Declarative Concurrency (CTM 4)

Carlos Varela
Rensselaer Polytechnic Institute

Adapted with permission from:
Seif Haridi
KTH
Peter Van Roy
UCL

October 28, 2016

Review of concurrent programming

- There are four basic approaches:
  - Sequential programming (no concurrency)
  - Declarative concurrency (streams in a functional language, Oz)
  - Message passing with active objects (Erlang, SALSA)
  - Atomic actions on shared state (Java)
- The atomic action approach is the *most difficult*, yet it is the one you will probably be most exposed to!
- But, if you have the choice, which approach to use?
  - Use the simplest approach that does the job: sequential if that is ok, else declarative concurrency if there is no observable nondeterminism, else message passing if you can get away with it.

Concurrency

- How to do several things at once
- Concurrency: running several activities each running at its own pace
- A *thread* is an executing sequential program
- A program can have multiple threads by using the thread instruction
- {Browse 99*99} can immediately respond while Pascal is computing

```
thread
   P in
   P = {Pascal 21}
   {Browse P}
end
{Browse 99*99}
```

State

- How to make a function learn from its past?
- We would like to add memory to a function to remember past results
- Adding memory as well as concurrency is an essential aspect of modeling the real world
- Consider {FastPascal N}: we would like it to remember the previous rows it calculated in order to avoid recalculating them
- We need a concept (memory cell) to store, change and retrieve a value
- The simplest concept is a (memory) cell which is a container of a value
- One can create a cell, assign a value to a cell, and access the current value of the cell
- Cells are not variables

```
declare
C = {NewCell 0}
{Assign C {Access C}+1}
{Browse {Access C}}
```

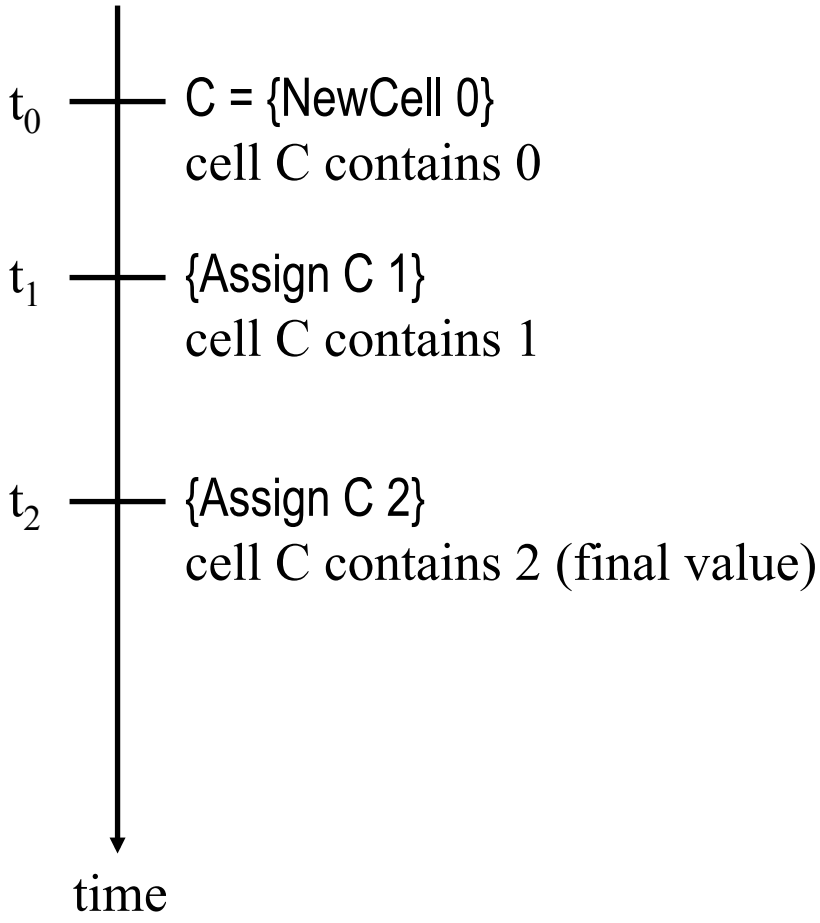
Nondeterminism

- What happens if a program has both concurrency and state together?
- This is very tricky
- The same program can give different results from one execution to the next
- This variability is called *nondeterminism*
- Internal nondeterminism is not a problem if it is not observable from outside

Nondeterminism (2)

```
declare
C = {NewCell 0}

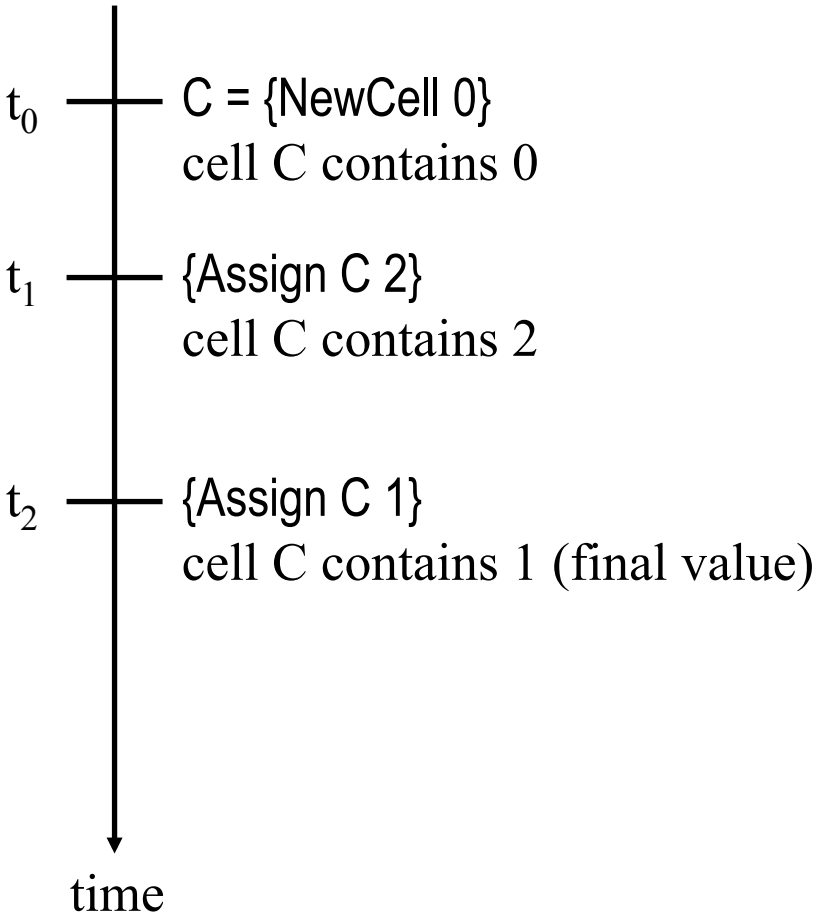
thread {Assign C 1} end
thread {Assign C 2} end
```



Nondeterminism (3)

```
declare
C = {NewCell 0}

thread {Assign C 1} end
thread {Assign C 2} end
```



Nondeterminism (4)

```
declare
C = {NewCell 0}

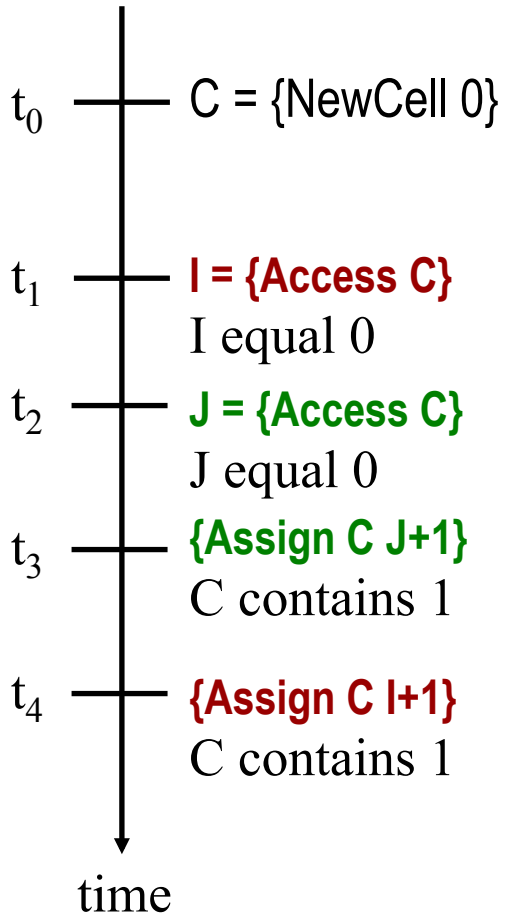
thread I in
   I = {Access C}
   {Assign C I+1}
end
thread J in
   J = {Access C}
   {Assign C J+1}
end
```

- What are the possible results?
- Both threads increment the cell C by 1
- Expected final result of C is 2
- Is that all?

