Teoría de Lenguajes
Teoría de la Programación

Clase 5: Concurrencia declarativa

Concurrencia

Actividades independientes

La ejecución no es secuencial, sino que se va solapando

No es en paralelo => otro procesador

Concurrencia declarativa

Determinístico

Se hacen cálculos en forma incremental

Ejemplo

```
local X0 X1 X2 X3 in
    thread X1=1 + X0 end
    thread X3=X1 + X2 end
    X0=4
    X2=2
    {Browse [X0 X1 X2 X3]}
end
```

Threads

Hilo independiente de ejecución

Tipos de threads dependiendo el contexto

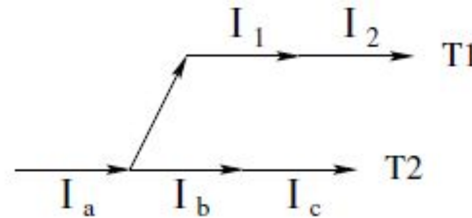
- Heavyweight: diferente proceso de SO
- Middleweight: mismo proceso, kernel level
- **Lightweight: User level - green threads**

Interleavings - Order

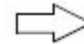
Los threads se ejecutan en forma intercalada (interleaving). No se solapan los computos

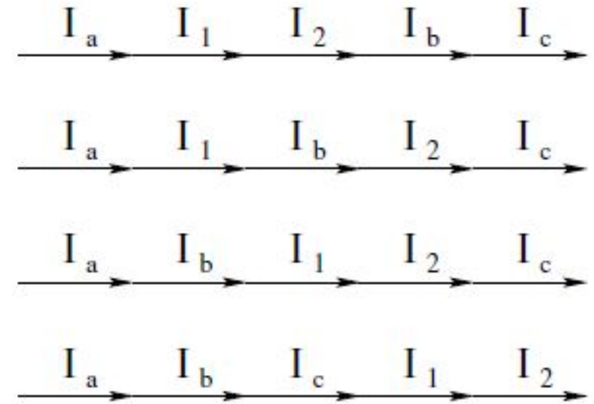
Total order

Causal order



Causal order





Some possible executions

Scheduler

Un scheduler define cual es el próximo thread a ejecutarse

Un thread al igual que un programa puede tener estados

- Listo
- Suspendido

El scheduler debe ser justo: Evitar starvation

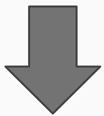
Sintaxis

Tenemos un nuevo statement válido

```
thread <s> end
```

Máquina abstracta

(ST,σ)



(MST,σ)

MST = $\{ST_1, ST_2, \ldots, ST_N\}$

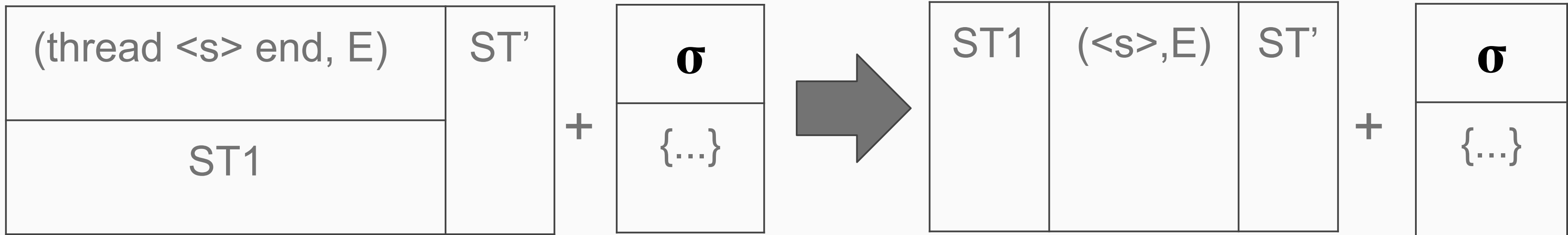
$$\underbrace{(\{\underbrace{[\ \overbrace{(\langle s \rangle, \phi)}^{\text{statement}}\ ]}_{\text{stack}}\},}_{\text{multiset}}\ \phi)$$

Semántica

En el tope de un ST tenemos el siguiente semantic statement (thread <s> end, E)

Se crea un nuevo ST que se agrega al MST con el tope de pila (<s>,E)

El σ sigue siendo único



Estados - Manejo de memoria

Un ST finalizado puede ser eliminado

Un ST bloqueado puede ser eliminado si la condición de activación es inalcanzable

Cheap concurrency & dataflow

Ejercicio

```
Hacer un Map pero que funcione en forma concurrente
```

Técnicas de programación concurrente

Problema

Programar un ejemplo de un ejercicio tipo productor consumidor

El productor ira generando numeros y poniendolos en una lista. Con un límite de números

El Consumidor irá recibiendo los números e imprimiendo solo los pares en pantalla

Problema

```
Pensemos a la función Produce como
un procedure
```

Streams

Técnica más usada para la programación concurrente declarativa.

