Sincronización entre procesos (2/2)
Problemas clásicos

Sergio Yovine

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

Sistemas Operativos, primer cuatrimestre de 2017

(2) Interacción entre procesos

Coordinación

Sincronización

Veamos algunos problemas clásicos *paradigmáticos*...

(3) Turnos: definición

- Problema:
  - 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: $s_0, \ldots, s_{N-1}$

- ¿Soluciones?

(4) Turnos: solución

- Podemos utilizar semáforos.

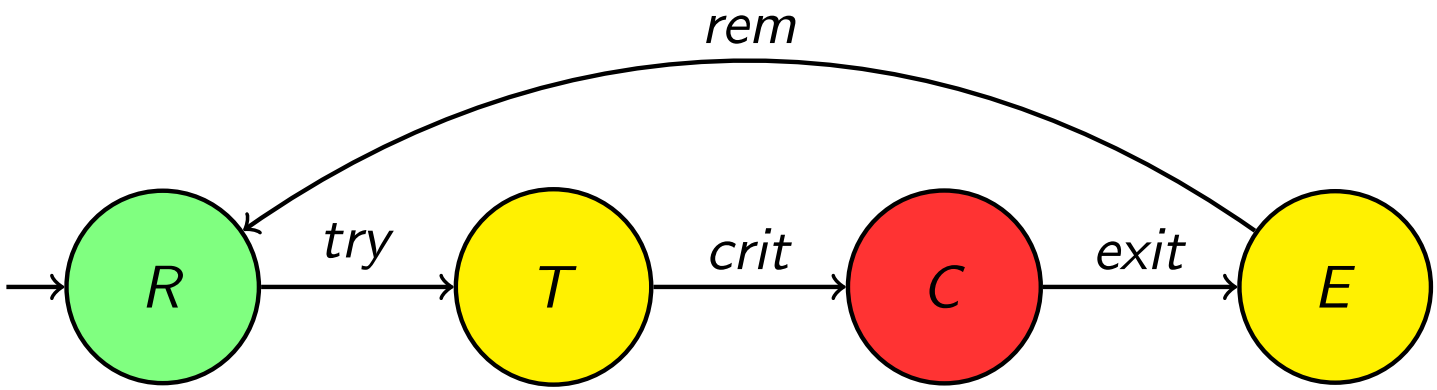
```
1  // Semáforos
2  semaphore sem[N+1];
3
4  // Inicialización
5  proc init() {
6    for(i=0; i<N+1; i++)
7      sem[i] = 0;
8
9    for (i=0; i<N; i++)
10     fork P(i);
11
12   sem[0].signal();
13 }
```

```
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  }
```

- ¿Esta solución es correcta?

(5) 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

(6) 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 $b(i)$

(7) 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?

(8) Rendezvous: solución

- $N-2$ procesos se quedan bloqueados en T (línea 10).
- ¿Por qué? Porque hay $N-1$ `wait()` (línea 10) y un único `signal()` (línea 13).
- Se viola la siguiente propiedad: si un proceso i está en T en un instante k entonces en el futuro (existe $k' > k$) i está en C:

$$\forall \tau.\ \forall k.\ \forall i.\ \tau_k(i) = T \implies \exists k' > k.\ \tau_{k'}(i) = C$$

  es decir, hay *deadlock*.
- Pregunta: ¿Se cumplían las condiciones de Coffman?

(9) Rendezvous: solución

- Sacar el `else` (línea 11) 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    // 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 }
```

(10) Problema: Sección crítica de a $M \leq N$

- Propiedad *SCM* a garantizar:
  A lo sumo $M \leq N$ procesos están simultáneamente en C

```
1  semaphore sem = M;
2  proc P(i) {
3    // T
4    sem.wait();
5    // C
6    sc(i);
7    // E
8    sem.signal();
9  }
```

- Ejercicio: formalizar y probar la propiedad en el modelo de Lynch

(11) Lectores/escritores: definición

- Se da mucho en bases de datos.
- Hay una variable compartida.
- Los escritores necesitan acceso exclusivo.
- Pero los lectores pueden leer simultáneamente.
- Propiedad *SWMR* (Single-Writer/Multiple-Readers):
  Si i es un escritor y está en C, entonces está solo

$$\forall \tau.\ \forall k.\ \exists i.\ \mathit{writer}(i) \wedge \tau_k(i) = C \implies \forall j \neq i.\ \tau_k(j) \neq C$$

(12) Lectores/escritores: solución

- Dos semáforos y un contador

```
semaphore wr = 1;
semaphore rd = 1;
int readers = 0;

proc writer(i) {
  // T
  wr.wait();
  // C
  write();
  // E
  wr.signal();
}
```