Nondeterminism (5)

- Another possible final result is the cell C containing the value 1

```
declare
C = {NewCell 0}
thread I in
    I = {Access C}
    {Assign C I+1}
end
thread J in
    J = {Access C}
    {Assign C J+1}
end
```



Lessons learned

- Combining concurrency and state is tricky
- Complex programs have many possible *interleavings*
- Programming is a question of mastering the interleavings
- Famous bugs in the history of computer technology are due to designers overlooking an interleaving (e.g., the Therac-25 radiation therapy machine giving doses thousands of times too high, resulting in death or injury)

1. If possible try to avoid concurrency and state together
2. Encapsulate state and communicate between threads using dataflow
3. Try to master interleavings by using *atomic operations*

Atomicity

- How can we master the interleavings?
- One idea is to reduce the number of interleavings by programming with coarse-grained atomic operations
- An operation is *atomic* if it is performed as a whole or nothing
- No intermediate (partial) results can be observed by any other concurrent activity
- In simple cases we can use a *lock* to ensure atomicity of a sequence of operations
- For this we need a new entity (a lock)

Atomicity (2)

```
declare
L = {NewLock}

lock L then
   sequence of ops 1
end
```

} Thread 1

```
lock L then
   sequence of ops 2
end
```

} Thread 2

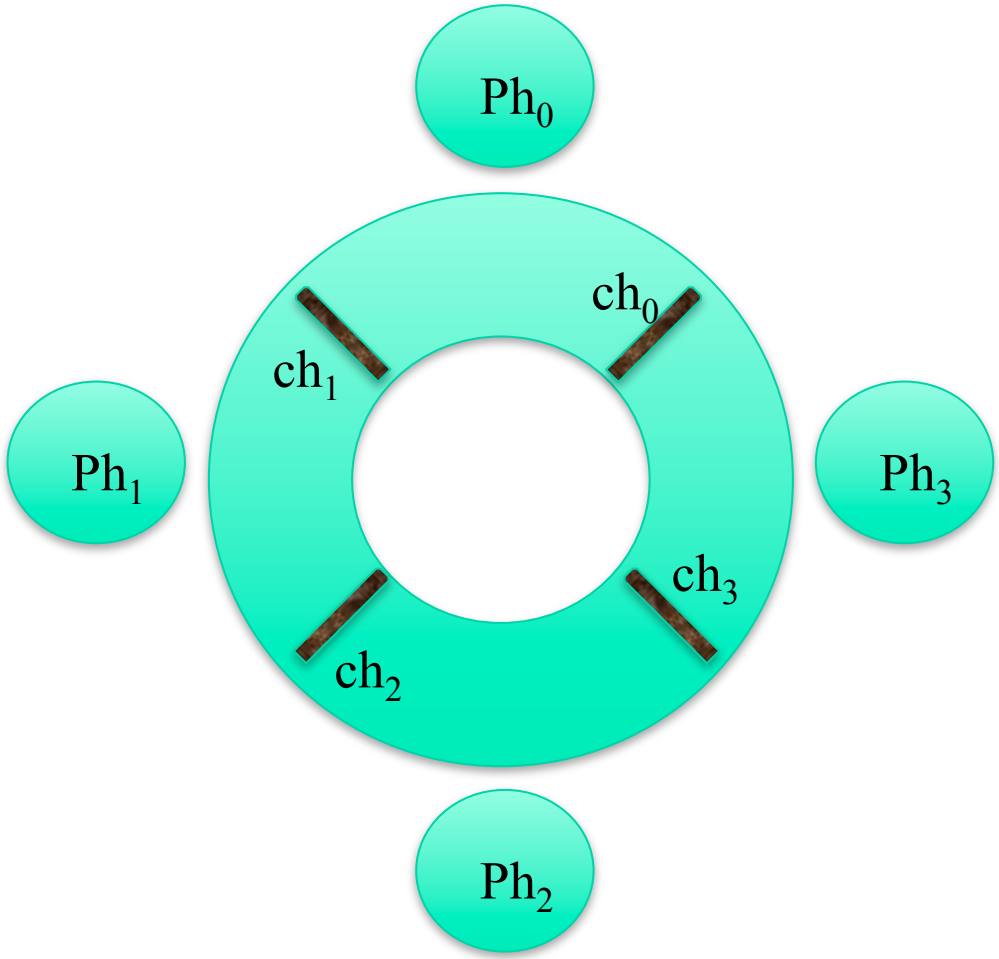
The program

```
declare
C = {NewCell 0}
L = {NewLock}

thread
   lock L then I in
      I = {Access C}
      {Assign C I+1}
   end
end
thread
   lock L then J in
      J = {Access C}
      {Assign C J+1}
   end
end
```

The final result of C is always 2

Locks and Deadlock: Dining Philosophers



Review of concurrent programming

- There are four basic approaches:
  - Sequential programming (no concurrency)
  - Declarative concurrency (streams in a functional language, Oz)
  - Message passing with active objects (Erlang, SALSA)
  - Atomic actions on shared state (Java)
- The atomic action approach is the *most difficult*, yet it is the one you will probably be most exposed to!
- But, if you have the choice, which approach to use?
  - Use the simplest approach that does the job: sequential if that is ok, else declarative concurrency if there is no observable nondeterminism, else message passing if you can get away with it.

Declarative Concurrency

- This lecture is about declarative concurrency, programs with no observable nondeterminism, the result is a function
- Independent procedures that execute on their pace and may communicate through shared dataflow variables

Single-assignment Variables

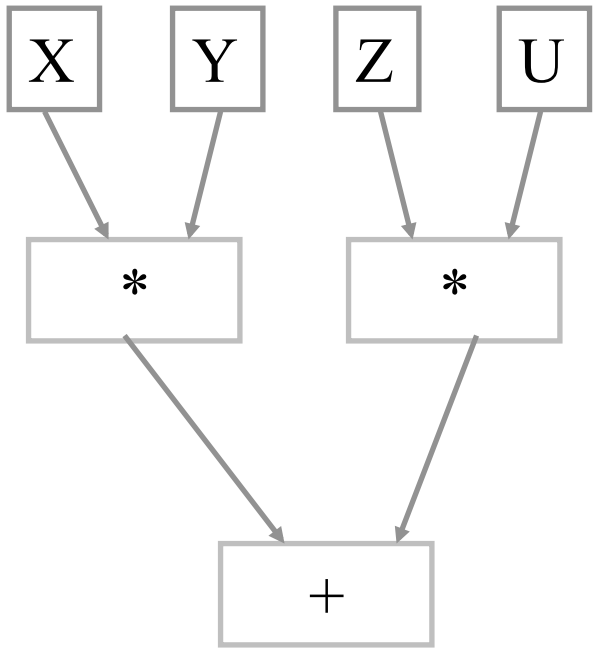
- Variables are short-cuts for values, they cannot be assigned more than once

```
declare
V = 9999*9999
{Browse V*V}
```

- Variable identifiers: is what you type
- Store variable: is part of the memory system
- The **declare** statement creates a store variable and assigns its memory address to the identifier 'V' in the environment

Dataflow

- What happens when multiple threads try to communicate?
- A simple way is to make communicating threads synchronize on the availability of data (data-driven execution)
- If an operation tries to use a variable that is not yet bound it will wait
- The variable is called a *dataflow variable*



Dataflow (II)

