

# Programming Paradigms

## Lecture 9

---

**Slides are from Prof. Chin Wei-Ngan from NUS**

Declarative Concurrency

# Reminder of Last Lecture

- Programming techniques
  - Types
  - Abstract data types
  - Haskell
  - Design methodology : functors + modules

# Overview

- Declarative concurrency
- Mechanisms of concurrent program
- Streams
- Demand-driven execution
  - execute computation, if variable needed
  - needs suspension by a thread
  - requested computation is running in new thread
- By-Need triggers
- Lazy functions

# The World is Concurrent!

- Concurrent programs
  - several activities execute simultaneously (concurrently)
- Most of the software used are concurrent
  - ❑ operating system: IO, user interaction, many processes, ...
  - ❑ web browser, Email client, Email server, ...
  - ❑ telephony switches handling many calls
  - ❑ ...

# Why Should We Care?

- Software must be concurrent...
  - ... for many application areas
- Concurrency can be helpful for constructing programs
  - organize programs into independent parts
  - concurrency allows to make them independent with respect to how to execute
  - essential: how do concurrent programs interact?
- Concurrent programs can run faster on parallel machines (including clusters and cores)

# Concurrency and Parallelism

- **Concurrency** is *logically simultaneous processing* which can also run on sequential machine.
- **Parallelism** is *physically simultaneous processing* and it involves multiple processing elements and/or independent device operations.
- A **computer cluster** is a group of connected computers that work together as a unit. One popular implementation is a cluster with nodes running Linux with support library (for parallelism).

# Concurrent Programming is Difficult...

- This is the traditional belief
- The truth is: concurrency is *very* difficult...
  - ... if used with inappropriate tools and programming languages
- Particularly troublesome : *state* and *concurrency*

# Concurrent Programming is Easy...

- Oz (as well as Erlang) has been designed to be very good at concurrency...
- Essential for concurrent programming here
  - data-flow variables
    - very simple interaction between concurrent programs, mostly automatic
  - light-weight threads

# Declarative Concurrent Programming

- What stays the same
  - the result of your program
  - concurrency does not change the result
- What changes
  - programs can compute incrementally
  - incremental input... (such as reading from a network connection) ... and incremental processing

# Threads

# Our First Concurrent Program

```
declare X0 X1 X2 X3  
thread X1 = 1 + X0 end  
thread X3 = X1 + X2 end  
{Browse [X0 X1 X2 X3]}
```

- Browser will show [X0 X1 X2 X3]
  - variables are not yet assigned

# Our First Program

```
declare X0 X1 X2 X3  
thread X1 = 1 + X0 end  
thread X3 = X1 + X2 end  
{Browse [X0 X1 X2 X3] }
```

- Both threads are suspended
  - $X1 = 1 + X0$  suspended;  $X0$  unassigned
  - $X3 = X1 + X2$  suspended;  $X1, X2$  unassigned

# Our First Program

```
declare X0 X1 X2 X3  
thread X1 = 1 + X0 end  
thread X3 = X1 + X2 end  
{Browse [X0 X1 X2 X3]}
```

- Feeding                    X0 = 4

# Our First Program

```
declare X0 X1 X2 X3  
thread X1 = 1 + X0 end  
thread X3 = X1 + X2 end  
{Browse [X0 X1 X2 X3]}
```

- Feeding                     $X0 = 4$ 
  - First thread can execute, binds  $X1$  to 5

# Our First Program

```
declare X0 X1 X2 X3  
thread X1 = 1 + X0 end  
thread X3 = X1 + X2 end  
{Browse [X0 X1 X2 X3]}
```

- Feeding                    X0 = 4
  - First thread can execute, binds X1 to 5
  - Browser shows [ 4 5 X2 X3 ]

# Our First Program

```
declare X0 X1 X2 X3  
thread X1 = 1 + X0 end  
thread X3 = X1 + X2 end  
{Browse [X0 X1 X2 X3]}
```

- Second thread is still suspended
  - Variable  $x_2$  is still not assigned

# Our First Program

```
declare X0 X1 X2 X3  
thread X1 = 1 + X0 end  
thread X3 = X1 + X2 end  
{Browse [X0 X1 X2 X3]}
```

- Feeding                 $X2 = 2$ 
  - Second thread can execute, binds  $X3$  to 7
  - Browser shows [4 5 2 7]

# Threads

- A **thread** is simply an executing program.
- A program can have more than one thread.
- A thread is created by :  
`thread <s> end`
- Threads compute
  - independently
  - as soon as their statements can be executed
  - interact by binding variables in store

