Sincronización entre procesos (1/2)

Sergio Yovine

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

Sistemas Operativos, segundo cuatrimestre de 2016

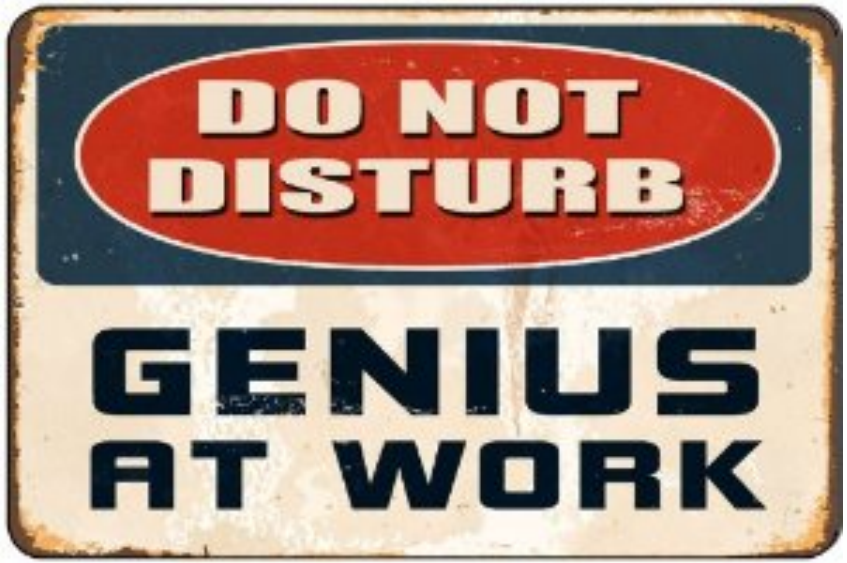
(2) Interacción entre procesos

Scheduling (Arbitraje)

Sincronización

Coordinación

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. 
- 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).

(7) Condición de carrera: No es un problema menor ...

Nasdaq's Facebook Glitch Came From Race Conditions

COMMENTS

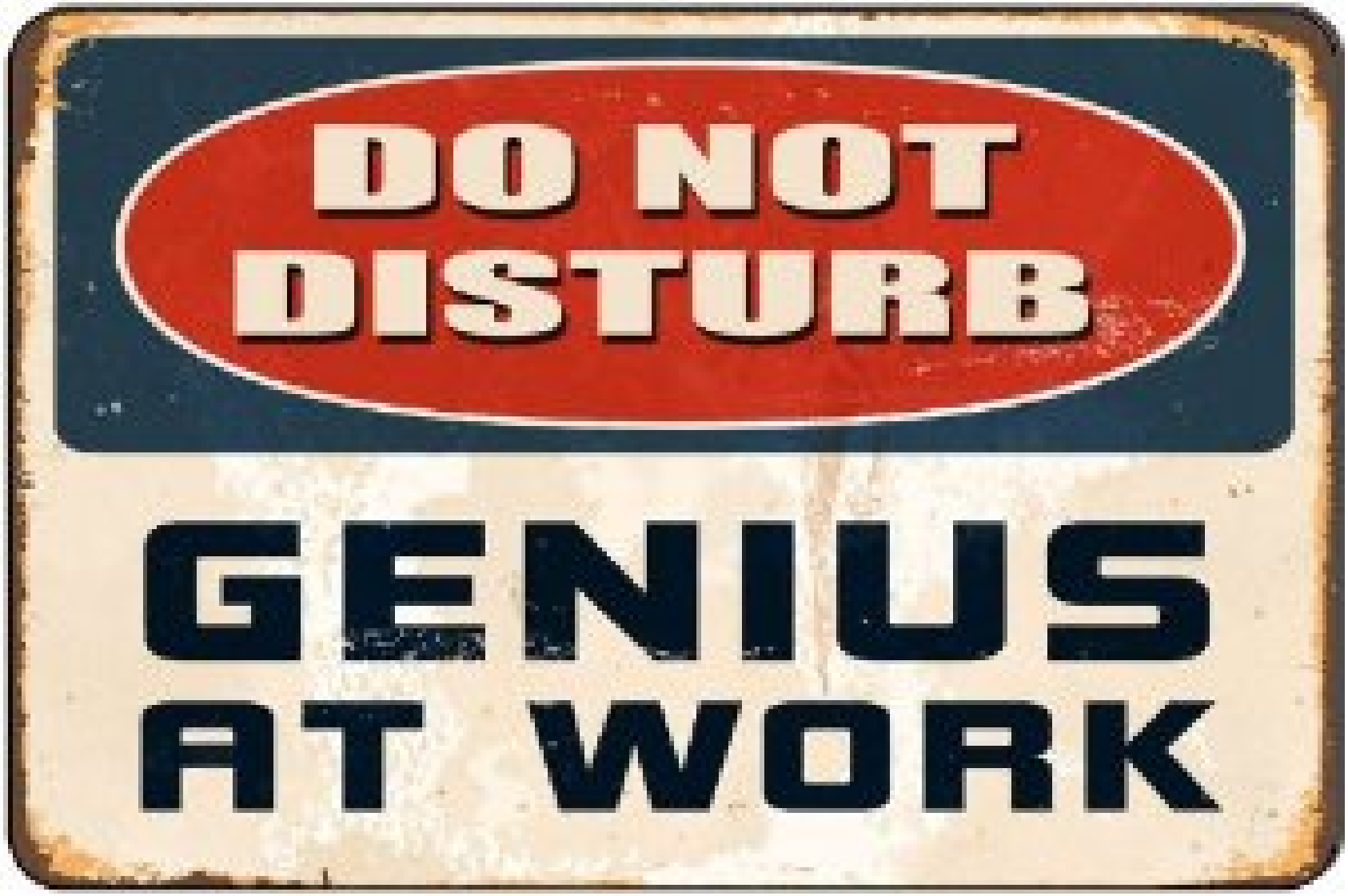
Joab Jackson
IDG News Service May 21, 2012 12:30 PM

