Modelo de Cómputo

Programación concurrente

Programa concurrente

- Programa concurrente: conjunto finito de *procesos (secuenciales)*.
- Un proceso secuencial está escrito con un conjunto finito de *acciones atómicas*.
- Ejecución: una secuencia de acciones atómicas obtenidas intercalando arbitrariamente acciones atómicas de los distintos programas
- Una computación es la ejecución de una secuencia (intercalada) de acciones.

Estados y transiciones

- Estado:
  - Análogo a la computación secuencial (modelo imperativo)
    - puntero o etiqueta de próxima instrucción
    - asignación de valores a variables
  - Estado de un programa con N procesos es una tupla que contiene:
    - N etiquetas de próxima instrucción (una para cada programa)
    - la asignación de valores a variables locales y globales (en general asumimos un lenguaje tipado)
- Transición: Existe una transición entre dos estados s_1 y s_2 si s_2 se obtiene ejecutando una de las próximas acciones de alguno de los programas.

Ejemplo

Considerar los threads

```
global x = 0

thread p
p1:    x = x + 1

thread q
q1:    x = x + 1
```

Diagrama de transición de estados

- Grafo dirigido (Estados, Transiciones) definido inductivamente de la siguiente manera
  - El estado inicial es un nodo del grafo
    - Cada etiqueta de próxima instrucción apunta a la primer acción del proceso correspondientes
    - La asignación a variables tienen el valor inicial (si está definido en el programa).
  - Si s_1 es un nodo del grafo y existe una transición entre s_1 y s_2, entonces s_2 está en el diagrama y el el arco (s_1,s_2) está en el grafo.

Ejemplo

Considerar los threads

```
global x = 0

thread p
p1:    x = x + 1

thread q
q1:    x = x + 1
```

¿Cuál es el diagrama de transición de estados?
Notar que en general puede no ser finito. ¿Cuándo?

Atomicidad

Considerar ahora los threads

```
global x = 0

thread p
temp
p1:    temp = x
p2:    x = temp + 1

thread q
q1:    temp = x
q2:    x = temp + 1
```

Un algoritmo concurrente es correcto dependiendo de cuáles son las acciones atómicas

Importante: Vamos a asumir que las acciones atómicas son: lecturas y asignaciones de valores a variables y la evaluación de expresiones booleanas en las estructuras de control.

Definiciones

Sección crítica

Llamamos sección crítica a una parte del programa que no puede ser ejecutada concurrentemente con otra sección critica del mismo programa.

Exclusión mutua

Llamamos exclusión mutua al problema de asegurar que dos (o mas) threads no ejecutan simultáneamente su sección crítica.

Esquema general

Existen N procesos que tienen la siguiente estructura

```
shared variables

thread id = i
while(true){
    seccion no critica
    preprotocol
    seccion critica
    postprotocol
}
```

- No hay variables compartidas entre sección crítica y no crítica.
- La sección crítica siempre termina.
- La no crítica no necesariamente termina.

Requerimientos de la exclusión mutua

1. **Mutex:** En cualquier momento hay como máximo un proceso en la región crítica.
2. **Ausencia de deadlocks y livelocks:** Si varios procesos intentan entrar a la sección crítica alguno lo logrará.
3. **Garantía de entrada:** Un proceso intentando entrar a su sección crítica tarde o temprano lo logrará.

Pregunta

¿Podemos resolver el problema de la exclusión mutua para dos procesos asumiendo que las únicas operaciones atómicas son la lectura y la escritura en de variables?

Algoritmo I

```
shared flag = false
```

```
thread id = 0
 while(true){
  // seccion no critica
  while (flag);
  flag = true
  // seccion critica
  flag = false
 }
```

```
thread id = 1
  while(true){
  // seccion no critica
  while (flag);
  flag = true
  // seccion critica
  flag = false
       }
```

- **Mutex:** No
- **Ausencia dead/live-locks:** Sí
- **Garantía de entrada:** Sí

Algoritmo II

```
shared flag = {false, false}
```

```
thread id = 0
 while(true){
  // seccion no critica
  otro = (id + 1) % 2
  flag[id] = true
  while (flag[otro]);
  // seccion critica
  flag[id] = false
}
```

```
thread id = 1
 while(true){
  // seccion no critica
  otro = (id + 1) % 2
  flag[id] = true
  while (flag[otro]);
  // seccion critica
  flag[id] = false
        }
```