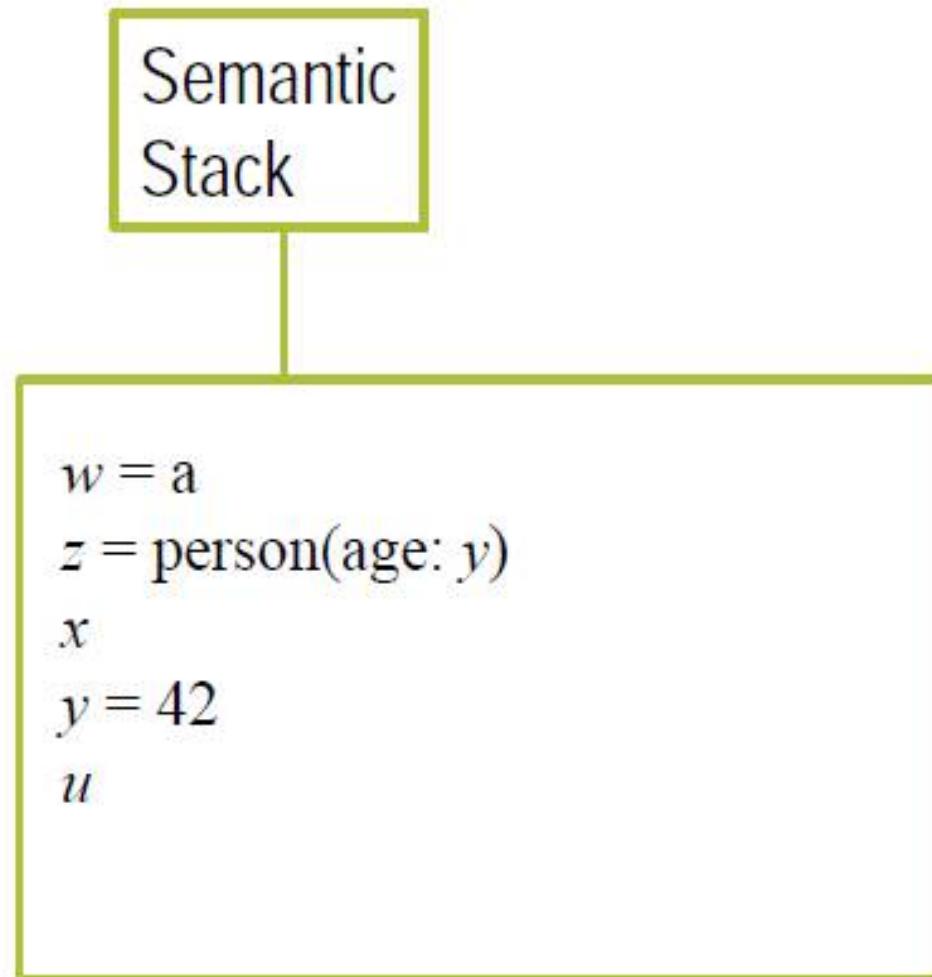
# The Browser

- Browser is implemented in Oz as a thread.
- It also runs whenever browsed variables are bound
- It uses some extra functionality to look at unbound variables

# Sequential Model

Statements are executed sequentially from a single semantic stack

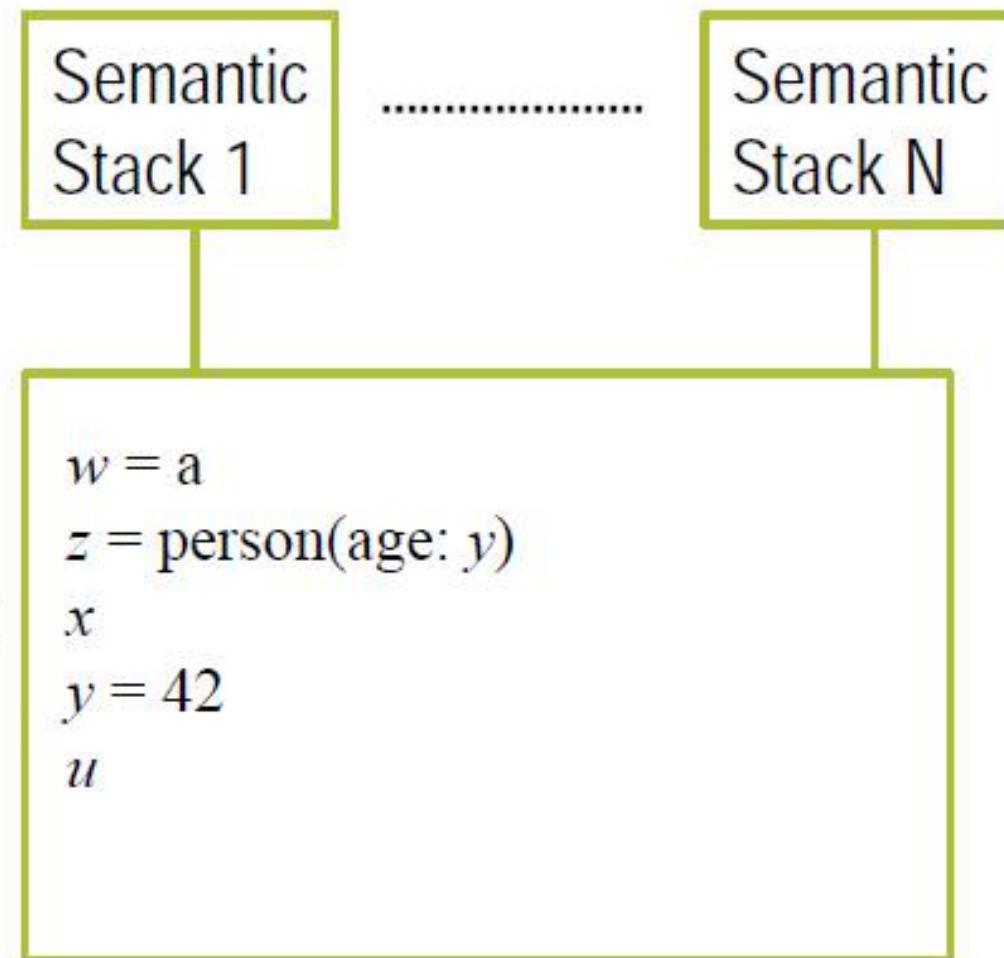
Single-assignment store



# Concurrent Model

Multiple semantic  
stacks (threads)

Single-assignment  
store



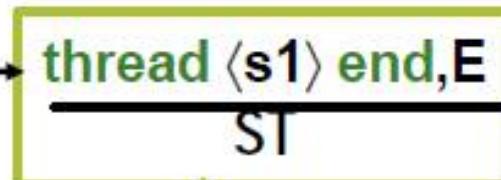
# Concurrent Declarative Model

Kernel language extended with thread creation

$\langle S \rangle ::=$	<b>skip</b>	<i>empty statement</i>
	$\langle X \rangle = \langle Y \rangle$	<i>variable-variable binding</i>
	$\langle X \rangle = \langle V \rangle$	<i>variable-value binding</i>
	$\langle S_1 \rangle \langle S_2 \rangle$	<i>sequential composition</i>
	<b>local</b> $\langle x \rangle$ <b>in</b> $\langle s_1 \rangle$ <b>end</b>	<i>declaration</i>
	<b>proc</b> $\{ \langle x \rangle \langle y_1 \rangle \dots \langle y_n \rangle \}$ $\langle s_1 \rangle$ <b>end</b>	<i>procedure introduction</i>
	<b>if</b> $\langle x \rangle$ <b>then</b> $\langle s_1 \rangle$ <b>else</b> $\langle s_2 \rangle$ <b>end</b>	<i>conditional</i>
	$\{ \langle x \rangle \langle y_1 \rangle \dots \langle y_n \rangle \}$	<i>procedure application</i>
	<b>case</b> $\langle x \rangle$ <b>of</b> $\langle \text{pattern} \rangle$ <b>then</b> $\langle s_1 \rangle$ <b>else</b> $\langle s_2 \rangle$ <b>end</b>	<i>pattern matching</i>
	<b>thread</b> $\langle s_1 \rangle$ <b>end</b>	<b><i>thread creation</i></b>

