

Programación de múltiples procesos

Arquitectura de Computadores – IIC2343



Un programa “ordinario”:

- declaraciones de datos
- asignaciones
- sentencias de control de flujo

Sentencias realmente ejecutadas:

- evaluación de expresiones
- movimiento de datos
- cambios al flujo de control

Instrucciones que aparecen en el código de máquina que resulta de la compilación

Estas instrucciones de máquina

- son ejecutadas secuencialmente
- tienen acceso a datos almacenados en las memorias principal y secundaria



Un programa en un multiprocesador:

- un conjunto de programas secuenciales
- ... que pueden ser ejecutados en paralelo

Cada programa secuencial es ahora (por claridad) un proceso

Paralelo:

- sistema en el cual las ejecuciones de varios procesos se traslapan en el tiempo
- ... ya que son ejecutados en procesadores diferentes

Concurrente:


- paralelismo potencial: la ejecución podría traslaparse
- el paralelismo puede ser sólo aparente: puede ser implementado compartiendo los recursos de unos pocos procesadores, a veces sólo uno



La concurrencia es una abstracción útil:

- podemos entender mejor un programa concurrente si suponemos que todos los procesos son ejecutados en paralelo
- y si los procesos son efectivamente ejecutados en paralelo en varios procesadores,
- ... entender su comportamiento es más fácil si imponemos un orden a las instrucciones que sea compatible con la ejecución compartida en un solo procesador

La gracia es que podemos estudiar y modelar el comportamiento de muchos sistemas reales sin detalles innecesarios



Los (dos o más) procesos que forman un programa trabajan en conjunto para realizar una tarea

... es decir, deben interactuar

Debido a esta interacción, es difícil escribir programas correctos incluso para problemas simples

El desafío proviene de la necesidad de *sincronizar* la ejecución de los diversos procesos

... y de permitirles que se *comuniquen*

Es difícil implementar sincronización y comunicación seguras y eficientes

Un programa secuencial trivial

```
n = 0
k1 = 1
k2 = 2
n = k1
n = k2
```

El valor final de **n** es uno solo: 2

Un programa concurrente trivial, formado por dos procesos: P y Q

```
n = 0
P:      Q:
k1 = 1   k2 = 2
n = k1   n = k2
```

El valor final de **n** puede ser 1 o 2, según en qué orden relativo se ejecuten las asignaciones a **n**

Comunicación:

- **memoria (variables) compartida** —un proceso escribe en una variable que es leída por otro proceso
- memoria distribuida o paso de mensajes —un proceso envía un mensaje que es recibido por otro proceso

Sincronización:

- **exclusión mutua** —asegurar que las secciones críticas de instrucciones no se ejecuten al mismo tiempo
- **sincronización por condición** —postergar un proceso hasta que una determinada condición sea verdadera

P.ej., la comunicación entre un productor y un consumidor se implementa usando un buffer (p.ej., una variable) compartido:

- el productor escribe un mensaje en el buffer (le asigna un valor)
- el consumidor lee un mensaje del buffer

La exclusión mutua permite asegurar que el productor y el consumidor no tienen acceso al buffer al mismo tiempo:

- un mensaje parcialmente escrito no es leído prematuramente

La sincronización de condición permite asegurar que el productor y el consumidor tienen acceso al buffer alternadamente:

- el productor no escribe un segundo mensaje si el primero no ha sido leído
- el consumidor no lee dos veces seguidas un mismo mensaje

El problema de la sección crítica

Cada uno de n procesos ejecuta repetidamente (en un *loop* infinito) una secuencia de sentencias, compuesta de dos subsecuencias:

- una *sección crítica*
- una sección no crítica

La **sección crítica** (SC) es una secuencia de sentencias que tienen acceso a alguna variable compartida (entre los n procesos)

... por lo que la ejecución de la sección crítica de un proceso debe hacerse bajo *exclusión mutua*:

... es decir, excluyendo a todos los otros procesos de ejecutar al mismo tiempo sus respectivas secciones críticas

Para lograr esto, la SC de cada proceso debe ser precedida por un *protocolo de entrada*

- código y variables adicionales para ayudar a que se cumpla la exclusión mutua y otras propiedades

... y seguida por un *protocolo de salida*

```
process P[i = 1 ... n]:  
  while true:  
    protocolo de entrada  
    SC  
    protocolo de salida  
    sección no crítica
```

Suponemos que un proceso que entra a su SC, sale:

- un proceso puede terminar su ejecución —por cualquier razón— sólo fuera de su SC

El problema consiste en diseñar los protocolos de entrada y de salida

... de manera que cumplan las siguientes cuatro propiedades:

Exclusión mutua:

- a lo más un proceso a la vez está ejecutando su SC

No hay bloqueo:

- si dos o más procesos están tratando de entrar a sus SC's, entonces al menos uno lo logrará

No hay demora innecesaria:

- si un proceso está tratando de entrar a su SC y los otros procesos no, entonces nada le impide al primer proceso entrar

Entrada:

- un proceso que está tratando de entrar a su SC (finalmente) lo logrará




Las cuatro propiedades anteriores son importantes,

... pero exclusión mutua es esencial para que el programa funcione correctamente:

- nos concentramos en exclusión mutua primero
- luego vemos cómo cumplir las otras propiedades también

Desarrollamos primero una solución para dos procesos:

- la solución es generalizable para n procesos



Sean **in1** e **in2** variables boolean, inicialmente falsas, que indican si los procesos están en sus SCs:

- cuando el proceso P1 está en su SC, ponemos **in1** en verdadera;
... similarmente, para P2 e **in2**
- el estado que queremos evitar es uno en que **in1** e **in2** son ambas verdaderas

El protocolo de entrada

Antes que **P1** entre a su *SC* —y ponga **in1** en verdadera— debe asegurarse de que **in2** sea falsa:

$$\langle \text{await} (!\text{in2}) \text{ in1} = \text{true} \rangle$$

... en que $\langle y \rangle$ especifican que la sentencia **await** debe ejecutarse indivisiblemente:

- la semántica de **await** (*cond*) *S* es que se espera hasta que la condición *cond* sea verdadera y entonces se ejecuta *S*
- “indivisiblemente”, es decir, cuando se escribe entre $\langle y \rangle$, significa que cuando *S* comienza a ejecutarse, está garantizado que *cond* es verdadera

Los procesos son simétricos:

- el protocolo de entrada en P2 es análogo

... y el de salida

Nunca es necesario postergar (hacer esperar) a un proceso que sale de su *SC*

No necesitamos custodiar (p.ej., mediante $\langle y \rangle$) la asignación de **false** a **in1** o a **in2**

in1 = false, in2 = false

process P1:

while true:

⟨await (!in2) in1 = true⟩

SC

in1 = false

sección no crítica

process P2:

while true:

⟨await (!in1) in2 = true⟩

SC

in2 = false

sección no crítica

La solución anterior cumple tres de las cuatro propiedades

Exclusión mutua:

- cumple, por construcción

No hay bloqueo —lo que se demuestra por contradicción:

- si ambos procesos estuvieran bloqueados en sus protocolos de entrada, entonces **in1 == in2 == true**
- ... pero si ambos procesos están en sus protocolos de entrada, entonces **in1 == in2 == false**

No hay demora innecesaria:

- un proceso es bloqueado sólo si el otro proceso está en su SC

Muchos computadores, y especialmente los multiprocesadores, tienen alguna instrucción especial que permite implementar estas *acciones indivisibles condicionales*:

La instrucción *Test and Set*:

```
TS(lock):  
    init = lock  
    lock = true  
    return init
```

La instrucción *Fetch and Add*:

```
FA(var,incr):  
    tmp = var  
    var = var+ incr  
    return tmp
```

