Taller de Programación Concurrente Declarativa

Modelos y Paradigmas de Programación

Escuela de Ingeniería de Sistemas y Computación

Universidad del Valle

Profesor Juan Francisco DIAZ FRIAS*

A continuación Usted encontrará una serie de ejercicios que debe resolver **individualmente**. No se admite ninguna consulta con sus compañeros de otros grupos de trabajo, pero sí con el profesor. Esa es la hipótesis básica sobre la cual se fundamenta esta tarea. Cualquier violación a esta regla será considerada un intento de copia y causará la anulación del taller.

La solución a todos y cada uno de los puntos del taller se debe almacenar en un archivo `.oz` tal como se describe en cada punto. Usted debe entregar como solución al examen, el conjunto de esos archivos empaquetado y comprimido en un solo archivo cuyo nombre debe ser análogo a `taller2JFDiaz.zip` o `taller2JFDiaz.tgz` o `taller2JFDiaz.tar.gz` o `taller2JFDiaz.rar`. Al desempaquetar este archivo, debo encontrar solamente los archivos siguientes: `sum_impares.oz`, `contador.oz`, `foobar.oz`, `logicaDigital.oz`, `terminacion.oz` y `primosPereza.oz`, con el contenido descrito abajo. Adicionalmente debe entregar un archivo `practica1.pdf`, con el desarrollo del taller que no corresponde a código Oz.

No envíe ningún archivo adicional, así lo haya utilizado para sus pruebas. Su solución debe ser enviada a través del Campus Virtual en el enlace correspondiente al Taller de Programación Concurrente Declarativa según las fechas indicadas allí. El sistema no permitirá que se envíen soluciones después de dicha fecha.

1. Explique lo que sucede cuando ejecuta el siguiente programa en *MOzArt* .

*juanfco.diaz@correounivalle.edu.co

```
declare A B C D in
thread D=C+1 end
thread C=B+1 end
thread A=1 end
thread B=A+1 end
{Browse D}
```

- En qué orden son creados los hilos?
- En qué orden son evaluados los hilos?

2. *Comportamiento de flujo de datos en un contexto concurrente.* Considere la función `{Filter En F}`, que devuelve los elementos de `En` para los cuales la función booleana `F` devuelve **`true`**. Esta es una posible definición de `Filter`:

```
fun {Filter En F}
  case En
  of X|En2 then
    if {F X} then X|{Filter En2 F}
    else {Filter En2 F} end
  else
    nil
  end
end
```

Al ejecutar la instrucción siguiente:

```
{Show {Filter [5 1 2 4 0] fun {$ X} X>2 end}}
```

se despliega

```
[5 4]
```

(Usamos el procedimiento `Show`, el cual despliega el valor instantáneo de su argumento en la ventana del emulador de Mozart. Es diferente al `Browse`, pues la salida de `Show` no se actualiza si el argumento se liga posteriormente.) Entonces `Filter` se comporta como se espera, en el caso de una ejecución secuencial y que todos los valores de entrada están disponibles. Ahora exploremos el comportamiento de flujo de datos de `Filter`.

a) ¿Qué pasa cuando se ejecuta lo siguiente?:

```
declare A
{Show {Filter [5 1 A 4 0] fun {$ X} X>2 end}}
```

Uno de los elementos de la lista es una variable `A` que no ha sido ligada aún a ningún valor. Recuerde que las declaraciones **`case`** e **`if`** suspenderán el hilo en que ellas se ejecutan, hasta que puedan decidir qué alternativa tomar.

b) ¿Qué pasa cuando se ejecuta lo siguiente?:

```
declare Sal A
thread Sal={Filter [5 1 A 4 0] fun {$ X} X>2 end} end
{Show Sal}
```

Recuerde que invocar `Show` despliega su argumento tal como esté al momento de la invocación. Se pueden desplegar diversos resultados. ¿Cuáles y por qué? ¿La función `Filter` es determinística? ¿Por qué sí o por qué no?

c) ¿Qué pasa cuando se ejecuta lo siguiente?:

```
declare Sal A
thread Sal={Filter [5 1 A 4 0] fun {$ X} X>2 end} end
{Delay 1000}
{Show Sal}
```

Recuerde que la invocación `{Delay N}` suspende su hilo por `N` ms al menos. Durante este tiempo, otros hilos listos pueden ser ejecutados.

d) ¿Qué pasa cuando se ejecuta lo siguiente?:

```
declare Sal A
thread Sal={Filter [5 1 A 4 0] fun {$ X} X>2 end} end
thread A=6 end
{Delay 1000}
{Show Sal}
```

¿Qué se despliega y por qué?

