Tema 3: Cerrojos y Barreras (Parte 1: El problema de la sección crítica)

Elvira Albert

Departamento de Sistemas Informáticos y Computación

Universidad Complutense de Madrid

elvira@sip.ucm.es

Madrid, Marzo, 2021

Introducción

- Los programas concurrentes emplean dos tipos de sincronización:
  - Exclusión mutua
  - Sincronización condicional
- Vamos a examinar dos problemas importantes para ilustrar como programar este tipo de sincronización
  - Secciones críticas
  - Barreras

Contenidos Tema 3

3.1 El problema de la sección crítica (SC)

- La mayor parte de los programas concurrentes tienen SCs de código
- Una solución al problema de la SC se puede utilizar para implementar await
- Formulación problema general: n procesos repetidamente ejecutan una SC y después una no crítica

```
process CS[i=1 to n]
while (true) {
  entry protocolo;
  SC;
  exit protocolo;
  no SC;
}
```

3.1 El problema de la sección crítica (SC)

Nuestro objetivo

Diseñar protocolos de entry y exit que garanticen 4 propiedades:

- SAFETY:
  - **Exclusión mutua:** como máximo un proceso cada vez ejecuta SC
  - **Ausencia de bloqueo (y livelock):** si 2 o más procesos intentan entrar en SC como mínimo uno lo conseguirá
  - **Ausencia retraso innecesario:** si un proceso intenta entra en SC y el resto están ejecutando sus no SC, el proceso entrará
- LIVENESS:
  - **Entrada eventual:** un proceso que intenta entrar en su SC eventualmente entrará (depende de la política de planificación)

3.1 El problema de la sección crítica (SC)

- Solución trivial <SC>
  - instrucción await incondicional
  - que propiedades cumple?
  - garantizado con política incondicionalmente justa
  - cómo implementar <>?
- Solución para dos procesos utilizando await
  - utilizar variables booleanas in1 e in2 que indiquen si p1 y p2 estan en SC
  - invariante global $\neg$ (in1 $\wedge$ in2)
  - propiedad 4 requiere política fuertemente justa

Generalización a "n" procesos

- Utilizar una variable booleana lock $==$ in1 $\vee \ldots \vee$ inn

```
bool lock=false;
process CS1
while (true) {
  <await (!lock) lock=true;>
  SC;
  lock=false;
  no SC;
}
```

- La relevancia de lock es que casi todas las máquinas tienen una instrucción especial para implementar dicha acción atómica

3.2 Secciones Críticas: Espera activa (spin locks)

Test and set

- se utiliza instrucción "test-and-set" que lee el valor de lock, pone lock a true y devuelve el valor inicial que tenía
- propiedad 4 requiere política fuertemente justa
- ineficiencia:
  - contención de memoria: continuamente todos los procesos comprueban lock
  - invalidación de cache: cada vez que lock es escrita, se invalida

Test and test and set

- modificar protocolo de entrada "while (lock) skip"
- contención de memoria se reduce

Implementación await

- Cualquier solución al problema de la SC se puede usar para implementar <S;>

  CSentry; S; CSexit;

- Para implementar <await (B) SC;>:

```
CSentry;
while (!B) {
  CSexit;
  Delay;
  CSentry}
CS;
CSexit;
```

3.3 Secciones Críticas: Soluciones justas

Motivación

- Las soluciones que hemos visto requieren un scheduling fuertemente justo
- Vamos a ver tres soluciones que requieren scheduling débilmente justo

Algoritmo "rompe-empate"

- Si todos los procesos están intentando entrar en SC, no hay control sobre cual entrará
- Solución justa: turnos para entrar
- "Tie-breaker": usar una variable para indicar que proceso fue el último en entrar

Algoritmo ticket

- El algoritmo tie-breaker es bastante complejo para "n" procesos
- Utilizar contadores enteros para ordenar los procesos
- Garantiza entrada eventual con política débilmente justa
- Implementación con instrucción fetch-and-add

Algoritmo panadería

- En el algoritmo de ticket, sino tenemos instrucción fetch-and-add, tenemos que usar protocolo no justo
- Los clientes se preguntan entre ellos y no hay máquina central
- Garantiza entrada eventual con política débilmente justa