Sincronización entre procesos (2/2)
Modelo teórico de razonamiento

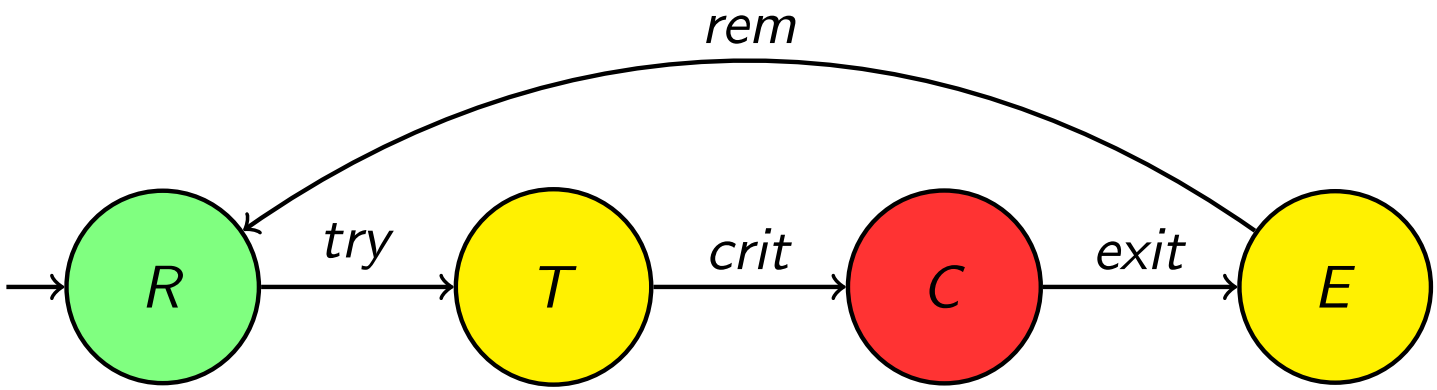
Sergio Yovine

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

Sistemas Operativos, segundo cuatrimestre de 2017

(2) Modelo de proceso

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



- Estado: $\sigma : [0 \ldots N-1] \mapsto \{R, T, C, E\}$
- Transición: $\sigma \xrightarrow{\ell} \sigma'$, $\ell \in \{rem, try, crit, exit\}$
- Ejecución: $\tau = \tau_0 \xrightarrow{\ell} \tau_1 \ldots$
- Garantizar *PROP*: Toda ejecución satisface *PROP*
- Notación: $\#S$ = cantidad de elementos del conjunto S

Exclusión mutua (EXCL)

Para toda ejecución τ y estado τ_k, no puede haber más de un proceso i tal que $\tau_k(i) = C$.

(4) Propiedades

Exclusión mutua (EXCL)

Notación:

$$\Box\ \#CRIT \leq 1$$

Progreso (PROG) (*lock-free*)

Para toda ejecución τ y estado τ_k,
si en τ_k hay un proceso i en T y ningún i' en C
entonces $\exists j > k$, t. q. en el estado τ_j algún proceso i' está en C.

(6) Propiedades

Progreso (PROG) (*lock-free*)

Notación:

$$\Box(\#TRY \geq 1 \wedge \#CRIT = 0 \implies \Diamond \#CRIT > 0)$$

Progreso global absoluto (WAIT-FREE)

Para toda ejecución τ, estado τ_k y todo proceso i,
si $\tau_k(i) = T$
entonces $\exists j > k$, tal que $\tau_j(i) = C$.

Progreso global absoluto (WAIT-FREE)

Notación:

$$\forall i. \Box\ IN(i)$$

donde $IN(i)$ es:

$$TRY(i) \implies \diamondsuit\ CRIT(i)$$

Progreso global dependiente (G-PROG)

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

Para toda ejecución τ,
si para todo estado τ_k y proceso i tal que $\tau_k(i) = C$,
$\exists j > k$, tal que $\tau_j(i) = R$
entonces para todo estado $\tau_{k'}$ y todo proceso i',
si $\tau_{k'}(i') = T$
entonces $\exists j' > k'$, tal que $\tau_{j'}(i') = C$.

Progreso global dependiente (G-PROG)
(*deadlock-*, *lockout-*, *o starvation-free*)

Notación:

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

donde $OUT(i)$ es:

$$CRIT(i) \implies \Diamond\ REM(i)$$

Justicia (FAIR) (*fairness*)

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

(12) Observaciones a tener en cuenta

- **EXCL** es una propiedad de safety
  – nada malo pueda pasar nunca.
- **PROG**, **G-PROG** y **WAIT-FREE** son propiedades de liveness
  – algo bueno debe pasar en el futuro.
- **FAIR** se asume
  – debe ser garantizada por el scheduler
- No hacer nada garantiza safety.
- Siempre hay que tener ambas propiedades.

(13) Turnos: definición

- Tenemos una serie de procesos:

$$P_i, i \in [0 \ldots N-1]$$

- Se están ejecutando en simultáneo.
- Cada proceso i ejecuta una tarea s_i.
- Propiedad *TURNOS* a garantizar:

los s_i ejecutan en orden: $s_0, \ldots, s_{N-1}$

(14) Turnos: solución

- Podemos utilizar semáforos.

```
1  // Semáforos
2  semaphore sem[N+1] = 0;
3
4  // Proceso i
5  proc P(i) {
6    // T: Esperar turno
7    sem[i].wait();
8    // C: Ejecutar
9    s(i);
10   // E: Avisar al próximo
11   sem[i+1].signal();
12   // R:
13 }
```

- ¿Esta solución es correcta?

(15) Barrera o Rendezvous: definición

- *Rendezvous* (punto de encuentro) o *barrera de sincronización*.
- Cada $P_i, i \in [0 \ldots N-1]$, 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: solución

- 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    // T:
7    // ¿Se puede ejecutar b?
8    if (cant.getAndInc() < N-1)
9      // No. Esperar
10     barrera.wait();
11   else
12     // Sí. Entrar y avisar
13     barrera.signal();
14   // C:
15   // Ejecutar b
16   b(i);
17 }
```

- ¿Esta solución es correcta?

(17) Rendezvous: solución

- $N - 2$ procesos se quedan bloqueados en T (línea 10).
- ¿Por qué?
  - hay $N - 1$ wait() (línea 10) y
  - un único signal() (línea 13).
- Se viola **G-PROG** (en este caso hay *deadlock*).

(18) Rendezvous: solución

- Sacar el `else` (línea 11)

```
1  atomic<int> cant = 0;  // Procs que terminaron a
2  semaphore barrera = 0; // Barrera baja
3
4  proc P(i) {
5    a(i);
6    // T:
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   // C:
14   // Ejecutar b
15   b(i);
16 }
```

(19) Bibliografía extra

- Nicholas Carriero, David Gelernter: Linda in Context. CACM, 32(4), 1989. `http://goo.gl/gfgbsQ`
- Andrew D. Birrell and Bruce Jay Nelson. 1984. Implementing remote procedure calls. ACM Trans. Comput. Syst. 2, 1 (February 1984), 39-59. `http://goo.gl/3eIskN`
- A. L. Ananda, E. K. Koh. A survey of asynchrounous RPC. `http://goo.gl/t96vFg`
- Andrew S. Tanenbaum. RPC. `http://goo.gl/9N3zKz`
- T.L. Casavant, J.G. Kuhl. A taxonomy of scheduling in general-purpose distributed computing systems. IEEE TSE 14(2):141-154, 1988.
- M. Singhal and N. G. Shivaratri. Advanced Concepts in Operating Systems. McGraw Hill, 1994.