

# Sincronización entre procesos 2: Razonamiento y problemas clásicos (primera parte)

Fernando Schapachnik


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

Sistemas Operativos, primer cuatrimestre de 2021

## (2) Créditos

Basado fuertemente en trabajo de Sergio Yovine.

### (3) Siempre los mismos problemas...

- En muchas áreas de la computación hay problemas clásicos.
- Eso significa que aparecen una y otra vez bajo diferentes formas, pero son básicamente el mismo problema.
- En el caso de sincronización entre procesos también sucede que muchos de estos problemas nos suelen inducir a pensar en “soluciones” incorrectas.
- Por eso está bueno estudiarlos y conocer buenas formas de solucionarlos.
- Además, analizarlos suele aportar una mirada bastante profunda sobre problemas fundamentales de la computación, y a veces, del razonamiento humano.
- **OJO:** conocer los problemas clásicos y sus soluciones no debe ser una excusa para la pereza. Los problemas concretos tienen particularidades y contextos que hacen necesario evaluarlos de manera individual y **pensar** en cada caso. 
- Dicho de otra forma, el conocimiento no reemplaza a la inteligencia. Ambos se complementan.

## (4) Turnos

- Tenemos una serie de procesos ejecutando en simultáneo:  
 $P_i, i \in [0 \dots N - 1]$
- Cada proceso  $P_i$  ejecuta una tarea  $s_i$ .
- Supongamos que consiste en imprimir “Soy el proceso  $i$ ”.
- Qué queremos:
  - Sí queremos: Soy el proceso 0. Soy el proceso 1. ... Soy el proceso  $N - 1$ .
  - No queremos: Soy el proceso 2. Soy el proceso 1. Soy el proceso 3 ...
  - Es decir, queremos garantizar que “van tomando turnos”.
- ¿Soluciones?

## (5) Turnos (cont.)

- Podemos utilizar semáforos.

```
1  // Semáforos
2  semaphore sem[N+1];
3
4  // Inicialización
5  proc init() {
6      for(i = 0; i<N+1; i++)
7          sem[i] = 0;
8
9      for (i = 0; i<N; i++)
10         spawn P(i);
11
12     sem[0].signal();
13 }
```

```
1  // Proceso i
2  proc P(i) {
3      // Esperar turno
4      sem[i].wait();
5      // Ejecutar
6      print("Soy el proc " + i);
7      // Avisar al próximo
8      sem[i+1].signal();
9  }
```

- ¿Es correcto este programa?

## (6) Razonando en paralelo

- ¿Cómo razonamos sobre un programa paralelo?
- Dicho de otro modo, ¿cómo hacemos para saber si es correcto?
- ¿Podemos derivar de una pre condición a un post condición?
- ¿Vale el teorema del invariante, por ejemplo?
- La dificultad es que los programas paralelos no tienen una, sino muchas ejecuciones posibles.
- Una ejecución es  $\tau = \tau_0 \rightarrow \tau_1 \dots$  donde los  $\tau_i$  son los diferentes estados del programa.
- Entonces la pregunta es si todas las ejecuciones del programa...
- ...¿llegan a la postcondición?
- Tampoco, porque:
  - Nos interesan ejecuciones que no necesariamente terminan.
  - Nos interesa considerar si abortan, si se traban, si mueren de inanición, etc.

## (7) Razonando en paralelo (cont.)

- En el caso de los programas paralelos la noción de *correcto* deja de ser únivoca para transformarse en un conjunto de propiedades que se plantean sobre toda ejecución.
- Y razonar sobre eso significa poder demostrar (o al menos argumentar) que se cumplen.
- (Para cualquier scheduling posible.)
- Hoy vamos a ver algunas propiedades comunes.
- A la vez que vamos a analizar también problemas clásicos (que muchas veces dan origen a esas mismas propiedades).

## (8) ¿Qué significa que un programa distribuido sea correcto?

- La manera de lidiar con la corrección de este tipo de sistemas suele ser:
  - Plantear propiedades de **safety**: cosas malas no suceden (por ejemplo, deadlock).
  - Plantear propiedades de **progreso** (o **liveness**): en algún momento algo bueno sucede.
  - Demostrar (o argumentar) que la combinación de esas propiedades implica el comportamiento deseado.



## (9) Tipos de propiedades

- Contraejemplo:
  - Sucesión de pasos que muestra una ejecución del sistema que viola cierta propiedad.
- Safety :
  - Intuición: “nada malo sucede”.
  - Ejemplos: exclusión mutua, ausencia de deadlock, no pérdida de mensajes, “los relojes nunca están más de  $\delta$  unidades desincronizados”.
  - Definición (un poco más) formal: tienen un contraejemplo finito.
- Liveness :
  - Intuición: “en algún momento algo bueno sí va a suceder”
  - Ejemplos: “si se presiona el botón de stop, el tren frena”, “cada vez que se recibe un estímulo, el sistema responde”, “siempre en el futuro el sistema avanza”, no inanición.
  - Definición (un poco más) formal: los contraejemplos no son finitos.