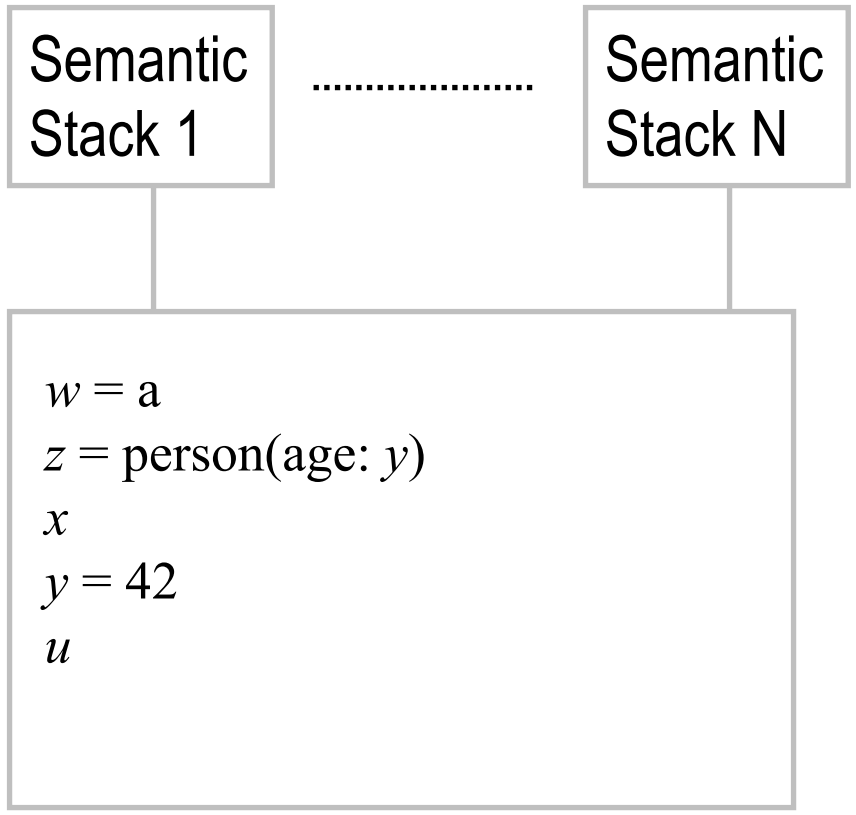
- Two important properties of dataflow
  - Calculations work correctly independent of how they are partitioned between threads (concurrent activities)
  - Calculations are patient, they do not signal error; they wait for data availability
- The dataflow property of variables makes sense when programs are composed of multiple threads

```
declare X
thread
   {Delay 5000} X=99
end
{Browse ‘Start’} {Browse X*X}
```

```
declare X
thread
   {Browse ‘Start’} {Browse X*X}
end
{Delay 5000} X=99
```

The concurrent model

Multiple semantic stacks (threads)



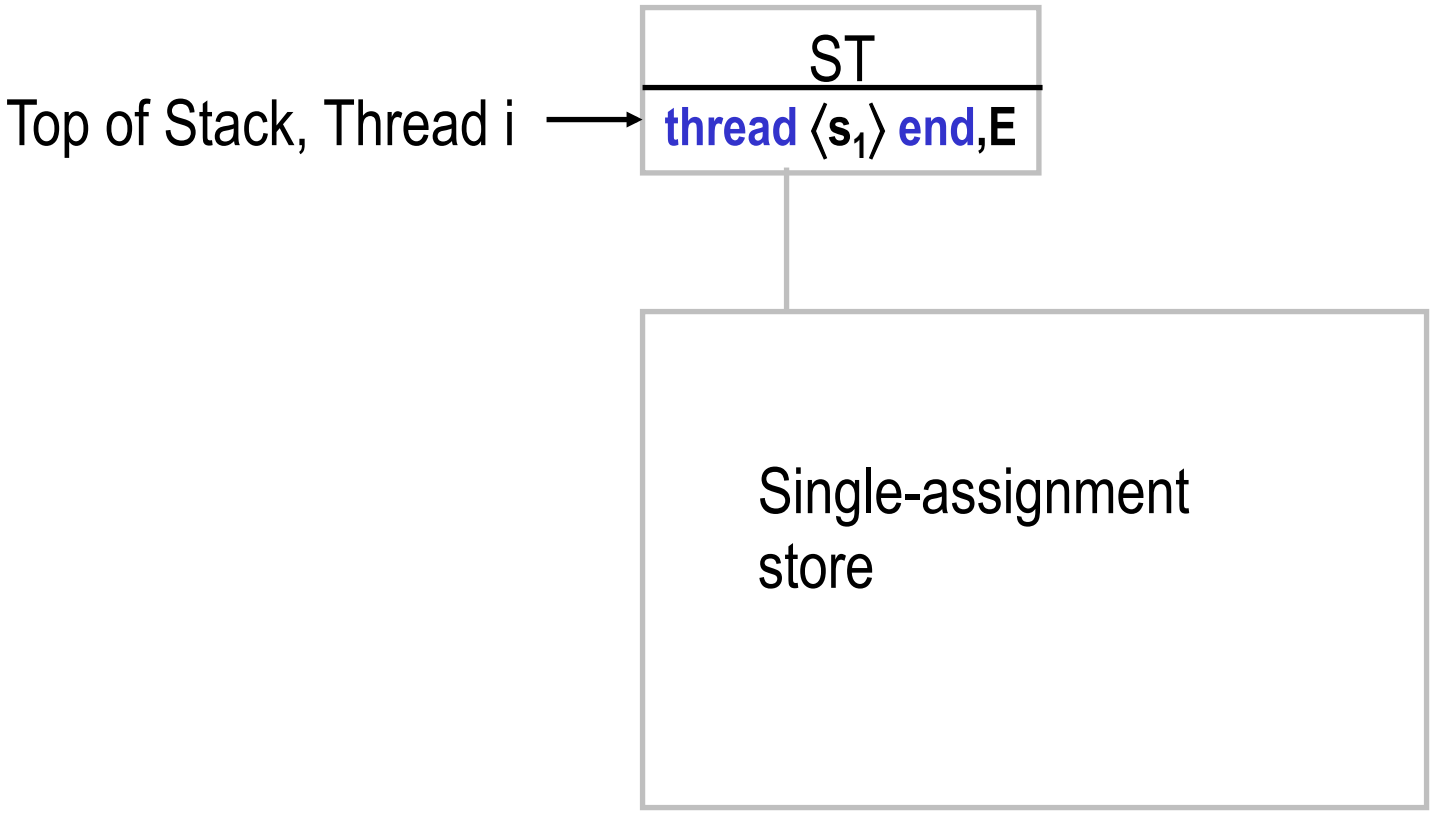
Single-assignment store

Concurrent declarative model

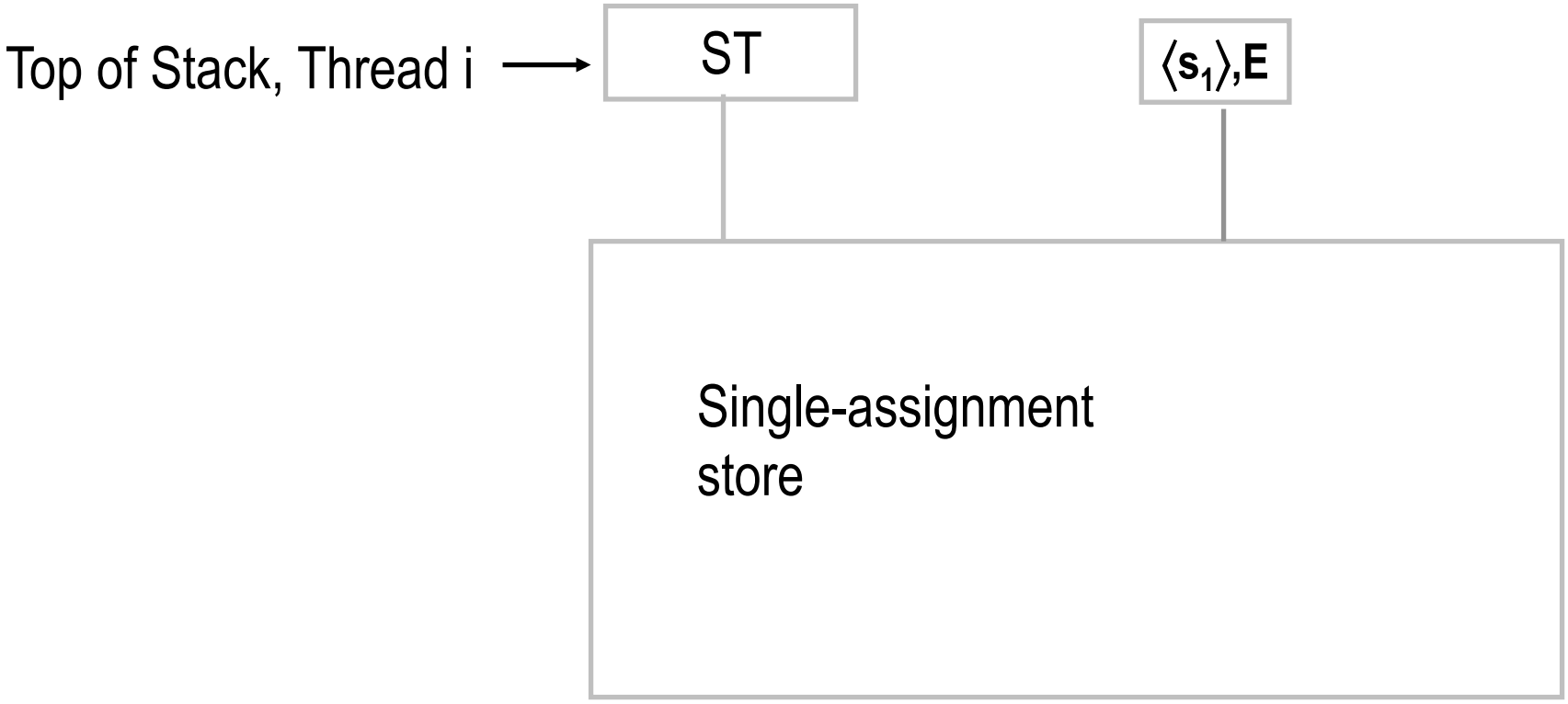
The following defines the syntax of a statement, ⟨s⟩ denotes a statement