P.ej., cuando se ejecuta la instrucción *Fetch and Add*, las tres instrucciones que la componen se ejecutan ininterrumpidamente, como si fueran una sola instrucción

... de modo que el proceso que la ejecuta recibe el valor que la variable **lock** tenía al momento de iniciarse la ejecución

... y al mismo tiempo coloca la variable **lock** en **true**

<await (!lock) lock = true> = while TS(lock)::

Con esto, podemos escribir la versión general, para n procesos, de la solución anterior:

- todo lo que nos interesa saber, cuando un proceso quiere entrar a su sección crítica, es si hay o no otro proceso ejecutando su propia sección crítica
- la variable **lock** es verdadera si y solo si hay exactamente un proceso ejecutando su sección crítica

```
bool lock = false
process P[i = 1 ... n]:
    while true:
        while TS(lock)::
            SC
            lock = false
            sección no crítica
```

Las propiedades de esta solución se analizan en la siguiente diapositiva

Exclusión mutua:

- sólo un proceso será el primero en cambiar el valor de **lock** de **false** a **true** y terminará su protocolo de entrada

No hay bloqueo:

- los procesos que esperan en sus protocolos de entrada, esperan que **lock** se vuelva **false**
... cuando esto ocurra, uno tendrá éxito en cambiarlo a **true**
- si **lock** está en **true** es porque hay un proceso en su *SC*
... este proceso garantizadamente saldrá de su *SC* asignando **false** a **lock**

No hay espera innecesaria:

- si ningún proceso está en su *SC* o tratando de entrar a ella, entonces **lock == false**
- en esta condición, si sólo un proceso quiere entrar a su sección crítica, nada se lo impide

Entrada:

- depende del *scheduling*

¿Qué pasa cuando dos o más procesos están tratando de entrar a sus SC 's?

- en la solución anterior no hay control acerca de cuál tendrá éxito

Solución (más) justa: los procesos deberían tomar turnos:

- el **algoritmo de Peterson** “desempata” usando una variable adicional que indica cuál proceso fue el último en entrar

Primero, ¿ cómo podemos implementar

```
⟨ await (!lock) lock = true ⟩
```

... usando sólo variables simples y sentencias convencionales ?

P.ej., en el siguiente código, las dos acciones no son ejecutadas indivisiblemente —y por lo tanto no se asegura exclusión mutua:

```
while in2;;      while in1;;  
in1 = true      in2 = true
```

El problema se debe a que las variables **in1** e **in2** son puestas en verdadero después de esperar, en las sentencias **while**, que se vuelvan falsas

Así, el primer paso de un protocolo de entrada es poner en verdadero la variable que indica que quiero entrar a mi sección crítica

P.ej., las siguientes dos acciones tampoco son ejecutadas indivisiblemente,

... pero aseguran exclusión mutua:

```
in1 = true          in2 = true
while in2::;        while in1::;
```

Sin embargo, podrían incurrir en bloqueo (*deadlock*) —si **in1** e **in2** son ambas **true**

Sea **last** una variable que indica cuál proceso fue el último en iniciar el protocolo de entrada:

- si **P1** y **P2** están tratando de entrar —**in1** e **in2** son ambas **true**— el último que inició el protocolo se posterga en favor del otro proceso

`in1 = false, in2 = false`
`last = 1` —*inicialización arbitraria*

Process P1
while true:
 `in1 = true`
 `last = 1`
 while `in2 ∧ last == 1`;;
 SC
 `in1 = false`
 sección no crítica

Process P2
while true:
 `in2 = true`
 `last = 2`
 while `in1 ∧ last == 2`;;
 SC
 `in2 = false`
 sección no crítica

Generalizar a n procesos la solución anterior es difícil

Veamos otra posibilidad

Los dos pasos fundamentales:

- un proceso que quiere entrar a su *SC* primero mira los turnos (un número entero > 0) de todos los otros procesos
... y luego define su propio turno como uno más que el de cualquier otro proceso
- el proceso luego espera comparando su turno con el de todos los otros procesos,
... hasta que el suyo sea el menor de todos (exceptuando los 0's)
- un proceso que no quiere entrar a su sección crítica tiene su turno en 0

```
turn[n] = {0 ... 0}
process P[i = 1 ... n]:
  while true:
    < turn[i] = max(turn[1], ..., turn[n]) + 1 >
    for j = 1 ... n:
      if j ≠ i:
        < await (turn[j] == 0 ∨ turn[i] < turn[j]) >
    SC
    turn[i] = 0
    sección no crítica
```

Los valores de **turn[i]** pueden llegar a ser muy grandes:

- aunque **turn[i]** seguirá creciendo sólo si siempre hay al menos un proceso tratando de entrar a su sección crítica, lo que es improbable

Sin embargo, el verdadero problema de esta solución es que no es posible implementar este algoritmo en computadores modernos:

- la asignación a **turn[i]** requiere calcular el máximo de n valores
- la sentencia **await** hace referencia a una variable compartida, **turn[j]**, dos veces

Primero, busquemos una solución para dos procesos

turn1 = 0, turn2 = 0

Process P1:

```
while true:
    turn1 = turn2 + 1
    while turn2 ≠ 0
        ∧ turn1 > turn2;;
    SC
    turn1 = 0
    sección no crítica
```

Process P2:

```
while true:
    turn2 = turn1 + 1
    while turn1 ≠ 0
        ∧ turn2 > turn1;;
    SC
    turn2 = 0
    sección no crítica
```

Ni las asignaciones en los protocolos de entrada ni la evaluación de las condiciones en las sentencias **while** son indivisibles:

- ambos procesos pueden colocar sus variables **turn** en 1,
... y luego ambos ven que sus variables **turn** no son mayores que la del otro proceso
(y entran a sus secciones críticas)

Si **turn1** y **turn2** son ambos 1, podríamos hacer que **P2** espere, cambiando su condición del **while** a

$$\mathbf{turn1} \neq 0 \wedge \mathbf{turn2} \geq \mathbf{turn1}$$

Pero consideremos el siguiente escenario (posible, ya que ni las asignaciones ni la evaluación de las condiciones son indivisibles):

- primero, **P1** lee **turn2** y obtiene 0
- luego, **P2** ve **turn1** aún en 0, coloca **turn2** en 1, y entra
- finalmente, **P1** coloca **turn1** en 1 y entra

turn1 = 0, turn2 = 0

Process P1:
while true:
 turn1 = 1
 turn1 = turn2 + 1
 while turn2 ≠ 0
 \wedge turn1 > turn2::
 SC
 turn1 = 0
 sección no crítica

Process P2:
while true:
 turn2 = 1
 turn2 = turn1 + 1
 while turn1 ≠ 0
 \wedge turn2 > turn1::
 SC
 turn2 = 0
 sección no crítica

Si el protocolo de entrada de cada proceso comienza asignando 1 a su variable **turn**, entonces es imposible que si ambos procesos quieren entrar a sus secciones críticas, ambos creen que la variable **turn** del otro proceso aún vale 0

```

turn[n] = {0 ... 0}
process P[i = 1 ... n]:
    while true:
        turn[i] = 1
        turn[i] = max(turn[1], ..., turn[n])+1
        for j = 1 ... n:
            if j ≠ i:
                while turn[j] ≠ 0
                    ∧ (turn[i],i) > (turn[j],j)::
                    SC
        turn[i] = 0
    sección no crítica

```

La asignación a **turn[i]** de **max(...)+1** no es indivisible, de modo que dos procesos pueden asignar el mismo valor a sus respectivos **turn[i]**

Esto se resuelve al hacer las comparaciones en el **while**: si los **turn[i]** de dos procesos son iguales, entonces se desempata según el número de proceso

El siguiente programa, compuesto por dos procesos, ¿terminará?

| <code>x = 10; c = true</code> | |
|---|---|
| <code>< await x == 0 ></code> <code>c = false</code> | <code>while c:</code> <code>< x = x-1 ></code> |

Solución. Llamemos P al proceso de la izquierda y Q al de la derecha.

