

# Sincronización entre procesos

D. Fernandez Slezak

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

Sistemas Operativos, 1er cuatrimestre de 2024

## (2) 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...).

### (3) ¿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.

## (4) 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.

## monoproceso

```
1  unsigned int ticket= 0;
2  unsigned int fondo= 0;
3
4  /* Devuelve el número de ticket y actualiza el fondo. */
5  unsigned int donar(unsigned int donacion)
6  {
7      fondo+= donacion;
8      ticket++;
9
10     return ticket;
11 }
```

monoproceso Veamos el programa traducido a pseudo assembler.

```
1 load fondo
2 add donacion
3 store fondo
4
5 load ticket
6 add 1
7 store ticket
8
9 return reg
```

## (7) Veamos un scheduling posible

Dos procesos  $P_1$  y  $P_2$  ejecutan el mismo programa, compartiendo las variables.


$P_1$	$P_2$	Variables
donar(10)	donar(20)	fondo==100, ticket==5
load fondo add 10		
	load fondo add 20	
store fondo		fondo==110, ticket==5
	store fondo load ticket add 1	fondo==120, ticket==5
load ticket add 1 store ticket		fondo==120, ticket==6
	store ticket	fondo==120, ticket==6
return 6		
	return 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 por donde se lo mire.
- No debería pasar. Toda ejecución paralela debería dar un resultado equivalente a alguna ejecución secuencial de los mismos procesos. ⚠
- El problema que ocurrió en este ejemplo se llama *condición de carrera* o *race condition*. ⚠
- I.e., una situación donde el resultado que se obtiene varía sustancialmente dependiendo de en qué momento se ejecuten las cosas (o de en qué orden se ejecuten).
- Una forma posible para solucionarlo es logrando la *exclusión mutua* mediante *secciones críticas*.





## (9) ¿Qué es una sección crítica?

- Un cacho de código tal que: 
  - 1 Sólo hay un proceso a la vez en su sección crítica.
  - 2 Todo proceso más tarde o más temprano va a poder entrar a su sección crítica.
  - 3 Ningún proceso fuera de su sección crítica 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?

## (10) Implementando secciones críticas

- 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.
- Una solución particular para este problema es el algoritmo de Peterson, para dos procesos. [Léanla del libro.](#)
- [La solución más general consiste en obtener un poquito de ayuda del HW.](#)

## (11) 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.

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

## 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 }
```

## (14) 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 jamás llegarán a empleados del mes... ⚠
- ...pero sus programadores sí...

## (15) Busy waiting



- Ex-programador que hacía busy waiting.
- Le conseguí el puesto yo mismo. ⚠
- No hagan busy waiting.
- Conviene poner un `sleep()` en el cuerpo del `while`.

## (16) 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.
- Edsger Wybe Dijkstra, *Cooperating Sequential Processes*, Technical Report EWD-123, Sept. 1965.





## (18) 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 nos vamos al infierno.
- Podríamos usar *sleep()-wakeup()*. ¿Podríamos?

## (19) 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++ 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*.

## (20) 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*. 

## -Consumidor con semáforos.

Código común:

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


```
1 void productor()  
2 {  
3     while (true)  
4     {  
5         item= producir_item();  
6         wait(vacios);  
7         // Hay lugar. Ahora  
8         // necesito acceso  
9         // exclusivo.  
10        wait(mutex);  
11        agregar(item, buffer);  
12        cant++;  
13        // Listo, que sigan  
14        // los demás.  
15        signal(mutex);  
16        signal(llenos);  
17    }  
18 }
```

