

# Sincronización entre procesos (1/2)

Sergio Yovine

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

Sistemas Operativos, segundo cuatrimestre de 2017

## (2) Interacción entre procesos

Scheduling

(Arbitraje)



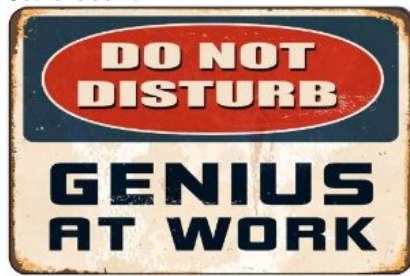
Coordinación



Sincronización



Uso exclusivo



### (3) Esta teórica

- Primera parte
  - Mecanismos para acceder de manera exclusiva a un recurso
- Segunda parte
  - Sincronización
  - Coordinación
- Veamos un ejemplo
  - Fondo de donaciones.
  - Sorteo entre los donantes.
  - Hay que dar números para participar del sorteo.

## (4) Ejemplo: Fondo de donaciones

- Programa en C/Java

```
int ticket = 0;
int fondo = 0;

int donar(int donacion) {
    fondo += donacion; // Actualiza el fondo
    ticket++;           // Incrementa el número de ticket
    return ticket;      // Devuelve el número de ticket
}
```

- En assembler



```
load fondo
add donacion
store fondo
load ticket
add 1
store ticket
return reg
```

## (5) Ejemplo: Fondo de donaciones

- Dos procesos  $P_1$  y  $P_2$  ejecutan el mismo programa
- $P_1$  y  $P_2$  comparten variables `fondo` y `ticket`

$P_1$	$P_2$	$r_1$	$r_2$	fondo	ticket	$ret_1$	$ret_2$
donar(10)	donar(20)			100	5		
load fondo		100		100	5		
add 10		110		100	5		
	load fondo	110	100	100	5		
	add 20	110	120	100	5		
store fondo		110	120	110	5		
	store fondo	110	120	!! 120	0		
	load ticket	110	5	120	5		
	add 1	110	6	120	5		
load ticket		5	6	120	5		
add 1		6	6	120	5		
store ticket		6	6	120	6		
	store ticket	6	6	20	!! 6		
return reg		6	6	120	6	6	
	return reg	6	6	120	6		6

## (6) Ejemplo: Fondo de donaciones ¿Qué pasó?

- Si las ejecuciones hubiesen sido secuenciales, los resultados posibles eran que el fondo terminara con **130** y cada usuario recibiera los tickets **6 y 7** en algún orden.
- Sin embargo, terminamos con un resultado inválido.
- Toda ejecución debería dar un resultado equivalente a **alguna** ejecución **secuencial** de los mismos procesos.   
( Pero, ¿a qué nivel de granularidad?! )
- Lo que ocurrió se llama **condición de carrera** o *race condition*. 
- Porque el resultado que se obtiene varía sustancialmente dependiendo de en qué momento se ejecuten las cosas (o de en qué orden se ejecuten).

## Nasdaq's Facebook Glitch Came From Race Conditions



Joab Jackson

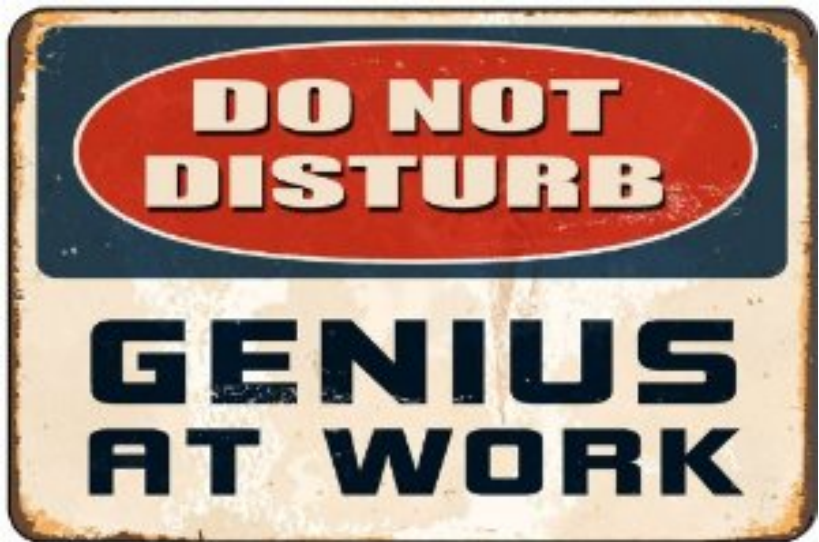
IDG News Service May 21, 2012 12:30 PM

The Nasdaq computer system that delayed trade notices of the Facebook IPO on Friday was plagued by race conditions, the stock exchange announced Monday. As a result of this technical glitch in its Nasdaq OMX system, the market expects to pay out US\$13 million or even more to traders.

Otros ejemplos notorios: MySQL, Apache, Mozilla, OpenOffice.