⟨s⟩ ::=	skip	*empty statement*
|	⟨x⟩ = ⟨y⟩	*variable-variable binding*
|	⟨x⟩ = ⟨v⟩	*variable-value binding*
|	⟨s_1⟩ ⟨s_2⟩	*sequential composition*
|	local ⟨x⟩ in ⟨s_1⟩ end	*declaration*
|	proc {⟨x⟩ ⟨y_1⟩ … ⟨y_n⟩ } ⟨s_1⟩ end	*procedure introduction*
|	if ⟨x⟩ then ⟨s_1⟩ else ⟨s_2⟩ end	*conditional*
|	{ ⟨x⟩ ⟨y_1⟩ … ⟨y_n⟩ }	*procedure application*
|	case ⟨x⟩ of ⟨pattern⟩ then ⟨s_1⟩ else ⟨s_2⟩ end	*pattern matching*
|	**thread ⟨s_1⟩ end**	***thread creation***

The concurrent model



The concurrent model



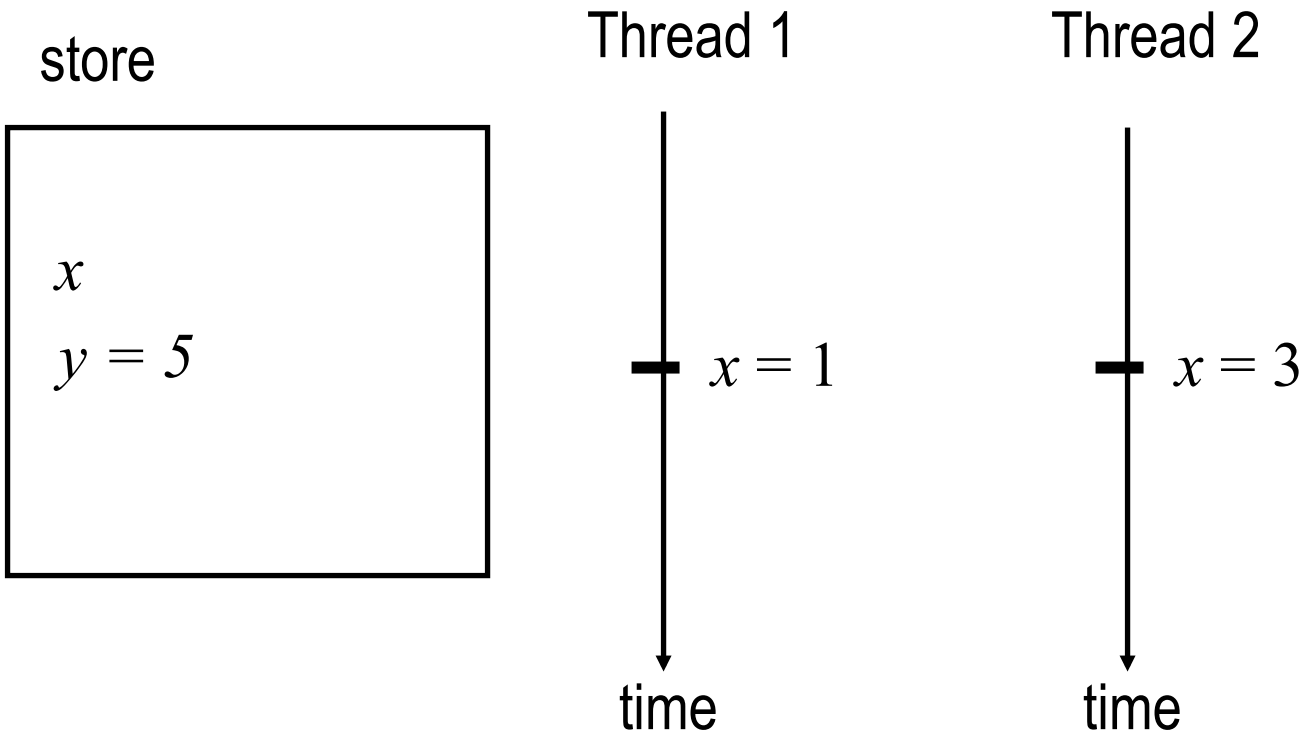
Basic concepts

- The model allows multiple statements to execute ”at the same time”
- 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, as well as reading the same variable
- Writing the same variable is done sequentially
- The above view is in fact equivalent to an *interleaving execution*: a totally ordered sequence of computation steps, where threads take turns doing one or more steps in sequence

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 is concurrent access to shared state

Example of nondeterminism



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

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 is concurrent access to shared state
- In the concurrent declarative model when there is only one binder for each dataflow variable or multiple compatible bindings (e.g., to partial values), the nondeterminism is not observable on the store (i.e. the store develops to the same final results)
- This means for correctness we can ignore the concurrency

Scheduling

- The choice of which thread to execute next and for how long is done by a part of the system called the *scheduler*
- A thread is *runnable* if its next statement to execute is not blocked on a dataflow variable, otherwise the thread is *suspended*
- A scheduler is fair if it does not starve a runnable thread, i.e. all runnable threads eventually execute
- Fair scheduling makes it easy to reason about programs and program composition
- Otherwise some correct program (in isolation) may never get processing time when composed with other programs

Example of runnable threads

```
proc {Loop P N}
   if N > 0 then
      {P} {Loop P N-1}
   else skip end
end
thread {Loop
           proc {$} {Show 1} end
           1000}
end
thread {Loop
           proc {$} {Show 2} end
           1000}
end
```

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

Dataflow computation

- Threads suspend on data unavailability in dataflow variables
- The **{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
```

Illustrating dataflow computation

```
declare X0 X1 X2 X3
{Browse [X0 X1 X2 X3]}
thread
   Y0 Y1 Y2 Y3
in
   {Browse [Y0 Y1 Y2 Y3]}
   Y0 = X0 + 1
   Y1 = X1 + Y0
   Y2 = X2 + Y1
   Y3 = X3 + Y2
   {Browse completed}
end
```

- Enter incrementally the values of X0 to X3
- When X0 is bound the thread will compute Y0=X0+1, and will suspend again until X1 is bound

Concurrent Map

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

- This will fork a thread for each individual element in the input list
- Each thread will run only if both the element X and the procedure F is known

Concurrent Map Function

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

- What this looks like in the kernel language:

```
proc {Map Xs F Rs}
   case Xs
   of nil then Rs = nil
   [] X|Xr then R Rr in
      Rs = R|Rr
      thread {F X R} end
      {Map Xr F Rr}
   end
end
```

How does it work?

- If we enter the following statements:
  ```
  declare F X Y Z
  {Browse thread {Map X F} end}
  ```
- A thread executing Map is created.
- It will suspend immediately in the case-statement because X is unbound.
- If we thereafter enter the following statements:
  ```
  X = 1|2|Y
  fun {F X} X*X end
  ```
- The main thread will traverse the list creating two threads for the first two arguments of the list

How does it work?

- The main thread will traverse the list creating two threads for the first two arguments of the list:

  thread {F 1} **end**, and **thread** {F 2} **end**,

After entering:

```
Y = 3|Z
Z = nil
```

the program will complete the computation of the main thread and the newly created thread **thread** {F 3} **end**, resulting in the final list [1 4 9].

Simple concurrency with dataflow

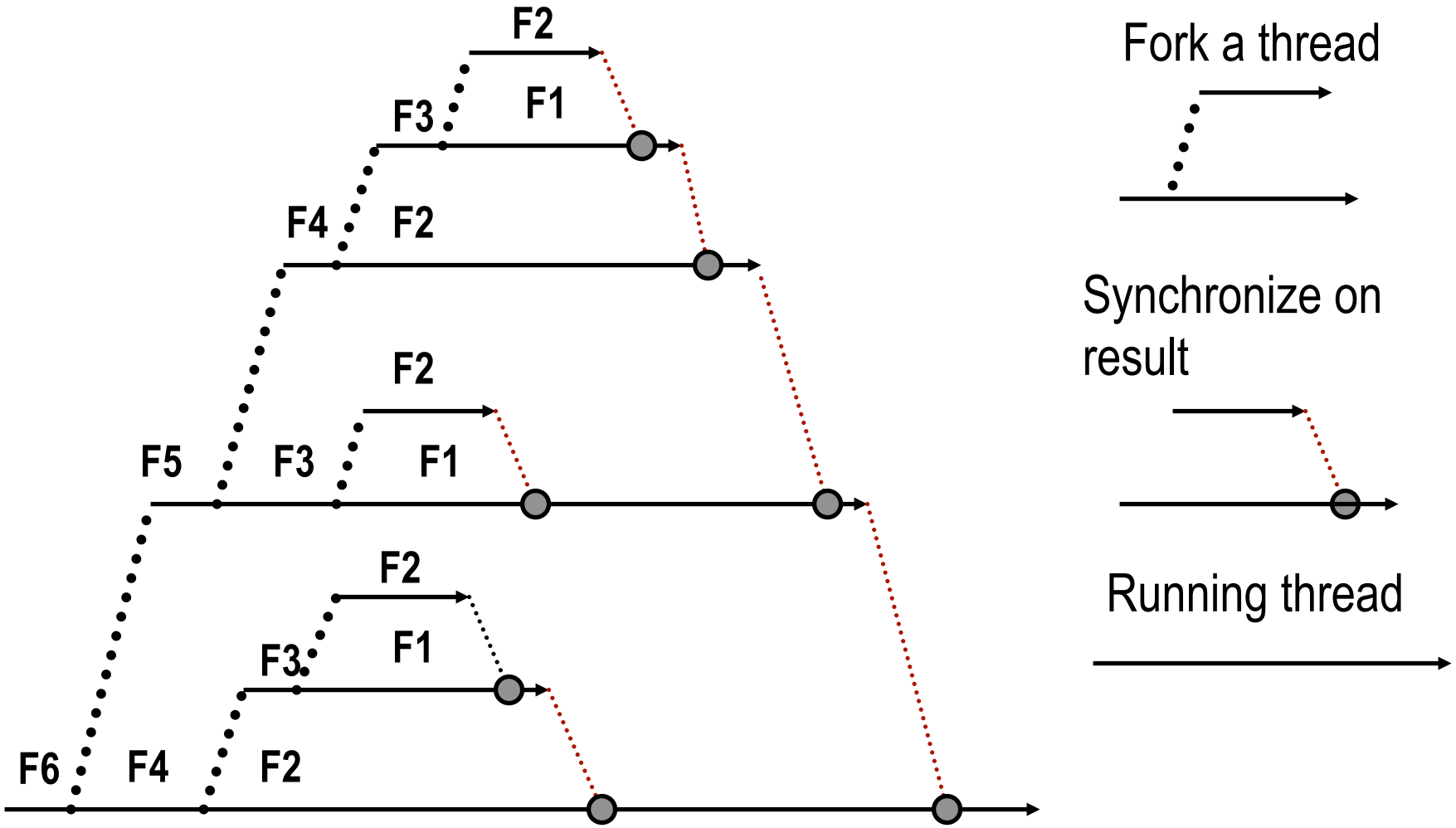
- Declarative programs can be easily made concurrent
- Just use the thread statement where concurrency is needed

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

Understanding why

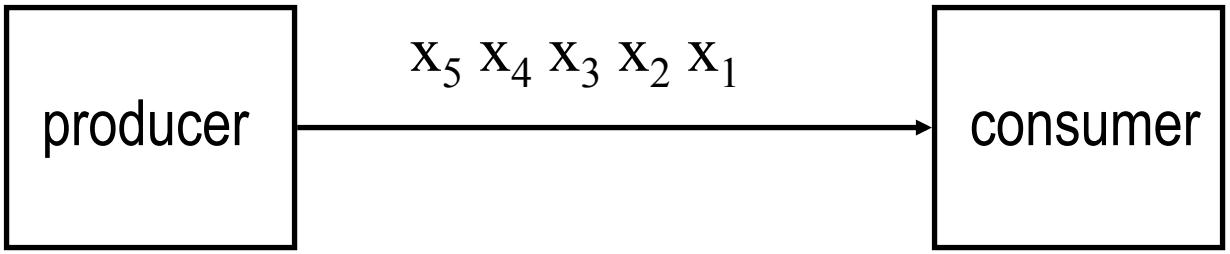
```
fun {Fib X}
   if X=<2 then 1
   else  F1 F2 in
      F1  =  thread {Fib X-1} end
      F2  =  {Fib X-2}
      F1  + F2
   end
end
```

Execution of {Fib 6}



Streams

- A stream is a sequence of messages
- A stream is a First-In First-Out (FIFO) channel
- The producer augments the stream with new messages, and the consumer reads the messages, one by one.

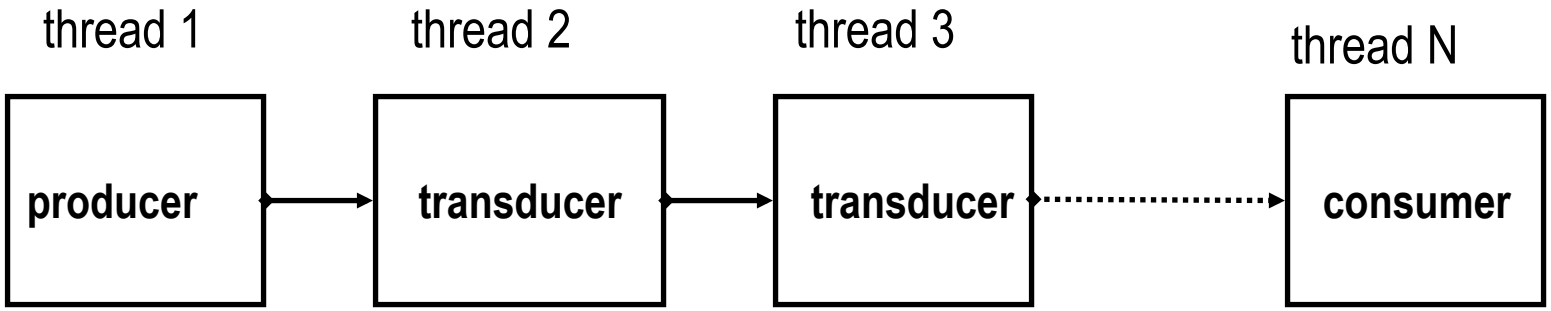


Stream Communication I

- The data-flow property of Oz easily enables writing threads that communicate through streams in a producer-consumer pattern.
- A stream is a list that is created incrementally by one thread (the producer) and subsequently consumed by one or more threads (the consumers).
- The consumers consume the same elements of the stream.

Stream Communication II

- **Producer**, produces incrementally the elements
- **Transducer**(s), transform(s) the elements of the stream
- **Consumer**, accumulates the results



Stream communication patterns

- The producer, transducers, and the consumer can, in general, be described by certain program patterns
- We show various patterns

Producer

```
fun {Producer State}
   if {More State} then
      X = {Produce State} in
      X | {Producer {Transform State}}
   else nil end
end
```

- The definition of *More*, *Produce*, and *Transform* is problem dependent
- State could be multiple arguments
- The above definition is not a complete program!

Example Producer

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

```
fun {Producer State}
   if {More State} then
      X = {Produce State} in
      X | {Producer {Transform State}}
   else nil end
end
```

- The State is the two arguments N and Limit
- The predicate More is the condition N=<Limit
- The Produce function is the identity function on N
- The Transform function (N,Limit) ⇒ (N+1,Limit)

Consumer Pattern

```
fun {Consumer State InStream}
   case InStream
   of nil then {Final State}
   [] X | RestInStream then
      NextState = {Consume X State} in
      {Consumer NextState RestInStream}
   end
end
```

The consumer suspends until InStream is either a cons or a nil

- *Final* and *Consume* are problem dependent

Example Consumer

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

```
fun {Consumer State InStream}
   case InStream
   of nil then {Final State}
   [] X | RestInStream then
      NextState = {Consume X State} in
      {Consumer NextState RestInStream}
   end
end
```

- The State is A
- `Final` is just the identity function on State
- `Consume` takes X and State ⇒ X + State

Transducer Pattern 1