## (10) Tipos de propiedades (cont.)

- A veces interesa probar **fairness**.
- Intuición: “Los procesos reciben su turno con infinita frecuencia”.
  - Incondicional: El proceso es ejecutado “regularmente” si está habilitado siempre.
  - Fuerte: El proceso es ejecutado “regularmente” si está habilitado con infinita frecuencia.
  - Débil: El proceso es ejecutado “regularmente” si está continuamente habilitado a partir de determinado momento.
- Intuición más débil: “No se van a dar escenarios poco realistas donde alguien es postergado para siempre”
- En general, fairness se suele suponer como válida para probar otras propiedades (liveness en general).

## (11) Tipos de propiedades (cont.)

- En la práctica se usan versiones acotadas de estas propiedades (por ejemplo, “el sistema responde en a lo sumo X unidades de tiempo”).
- Para trabajar con estas propiedades se inventaron *lógicas temporales* como LTL, CTL, TCTL, ITL, etc.

## (12) Demostrando propiedades

- En muchos casos no vamos a demostrar, simplemente vamos a argumentar.
- Si queremos demostrar, tenemos que tener un modelo formal del comportamiento del sistema: se suelen usar para eso distintos tipos de autómatas.
- También tenemos que formalizar la propiedad en alguna de estas lógicas.
- Podemos demostrar “a mano”, usando las reglas propias de estas lógicas...
- O usar herramientas:
  - Theorem provers.
  - Model checkers.

## (13) Volvamos a los turnos

- Entonces, ¿nuestra solución es correcta?
- Primero deberíamos formalizarla:
  - Tenemos una serie de procesos ejecutando en simultáneo:  
 $P_i, i \in [0 \dots N - 1]$
  - Cada proceso  $P_i$  ejecuta una tarea  $s_i$ .
  - Toda ejecución debe garantizar la siguiente propiedad (**TURNOS**):  
los  $s_i$  ejecutan en orden:  $s_0, s_1, \dots, s_{N-1}$
- Ahora sí analicemos la solución propuesta:

## (14) Turnos (cont.)



```
1 // Semáforos
2 semaphore sem[N+1];
3
4 // Inicialización
5 proc init() {
6     for(i = 0; i<N+1; i++)
7         sem[i] = 0;
8
9     for (i = 0; i<N; i++)
10         spawn P(i);
11
12     sem[0].signal();
13 }
```

```
1 // Proceso i
2 proc P(i) {
3     // Esperar turno
4     sem[i].wait();
5     // Ejecutar
6     s(i);
7     // Avisar al próximo
8     sem[i+1].signal();
9 }
```

- ¿Cumple *TURNOS*?
- Demostración por el absurdo.
- ¿Qué tipo de propiedad es?
- Veamos otro problema clásico.

## (15) Barrera o rendezvous: definición

- *Rendezvous* (punto de encuentro), o *barrera de sincronización*.
- Cada  $P_i, i \in [0 \dots N - 1]$ , tiene que ejecutar  $a(i); b(i)$ .
- Propiedad **BARRERA** a garantizar:

$b(j)$  se ejecuta después de **todos** los  $a(i)$

- Es decir, queremos *poner una barrera* entre los  $a$  y los  $b$ .
- Pero, no hay que restringir de más:

no hay que imponer ningún orden entre los  $a(i)$  ni los  $b(i)$

## (16) Rendezvous (cont.)

- Un objeto atómico y un semáforo

```
1  atomic<int> cant = 0; // Procs que terminaron a
2  semaphore barrera = 0; // Barrera baja
3
4  proc P(i) {
5      a(i);
6      // ¿Se puede ejecutar b?
7      if (cant.getAndInc() < N-1)
8          // No. Esperar.
9          barrera.wait();
10     else
11         // Sí. Entrar y avisar
12         barrera.signal();
13     // Ejecutar b (sección crítica).
14     b(i);
15 }
```

- ¿Esta solución es correcta?

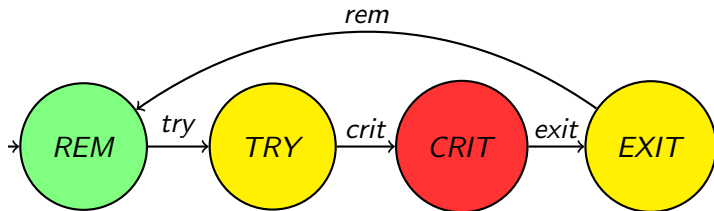


## (17) Rendezvous (cont.)

- ¿Algún  $b(i)$  se ejecuta antes de que terminen todos los  $a(i)$ ?  
No.
- Pero...  $N - 1$  procesos se quedan bloqueados en la línea ??.
- ¿Por qué? Porque hay un único `signal()` (línea ??).
- (Tarea: leer del libro las condiciones de Coffman. ¿Se cumplían las condiciones de Coffman?)
- Es decir, no alcanza con la propiedad de safety, necesitamos alguna propiedad de progreso
- Un primer intento sería: “todo proceso que intenta acceder a la sección crítica, en algún momento lo logra”.
- Mejorado: “todo proceso que intenta acceder a la sección crítica, en algún momento lo logra, **cada vez que lo intenta**”.
- ¿Podemos formalizar la escritura de esta propiedad?