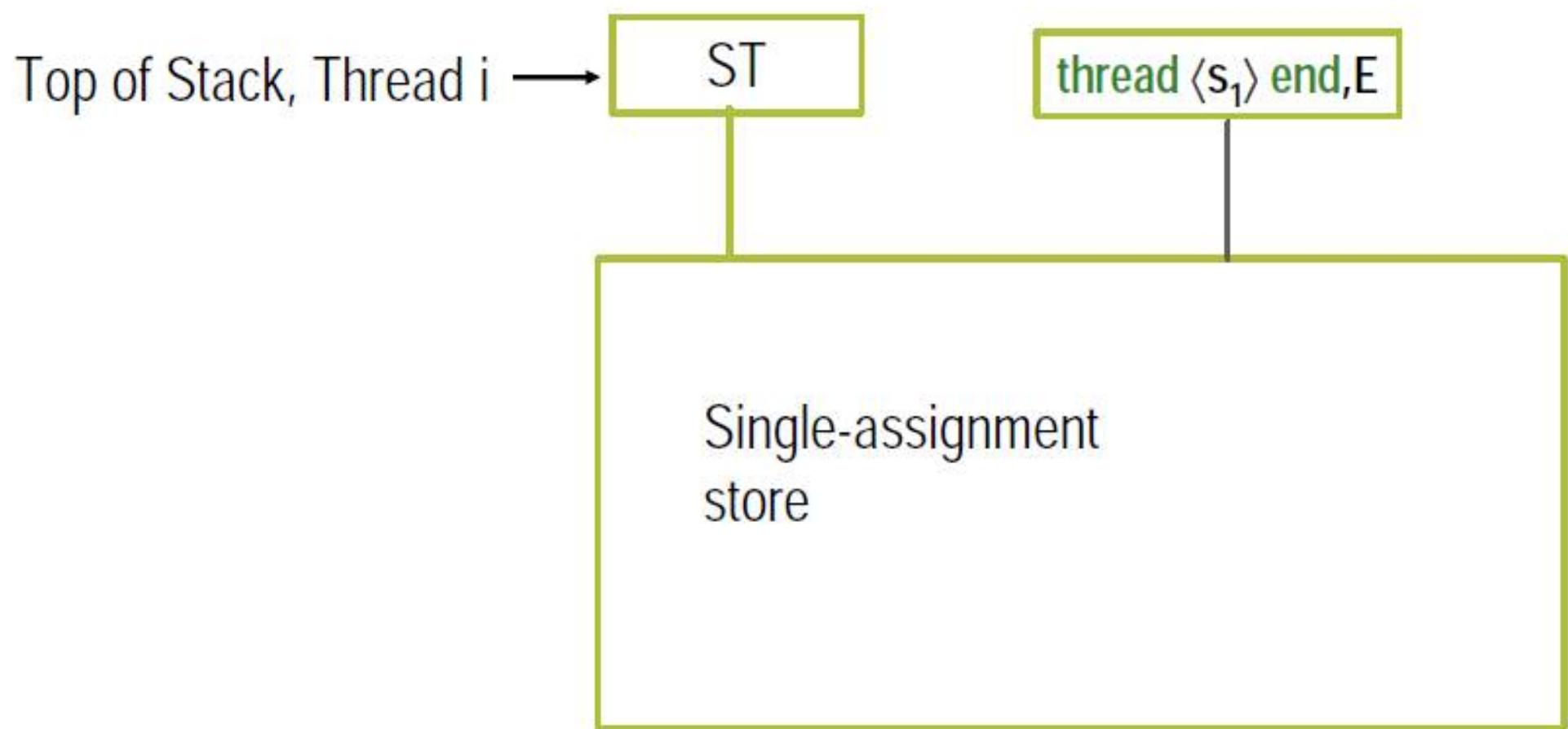
# The Concurrent Model

Top of Stack, Thread i →



Single-assignment  
store

# The Concurrent Model



# Basic Concepts

- Model allows multiple statements to execute "*simultaneously*" ?
- Can imagine that these threads really execute in parallel, each has its own processor, but share the same memory
- Reading and writing different variables can be done simultaneously by different threads
- Reading the same variable can be done *concurrently*.
- Writing to the same variable to be done *sequentially*.

# Causal Order

- In a sequential program, all execution states are *totally ordered*
- In a concurrent program, all execution states *of a given thread* are totally ordered
- But, ... the execution state of the concurrent program as a whole is **partially ordered**

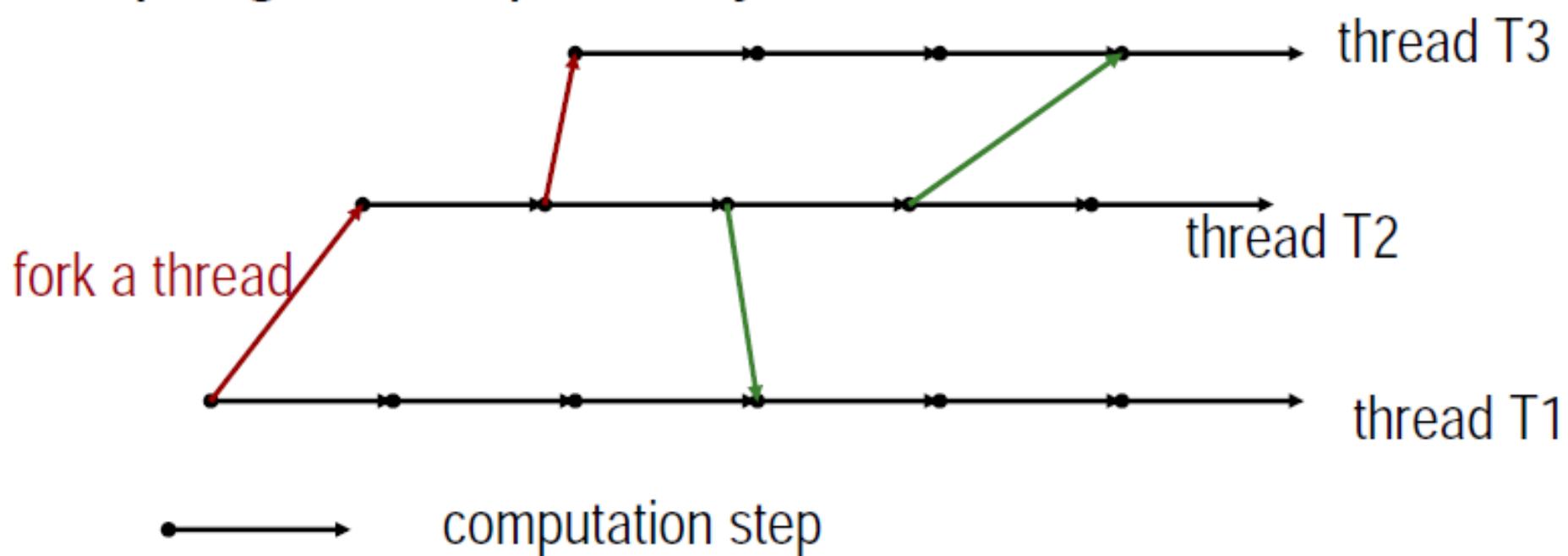
# Total Order

- In a sequential program all execution states are *totally ordered*
- Computation step: transition between two consecutive execution states

