

# Sincronización entre procesos (1/2)

Sergio Yovine

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

Sistemas Operativos, primer cuatrimestre de 2017

## (2) Interacción entre procesos

Scheduling

(Arbitraje)



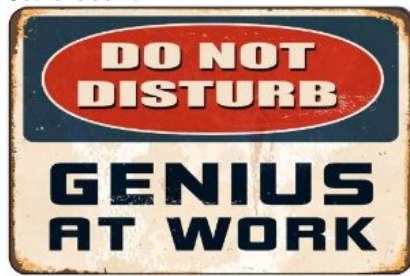
Coordinación



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 <sub>1</sub>	ret <sub>2</sub>
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

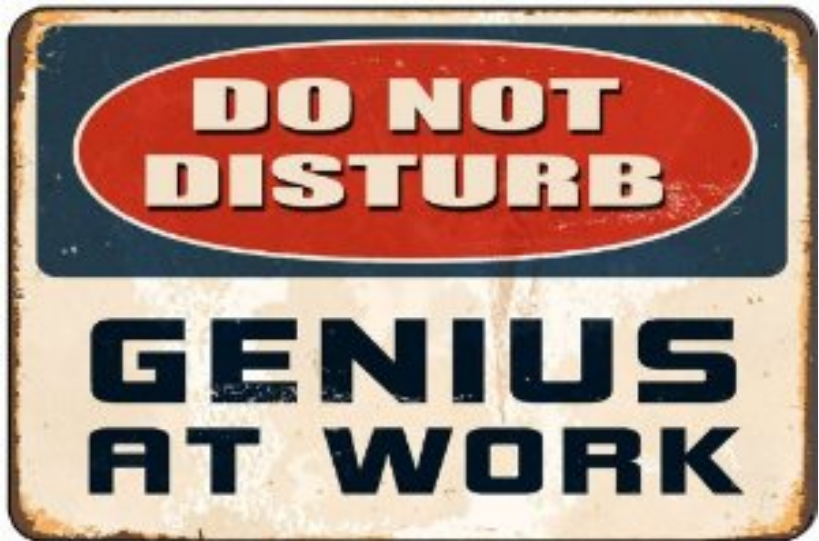
IDG News Service May 21, 2012 12:30 PM

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) Problema: garantizar exclusión mutua

- Solución posible: **sección crítica** (*CRIT*) 
  - Entrar es como poner el cartelito de no molestar en la puerta
  - Salir es sacar el cartelito
- ¿Dónde?
  - La sección crítica es toda la función  $\implies$  menor concurrencia

```
1 criticalsection int donar(int donacion) { ... }
```
  - Una sección crítica es un bloque  $\implies$  mayor concurrencia

```
1 int donar(int donacion) {
2     int tmp;
3     criticalsection { fondo += donacion; }
4     criticalsection { tmp = ++ticket; }
5     return tmp;
6 }
```

## (10) Exclusión mutua: ¿Cómo se implementa?

- Propiedades

- 1 Sólo hay un proceso a la vez en *CRIT*
- 2 Todo proceso que esté esperando entrar a *CRIT* va a entrar
- 3 Ningún proceso fuera de *CRIT* puede bloquear a otro


- Dos llamados:

- uno para entrar
- otro para salir

```
1  begincriticalsection()  
2  fondo += donacion;  
3  endcriticalsection()
```

- La pregunta es: ¿cómo se implementan?

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

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

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

## (12) Exclusión mutua: TASLocks

- Spin lock (TASLock)

```
1  atomic<bool> reg;  
2  
3  void create() {  
4      reg.set(false);  
5  }  
6  
7  void lock() {  
8      while (reg.testAndSet()) {}  
9  }  
10  
11 void unlock() {  
12     reg.set(false);  
13 }
```

## (13) Exclusión mutua: TASLocks

- Ejemplo de uso

```
1  TASLock l;  
2  int donar(int donacion) {  
3      int tmp;  
4      // Inicio de la seccion critica  
5      l.lock();  
6      fondo += donacion;  
7      l.unlock();  
8      // Fin de la seccion critica  
9      // Inicio de la seccion critica  
10     l.lock();  
11     tmp = ++ticket;  
12     l.unlock();  
13     // Fin de la seccion critica  
14     return tmp;  
15 }
```