## (18) Modelo de proceso

- N. Lynch, Distributed Algorithms, 1996 (cap. 10)



- Estado:  $\sigma : [0 \dots N - 1] \mapsto \{REM, TRY, CRIT, EXIT\}$
- Transición:  $\sigma \xrightarrow{\ell} \sigma', \ell \in \{rem, try, crit, exit\}$
- Ejecución:  $\tau = \tau_0 \xrightarrow{\ell} \tau_1 \dots$

## (19) WAIT-FREEDOM

- “Todo proceso que intenta acceder a la sección crítica, en algún momento lo logra, cada vez que lo intenta”.
- Esta propiedad se llama **WAIT-FREEDOM**.

$$\forall \tau. \forall k. \forall i. \tau_k(i) = TRY \implies \exists k' > k. \tau_{k'}(i) = CRIT$$

- Intuición: “libre de procesos que esperan (para siempre)”.
- Es una garantía muy fuerte.

## (20) Rendezvous: solución

- Sacar el else (línea ??) para que haya un `signal()` después de cada `wait()`

```
1  atomic<int> cant = 0;  // Procs que terminaron a
2  semaphore barrera = 0; // Barrera baja
3
4  proc P(i) {
5      a(i);
6      // TRY
7      // ¿Se puede ejecutar b?
8      if (cant.getAndInc() < N-1)
9          // No. Esperar.
10         barrera.wait();
11     // Sí. Entrar y avisar.
12     barrera.signal();
13     // CRIT
14     // Ejecutar b.
15     b(i);
16 }
```

- El modelo de Lynch puede ayudarnos a formalizar algunas propiedades.

## (22) Ecuanimidad (a veces llamada justicia)

### FAIRNESS

Para toda ejecución  $\tau$  y todo proceso  $i$ ,  
**si**  $i$  **puede** hacer una transición  $\ell_i$  en una cantidad  
infinita de estados de  $\tau$   
**entonces** existe un  $k$  tal que  $\tau_k \xrightarrow{\ell_i} \tau_{k+1}$ .

## (23) Exclusión mutua

Para toda ejecución  $\tau$  y estado  $\tau_k$ , no puede haber más de **un** proceso  $i$  tal que  $\tau_k(i) = CRIT$ .

$$EXCL \equiv \square \#CRIT \leq 1$$

## (24) Progreso (del sistema)

Para toda ejecución  $\tau$  y estado  $\tau_k$ ,  
**si** en  $\tau_k$  hay **un** proceso  $i$  en  $TRY$  y **ningún**  $i'$  en  $CRIT$

**entonces**  $\exists j > k$ , t. q. en el estado  $\tau_j$  **algún** proceso  $i'$  está en  $CRIT$ .

**LOCK-FREEDOM**  $\equiv$

$$\square (\#TRY \geq 1 \wedge \#CRIT = 0 \implies \diamond \#CRIT > 0)$$



## (25) Dos predicados auxiliares

- Lograr entrar:

Def.:  $IN(i) \equiv i \in TRY \implies \diamond i \in CRIT$

- Salir:

Def.:  $OUT(i) \equiv i \in CRIT \implies \diamond i \in REM$

(*deadlock-, lockout-, o starvation-freedom*)

Para toda ejecución  $\tau$ ,

**si** para todo estado  $\tau_k$  y proceso  $i$  tal que

$\tau_k(i) = CRIT$ ,

$\exists j > k$ , tal que  $\tau_j(i) = REM$

**entonces** para todo estado  $\tau_{k'}$  y **todo** proceso  $i'$ ,

**si**  $\tau_{k'}(i') = TRY$

**entonces**  $\exists j' > k'$ , tal que  $\tau_{j'}(i') = CRIT$ .

**STARVATION-FREEDOM**

$\equiv \forall i. \Box OUT(i) \implies \forall i. \Box IN(i)$

Para toda ejecución  $\tau$ , estado  $\tau_k$  y **todo** proceso  $i$ ,  
**si**  $\tau_k(i) = TRY$   
**entonces**  $\exists j > k$ , tal que  $\tau_j(i) = CRIT$ .

$$\text{WAIT-FREEDOM} \equiv \forall i. \square IN(i)$$

- Vimos
  - Algunos problemas comunes de sincronización.
  - Propiedades a garantizar.
  - Cómo razonar sobre programas paralelos (distribuidos en realidad).
- Práctica: ejercicios de sincronización.
- En la próxima teórica vamos a ver más problemas comunes de sincronización.

## (29) Bibliografía adicional

- M. Herlihy, N. Shavit. *The Art of Multiprocessor Programming*. Morgan Kaufmann, 2008.
- N. Lynch. *Distributed Algorithms*. Morgan Kaufmann, 1996.
- Valgrind tool. <http://valgrind.org/>
- Java Pathfinder (JPF).  
<http://babelfish.arc.nasa.gov/trac/jpf>