

Sincronización entre procesos 1: Motivación y herramientas

Rodolfo Baader

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

Sistemas Operativos, segundo cuatrimestre de 2024

(2) Créditos

Basado fuertemente en trabajo de Sergio Yovine.

(3) Muchas manos en un plato...

- “The more the merrier”
- Pero cuando somos muchos hay que ser cuidadosos, porque como dijo Fontanarrosa:
- “Si la que crees tu pierna no es tu pierna, no es orgía, es desorden”
- Vamos a preocuparnos por hacer que los procesos puedan cooperar sin estorbarse (o haciéndolo lo menos posible...).

(4) ¿Por qué vemos esto?

- Contención y concurrencia son dos problemas fundamentales en un mundo donde cada vez más se tiende a la programación distribuida y/o paralela.
- Eg, clusters, grids, multicores.
- Pero además, es importantísimo para los SO: muchas estructuras compartidas, mucha contención.
- ¿Recuerdan lo que dijimos al comienzo?
- Los SO tienen que manejar la contención y la concurrencia de manera tal de lograr:
 - Hacerlo correctamente.
 - Hacerlo con buen rendimiento.

(5) Veamos un ejemplo

- Fondo de donaciones. Sorteo entre los donantes. Hay que dar números.
- Dos procesos, tienen que incrementar el número de ticket y manejar el fondo acumulado.

(6) 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
```

(7) Veamos un scheduling posible

- 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	5		
	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	120	!! 6		
return reg		6	6	120	6	6	
	return reg	6	6	120	6		6

(8) ¿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. 
- 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



COMMENTS

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>

(10) Una solución

- Una forma posible para solucionarlo es logrando la *exclusión mutua* mediante **secciones críticas** (alias *CRIT*).

(11) ¿Qué es una sección crítica?

- Un cacho de código tal que: 
 - 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.
- A nivel de código se implementaría con dos llamados: uno para entrar y otro para salir de la sección crítica.
- Entrar a la sección crítica es como poner el cartelito de no molestar en la puerta...
- Si logramos implementar exitosamente secciones críticas, contamos con herramientas para que varios procesos puedan compartir datos sin estorbarse.
- La pregunta es: ¿cómo se implementa una sección crítica?

(12) Implementando secciones críticas

- Se soluciona revoleando booleanos?
- Una alternativa podría ser la de suspender todo tipo de interrupciones adentro de la sección crítica. Esto elimina temporalmente la multiprogramación. Aunque garantiza la correcta actualización de los datos compartidos trae todo tipo de problemas.
- Otra alternativa es usar *locks*: variables booleanas, compartidas también. Cuando quiero entrar a la sección crítica la pongo en 1, al salir, en 0. Si ya está en 1, espero hasta que tenga un 0.
- Buenísimo... Excepto porque tampoco funciona así como está.
- Podría suceder que cuando veo un 0 se me acaba el quantum y para cuando me toca de nuevo ya se metió otro proceso sin que me dé cuenta.
- La solución más general consiste en obtener un poquito de ayuda del HW.

(13) TAS

- El HW suele proveer una instrucción que permite establecer atómicamente el valor de una variable entera en 1.
- Esta instrucción se suele llamar *TestAndSet*. ⚠
- La idea es que pone 1 y devuelve el valor anterior, pero de manera *atómica*.
- Que una operación sea atómica significa que es indivisible, incluso si tenemos varias CPUs. ⚠
- Veamos como se usaría.

(14) TAS

```
1  /* RECORDAR: esto es pseudocódigo. TAS suele ser una
2   * instrucción assembler y se ejecuta de manera atómica.
3   */
4  bool TestAndSet(bool *destino)
5  {
6      bool resultado = *destino;
7      *destino = TRUE;
8      return resultado;
9  }
```

(15) Usando TAS para obtener locks