```
proc reader(i) {
  // T
  rd.wait();
  if (++readers == 1)
    wr.wait();
  rd.signal();
  // C
  read();
  // E
  rd.wait();
  if (--readers == 0)
    wr.signal();
  rd.signal();
}
```

- ¿Esta solución es correcta?

(13) Lectores/escritores: solución

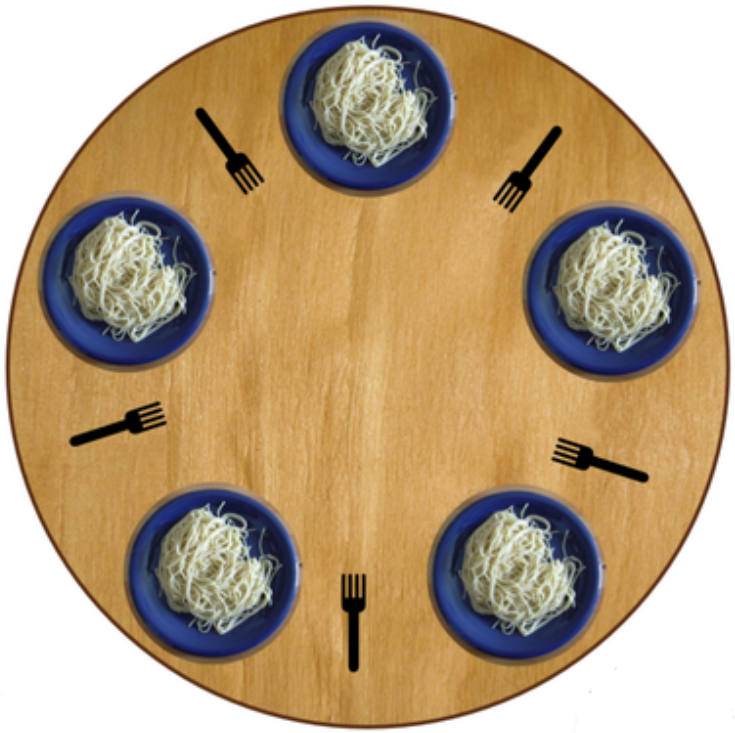
- Puede haber inanición de escritores.
- ¿Por qué? Puede ser que haya siempre (al menos) un lector.
- Se viola la propiedad de *progreso global dependiente* G-PROG
- Esto es, si todo proceso sale de C entonces todo proceso que está en T entra inevitablemente a C.

$$\forall \tau.$$
$$\forall k.\ \forall i.\ \tau_k(i) = C \implies \exists k' > k.\ \tau_{k'}(i) = R$$
$$\implies$$
$$\forall k.\ \forall i.\ \tau_k(i) = T \implies \exists k' > k.\ \tau_{k'}(i) = C$$

- Tarea: pensar cómo garantizar *G-PROG* en este caso.

(14) Filósofos que cenan: definición

- Dining Quintuple/Philosophers, Dijkstra, 1965.
- 5 filósofos en una mesa circular.
- 5 platos de fideos y 1 tenedor entre cada plato.
- Para comer necesitan dos tenedores.



(15) Filósofos que cenan: definición

- Código de los filósofos

```
proc Filósofo(i)
while (true) {
  think();          // R
  getforks(i);      // T
  eat();            // C
  leaveforks(i);    // E
}
```

- Problema: programar `getforks` y `getforks` satisfaciendo:

EXCL-FORK	Los tenedores son de uso exclusivo.
LOCK-FREE	No haya deadlock.
G-PROG	No haya inanición.
EAT	Más de un filósofo esté comiendo a la vez. (Variante de *SCM*).

- Tarea: Escribir formalmente las propiedades.

(16) Filósofos que cenan: solución (ingenua)

- Un arreglo de N semáforos

```
#define left(i)   i
#define right(i) (i  % N)
semaphore forks[N] = 1;

void getforks(i) {
 forks[left(i)].wait();
 forks[right(i)].wait();
}

void leftforks(i) {
   forks[left(i)].signal();
   forks[right(i)].signal();
}
```

- ¿Esta solución es correcta?

(17) Filósofos que cenan: solución (ingenua)

- Propiedades

EXCL-FORK	OK.
LOCK-FREE	NOK.
G-PROG	NOK.
EAT	NOK.

- Resultado general (N. Lynch, Dist. Algorithms, Cap. 11)

  NO existe ninguna solución en la que todos los filósofos hacen lo mismo (no existe *solución simétrica*).

- Tarea: pensar en soluciones. Buscar las ya existentes.

(18) Barbero: definición

- En una peluquería hay un único peluquero
- La peluquería tiene dos salas
  - una de espera, con N sillas
  - otra donde está la única silla para cortar el pelo.
- Cuando no hay clientes, el peluquero se duerme una siesta.
- Cuando entra un cliente
  - si no hay lugar en la sala de espera, se va.
  - si el peluquero está dormido, lo despierta.
- Ejercicio:
  - Formalizar las propiedades a garantizar.
  - Probar que la solución propuesta las satisface.

(19) Barbero: solución

- Vamos a usar dos semáforos y un objeto atómico:

```
semaphore aclient = 0;
semaphore served = 0;
atomic<int> clientes = 0;
```

- El peluquero es sencillo:

```
proc Peluquero
while (true) {
  // T
  aclient.wait();
  served.signal();
  // C
  cuthair();
  // E
}
```

(20) Barbero: solución

- Veamos a los clientes:

```
proc Cliente()
  // T
  // ¿Hay lugar?
  if (clientes.getAndInc() >= N + 1) { // No
    clients.getAndDec();
    return;
  }
  // Sí. Avisarle al peluquero
  aclient.signal();
  // Esperar a ser atendido
  served.wait();
  // C
  gethaircut();
  // E
  clients.getAndDec();
}
```

(21) Dónde estamos

- Vimos
  - Mecanismos de sincronización.
  - Problemas comunes de sincronización.
  - Propiedades a garantizar.
- Práctica: Ejercicios de sincronización.
- En la teórica de sistemas distribuidos veremos
  - Sincronización sin locks y semáforos
  - Más sobre objetos atómicos y sus propiedades

(22) Bibliografía adicional

- Allen B. Downey. *The Little Book of Semaphores*. http://goo.gl/ZB9zYl
- 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.
- Valgrind tool. http://valgrind.org/
- Java Pathfinder (JPF). http://babelfish.arc.nasa.gov/trac/jpf