```
fun {Transducer State InStream}
   case InStream
   of nil then nil
   [] X | RestInStream then
      NextState#TX = {Transform X State}
      TX | {Transducer NextState RestInStream}
   end
end
```

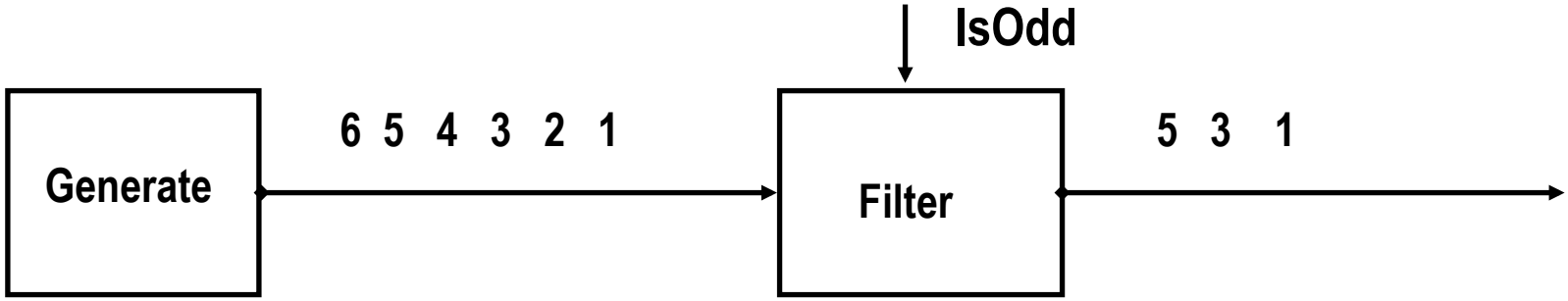
- A transducer keeps its state in State, receives messages on InStream and sends messages on OutStream

Transducer Pattern 2

```
fun {Transducer State InStream}
   case InStream
   of nil then nil
   [] X | RestInStream then
      if {Test X#State} then
         NextState#TX = {Transform X State}
         TX | {Transducer NextState RestInStream}
      else {Transducer State RestInStream} end
   end
end
```

- A transducer keeps its state in State, receives messages on InStream and sends messages on OutStream

Example Transducer



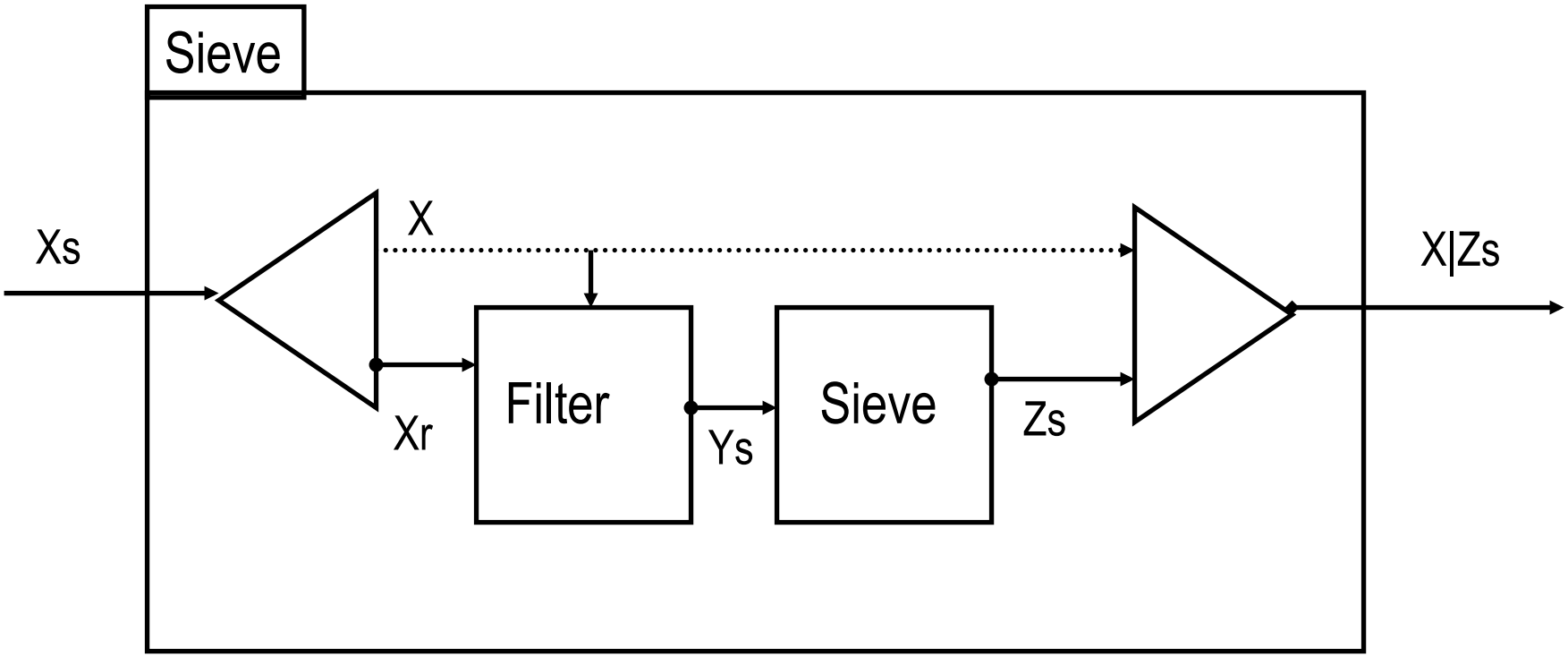

```
fun {Filter Xs F}
   case Xs
   of nil then nil
   [] X|Xr then
      if {F X} then X|{Filter Xr F}
      else {Filter Xr F} end
   end
end
```

Filter is a transducer that takes an Instream and incrementally produces an Outstream that satisfies the predicate F

```
local Xs Ys in
   thread Xs =  {Generate 1 100} end
   thread Ys = {Filter Xs IsOdd} end
   thread {Browse Ys} end
end
```

Larger example: The sieve of Eratosthenes

- Produces prime numbers
- It takes a stream 2...N, peals off 2 from the rest of the stream
- Delivers the rest to the next sieve



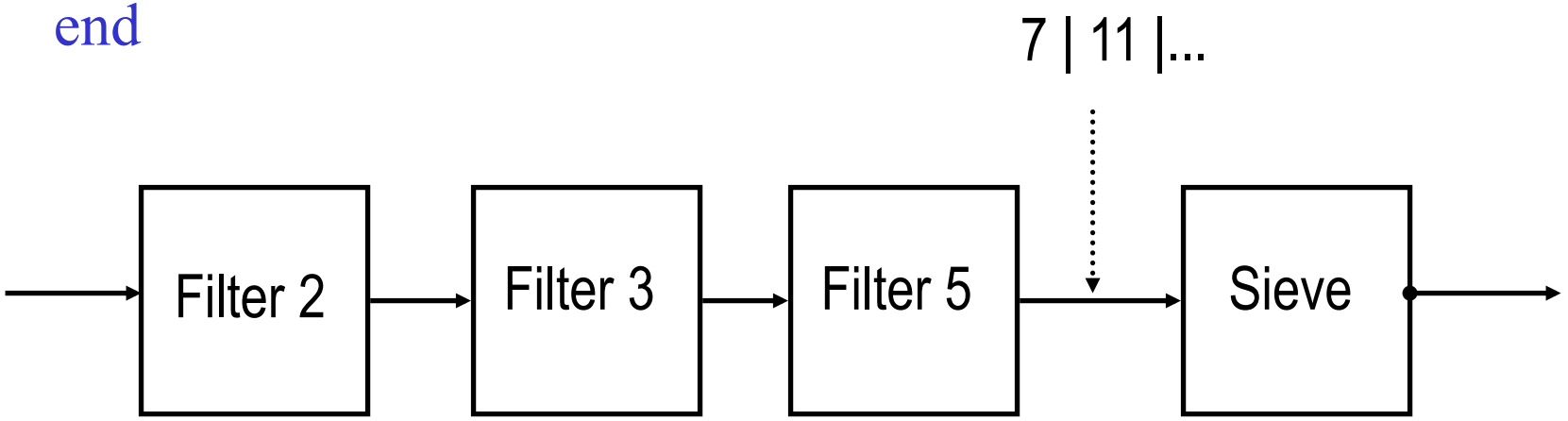
Sieve