## (14) 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()`

# (15) Exclusión mutua: TTASLocks

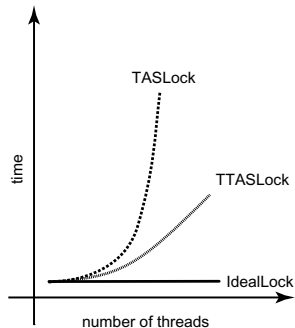
- Spin lock (TTASLock)

```
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.



## (16) 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 }
```



## (17) Volvamos al fondo de donaciones

- Usando los objetos atómicos anteriores ...

```
1  atomic<int> fondo;  
2  atomic<int> ticket;  
3  
4  fondo.set(0);  
5  ticket.set(0);  
6  
7  int donar(int donacion) {  
8      fondo.getAndAdd(donacion);  
9      return 1 + ticket.getAndInc();  
10 }
```

- No hay busy waiting
- Hay más concurrencia
- Esta solución es *wait-free* (lo veremos más en detalle después)

## (18) ¿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>

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

- Requiere acceso al **kernel**

```
1 void wait() { // adquirir el lock del kernel
2     while(!capacidad) { // ocupado (cuidado: STARVATION)
3         fila.enqueue(self); // encolarse
4         towaiting(self); // liberar el lock del kernel y do
5         // SIGNALED (recupera el 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)

## (20) Productor-Consumidor con semáforos

- Código común

```
1  bag<T> buffer; // requiere exclusión mutua interna
2  semaphore filled = 0; // cantidad de items en buffer
3  semaphore empty = N; // lugares libres en el buffer
```

- Productor y consumidor

```
1  void productor() {
2      while (true) {
3          T item = produce();
4          // ¿Hay lugar?
5          empty.wait();
6          // Agregar al buffer
7          buffer.put(item);
8          // Avisar que hay algo
9          filled.signal();
10     }
11 }
```

```
1  void consumidor() {
2      while (true) {
3          // ¿Hay algo?
4          filled.wait();
5          // Sacar del buffer
6          buffer.get(&item);
7          // Avisar que hay lugar
8          empty.signal();
9          consumir(item);
10     }
11 }
```

## (21) 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** 

## (22) Lock reentrante o recursivo


- Esquema de implementación

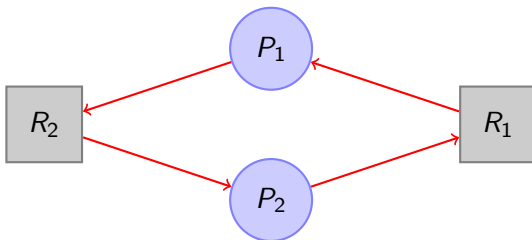
```
1  int  calls;
2  atomic<int>  owner;
3
4  void  create() { owner.set(-1); calls = 0; }
5
6  void  lock() {
7      if (owner.get() != self) {
8          while (owner.compareAndSwap(-1, self) != self) {}
9      }
10     // owner == self
11     calls++;
12 }
13
14 void  unlock() { } if (--calls ==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

- Ejercicio: hacerlo con local spinning y semáforos

## (23) Deadlock: Modelo

- Grafos bipartito
  - Nodos: procesos  $P$  y recursos  $R$ .
  - Arcos:
    - De  $P$  a  $R$  si  $P$  *solicita*  $R$
    - De  $R$  a  $P$  si  $P$  *adquirió*  $R$

- Deadlock = ciclo 



## (24) Deadlock: Condiciones necesarias (o de Coffman)

- Coffman et al. 1971.  <http://goo.gl/qW05ft>

Exclusión mutua :

Un recurso no puede estar asignado a más de un proceso.

Hold and wait :

Los procesos que ya tienen algún recurso pueden solicitar otro.

No preemption :

No hay mecanismo compulsivo para quitarle los recursos a un proceso.

Espera circular :

Tiene que haber un ciclo de  $N \geq 2$  procesos, tal que  $P_i$  espera un recurso que tiene  $P_{i+1}$ .



## (25) Problemas 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

- Conveying (demora en cascada):

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

## (26) Problemas 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)

## (27) Dónde estamos

- Vimos
  - Condiciones de carrera.
  - Secciones críticas.
  - TestAndSet.
  - Busy waiting / sleep.
  - Productor - Consumidor.
  - Semáforos.
  - Deadlock.
  - Monitores. (Estudiar de la bibliografía)
  - Variables de condición. (Estudiar de la bibliografía)
- Práctica: Ejercicios de sincronización.
- Próxima teórica: Problemas comunes de sincronización.
- En la teórica de sistemas distribuidos veremos
  - Sincronización sin locks y semáforos
  - Más sobre objetos atómicos y sus propiedades

## (28) 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>