La respuesta es no.

La acción $\langle x = x-1 \rangle$ de Q se va a ejecutar una y otra vez, mientras c permanezca *true*.

La única forma de que c se vuelva *false* es mediante la ejecución de $c = \text{false}$ en P , pero esto requiere la ejecución de la acción condicional $\langle \text{await } x == 0 \rangle$.

Sin embargo, esta ejecución está garantizada sólo si la condición $x == 0$, una vez que se vuelve verdadera, permanece verdadera (por lo menos, hasta que se ejecute $\langle \text{await } x == 0 \rangle$).

Pero esto no es cierto: puede ser que cuando $x == 0$, P no tenga la oportunidad de ejecutarse, pero sí Q , que decrementará el valor de x , volviendo nuevamente falsa la condición anterior.

Prueba que la siguiente solución al problema de la sección crítica para los procesos P y Q cumple las propiedades de
a) exclusión mutua; b) ausencia de *deadlock*; y c) entrada

| wantP = false, wantQ = false turn = 1 | |
|--|--|
| <p>P</p> <pre>while true: sección no crítica wantP = true while wantQ: if turn == 2: wantP = false while turn == 2;; wantP = true sección crítica turn = 2 wantP = false</pre> | <p>Q</p> <pre>while true: sección no crítica wantQ = true while wantP: if turn == 1: wantQ = false while turn == 1;; wantQ = true sección crítica turn = 1 wantQ = false</pre> |

Solución.

a) Resumidamente, la variable **turn** asegura la exclusión mutua; supongamos que vale 1. Si ambos procesos están tratando de entrar a sus secciones críticas, **wantP = wantQ = true**. Como **turn = 1**, Q hace **wantQ = false** y se queda esperando en el siguiente **while (turn == 1) { ... }** (de donde podrá salir sólo cuando P haga **turn = 2**, una vez que salga de su sección crítica).

Entonces, con **wantQ = false**, P termina de ejecutar su **while (wantQ): ...** y entra a su sección crítica, mientras Q espera.

b) Similarmente, **turn** asegura que no habrá *deadlock*. Sólo habría *dead-lock* si ambos procesos deben esperar indefinidamente en sus protocolos de entrada, lo que sólo es posible si **wantP = wantQ = true**; pero, entonces, dependiendo del valor de **turn**, exactamente uno de **wantP** y **wantQ** se volverá **false**.

c) **turn** también asegura entrada, por el mismo raciocinio de a). Si ambos procesos quieren entrar repetidamente, **turn** les da la entrada alternadamente. Si sólo un proceso quiere entrar, digamos Q, **wantP** estará en **false** y Q no tendrá impedimento para entrar.

Considera la sentencia atómica **exchange**, definida como el intercambio ininterrumpible de los valores de dos variables (al estilo de *Test and Set* y *Fetch and Add*):

```
exchange(a,b):  
    tmp = a  
    a = b  
    b = tmp
```

Escribe una solución al problema de la sección crítica empleando **exchange**

Solución.

| int c = 1 | |
|--|--|
| P | Q |
| <pre>local1 = 0 while true: <i>sección no crítica</i> exchange(c, local1) while local1 ≠ 1: exchange(c, local1) <i>sección crítica</i> exchange(c, local1)</pre> | <pre>local2 = 0 while true: <i>sección no crítica</i> exchange(c, local2) while local2 ≠ 1: exchange(c, local2) <i>sección crítica</i> exchange(c, local2)</pre> |