDOM**, pero no **WAIT-FREEDOM**.

(20) EXCL con registros RW: Panadería de Lamport

- Registros
 - choosing[i], number[i]: atomic single-writer / multi-reader

- Proceso i

```
1  /* TRY */
2  choosing[i] = 1;
3  number[i] = 1 + maxj≠i number[j];
4  choosing[i] = 0;
5  foreach  $j \neq i$  {
6      waitfor choosing[j]==0;
7      waitfor number[j]==0 ||
8          (number[i], i) < (number[j], j);
9  }
10 /* CRIT */
11 ...
12 /* EXIT */
13 number[i] = 0;
```

- Garantiza **EXCL**, **LOCK-FREEDOM** y **WAIT-FREEDOM**.

(21) Analicemos a los registros RW

- Complejidad
 - Los algoritmos vistos requieren $\mathcal{O}(n)$ registros RW.

- Teorema (Burns & Lynch)

No se puede garantizar *EXCL* y *LOCK-FREEDOM* con menos de n registros RW

- ¿Se puede hacer algo mejor?
 - Sí, pero suponiendo restricciones de tiempo.
 - Algoritmo de Michael Fischer.

(22) Exclusión mutua con registros RW: Fischer

- Registros
 - turn: multi-writer / multi-reader

- Proceso i

```
1  /* TRY */
2  L: waitfor turn = 0;
3  turn = i;  tarda a lo sumo  $\delta$ 
4  pause  $\Delta$ ;
5  if turn  $\neq i$  goto L;
6  /* CRIT */
7  ...
8  /* EXIT */
9  turn = 0;
```

- Garantiza **EXCL.**
- Suponiendo **FAIRNESS**, garantiza **LOCK-FREEDOM** si $\Delta > \delta$.

(23) ¿Qué tan bueno es todo esto?

- ¿Qué tan bueno son los registros RW?
- ¿Qué tan bueno es TAS?
- El problema del consenso.
- Es una formalización de la pregunta “¿pueden n procesos acordar sobre un estado booleano?”

(24) Consenso

- Datos:

Valores $V = \{0, 1\}$

Inicio Todo proceso i empieza con $init(i) \in V$.

Decisión Todo proceso i decide un valor $decide(i) \in V$.

- El problema de consenso requiere:

Acuerdo Para todo $i \neq j$, $decide(i) = decide(j)$.

Validez Existe i , tal que $init(i) = decide(i)$.

Terminación Todo i decide en un número finito de transiciones (**WAIT-FREEDOM**).

- Teorema (Herlihy, Lynch)

No se puede garantizar consenso para un n arbitrario con registros RW atómicos.

Jerarquía de mecanismos de sincronización (Herlihy).


Consensus number:

Cantidad de procesos para los que resuelve consenso

- Registros RW atómicos = 1
- Colas, pilas = 2
- (TAS) `getAndSet()` = 2

¿Existen objetos atómicos con consensus number mayor?

(26) Consenso: CompareAndSwap

- CompareAndSwap/CompareAndSet 

```
1  atomic   T compareAndSwap(T registro, T esperado,
2                                T nuevo) {
3      T actual = registro;
4      if (actual == esperado) registro = nuevo;
5      return actual;
6  }
7
8  atomic   bool compareAndSet(T registro, T esperado,
9                                T nuevo) {
10     if (registro == esperado) {
11         registro = nuevo;
12         return true;
13     }
14     else return false;
15 }
```

- compareAndSwap() en HW: Intel x86 cmpxchg.

(27) Consenso: Compare-and-swap

- `compareAndSwap()` tiene consensus number infinito.

```
atomic<int> decisor = -1;
T[] proposed;

T decide(int i) {

    proposed[i] = init(i);

    if (decisor.compareAndSet(-1, i))
        // Soy el decisor.
        return proposed[i];

    // No soy el decisor, "decido" lo que
    // decidió el decisor.
    return proposed[decisor.get()];
}
```

(28) Dónde estamos

- Vimos:
 - Mecanismos de sincronización (para memoria compartida).
 - Problemas comunes de sincronización.
 - Propiedades comunes.
 - Jerarquía de registros.
 - Consenso.
- Hacer la práctica 3 para ejercitarse en el tema.
- Veremos:
 - Qué pasa cuando no tenemos memoria compartida.
 - Es decir, sincronización sin locks, semáforos ni registros.

(29) Bibliografía extra

- “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>.
- M. Herlihy, N. Shavit. The Art of Multiprocessor Programming. Morgan Kaufmann, 2008.
- M. Abadi, L. Lamport. An Old-Fashioned Recipe for Real Time. ACM TOPLAS 16:5, 1994. <http://goo.gl/t0Uir8>
- M. Herlihy. Impossibility and universality results for wait-free synchronization. ACM PODC, 1988. <http://goo.gl/arpWeP>
- L. Lamport. A new solution of Dijkstra’s concurrent programming problem. CACM 17:8,1974. <http://goo.gl/AZpjw0>
- N. Lynch, N. Shavit. Timing-Based Mutual Exclusion. IEEE 13th RTSS, 1992. <http://goo.gl/M1EtQD>