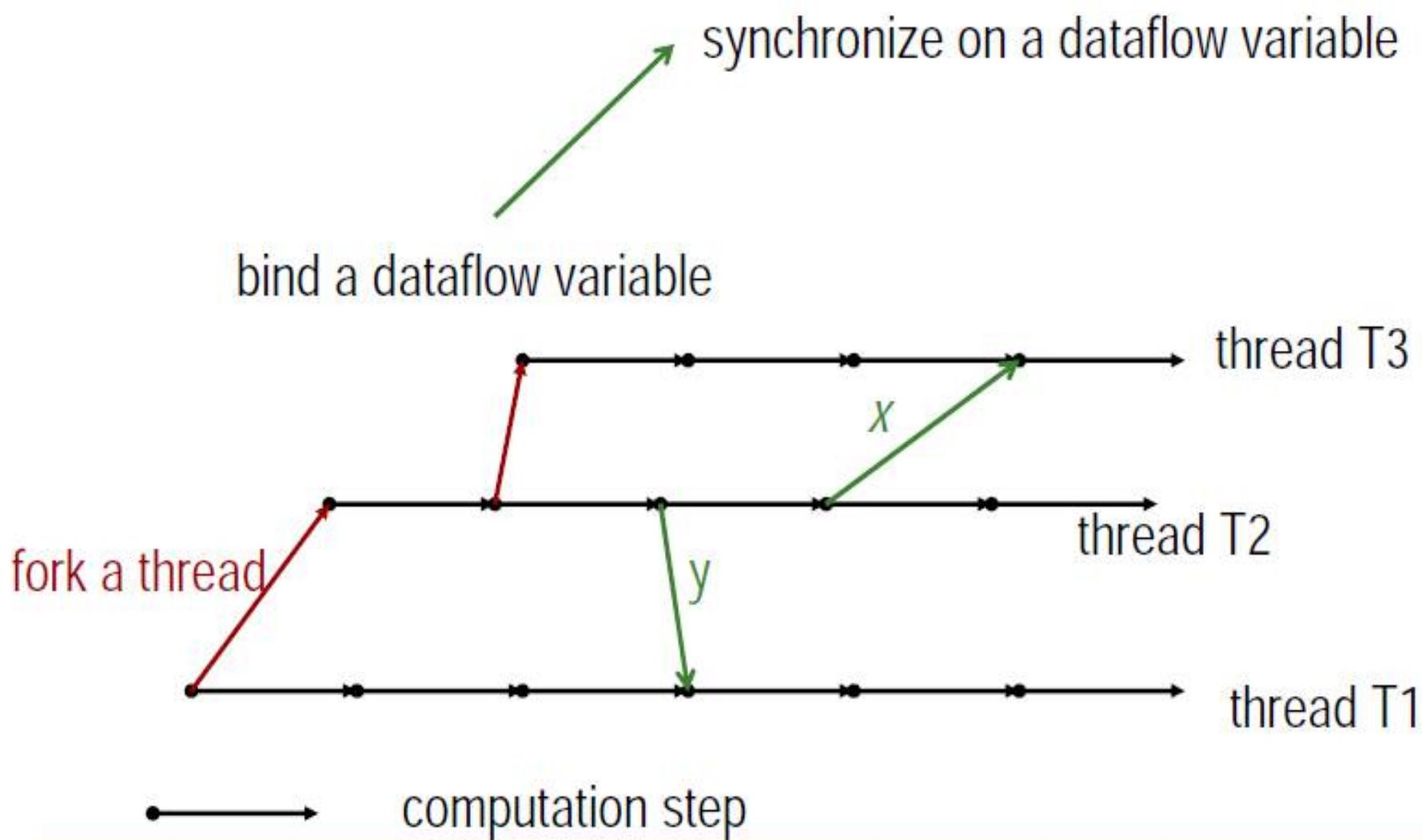


# Causal Order in the Declarative Model

- In a concurrent program all execution states of a given thread are totally ordered
- The execution state of the concurrent program is *partially ordered*