```
1  boolean lock; // Compartida.
2
3  void main(void)
4  {
5      while (TRUE)
6      {
7          while (TestAndSet(&lock))
8              // Si da true es que ya estaba lockeado.
9              // No hago nada.
10             ;
11
12             // Estoy en la sección crítica,
13             // hago lo que haga falta.
14
15             // Salgo de la sección crítica.
16             lock = FALSE;
17
18             // Si hay algo no crítico lo puedo hacer acá.
19         }
20     }
```

(16) Algo sobre TAS

- Notemos el while interno.
- Dado que el cuerpo del while está vacío, el código se la pasa intentando obtener un lock.
- Es decir, consume muchísima CPU.
- Eso se llama *espera activa* o *busy waiting*. ⚠
- Hay que ser muy cuidadoso. Es una forma muy agresiva (¡y costosa!) de intentar obtener un recurso.
- Perjudica al resto de los procesos, muchas veces sin razón.
- Una forma de verlo es que dichos procesos no reciben muchos mensajes el día del amigo... ⚠
- Solución: tomarlo con calma.
- Solución?

(17) Sleep. ¿Sleep?

- Ponemos el sleep.
- ¿Pero cuánto?
 - Si es mucho, perdemos tiempo.
 - Si es poco, igual desperdiciamos CPU (aunque mucho menos que antes).
- ¿No estaría bueno poder decirle al SO que queremos continuar sólo cuando `lock==0`?
- Por suerte, una vez más, viene Súper Dijkstra al rescate.

(18) Dijkstra



(19) Productor-Consumidor

- Ambos comparten un buffer de tamaño limitado más algunos índices para saber dónde se colocó el último elemento, si hay alguno, etc. A este problema a veces se lo conoce como *bounded buffer* (*buffer acotado*).
- Productor pone elementos en el buffer.
- Consumidor los saca.
- De nuevo tenemos un problema de concurrencia. Ambos quieren actualizar las mismas variables.
- Pero en este caso hay un problema adicional.
- Si Productor quiere poner algo cuando el buffer está lleno o Consumidor quiere sacar algo cuando el buffer está vacío, deben esperar.
- ¿Pero cuánto?
- Podríamos hacer busy waiting. Pero podemos perder mucho tiempo.
- Podríamos usar *sleep()-wakeup()*. ¿Podríamos?

(20) Productor-Consumidor (cont.)

- Pensemos en el consumidor: `if (cant==0) sleep();`
- Y el productor: `agregar(item, buffer); cant++;`
`wakeup();`
- Miremos un posible interleaving:

Consumidor	Productor	Variables
		<code>cant==0, buffer==[]</code>
	<code>agregar(i1, buffer)</code>	<code>cant==0, buffer==[i1]</code>
<code>¿ cant==0 ?</code>		
	<code>cant++</code> <code>wakeup()</code>	<code>cant==1, buffer==[i1]</code>
<code>sleep()</code>		

Resultado: el `wakeup()` se pierde, el sistema se traba...

A este problema se lo conoce como el *lost wakeup problem*.

(21) Semáforos: definición



- Dijkstra inventó los *semáforos* 



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

<https://goo.gl/PqDzpm>

(22) Semáforos

- Para solucionar este tipo de problemas Dijkstra inventó los *semáforos*. 
- Un semáforo es una variable entera con las siguientes características:
 - Se la puede inicializar en cualquier valor.
 - Sólo se la puede manipular mediante dos operaciones:
 - `wait()` (también llamada `P()` o `down()`).
 - `signal()` (también llamada `V()` o `up()`).
 - `wait(s): while (s<=0) dormir(); s- -;`
 - `signal(s): s++; if (alguien espera por s) despertar a alguno;`
 - Ambas se implementan de manera tal que ejecuten sin interrupciones.
- Un tipo especial de semáforo que tiene dominio binario se llama *mutex*, de *mutual exclusion*. 

(23) Productor-Consumidor con semáforos.


Código común:

```
semáforo mutex = 1;
semáforo llenos = 0;
semáforo vacios = N; // Capacidad del buffer.
```

```
1 void productor() {
2     while (true) {
3         item = producir_item();
4         wait(vacios);
5         // Hay lugar. Ahora
6         // necesito acceso
7         // exclusivo.
8         wait(mutex);
9         agregar(item, buffer);
10        cant++;
11        // Listo, que sigan
12        // los demás.
13        signal(mutex);
14        signal(llenos);
15    }
16 }
```

```
void consumidor() {
    while (true) {
        wait(llenos);
        // Hay algo. Ahora
        // necesito acceso
        // exclusivo.
        wait(mutex);
        item = sacar(buffer);
        cant--;
        // Listo, que sigan
        // los demás.
        signal(mutex);
        signal(vacios);
        hacer_algo(item);
    }
}
```

(24) Puede fallar...

- Volvamos a nuestra solución basada en semáforos.
- ¿Qué pasa si me olvido un signal o invierto el orden?
- El sistema se traba, porque el proceso A se queda esperando que suceda algo que sólo B puede provocar. Pero B a su vez está esperando algo de A.
- (Ella: no lo voy a llamar hasta que él no me llame. Él: si ella no me llama yo no la llamo. ¿Suenan familiares?)
- Esta situación se llama *deadlock*. 
- Y es una de las plagas de la concurrencia. No solamente de los SO.
- En algunos libros figura como *interbloqueo* o *abrazo mortal*. Traduttore, traditore.

(25) Deadlock



(26) De vuelta al siglo XXI

- Hoy en día los lenguajes de alto nivel proveen distintas alternativas para implementar secciones críticas.
 - bool atómicos.
 - ints atómicos.
 - colas atómicas.

(27) Exclusión mutua: TAS con objetos


- Objeto (o registro) **atómico** básico.
- Provee `getAndSet()` y `testAndSet()`.
- Implementa operaciones **indivisibles** (a nivel de hardware).