Comunicación entre threads utilizando listas con un final sin ligar.

```
local Xs1 Xs2 Xs3 in
   Xs1 = 1|2|3|Xs2
   {Browse Xs1}
   Xs2 = 4|5|6|Xs3
   Xs3 = 7|8|9
end
```

Posibles inconvenientes

Problema 1: El que produce, lo hace mucho más rápido que el que consume.

Solución: Demand driven concurrency

Problema 2: Por utilizar demand driven no estoy aprovechando del todo la capacidad del productor, recién cuando necesito otro elemento lo produzco.

Solución: Bounded buffer

Coroutines

Nonpreemptive threads

Funciona sin scheduler

Cada corrutina tiene la responsabilidad de delegar el control

Lazy evaluation / execution

Ejemplo

```
local F1 F2 F3 A B C D in
   fun lazy {F1 X} X*X  end
   fun lazy {F2} 125 end
   fun lazy {F3 X Y} X+Y end
   A = {F1 4}
   B = {F2}
   C = {F3 A B}
   D = A + B
   {Browse 'D:'#D}
   {Browse 'C:'#C}
end
```

Necesitar una variable

Una variable es necesitada si:

- Por no estar determinada suspende la ejecución de algún statement
- Ya está determinada

Una vez necesitada -> Siempre necesitada

Ejemplo II

```
local F1 A in
    fun lazy {F1}
     {Browse 'En F1'}
     125
    end
    A = {F1}
    A = 100
    {Browse 'A:'#A}
end
```

¿Que sucede en este caso?

Sintaxis

By Need triggers

Se agrega un nuevo statement válido:

Trigger creation

```
{ByNeed <x> <y> }
```

Variable en la cual se guardará el valor

Procedimiento por el cual se calcula el valor

Sintaxis

By Need triggers

```
{ByNeed <x> <y> }
```

Trigger creation:

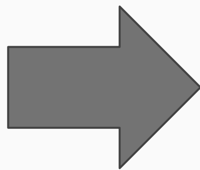
Cuando se necesite el valor será como ejecutar

```
thread <y>={<x>} end
```

```
thread {<x> <y>} end
```

By Need triggers

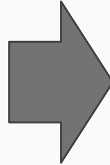
```
local F1 A B in
    fun lazy {F1} 125 end
    A = {F1}
    B = A+10
    {Browse B}
end
```



```
local F1 A B in
    proc {F1 X} X=125 end
    {ByNeed F1 A}
    B = 10+A
    {Browse B}
end
```

By Need triggers

```
local F1 A B in
    fun lazy {F1 X Y} X*Y end
    A ={F1 2 4}
    B = A + 10
    {Browse B}
end
```



```
local F1 A B in
    proc {F1 X Y Z}
     local T in
       T = proc{$ R} R=X*Y end
       {ByNeed T Z}
     end
    end
    {F1 2 4 A}
    B = A + 10
    {Browse B}
end
```

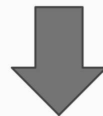
Semántica

Trigger store

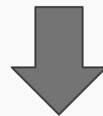
τ

Además del single-assignment store (σ), aparece un nuevo store, trigger store (τ)

(ST,σ)



(MST,σ)



(MST,σ,τ)

Semántica - Trigger creation

En el tope tenemos el siguiente semantic statement
({ByNeed <x> <y>}, E)

- Si E(<y>) no está determinada. Se agrega el par: *trig(E(<x>),E(<y>))* a τ
- Si E(<y>) está determinada, se crea un nuevo thread con el siguiente semantic statement en el tope *({<x> <y>},E)*

Semántica - Trigger activation

Existe un par en τ con la forma *trig(x,y)* y se detecta la necesidad de *y*

- Se saca el trigger del store τ
- Se crea un nuevo thread con el siguiente semantic statement en el tope *({<x> <y>},{<x>->x, <y>->y})*

Ejemplo

```
local Generate L in
   fun lazy {Generate N} N|{Generate N+1} end
   L = {Generate 1}
   {Browse L.2.2.1}
end
```

Ejemplo - A lenguaje Kernel

```
local PGen L in
   PGen = proc{$ N S}
       local Trig in
           Trig = proc{ $ Res}
                   local A in
                       Res = N|A
                       {PGen N+1 A}
                   end
                 end
           {ByNeed Trig S}
       end
   end
   {PGen 1 L}
   {Browse L.2.2.2.2.1}
end
```

Bibliografía

- **Concepts, Techniques, and Models of Computer Programming - Capítulo 4**, Peter Van Roy and Seif Haridi
- **Extras:**
  - **Principles of Concurrent and Distributed Programming,** M. Ben-Ari.