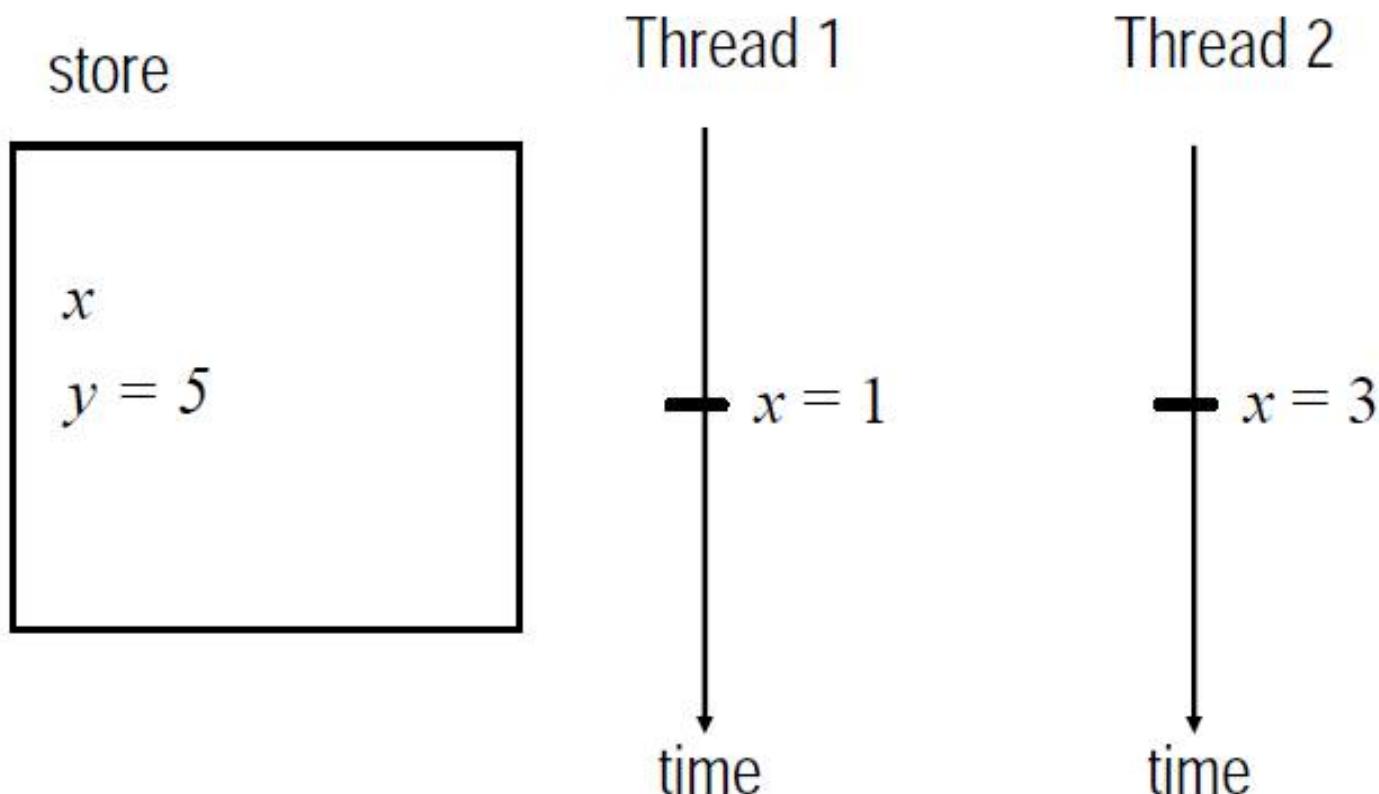
# Causal Order in the Declarative Model



# Nondeterminism

- An execution is *nondeterministic* if there is a computation step in which there is a **choice** what to do next
- Nondeterminism appears naturally when there are multiple concurrent states

# Example of Nondeterminism



- The thread that binds  $x$  first will continue, the other thread will raise an exception

## Nondeterminism

- If there is only one binder for each dataflow variable, nondeterminism is not *observable* on the store.
- That is the store has the same final results.
- Hence, for correctness we can ignore the concurrency
- This concept is known as "Declarative Concurrency".

# Declarative concurrency

- *Declarative programming (Reminder):*
  - the output of a declarative program should be a mathematical function of its input.
- *Functional programming (Reminder):*
  - the program executes with some input values and when it terminates, it has returned some output values.
- *Data-driven concurrent model:* a **concurrent** program is **declarative** if all executions with a given set of inputs have one of two results:
  - (1) they all do not terminate or
  - (2) they all eventually reach partial termination and give results that are logically equivalent.

# Partial Termination. Example

```
fun {Double Xs}  
case Xs of  
    nil then nil  
    [] X|Xr then 2*X|{Double Xr} end  
end  
Ys={Double Xs}
```

- As long as input stream  $Xs$  grows, then output stream  $Ys$  grows too. The program never terminates.
- However, if the input stream stops growing, then the program will eventually stop executing too.
- The program does a *partial termination*.

# Partial Termination. Examples

- If the inputs are bound to some partial values, then the program will eventually end up in partial termination. Also, the outputs will be bound to some partial values.
- What is the relation of outputs in terms of inputs when we consider partial values?
- Example:

$Xs=1|2|3|Xr \rightarrow Ys$  will be bound to  $2|4|6|_$

- Having  $Xr=4|5|Xr1$ , we get  $Ys$  bound to  $2|4|6|8|10|_$
- Making  $Xr1=nil$ , we get  $Ys$  bound to  $[2\ 4\ 6\ 8\ 10]$

# Logical Equivalence. Examples

- What does store contents being “the same” means?
- **Example 1:**
  - Case 1:  $X=1 \ Y=X$
  - Case 2:  $Y=X \ X=1$
- The store contents is the same for both cases
- **Example 2:**
  - Case 1:  $X=\text{foo}(Y \ W) \ Y=Z$
  - Case 2:  $X=\text{foo}(Z \ W) \ Y=Z$
- The store contents is the same for both cases

# Logical Equivalence

- A set of store bindings is called a **constraint**.
- For each variable  $x$  and constraint  $c$ , we define  $\text{values}(x, c)$  to be the set of all possible values  $x$  can have, given that  $c$  holds.

Example:  $\text{values}(x, 2 < x < 8) = \{3, 4, 5, 6, 7\}$

*arbitrary constraint*

# Logical Equivalence

- Two constraints  $c_1$  and  $c_2$  are *logically equivalent* if:
  - (1) they contain the same set of variables, and
  - (2) for each variable  $x$ ,  $\text{values}(x, c_1) = \text{values}(x, c_2)$ .

# Logical Equivalence. Example

## ■ Example:

- suppose that  $x$ ,  $y$ ,  $z$ , and  $w$  are store variables.
- the constraint

$$x = \text{foo}(y \ w) \wedge y = z$$

- is *logically equivalent* to the constraint

$$x = \text{foo}(z \ w) \wedge y = z.$$

- ## ■ Reason:
- $y = z$  forces  $y$  and  $z$  to have the same set of possible values, so that  $\text{foo}(y \ w)$  defines the same set of values as  $\text{foo}(z \ w)$ .

# Scheduling

- The choice of which thread to execute next and for how long is done by the *scheduler*
- A thread is *Runnable* if its next statement to execute is not blocked on a dataflow variable, otherwise the thread is *suspended*

# Scheduling

- A scheduler is *fair* if it does not starve each runnable thread
  - All runnable threads execute eventually
- Fair scheduling makes it easier to reason about programs
- Otherwise some runnable programs will never get its turn for execution.