Shan Lu et al. *Learning from Mistakes - A Comprehensive Study on Real World Concurrency Bug Characteristics*. ASPLOS'08. <http://goo.gl/e1rJ7f>

## (8) Solución: garantizar exclusión mutua






## (9) Volvamos a las donaciones ...

- Se quiere implementar un sitio web para recolectar las donaciones.
- La página tiene un único botón **DONAR**.
- Un campo de texto para ingresar la **donación**
- Un campo de texto para retornar el número de **ticket**
- La operación está completa cuando la página muestra el número de ticket.
- Sólo se puede aceptar una donación a la vez.
- La donación sólo puede completar o fracasar de manera **inmediata**.
- Si fracasa, el donante debe reiterar la donación.

## (10) Exclusión mutua: Test-And-Set (TAS)

- Objeto **atómico** básico *get/test-and-set*
- Operaciones **indivisibles no bloqueantes** (wait-free) 

```
1 private atomic<bool> reg;
2
3 atomic bool get() { return reg; }
4
5 atomic void set(bool b) { reg = b; }
6
7 atomic bool getAndSet(bool b) {
8     bool m = reg;
9     reg = b;
10    return m;
11 }
12
13 atomic bool testAndSet() { // TAS
14     return getAndSet(true);
15 }
```

## (11) Ejemplo: fondo de donaciones

- Implementación usando **TAS**

```
1  private atomic<bool> reg = false;
2
3  void donar(int donacion) {
4      if ( ! reg.testAndSet() ) { // TAS
5          // assert(reg)
6          fondo += donacion;
7          ticket++;
8          // mostrar ticket en pantalla
9          ...
10         // finalizar
11         reg.set(false);
12     }
13 }
```

- Garantiza *exclusión mutua*
- Al menos un donante completa su donación.
- Ningún donante se bloquea (*wait free*)
- Puede ocurrir que un donante **nunca** pueda hacer su donación.

## (13) Y dale con las donaciones ...

- Se quiere implementar un sitio web para recolectar las donaciones.
- La página tiene un único botón **DONAR**.
- Un campo de texto para ingresar la **donación**
- Un campo de texto para retornar el número de **ticket**
- La operación está completa cuando la página muestra el número de ticket.
- Sólo se puede aceptar una donación a la vez.
- Mientras la operación no completa, la página muestra un **spinner**.

## (14) Exclusión mutua: TASLocks

- Spin lock (TASLock)

```
1
2 public class TASLock {
3
4     private atomic<bool> reg;
5
6     public void create() {
7         reg.set(false);
8     }
9
10    public void lock() {
11        while (reg.testAndSet()) {}
12    }
13
14    public void unlock() {
15        reg.set(false);
16    }
17
18 }
```

## (15) Ejemplo: fondo de donaciones con spinner

- Implementación usando **TASLock**

```
1  TASLock l;  
2  void donar(int donacion) {  
3      l.lock();  
4      // assert(reg)  
5      fondo += donacion;  
6      ticket++;  
7      // mostrar ticket en pantalla  
8      ...  
9      // finalizar  
10     l.unlock();
```

- Pregunta: qué pasa si hacemos `assert(!reg)` *después* del `l.unlock()` de la línea 10?

## (16) Propiedades de la solución con spinner

- Garantiza *exclusión mutua*
- Al menos **un** donante *progres*a (y efectúa la donación) (es *lock-free*)
- Puede ocurrir que un donante se bloquee por tiempo indefinido (no es *wait free*)
- Puede ocurrir que un donante **nunca** pueda hacer su donación.



## (17) Exclusión mutua: Notas importantes sobre TASLock

- `testAndSet()` es provisto por el hardware
- Se implementa en *user mode*
- `lock()` introduce espera no acotada y **NO** es atómico.
- **NO** hay que olvidarse de hacer `unlock()`
- Hay espera activa o *busy waiting*

- Poner un `sleep()` en el cuerpo del while ¿de cuánto?

```
void lock_t(time_t delay) {  
    while (reg.testAndSet()) { sleep(delay); }  
}
```

- Evitar iterar sobre `testAndSet()`

## (18) Exclusión mutua: TTASLocks

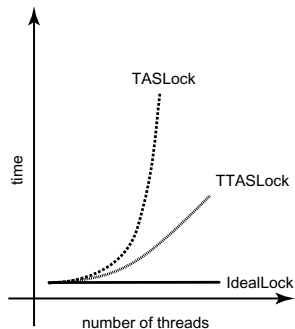
- Spin lock (TTASLock)

```
1 void lock() {  
2     while (true) {  
3         while (reg.get()) {} // espera activa  
4         if ( ! reg.testAndSet() ) return;  
5     }  
6 }
```

Efecto en la memoria *cachée*

- *cache hit* mientras es *true*
- *cache miss* cuando hay *unlock*

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



## (19) Otros objetos atómicos

- Registros Read-Modify-Write atómicos

```
1  atomic int getAndInc() {
2      int tmp = reg;
3      reg++;
4      return tmp;
5  }
6
7  atomic int getAndAdd(int v) {
8      int tmp = reg;
9      reg = reg + v;
10     return tmp;
11 }
12
13 atomic T compareAndSwap(T u, T v) { // CAS
14     T tmp = reg;
15     if (u == tmp) reg = v;
16     return tmp;
17 }
```

## (20) ¿Se puede evitar la espera activa?

- **Sémaforo**

- Una variable entera: `capacidad`  
= cantidad de procesos admisibles en CRIT al mismo tiempo
- Una `fila` de procesos en espera
- Dos operaciones:
  - `wait()` (`P()` o `down()`): Esperar hasta que se pueda entrar.
  - `signal()` (`V()` o `up()`): Salir y dejar entrar a alguno.