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

(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 $\Longrightarrow$ menor concurrencia

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

  - Una sección crítica es un bloque $\Longrightarrow$ 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 (o registro) atómico básico *get/test-and-set*
- 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 boolean getAndSet(bool b) {
8    bool m = reg;
9    reg = b;
10   return m;
11 }
12
13 atomic boolean testAndSet() {
14   return getAndSet(true);
15 }
```

(12) Exclusión mutua: Locks (o `mutex`)

- Spin lock (TASLock)

```
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 

(13) Exclusión mutua: Locks (o mutex)

- Ejemplo de uso

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

(14) Exclusión mutua: Inconvenientes de TASLock

- No hay que olvidarse de hacer unlock 
- Espera activa o *busy waiting* 
  - En el `while` el proceso se la pasa intentando obtener el lock.
  - Consume CPU.
  - Más importante: los procesos se *estorban* mutuamente.
  - Esos procesos jamás llegarán a empleados del mes...

(15) Busy waiting

- ...pero sus programadores sí...

- Ex-programador que hacía busy waiting.
- Le conseguí el puesto yo (DFS) mismo. ⚠
- No hagan busy waiting... si lo pueden evitar.

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

(16) Exclusión mutua: TTASLocks

- Spin lock (TTASLock)

```
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 *cachée* mientras sea verdadero (*cache hit*)
  - Cuando un proceso hace `unlock` hay *cache miss*
  - Igual es caro ...

(17) Exclusión mutua: TASLock vs TTASLock

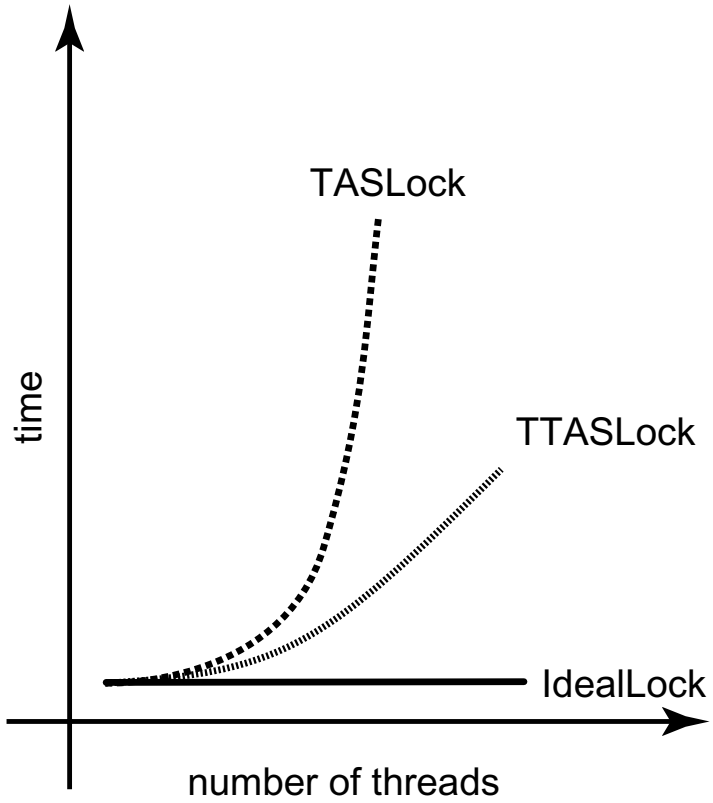


Figure 7.4 Schematic performance of a TASLock, a TTASLock, and an ideal lock with no overhead.

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

(18) 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) {
14   T tmp = reg;
15   if (u == tmp) reg = v;
16   return tmp;
17 }
```