# Example of Runnable Threads

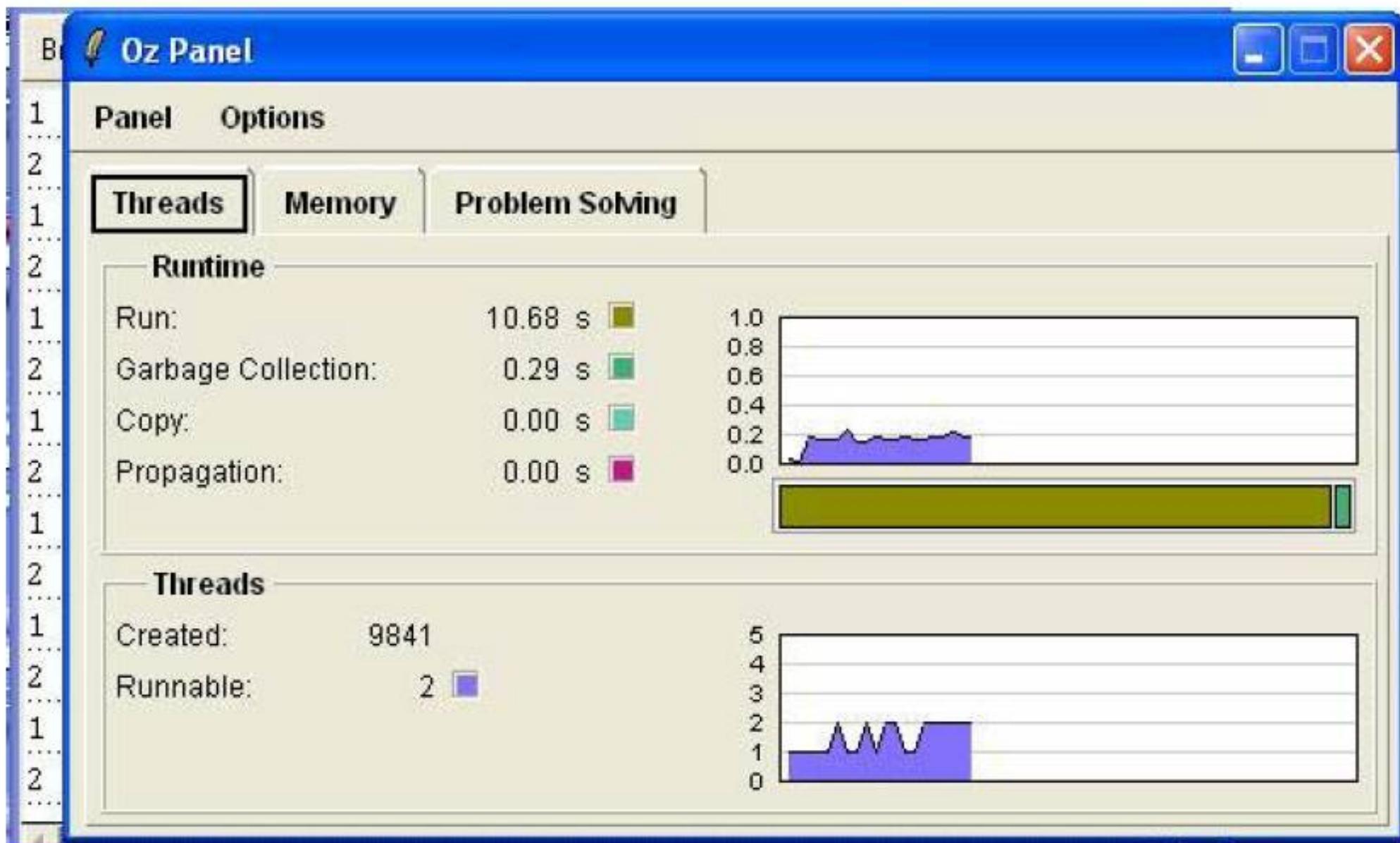
```
thread
  for I in 1..10000 do {Browse 1} end
end
thread
  for I in 1..10000 do {Browse 2} end
end
```

# Example of Runnable Threads

```
thread
  for I in 1..10000 do {Browse 1} end
end
thread
  for I in 1..10000 do {Browse 2} end
end
```

- This program will interleave the execution of two threads, one printing 1, and the other printing 2
- fair scheduler

# Example of Runnable Threads



# Dataflow Computation

- Threads suspend when dataflow variables needed are not yet bound
- {Delay X} primitive makes the thread suspends for X milliseconds, after that the thread is runnable

```
declare X
{Browse X}
local Y in
    thread {Delay 1000} Y = 10*10 end
    X = Y + 100*100
end
```

# Concurrency is Transparent

Example : a concurrent map operation

```
fun {CMap Xs F}
  case Xs
    of nil  then nil
    [] X|Xr then
      thread {F X} end | {CMap Xr F}
    end
  end
```

# Concurrency is Transparent

```
fun {CMap Xs F}  
  case Xs  
    of nil  then nil  
    [] X|Xr then  
      thread {F X} end | {CMap Xr F}  
    end  
end
```

thread ... end  
can also be used  
as expression

# Concurrency is Transparent

- What happens:

```
declare F
```

```
{Browse {CMap [1 2 3 4] F} }
```

- Browser shows [\_\_\_\_\_]

- CMap computes the list skeleton
  - newly created threads suspend until F becomes bound

# Concurrency is Transparent

- What happens:

```
F = fun { $ X } X+1 end
```

- Browser shows [2 3 4 5]

# Cheap Concurrency and Dataflow

- Declarative programs can be easily made concurrent
- Just use the `thread` statement where concurrency is needed

# Cheap Concurrency and Dataflow

```
fun {Fib X}
    if X==0 then 0
    elseif X==1 then 1
    else
        thread {Fib X-1} end + {Fib X-2}
    end
end
```

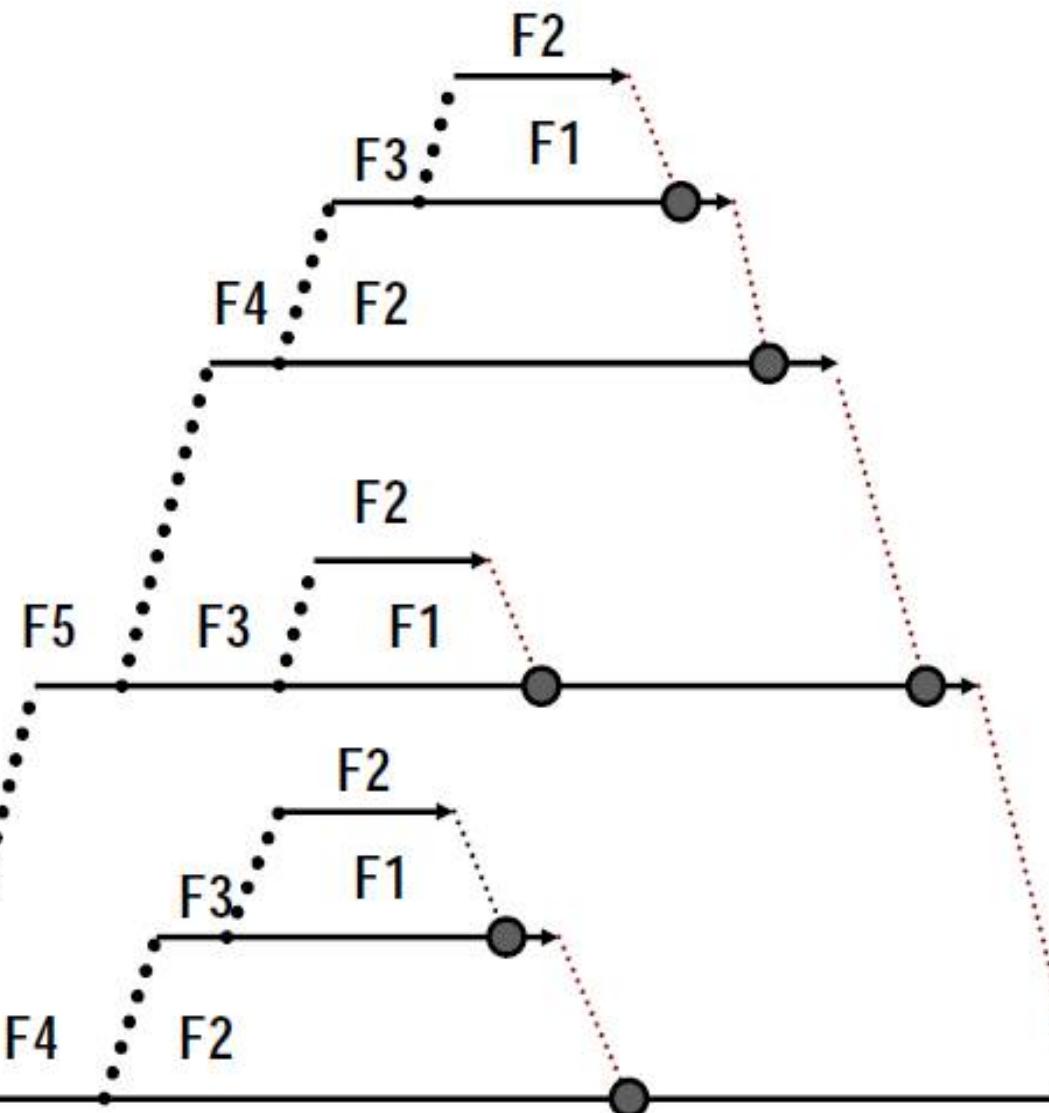
# Understanding why

```
fun {Fib X}
    if X==0 then 0 elseif X==1 then 1
    else Y1 Y2 in
        [ Y1 = thread {Fib X-1} end
        Y2 = {Fib X-2}
        Y1 + Y2
    end
end
```