```
fun {Sieve Xs}
   case Xs
   of nil then nil
   [] X|Xr then Ys in
      thread Ys = {Filter Xr fun {$ Y} Y mod X \= 0 end} end
      X | {Sieve Ys}
   end
end
```

- The program forks a filter thread on each sieve call

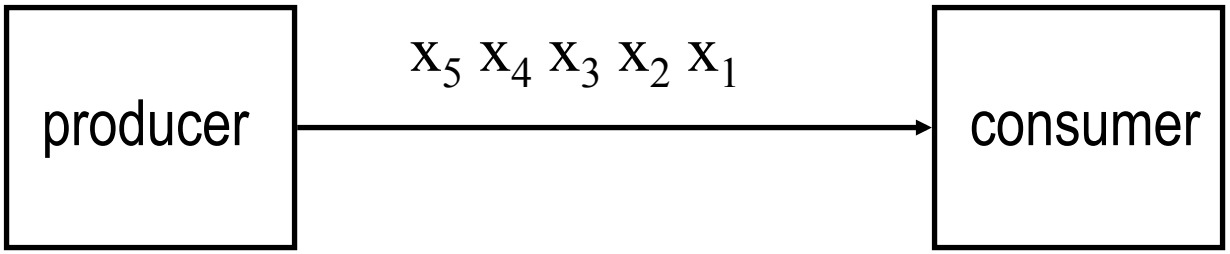
Example call

```
local Xs Ys in
   thread Xs =  {Generate 2 100000} end
   thread Ys = {Sieve Xs} end
   thread for Y in Ys do {Show Y} end end
end
```



Limitation of eager stream processing Streams

- The producer might be much faster than the consumer
- This will produce a large intermediate stream that requires potentially unbounded memory storage



Solutions

There are three alternatives:

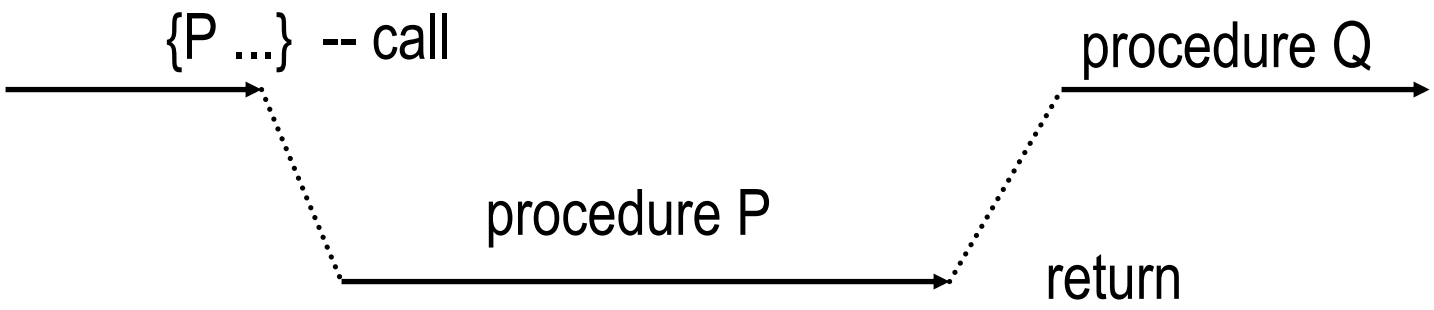
1. Play with the speed of the different threads, i.e. play with the scheduler to make the producer slower
2. Create a bounded buffer, say of size N, so that the producer waits automatically when the buffer is full
3. Use demand-driven approach, where the consumer activates the producer when it needs a new element (**lazy evaluation**)

- The last two approaches introduce the notion of flow-control between concurrent activities (very common)

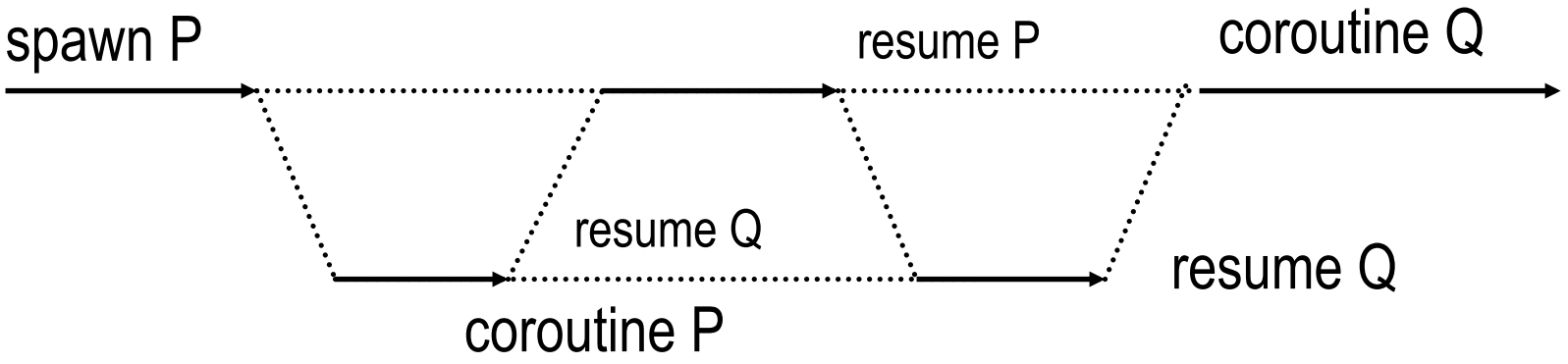
Coroutines I

- Languages that do not support concurrent threads might instead support a notion called **coroutining**
- A coroutine is a nonpreemptive thread (sequence of instructions), there is no scheduler
- Switching between threads is the programmer's responsibility

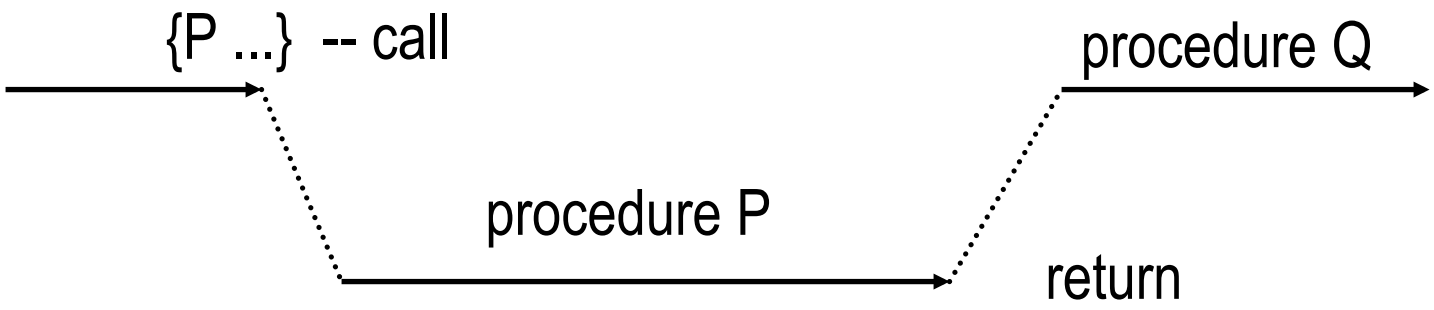
Coroutines II, Comparison



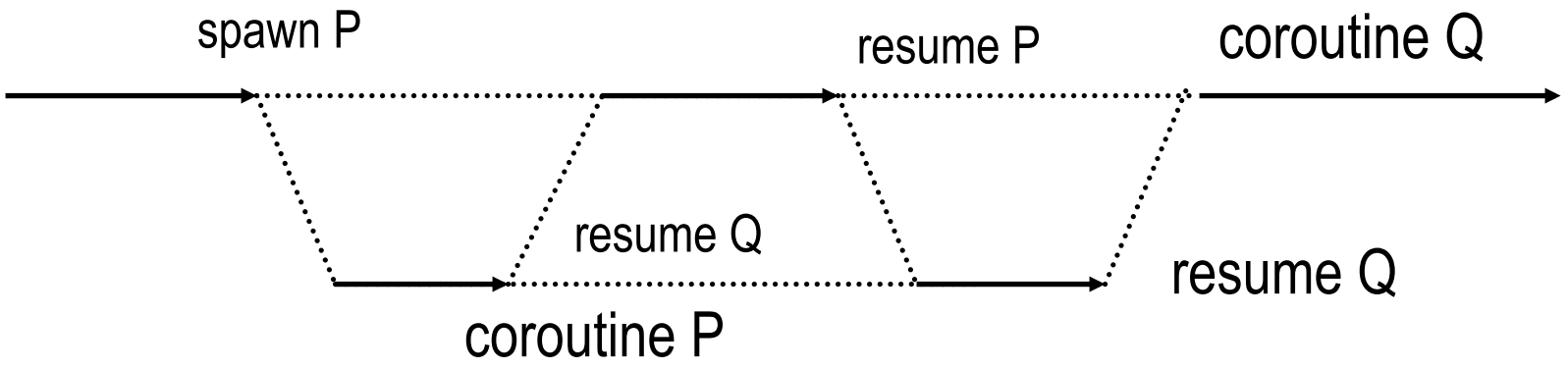
Procedures: one sequence of instructions, program transfers explicitly when terminated it returns to the caller



Coroutines II, Comparison



Coroutines: New sequences of instructions, programs explicitly do all the scheduling, by spawn, suspend and resume



Time

- In concurrent computation one would like to handle time
- `proc {Time.delay T}` – The running thread suspends for T milliseconds
- `proc {Time.alarm T U}` – Immediately creates its own thread, and binds U to unit after T milliseconds

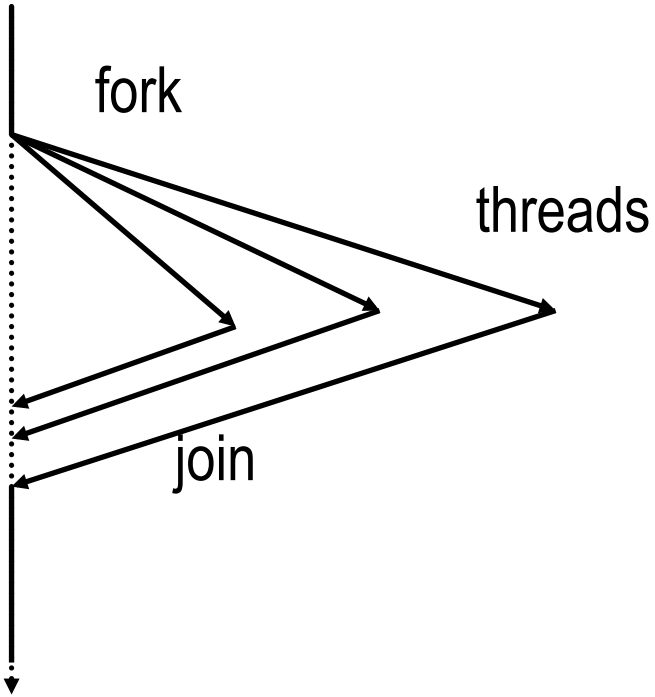
Example