```
1 void consumidor()  
2 {  
3     while (true)  
4     {  
5         wait(llenos);  
6         // Hay algo. Ahora  
7         // necesito acceso  
8         // exclusivo.  
9         wait(mutex);  
10        item= sacar(buffer);  
11        cant--;  
12        // Listo, que sigan  
13        // los demás.  
14        signal(mutex);  
15        signal(vacios);  
16        hacer_algo(item);  
17    }  
18 }
```

## (22) Otra alternativa

- Hay otras primitivas de sincronización además de los semáforos.
- Una de ellas son los *contadores de eventos*.
- Léanlos del libro.

## (23) 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

- Definición formal: un conjunto de procesos está en deadlock si cada proceso del conjunto está esperando por un evento que sólo otro de los procesos del conjunto puede causar.
- Notar: a veces el deadlock puede involucrar a muchos procesos.
- Suele ser muy difícil de detectar si no se planifica adecuadamente.

## (26) Deadlock

- Condiciones (Coffman y cía, 1971): 
  - Exclusión mutua: cada recurso está asignado a un proceso o está disponible.
  - *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 que ya los tiene. El proceso debe liberarlos explícitamente.
  - Espera circular: tiene que haber una cadena de dos o más procesos, cada uno de los cuales está esperando algún recurso que tiene el miembro siguiente.
- Las cuatro deben estar presente para que haya deadlock. Son condiciones **necesarias**. 
- Estas condiciones se pueden modelar como grafos con dos tipos de nodos: procesos y recursos. Hay deadlock si se encuentra un ciclo en el grafo.

## (27) ¿Qué hacer con el deadlock?

- Un primer enfoque. Algoritmo del avestruz:
  - Escondo la cabeza abajo de la tierra.
  - ¿Deadlock? ¿Qué deadlock?
  - Avestruz = Ostrich  $\neq$  Ostra
  - Por más que algún libro insista...
- A veces se puede evitar por diseño: se desarrolla el sistema de manera tal que nunca entre en deadlock.
  - Este tipo de alternativa es válida en entornos controlados.
  - Suele implicar disminuir las libertades del programador: sólo se le brindan primitivas seguras.
- Otro enfoque consiste en detectarlo en tiempo de ejecución:
  - Para esto hay que ir tomando nota de qué procesos piden qué recursos.
  - Lo cual toma tiempo.
  - Además, cuando se detecta el deadlock, ¿cómo se soluciona?
- Más adelante vamos a ver algoritmos de detección y de recuperación de deadlock.

## (28) Otros problemas

- Starvation - Inanición
  - Un proceso puede no ser nunca removido de la cola del semáforo en la que se encuentra suspendido (por ej: si uso LIFO).
- Inversión de Prioridades
  - Problema de scheduling cuando un proceso de baja prioridad retiene un lock que necesita uno de mayor prioridad.
  - Se puede resolver con un protocolo de herencia de prioridades

- Notemos que introdujimos los locks para evitar las condiciones de carrera.
- Pero los locks trajeron a los deadlocks y otros inconvenientes
- Dura la vida del programador...

## (30) Monitores


- Una solución para evitar caer en deadlocks es *monitores*.
  - Hoare, C. *Monitors: an operating system structuring concept*, Comm. ACM 17 (10): 549-557, 1974.
- No sirven para todos los casos, pero pueden ser útiles a veces.
- La idea es tener bloques de código (llamados monitores) que contengan también estructuras de datos propias.
- Sólo un proceso puede estar en un monitor a la vez. Esto garantiza exclusión mutua.
- Como estas estructuras de datos sólo se pueden acceder desde el propio monitor, y sólo un proceso puede estar en un monitor en cada momento dado, nunca puede un proceso estar esperando algún recurso teniendo otro tomado.
- La implementación se puede hacer con semáforos. Pero la hace el compilador.
- Sin embargo, no todo se puede resolver con monitores.
- Ejemplo: cómo hace el consumidor para bloquearse cuando no hay ítems.

```
monitor nombre-monitor
{
    // Declaración de variables locales
    procedure P1 (...) { .... }

    procedure Pn (...) {.....}

    Initialization code (...) { ..... }
}
```

## (32) Variables de condición

- Otra primitiva de sincronización se llama *variable de condición*. 
- Las variables de condición permiten a un proceso hacer `sleep()` hasta que algún otro le avise que cierto evento ya sucedió mediante un `signal()`.
- A diferencia de los semáforos, acá no se lleva la cuenta. Si el `signal()` se hace antes que el `sleep()`, se pierde.
- Son menos poderosas, pero más fáciles de implementar, y toman mucho menos tiempo.
- Antes hablamos de `sleep()`-`wakeup()`. La idea es muy parecida. Más allá de detalles, como que puede haber varios procesos dormidos en la misma variable de condición, la diferencia es que se las suele pensar en el contexto de monitores.
- **CUIDADO:** en el contexto de algunos entornos de programación concurrente, como por ejemplo `pthread`s, las variables de condición no gozan de la exclusión mutua que brindan los monitores.



- Tres problemas más:
  - Hay una potencial pérdida de paralelismo que podría llegar a ser innecesaria. De todas maneras esto podría ser un precio aceptable por la corrección.
  - No están disponibles en todos los lenguajes.
  - Cuando los procesos además de ser concurrentes son distribuidos (ie, máquinas distintas), no se pueden usar.

## (34) Dónde estamos

- Vimos
  - Condiciones de carrera.
  - Secciones críticas.
  - TestAndSet.
  - Busy waiting / sleep.
  - Productor - Consumidor.
  - Semáforos.
  - Deadlock.
  - Monitores.
  - Variables de condición.
- En la práctica:
  - Ejercicios de sincronización entre procesos.
- En la próxima teórica:
  - Problemas comunes de sincronización entre procesos.