E. W. Dijkstra, *Cooperating Sequential Processes*.  
Technical Report EWD-123, Sept. 1965.

<https://goo.gl/PqDzpm>

## (21) Semáforos: esquema de implementación (naive)

- Requiere acceso al **kernel**

```
1 void wait() { // adquirir lock del kernel
2     while(!capacidad) { // ocupado (espera no acotada!!)
3         fila.enqueue(self); // encolarse
4         towaiting(self); // liberar lock y dormir
5         // SIGNALED (recupera lock del kernel)
6     }
7     capacidad--;
8     // liberar el lock del kernel
9 }
10
11 void signal() { // adquirir el lock del kernel
12     capacidad++; // liberar semáforo
13     if(q.dequeue(&p)) { toready(p); // despertarlo }
14     // liberar el lock del kernel
15 }
```

- ¿Son atómicas? (respuesta en la clase de sist. distribuidos)

## (22) Deadlock

- ¿Qué pasa si un proceso no hace `unlock()`?  
Cualquier proceso que haga `lock()` se bloquea para siempre!
- ¿Qué pasa si se ejecuta `f()` en esta situación?


```
1 void f() {  
2     l.lock();  
3     f();  
4     l.unlock();  
5 }
```

El proceso queda bloqueado para siempre!

- ¿Qué pasa con  $P_1$  y  $P_2$  en el siguiente caso?

1 // Proceso 1	1 // Proceso 2
2 l_A.lock();	2 l_B.lock();
3 l_B.lock();	3 l_A.lock();
4 ...	4 ...

Si  $P_1$  y  $P_2$  están ambos en 3, ninguno puede continuar!

- Estas situaciones se llaman **deadlock** 

## (23) Lock reentrante o recursivo

- Esquema de implementación

```
1  int  calls;
2  atomic<int>  owner;
3
4  void  create() { owner.set(-1); calls = 0; }
5
6  void  lock() {
7      if (owner.get() != self) {
8          while (owner.compareAndSwap(-1, self) != self) {}
9      }
10     // owner == self
11     calls++;
12 }
13
14 void  unlock() { } if (--calls == 0) owner.set(-1); }
```

- Ejercicio: hacerlo con local spinning y semáforos

## (24) Errores de sincronización

- Race condition (condición de carrera):

El resultado no corresponde a ninguna *secuencialización*

- Deadlock (bloqueo para siempre):

Uno o más procesos quedan bloqueados para siempre

- Starvation (innanición):

Un proceso espera un tiempo no acotado para adquirir un lock porque otro(s) proceso(s) le gana(n) de mano

- Convoying (demora en cascada):

Un proceso que detiene un lock es sacado de running antes de liberar el lock



## (25) Errores de sincronización: ¿Qué hacer?

- Prevención
  - Patrones de diseño
  - Reglas de programación
  - Prioridades
  - Protocolos (e.g., Priority Inheritance)
- Detección
  - Análisis de programas
    - Análisis estático
    - Análisis dinámico
  - En tiempo de ejecución
    - Preventivo (antes que ocurra)
    - Recuperación (deadlock recovery)

## (26) Bibliografía adicional

- Hoare, C. *Monitors: an operating system structuring concept*, Comm. ACM 17 (10): 549-557, 1974. <http://goo.gl/eVaeao>
- M. Herlihy, N. Shavit. *The Art of Multiprocessor Programming*. Morgan Kaufmann, 2008.
- N. Lynch. *Distributed Algorithms*. Morgan Kaufmann, 1996.
- Ch. Kloukinas, S. Yovine. *A model-based approach for multiple QoS in scheduling: from models to implementation*. Autom. Softw. Eng. 18(1): 5-38 (2011). <https://goo.gl/5FuU6x>
- M. C. Rinard. *Analysis of Multithreaded Programs*. SAS 2001: 1-19 <http://goo.gl/pyfg0G>
- L. Sha, R. Rajkumar, J. P. Lehoczky. *Priority Inheritance Protocols: An Approach to Real-Time Synchronization*. IEEE Transactions on Computers, September 1990, pp. 1175-1185. <http://goo.gl/0Qeujs>
- Valgrind tool. <http://valgrind.org/>
- Java Pathfinder (JPF). <http://babelfish.arc.nasa.gov/trac/jpf>