```
local
   proc {Ping N}
      for I in 1..N do
            {Delay 500} {Browse ping}
      end
      {Browse 'ping terminate'}
   end
   proc {Pong N}
      for I in 1..N do
           {Delay 600} {Browse pong}
      end
      {Browse 'pong terminate'}
   end
in .... end
```

```
local
....
in
   {Browse 'game started'}
   thread {Ping 1000} end
   thread {Pong 1000} end
end
```

Concurrent control abstraction

- We have seen how threads are forked by 'thread ... end'
- A natural question to ask is: how can we join threads?



Termination detection

- This is a special case of detecting *termination of multiple threads*, and making another thread wait on that event.
- The general scheme is quite easy because of dataflow variables:

```
thread ⟨S1⟩ X1 = unit end
thread ⟨S2⟩  X2 = X1 end
  ...
thread ⟨Sn⟩ Xn = Xn-1 end
{Wait Xn}
% Continue main thread
```

- When all threads terminate the variables $X_1 \ldots X_N$ will be merged together labeling a single box that contains the value **unit**.
- $\{\text{Wait } X_N\}$ suspends the main thread until X_N is bound.

Concurrent Composition

conc S_1 **[]** S_2 **[]** … **[]** S_n **end**

```
{Conc     [ proc{$} S1 end
            proc{$} S2 end
               ...
            proc{$} Sn end]  }
```

- Takes a single argument that is a list of nullary procedures.
- When it is executed, the procedures are forked concurrently. The next statement is executed only when all procedures in the list terminate.

Conc

```
local
   proc {Conc1 Ps I O}
      case Ps of P|Pr then
         M in
         thread {P} M = I end
         {Conc1 Pr M O}
      [] nil then O = I
      end
   end
in
   proc {Conc Ps}
     X in  {Conc1 Ps unit X}
     {Wait X}
    end
end
```

This abstraction takes a list of zero-argument procedures and terminate after all these threads have terminated

Example

```
local
   proc {Ping N}
      for I in 1..N do
            {Delay 500} {Browse ping}
      end
      {Browse 'ping terminate'}
   end
   proc {Pong N}
      for I in 1..N do
           {Delay 600} {Browse pong}
      end
      {Browse 'pong terminate'}
   end
in .... end
```

```
local
....
in
   {Browse 'game started'}
   {Conc
      [ proc {$} {Ping 1000} end
        proc {$} {Pong 1000} end  ]}
    {Browse 'game terminated'}
end
```

Futures

- A **future** is a read-only capability of a single-assignment variable. For example to create a future of the variable `X` we perform the operation `!!` to create a future `Y`: `Y = !!X`
- A thread trying to use the value of a future, e.g. using `Y`, will suspend until the variable of the future, e.g. `X`, gets bound.
- One way to execute a procedure lazily, i.e. in a demand-driven manner, is to use the operation `{ByNeed +P ?F}`.
- `ByNeed` takes a zero-argument function `P`, and returns a future `F`. When a thread tries to access the value of `F`, the function `{P}` is called, and its result is bound to `F`.
- This allows us to perform demand-driven computations in a straightforward manner.

Example

- ```
  declare Y
  {ByNeed fun {$} 1 end Y}
  {Browse Y}
  ```
- we will observe that `Y` becomes a future, i.e. we will see `Y<Future>` in the Browser.
- If we try to access the value of `Y`, it will get bound to `1`.
- One way to access `Y` is by perform the operation `{Wait Y}` which triggers the producing procedure.
  ```

Thread Priority and Real Time

- Try to run the program using the following statement:
  - `{Sum 0 thread {Generate 0 100000000} end}`
- Switch on the panel and observe the memory behavior of the program.
- You will quickly notice that this program does not behave well.
- The reason has to do with the asynchronous message passing. If the producer sends messages i.e. create new elements in the stream, in a faster rate than the consumer can consume, increasingly more buffering will be needed until the system starts to break down.
- One possible solution is to control experimentally the rate of thread execution so that the consumers get a larger time-slice than the producers do.

Priorities

- There are three priority levels:
  - *high*,
  - *medium,* and
  - *low* (the default)
- A priority level determines how often a runnable thread is allocated a time slice.
- In Oz, a high priority thread cannot starve a low priority one. Priority determines only how large piece of the processor-cake a thread can get.
- Each thread has a unique name. To get the name of the current thread the procedure `Thread.this/1` is called.
- Having a reference to a thread, by using its name, enables operations on threads such as:
  - terminating a thread, or
  - raising an exception in a thread.
- Thread operations are defined the standard module `Thread`.

Thread priority and thread control

```
fun {Thread.state T}                    %% returns thread state
proc{Thread.injectException T E}        %% exception E injected into thread
fun {Thread.this}                       %% returns 1st class reference to thread
proc{Thread.setPriority T P}            %% P is high, medium or low
proc{Thread.setThisPriority P}          %% as above on current thread

fun{Property.get priorities}            %% get priority ratios
proc{Property.put priorities(high:H medium:M)}
```

Thread Priorities

- Oz has three priority levels. The system procedure

```
{Property.put priorities p(medium:Y high:X)}
```

  - Sets the processor-time ratio to `X:1` between high-priority threads and medium-priority thread.
  - It also sets the processor-time ratio to `Y:1` between medium-priority threads and low-priority threads. `X` and `Y` are integers.
  - Example:

```
{Property.put priorities p(high:10 medium:10)}
```

- Now let us make our producer-consumer program work. We give the producer low priority, and the consumer high. We also set the priority ratios to `10:1` and `10:1`.

The program with priorities

```
local L in
   {Property.put priorities p(high:10 medium:10)}
   thread
      {Thread.setThisPriority low}
      L = {Generate 0 100000000}
   end
   thread
      {Thread.setThisPriority high}
      {Sum 0 L}
   end
end
```

Exercises

67. SALSA asynchronous message passing enables to tag messages with properties: *priority*, *delay*, and *waitfor*. Erlang uses a selective receive mechanism that can be used to implement priorities and delays. Compare these mechanisms with Oz thread priorities, time delays and alarms, and futures.
68. How do SALSA tokens relate to Oz dataflow variables and futures?
69. What is the difference between multiple thread termination detection in Oz, process groups in Erlang, and join blocks in SALSA?
70. CTM Exercise 4.11.3 (page 339)
    - Compare the sequential and concurrent execution performance of equivalent SALSA and Erlang programs.
71. CTM Exercise 4.11.5 (page 339)