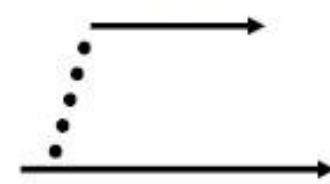
.....  
Dataflow dependency

# Execution of {Fib 6}

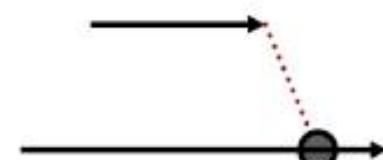
{Fib 6} is denoted as F6,...



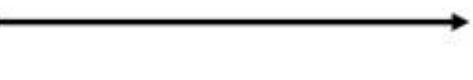
Fork a thread



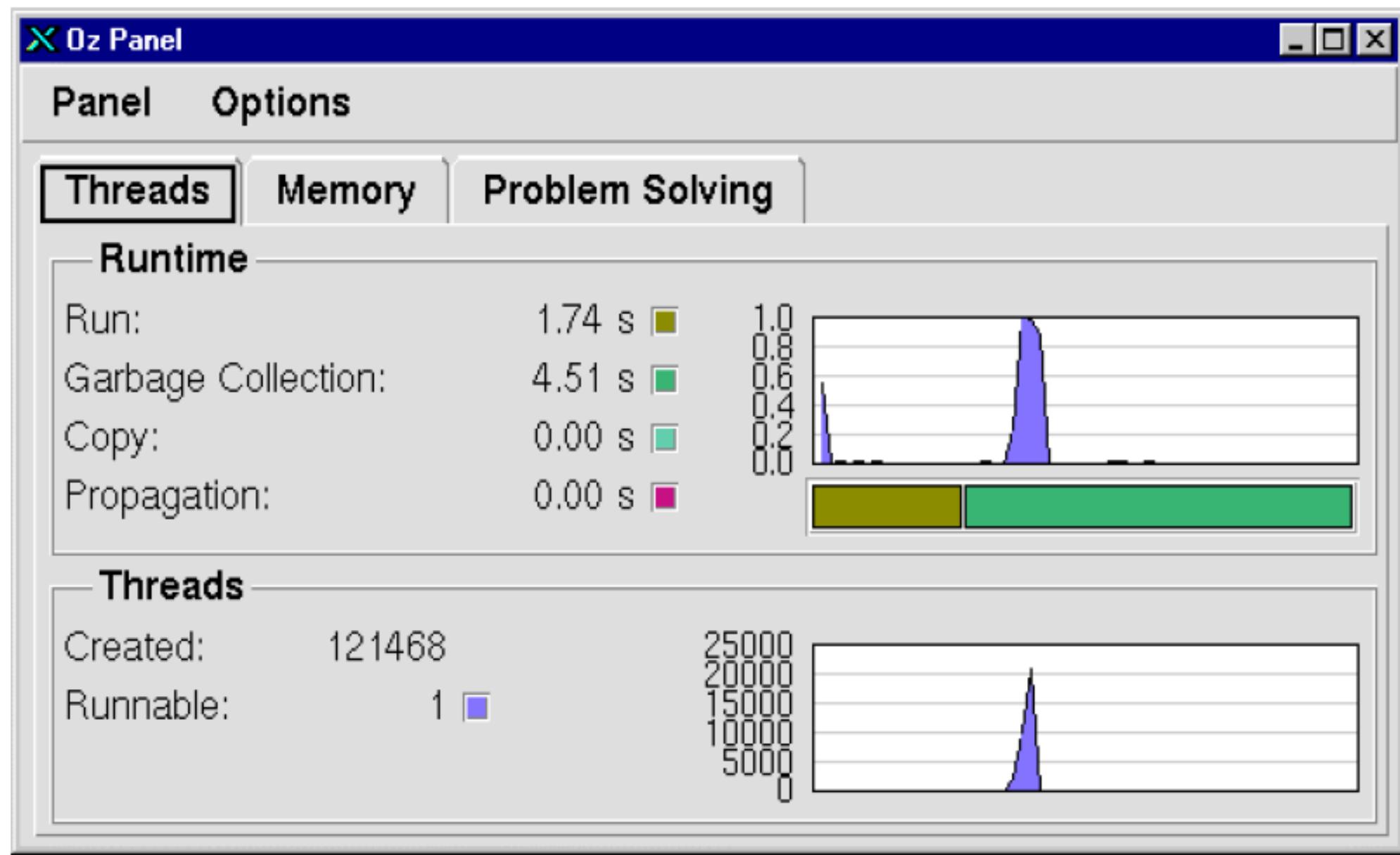
Synchronize on result



Running thread



# Fib



# Streams

# Streams

- A most useful technique for declarative concurrent programming to use **streams** to communicate between threads.
- A **stream** is a potentially unbounded list of messages, i.e., it is a list whose tail is an unbound dataflow variable.
- A thread communicating through streams is a kind of “active object”, also called **stream object**.
- A sequence of stream objects each of which feeds the next is called a **pipeline**.
- **Deterministic stream programming**: each stream object always knows for each input where the next message will come from.

