

Sistemas Distribuidos

Comunicación por memoria compartida

Sergio Yovine

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

Sistemas Operativos, primer cuatrimestre de 2017

(2) Exclusión mutua

- Se puede garantizar exclusión mutua con TAS
- ¿ Es posible hacerlo sin TAS ?

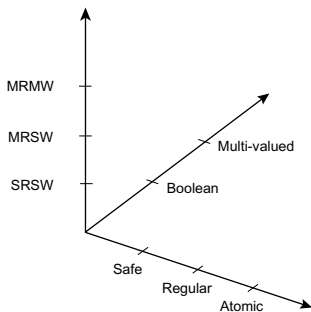
(3) Registros RW

Objeto atómico básico:

read-write register

Procesos por operación:

single/multiple



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

(4) Registros RW

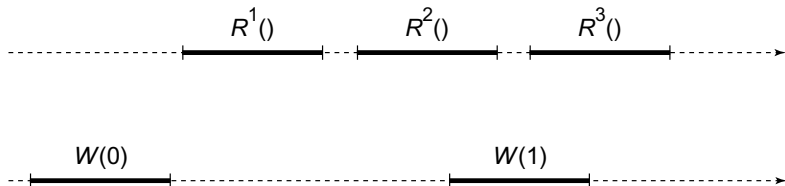


Figure 4.3 A single-reader, single-writer register execution: R^i is the i^{th} read and $W(v)$ is a write of value v . Time flows from left to right. No matter whether the register is *safe*, *regular*, or *atomic*, R^1 must return 0, the most recently written value. If the register is *safe* then because 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.

(5) EXCL con registros RW: Dijkstra

- Registros

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

- Process *i*

```
1  /* Try */
2  L: flag[i] = 1;
3  while (turn ≠ i)
4      if (flag[turn] == 0) turn = i;
5  flag[i] = 2;
6  foreach j ≠ i
7      if (flag[j] == 2) goto L;
8  /* Crit */
9  ...
10 /* Exit */
11 flag[i] = 0;
```

- Garantiza EXCL

- Asumiendo FAIR, garantiza PROG, pero no G-PROG 

(6) EXCL con registros RW: Panadería de Lamport

- Registros

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

- Process 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, PROG y G-PROG 

(7) Exclusión mutua con registros RW: Resumen

- Registros *atomic multi-writer / multi-reader*
 - EXCL y PROG, pero no G-PROG
 - Dijkstra
 - EXCL, PROG y G-PROG
 - Peterson
 - Tournament
- Registros *atomic single-writer / multi-reader*
 - EXCL y PROG, pero no G-PROG
 - Burns
 - EXCL, PROG y G-PROG
 - Lamport (Panadería)
 - Usa contadores (*time-stampping*) no acotados
 - Hay soluciones con contadores acotados

(8) Exclusión mutua con registros RW: Propiedades

Complejidad

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

Teorema (Burns & Lynch)

No se puede garantizar EXCL y PROG con menos de n registros RW


¿Se puede hacer algo mejor?

- Sí, pero asumiendo restricciones de tiempo
- Algoritmo de Michael Fischer

(9) Exclusión mutua con registros RW: Fischer

- Registros
 - turn: multi-writer / multi-reader
- Process 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
- Asumiendo FAIR, garantiza PROG si $\Delta > \delta$ 

(10) Consenso

Descripción

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

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

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

Validez Existe i , tal que $in(i) = decide(i)$

Terminación Todo i decide en un número finito de transiciones
(*wait-free*)

Teorema (Herlihy, Lynch)

No se puede garantizar consenso con registros RW atómicos

Jerarquía de objetos atómicos (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?

(12) Consenso: Compare-and-swap

- Compare-and-swap/set

```
1  atomic T compareAndSwap(T u /* expected */,  
2                          T v /* update */ ) {  
3      T w = register;  
4      if (u == w) register = v;  
5      return w;  
6  }  
7  
8  atomic bool compareAndSet(T u /* expected */,  
9                          T v /* update */ ) {  
10     if (u == register) {  
11         register = v;  
12         return true;  
13     }  
14     else return false;  
15 }
```

- compareAndSwap() en HW: Intel x86 cmpxchg

(13) Consenso: Compare-and-swap

- `compareAndSwap()` tiene consensus number infinito

```
atomic<int> reg = -1;
```

```
T[] proposed;
```

```
T decide(int i) {
```

```
    proposed[i] = in(i);
```

```
    if (r.compareAndSet(-1, i)) return proposed[i];
```

```
    return proposed[r.get()];
```

```
}
```

(14) Bibliografía extra

- 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>