- **Mutex:** Sí
- **Ausencia dead/live-locks:** No
- **Garantía de entrada:** No

Algoritmo III

```
shared turno = 0
```

```
thread id = 0
 while(true){
  // seccion no critica
  while (turno != id);
  // seccion critica
  turno = (id + 1) % 2
  // seccion no critica
 }
```

```
thread id = 1
  while(true){
  // seccion no critica
  while (turno != id);
  // seccion critica
  turno = (id + 1) % 2
  // seccion no critica
        }
```

- **Mutex:** Sí
- **Ausencia dead/live-locks:** Sí
- **Garantía de entrada:** No (si el thread de id 0 falla)

Algoritmo de Dekker (II + III)

```
shared turno = 0
shared flag  = {false, false}


thread id = 0                          thread id = 1
  // seccion no critica                  ...
  otro = (id + 1) % 2
  flag[id] = true
  while (flag[otro])
    if (turno == otro)
      flag[id] = false
      while (turno != id);
      flag[id] = true

  // seccion critica

  turno = otro
  flag[id] = false
  // seccion no critica
```

Algoritmo de Peterson

```
shared turno = 0
shared flag  = {false, false}


thread id = 0
  // seccion no critica
  otro = (id + 1) % 2
  flag[id] = true
  turno = otro
  while (flag[otro] && turno == otro);

  // seccion critica

  flag[id] = false
  // seccion no critica
```

Dekker y Peterson

- **Mutex:** Sí
- **Ausencia dead/live-locks:** Sí
- **Garantía de entrada:** Sí

Sólo sirven para dos procesos.

Algoritmo de Bakery

```
shared entrando[n] = {false, ..., false}
shared numero[n]   = {0, ..., 0}


thread id = 0
  // seccion no critica
  entrando[id] = true
  numero[id] = 1 + max(numero[1], ..., [n])
  entrando[id] = false
  for (j = 1; j <= n; ++j)
    while (entrando[j]);
    while (numero[j] != 0 && (numero[j] < numero[id] ||
      (numero[j] == numero[id] && j < id)));

  // seccion critica

  numero[id] = 0
  // seccion no critica
```

Este algoritmo resuelve el problema para *n threads*.

Pregunta

¿Qué otras acciones atómicas pueden idearse para resolver el problema de la exclusión mutua?

Test and set

```
function test-and-set(ref comp, ref local)
    local = comp
    comp  = 1
```

```
shared comp = 0
```

```
thread id = 0
  int local
  // seccion no critica
  repeat
    test-and-set(comp, local)
  until (local == 0)
  // seccion critica
  comp = 0
  // seccion no critica
```

```
thread id = 1
  int local
  // seccion no critica
  repeat
    test-and-set(comp, local)
  until (local == 0)
  // seccion critica
  comp = 0
  // seccion no critica
```

Exchange

```
function exchange(ref comp, ref local)
    temp  = comp
    comp  = local
    local = temp
```

```
shared comp = 0
```

```
thread id = 0
  int local = 1
  // seccion no critica
  repeat
    exchange(comp, local)
  until (local == 0)
  // seccion critica
  comp = 0
  // seccion no critica
```

```
thread id = 1
  int local = 1
  // seccion no critica
  repeat
    exchange(comp, local)
  until (local == 0)
  // seccion critica
  comp = 0
  // seccion no critica
```

Compare-and-swap

```
function compare-and-swap(ref comp, ref viejo, ref nuevo)
    temp = comp
    if (comp == viejo)
        comp = nuevo
    return temp
```

```
shared comp = false
```

```
thread id = 0
  // seccion no critica
  while( compare-and-swap(
    comp, false, true) );
  // seccion critica
  comp = false
  // seccion no critica
```

```
thread id = 1
  // seccion no critica
  while( compare-and-swap(
    comp, false, true) );
  // seccion critica
  comp = false
  // seccion no critica
```

Fetch-and-add

```
function fetch-and-add(ref comp, ref local, ref x)
    local = comp
    comp  = comp + x
```

```
shared ticket = 0
shared turno  = 0
```

```
thread id = 0
  int miTurno
  // seccion no critica
  fetch-and-add(
    ticket, miTurno, 1)
  while (turno != miTurno);
  // seccion critica
  fetch-and-add(
    turno, miTurno, 1)
  // seccion no critica
```

```
thread id = 1
  int miTurno
  // seccion no critica
  fetch-and-add(
    ticket, miTurno, 1)
  while (turno != miTurno);
  // seccion critica
  fetch-and-add(
    turno, miTurno, 1)
  // seccion no critica
```

Busy waiting

Todas las soluciones vistas en esta clase son ineficientes dado que consumen tiempo de procesador en las esperas.

Sería deseable suspender la ejecución de un proceso que intenta acceder a la sección crítica hasta tanto sea posible.