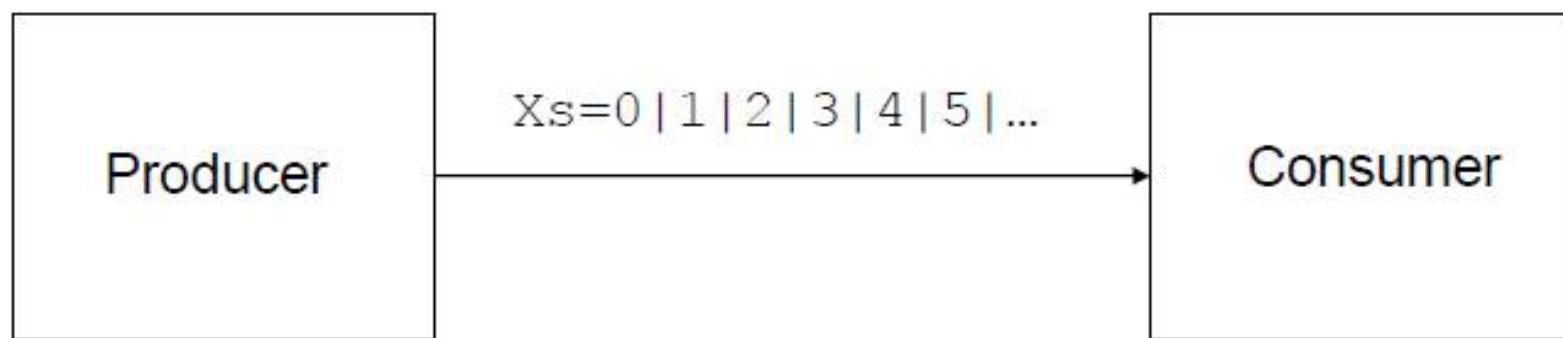
# Producer ↔ Consumer

```
thread X={Produce} end
```

```
thread Result={Consume X} end
```

- Typically, what is produced will be put on a list that never ends (without `nil`), called **stream**
- **Consumer** (also called **sink**) consumes as soon as **producer** (also called **source**) produces

# Producer/Consumer Stream



$Xs = \{\text{Produce } 0 \text{ Limit}\}$

$S = \{\text{Consume } Xs \text{ 0}\}$

## Example: Producer $\leftrightarrow$ Consumer

```
fun {Produce N Limit}
    if N<Limit then
        N | {Produce N+1 Limit}
    else nil end
end

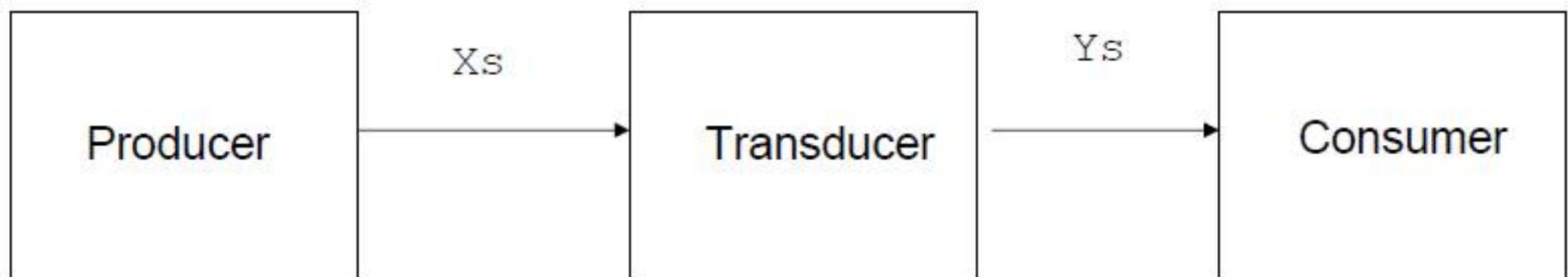
fun {Consume Xs Acc}
    case Xs of X|Xr then
        {Consume Xr Acc+X}
    [] nil then Acc
end
end
```

# Stream Transducer. Example

```
thread Stream={Produce 0 1000} end  
thread FilterResult={Filter Stream IsOdd} end  
thread Result={Consume FilterResult 0} end
```

- **Transducer:** a stream which reads the producer's output and computes a filtered stream for the consumer.
- Can be: filtering, mapping, ...
- Advantages of pipeline:
  - there is no need to wait the final value of the producer
  - producer, transducer, and consumer are executed concurrently

# Simple Pipeline



$Y_s = \{ \text{Filter } X_s \dots \}$

# Client $\Leftrightarrow$ Server

- Similar to producer  $\Leftrightarrow$  consumer
- Typical scenario:
  - more clients than servers
  - server has a fixed identity
  - clients send messages to server
  - server replies
- See Next Lecture: message sending

# Fairness

- Essential that even though producer can always produce, consumer also gets a chance to run
- Threads are scheduled with **fairness**
  - if a thread is runnable, it will eventually run

# Thread Scheduling

- More guarantees than just fairness
- Threads are given a time slice to run
  - approximately 10ms
  - when time slice is over: thread is **preempted**
  - next runnable thread is **scheduled**
- Can be influenced by priorities
  - high, medium, low

# Summary so far

- Threads
  - suspend and resume automatically
  - controlled by **data-flow variables**
  - cheap
  - execute fairly according to time-slice
- Pattern
  - producer  $\Leftrightarrow$  transducer  $\Leftrightarrow$  consumer

# Demand Driven Execution

# How to Control Producers?

- *Eager model*: the producer decides when enough data has been sent
- *Possible problem*: producer should not produce more than needed
- *One attempt*: make consumer the driver
  - consumer produces stream skeleton
  - producer fills skeleton

# Make Consumer be the Driver

```
fun {DConsume ?Xs A Limit}
    if Limit>0 then
        local X Xr in
            Xs=X|Xr {DConsume Xr A+X Limit-1}
    else A end
end

proc {DProduce N Xs}
    case Xs of X|Xr then
        X=N
        {DProduce N+1 Xr}
    end
end
```

## Overall program :

```
local Xs S in
  thread {DProduce 0 Xs} end
  thread S={DConsume Xs 0 150000} end
  {Browse S}
end
```

Note that consumer controls how many elements are needed.