(19) Otros objetos atómicos

- Fila

```
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 }
```

(20) 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)

(21) Sleep. ¿Sleep?

- Ponemos el sleep. ¿Pero cuánto?
  - Si es mucho, hay riesgo de perder tiempo durmiendo cuando el recurso compartido está disponible.
  - Si es poco, hay riesgo de desperdiciar CPU intentando entrar a la sección crítica cuando está ocupada
- Solución: despertarse *apenas* se pueda entrar 
  - ¿Cómo? notificando al que espera cuando se sale de *CRIT*
  - ¿A quién? a cualquiera o a todos

(22) Semáforos: definición

- Dijkstra inventó los *semáforos* ⚠

E. W. Dijkstra, *Cooperating Sequential Processes*.
Technical Report EWD-123, Sept. 1965.
`https://goo.gl/PqDzpm`

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

(23) Semáforos: implementación (naive)

- Objetos atómicos

```
atomic int getAndDec() {
  int tmp = cap;
  if (cap>0) --cap;
  return tmp;
}
```

```
// Capacidad del semáforo
atomic<int> cap;
cap.set(N);
// Fila de proc. en espera
atomic<queue<int>> q;
```

- Esquema de implementación (naive)

```
1  void wait() {
2    // ¿Se puede entrar?
3    if(cap.getAndDec()<=0) {
4      // semáforo ocupado
5      q.enqueue(self);
6      // dormir: WAITING
7      towaiting();
8      // SIGNALED
9    }
10   // semáforo adquirido: CRIT
11 }
```

```
1  void signal() {
2    // liberar semáforo
3    cap.getAndInc();
4
5    // ¿Alguien esperando?
6    if(q.dequeue(&next)) {
7      // despertarlo
8      toready(next);
9    }
10
11 }
```

(24) Semáforos: implementación (naive)

- Problema: no garantiza exclusión mutua
- Podría haber más procesos en el semáforo que su capacidad
- Ejemplo de ejecución posible

Estado	`cap`	`q`	P_1	P_2
A	1	$\{\}$	`wait`	
B	0	$\{\}$	CRIT	`wait`
C	0	$\{P_2\}$	CRIT	WAITING
D	0	$\{P_2\}$	`signal`	WAITING
E	1	$\{\}$		SIGNALED
F	1	$\{\}$	`wait`	SIGNALED
G	0	$\{\}$	CRIT	SIGNALED
H	0	$\{\}$	CRIT	CRIT !!

(25) Semáforos: implementación (naive)

- Solución: cambiar `if` por `while` en la línea **??** de `wait()`
- El proceso en SIGNALED vuelve a preguntar si puede entrar

```
1  void wait() {
2    // ¿Se puede entrar?
3    while(cap.getAndDec()<=0) {
4      // semáforo ocupado
5      q.enqueue(self);
6      // dormir: WAITING
7      towaiting();
8      // SIGNALED
9    }
10   // semáforo adquirido: CRIT
11 }
```

```
1  void signal() {
2    // liberar semáforo
3    cap.getAndInc();
4
5    // ¿Alguien esperando?
6    if(q.dequeue(&next)) {
7      // despertarlo
8      toready(next);
9    }
10
11 }
```

- ¿Es correcta?