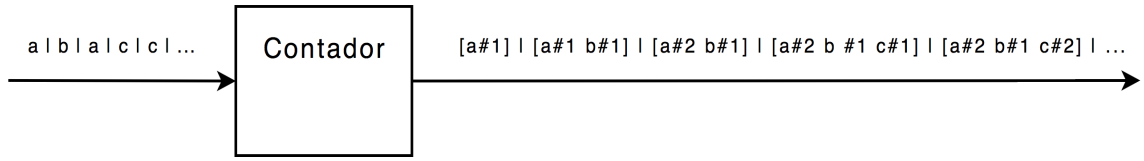
3. Implemente un programa que asincrónicamente genere un flujo de 10000 enteros, filtre los números impares del flujo y sume los elementos del flujo resultante. El productor, el filtro y el consumidor deben ser ejecutados en hilos independientes.

   La siguiente figura describe el algoritmo, donde `Productor` es el proceso que genera el flujo de enteros y `Consumidor` el proceso que suma los elementos filtrados.

   Su solución debe guardarla en un archivo de nombre `sum_impares.oz`.

4. Implemente un programa que registre las ocurrencias de caracteres en un flujo de entrada. El programa debe generar un flujo de salida donde cada posición del flujo corresponde al total de ocurrencias hasta ese momento. Cada posición del flujo de salida es una lista de tuplas `C#N`, donde `N` es el número de veces que `C` aparece en el flujo de entrada.

   Ejemplo:

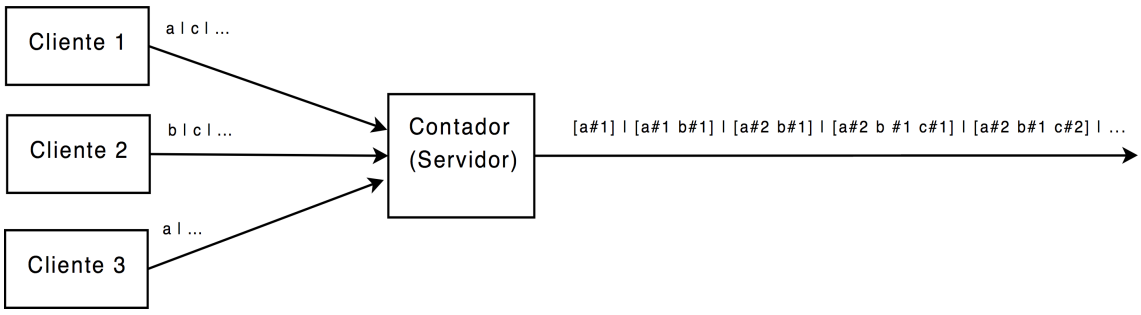


El `Contador` debe ser un proceso concurrente. Si las siguientes líneas son ejecutadas:

```
local
   InS
in
   {Browse {Counter InS}}
   InS=a|b|a|c|_
end
```

Debería aparecer en el browser `[a#1]|[a#1 b#1]|[a#2 b#1]|[a#2 b#1 c#1]|_`.

Ahora, piense en este contador como un servidor recibiendo la entrada de muchos clientes (ver siguiente figura). Implemente el servidor y los clientes en el modelo concurrente declarativo. Puede empezar implementando solo 2 clientes.



- Que sucede si uno de los clientes deja de enviar caracteres?
- Puede extender la solución a N clientes?

Su solución debe guardarla en un archivo de nombre `contador.oz`.

5. Foo fue un estudiante legendario famoso por su habilidad para beber grandes cantidades de cerveza. En efecto, nadie conoció su límite real. Además, Foo era capaz de beber una pinta (aproximadamente medio litro de cerveza) en 12 segundos.

   Bar fue un renombrado mesero capaz de servir una cerveza en 3 segundos. Las sesiones de bebida de Foo eran realizadas de la siguiente manera. Bar permanentemente servía cervevaz a Foo durante una hora. Foo bebía una cerveza tras otra. Para evitar que la cerveza se calentara, Bar nunca ponía más de 5 cervezas en la mesa al mismo tiempo.

Modele este escenario como un algoritmo productor/consumidor. Explique por que es necesario utilizar memoria intermedia limitada.

Su solución debe guardarla en un archivo de nombre `foobar.oz`.