```
1 private 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() {
14     return getAndSet(true);
15 }
```

(28) Spin locks

- En base a los bool atómicos con `testAndSet()` puedo construir un mutex llamado TASLock.
- También conocido como *spin lock*.

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

- Cuidado, `lock()` no es atómico. 

(29) Spin lock (cont.)

- Ejemplo de uso:

```
1  TASLock mtx;
2  int donar(int donacion) {
3      int res;
4      // Inicio de la sección crítica.
5      mtx.lock();
6      fondo += donacion;
7      mtx.unlock();
8      // Fin de la sección crítica.
9
10     // Inicio de otra sección crítica.
11     mtx.lock();
12     res = ticket; ticket++;
13     mtx.unlock();
14     // Fin de la sección crítica.
15
16     return res;
17 }
```

(30) Inconvenientes de TASLock

- Ya lo dijimos, pero...
- No hay que olvidarse de hacer `unlock()`. ⚠
- Produce espera activa o *busy waiting*. ⚠
- Sin embargo, su overhead puede ser menor que el de usar semáforos. ⚠
- Una forma de minimizar el impacto: testear antes de testear y setear (TTASLock).

(31) TTASLock

- Idea: probar antes de mandarse con el TAS().
- También llamada *local spinning*.

```
1 void create() { mtx.set(false); }
2
3 void lock() {
4     while (true) {
5         while (mtx.get()) {}
6         if (!mtx.testAndSet()) return;
7     }
8 }
9
10 void unlock() { mutex.set(false); }
```

- *Local spinning* es más eficiente
 - El while hace `get()` en lugar de `testAndSet()`.
 - Lee la memoria *cache* mientras sea verdadero (*cache hit*).
 - Cuando un proceso hace `unlock()` hay *cache miss*.
 - Igual es caro...

(32) TASLock vs TTASLock

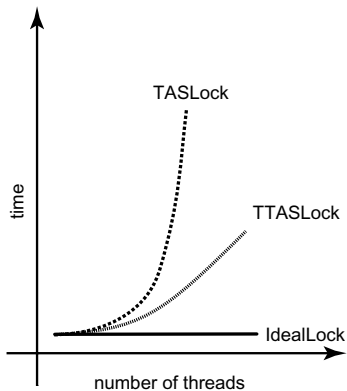


Figure 7.4 Schematic performance of a TASLock, a TTASLock, and an ideal lock with no overhead.

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

(33) Otros objetos atómicos

- Registros Read-Modify-Write atómicos:

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

(34) Otros objetos atómicos

- Cola

```
1  atomic enqueue(T item) {
2      mtx.lock();
3      q.push(item);
4      mtx.unlock();
5  }
6
7  atomic bool dequeue(T *pitem) {
8      bool success;
9      mtx.lock();
10     if (q.empty()) {
11         pitem = null;  success = false;
12     }
13     else {
14         pitem = q.pop(); success = true;
15     }
16     mtx.unlock();
17     return success;
18 }
```

(35) 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).

(36) Volvamos al lock

- ¿Qué pasa si un proceso usa **TASlock** recursivamente?

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

El proceso queda *bloqueado*: **deadlock** ⚠

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

```
1 // Proceso 1  
2 mtxA.lock();  
3 mtxB.lock();  
4 ...
```

```
1 // Proceso 2  
2 mtxB.lock();  
3 mtxA.lock();  
4 ...
```

Si P_1 y P_2 están ambos en 3, no pueden seguir: **deadlock** ⚠

(37) Mutex 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     // ower == self
11     calls++;
12 }
13
14 void unlock() {
15     if (--calls == 0) owner.set(-1);
16 }
```

- Ejercicio: hacerlo con local spinning

(38) Problema: garantizar exclusión mutua

- ¿Dónde?

- La sección crítica es toda la función \Rightarrow menor concurrencia

```
1  criticalsection int donar(int donacion) { ... }
```

- Una sección crítica es un bloque \Rightarrow mayor concurrencia

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

(39) Condiciones de Coffman

- Coffman et al. 1971. <http://goo.gl/qW05ft>
- Postula una serie de condiciones necesarias para la existencia de un deadlock.
- El modelo tiene algunas falencias.

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.

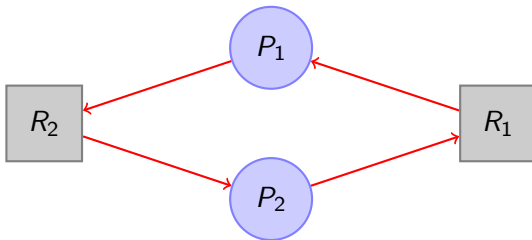
Espera circular

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

(40) Modelo de grafos bipartitos

- Grafos bipartito para detectar deadlock.
 - 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 ⚠



(41) Problemas de sincronización: ¿qué hacer?

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

(42) 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 y cómo razonar sobre nuestras soluciones.

(43) Bibliografía adicional

- Hoare, C. *Monitors: an operating system structuring concept*, Comm. ACM 17 (10): 549-557, 1974. <http://goo.gl/eVaeao>
- Allen B. Downey. *The Little Book of Semaphores*.
<https://greenteapress.com/wp/semaphores/>
- Edgar W. Dijkstra. *Cooperating sequential processes*.
<https://goo.gl/PqDzpm>.
- 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>