(26) Semáforos: implementación (naive)

- Problema: inanición (*starvation*)
- Un proceso podría quedarse esperando indefinidamente
- Ejemplo de ejecución posible

Estado	`cap`	`q`	P_1	P_2
A	1	$\{\}$	`wait`	
B	0	$\{\}$	CRIT	`wait`
C	0	$\{P_2\}$	CRIT	WAITING
D	0	$\{P_2\}$	`signal`	WAITING
E	1	$\{\}$		SIGNALED
F	1	$\{\}$	`wait`	SIGNALED
G	0	$\{\}$	CRIT	SIGNALED
C	0	$\{P_2\}$	CRIT	WAITING ciclo

(27) Semáforos: implementación (sin inanición)

- Solución:
  - Preguntar si hay alguien esperando antes de liberar el semáforo en `signal()`
  - Cambiar el `while` a `if` de nuevo en la línea **??** de `wait()`

```
1  void wait() {
2    // ¿Se puede entrar?
3    if(cap.getAndDec()<=0) {
4      // semáforo ocupado
5      q.enqueue(self);
6      // dormir: WAITING
7      towaiting();
8      // SIGNALED
9    }
10   // semáforo adquirido: CRIT
11 }
```

```
1  void signal() {
2    // ¿Alguien esperando?
3    if(q.dequeue(&next)) {
4     // despertarlo
5     toready(next);
6    }
7    else {
8      // liberar semáforo
9      cap.getAndInc();
10   }
11 }
```

(28) Semáforos: implementación (¿correcta?)

- P_1 no puede volver a entrar antes que P_2

Estado	`cap`	`q`	P_1	P_2
A	1	$\{\}$	`wait`	
B	0	$\{\}$	CRIT	`wait`
C	0	$\{P_2\}$	CRIT	WAITING
D	0	$\{P_2\}$	`signal`	WAITING
E	0	$\{\}$		SIGNALED
F	0	$\{\}$	`wait`	SIGNALED
G	0	$\{P_1\}$	WAITING	SIGNALED
H	0	$\{P_1\}$	WAITING	CRIT

- Por supuesto, esto no *prueba* que:
  - No tiene inanición.
  - Garantiza exclusión mutua.
- Ejercicio: probarlo.
- En la segunda parte se dan más fundamentos para esto.

(29) Productor-Consumidor con semáforos

- Código común

```
1  atomic<queue<T>> buffer;
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.enqueue(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.dequeue(&item);
7      // Avisar que hay lugar
8      empty.signal();
9      consumir(item);
10   }
11 }
```

(30) 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 

(31) 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

(32) 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.

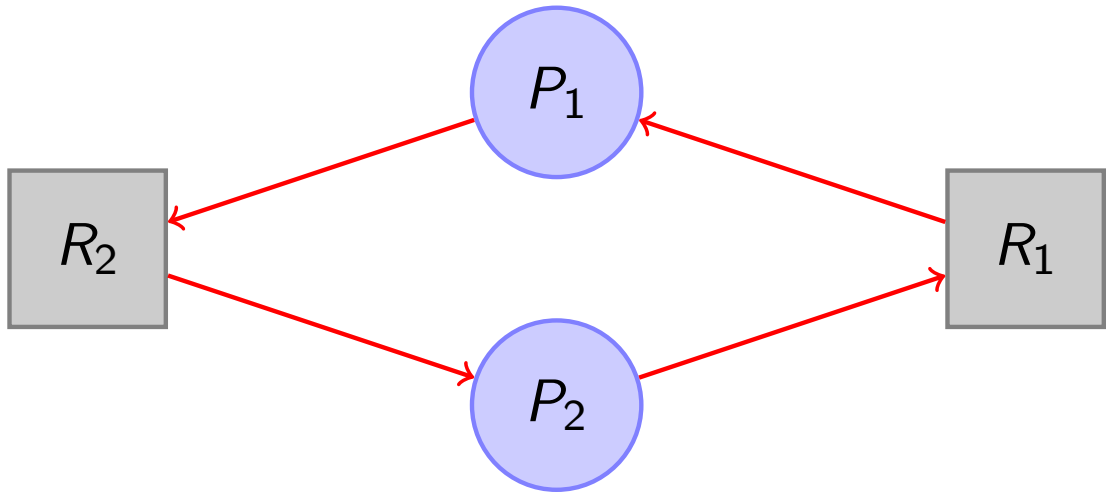
No preemption :
: No hay mecanismo compulsivo para quitarle los recursos a un proceso.

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

(33) 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 ⚠



(34) 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)

(35) 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

(36) Bibliografía adicional

- Hoare, C. *Monitors: an operating system structuring concept*, Comm. ACM 17 (10): 549-557, 1974. http://goo.gl/eVaeeo
- 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