6. *Simulación de la lógica digital.* Una compuerta lógica es una función concurrente que toma uno o más flujos de valores binarios como entrada, aplica una función lógica a la entrada y retorna el flujo resultante como salida.

   a) La compuerta lógica más simple es la compuerta Not, la cual toma un flujo como entrada, y niega cada uno de sus elementos. Implemente la `CompuertaNot` comenzando por definir la función auxiliar `Not` que devuelve el negativo de un elemento.

   b) Implemente las compuertas lógicas And y Or. Ambas reciben como entrada un dos flujos y devuleven del resultado de aplicar las operaciones lógicas And y Or a cada pareja de las entradas.

   c) Las compuertas lógicas pueden ser representadas con registros de la forma:

   ```
   gate(value:<logic operator> <input 1> ... <input n>)
   ```

   Nótese que la cantidad de entradas varian de acuerdo al operador lógico. Las entradas pueden ser representadas como otras compuertas lógicas o como `input(v)`, donde `v` corresponde a uno de los flujos de entrada. Una forma de escribir la `CompuertaNot` usando registros es la siguiente:

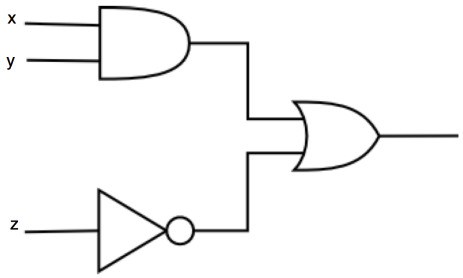
   ```
   gate(value:'not' input(x))
   ```

   Implemente una función `{Simulate G Ss}` que retorna la salida del circuito correspondiente a `G` bajo la entrada `Ss`.

   Ejemplo. Asuma que G está ligado al siguiente registro:

   ```
   gate(value:'or'
        gate(value:'and'
             input(x)
             input(y))
        gate(value:'not'
             input(z)))
   ```

   que corresponde al siguiente circuito:

Un llamado a `{Simulate G input(x:1|0|1|0|_ y:0|1|0|1|_ z:1|1|0|0|_ )}` debería retornar `0|0|1|1|_`. Note que `Simulate` debe ser un proceso concurrente.

d) Utilice la función `Simulate` para implementar el sumador completo descrito en la sección 4.3.5.1 de CTM.

Su solución a este ejercicio debe ser entregada en el archivo `logicaDigital.oz`.

7. Considere el siguiente código *MOzArt* :

```
declare
L1 L2 L3 L4
L1 = [1 2 3]
thread L2 = {Map L1 fun {$ X} {Delay 200} X*X end} end
thread L3 = {Map L1 fun {$ X} {Delay 200} 2*X end} end
thread L4 = {Map L1 fun {$ X} {Delay 200} 3*X end} end
{Show L2#L3#L4}
```

El problema es que si `Show` es llamado antes de que todos los hilos hayan finalizado, el resultado no aparecera en el emulador, porque `L2`, `L3` y `L4` no estarán ligados con la lista resultante. Use la función `{Wait X}` que espera hasta que la variable `X` esté ligada a un valor, así el llamado a `{Show L2#L3#L4}` solo será ejecutado cuando todos los hilos hayan finalizado su ejecución.

Su solución a este ejercicio debe ser entregada en el archivo `terminacion.oz`.

8. **Pereza**

a) Escriba una función `{Gen I}` que genera perezosamente una lista de todos los enteros comenzando en `I`.

b) Implemente una función `{Primes}` que perezosamente devuelva la lista infinita de los números primos en orden ascendente. Con el fín de implementar `Primes` escriba una versión perezosa de`Filter` y de la criba de Eratóstenes mostrada en el capítulo 4 de CTM. Genere una lista infinita ascendente de números naturales usando la función `Gen` para alimentar la criba.

c) Implemente una función `{ShowPrimes N}` que use `{Primes}` para mostrar los primeros `N` números primos.

Su solución a este ejercicio debe ser entregada en el archivo `primosPereza.oz`.

9. Este ejercicio explora un poco más el comportamiento concurrente de la ejecución perezosa. Ejecute lo siguiente:

```
fun lazy {HagaX} {Browse x} {Delay 3000} 1 end
fun lazy {HagaY} {Browse y} {Delay 6000} 2 end
fun lazy {HagaZ} {Browse z} {Delay 9000} 3 end
X={HagaX}
Y={HagaY}
Z={HagaZ}
{Browse (X+Y)+Z}
```

Esto despliega `x` y `y` inmediatamente, `z` después de seis segundos, y el resultado, 6, después de 15 segundos. Explique este comportamiento. ¿Qué pasa si reemplazamos `(X+Y)+Z` por `X+(Y+Z)` o por **`thread`** `X+Y` **`end`** `+ Z`? ¿En cuál forma se presenta más rápido el resultado final? ¿Cómo programaría la suma de n números enteros $i_1, \ldots, i_n$, sabiendo que el entero i_j estará disponible sólo después de t_j ms, de manera que el resultado final se produzca lo más rápido posible?