Processes and Channels

Occam

Occam

- Occam based on Communicating Sequential Processes (CSP) formalism developed by Tony Hoare, Oxford, UK, and an experimental language by David May, Bristol, UK
- Designed to have a formal semantics suitable for automatic program transformations
- Many groups investigated direct translation of Occam into hardware

Cosmetics

- Keywords are in CAPITAL letters
- Variables may include '.'s
  - This.is.a.variable.name
- Scope is marked with indent spaces
  - No { } is uses
- Occam is line oriented
- Comments are anything following --

Occam Processes

- Not processes as we know them from operating systems
- More like procedures
- Or atomic blocks
- Think of them as structured actions
- EVERY LINE IS A PROCESS!!!

Structure

The structure of a program is a process with declarations preceding it.

<declares>
<process>

```
INT j:
SEQ
  j := 1
  j := j + 1
```

Precedents

- Nasty surprise – there is no mathematical precedents rules!!! ☹
- All ”complex” formulas must be parameterized to make the expression non-ambivalent
  - `x := 2 * y + 1 -- IS ILLEGAL`
  - `x := (2 * y) + 1 -- Is legal`

Conditionals - IF

- `IF` <<condition> <expression>>+
- ONE condition must be true!!!
  - Otherwise the process stops

```
IF
  n < 0
    sign := -1
  n = 0
    sign := 0
  n > 0
    sign := 1
```

```
IF
  n < 0
    sign := -1
  TRUE
    sign := 1
```

Interval IF

- ```
  IF
    i=0
      j:=2
    i=1
      j:=1
    i=2
      j:=0
  ```
- ```
  IF k = 0 FOR 2
    k=i
      j:=2-k
  ```

Channels

- Channels connects processes and are the basis for the Communication part of the CSP implementation
- Channels are all rendez-vous

Channels

- Channels are of a given type
  - `CHAN OF INT q:`
  - `CHAN OF ANY link:`
  - `[n+1] CHAN OF ANY links:`

Send !

- Send the value of a variable down a channel
- <channel> ! <value>
- `ch ! y + 1`
- A set of values can also be sent
  - `ch ! x; y; x + y`

Receive ?

- Receive from channel into a variable
- ? <variable>
- `Ch ? X`
- A set of values can also be received
  - `ch ? x; y; z`

SEQ

- A sequential block of code can be defined by a `SEQ` statement
- ```
  IF
    c < 2
      SEQ
        a:=1
        b:=2
    TRUE
        c:=3
  ```
  ```

PAR

- Parallel blocks of code can be defined using `PAR`
- After a PAR each line of code is an individual process
- ```
  IF
    c<2
      PAR
        a ! 1
        b ! 2
    TRUE
      PAR
        a ? x
        b ? y
  ```
  ```

PAR

```
PAR
  a ? x
  a ! 2
```

What does this mean???

```
x:=2
```

Procedures

- We can structure our programs further by using procedures
- **Call by reference!**
- PROC <name> {(params)}<body>:
-
```
PROC add.one(INT param)
  param:=param+1
:
```

Call by value

- Call by value can be forced by adding the VAL keyword before a parameter
- ```
  PROC add.one(VAL INT param)
    new.param:=param+1
  : -- this is now pointless:)
  ```
  ```

A Procedure Example

```
PROC Average ([]REAL32 Data, REAL32 Res)
  SEQ
    Res:=0.0
    SEQ i = 0 FOR SIZE Data
      Res := Res + Data[i]
    Res := Res/(REAL32 ROUND (SIZE Data))
:
```

Hello World

```
PROC hello.world (CHAN OF BYTE keyboard, screen, error)
  --{{{
  SEQ
    screen ! 'H'
    screen ! 'e'
    screen ! 'l'
    screen ! 'l'
    screen ! 'o'
    screen ! ' '
    screen ! 'W'
    screen ! 'o'
    screen ! 'r'
    screen ! 'l'
    screen ! 'd'
    screen ! '*c'
    screen ! '*n'
  --}}}
:
```

Hello World (2)

```
PROC hello.world (CHAN OF BYTE keyboard, screen, error)
  --{{{
  VAL []BYTE greeting IS "Hello World*c*n":
  SEQ i = 0 FOR SIZE greeting
    screen ! greeting[i]
  --}}}
:
```

Hello World (3)

```
#USE "course.lib"

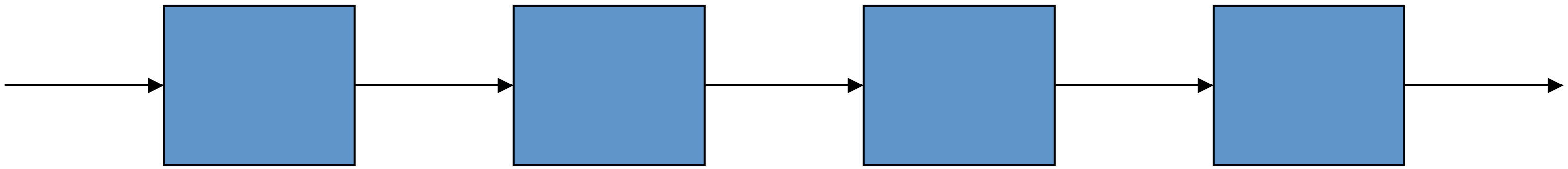
PROC hello.world (CHAN OF BYTE keyboard, screen, error)
  --{{{
  out.string ("Hello World*c*n", 0, screen)
  --}}}
:
```

Echo

```
#INCLUDE "consts.inc"

PROC echoing (CHAN OF BYTE keyboard, screen, error)
  --{{{
  BYTE ch:
  SEQ
    ch := ' '
    WHILE ch <> 'Z'
      SEQ
        keyboard ? ch
        screen ! ch
        screen ! FLUSH
    screen ! '*c'
    screen ! '*n'
  --}}}
:
```

A simple FIFO buffer



A simple FIFO buffer

```
VAL INT N IS 4:
[N + 1] CHAN OF INT C:
PAR P = 0 FOR N
  INT Value:
  WHILE TRUE
    SEQ
      C[P] ? Value
      C[P + 1] ! Value
```

Compile time limitations

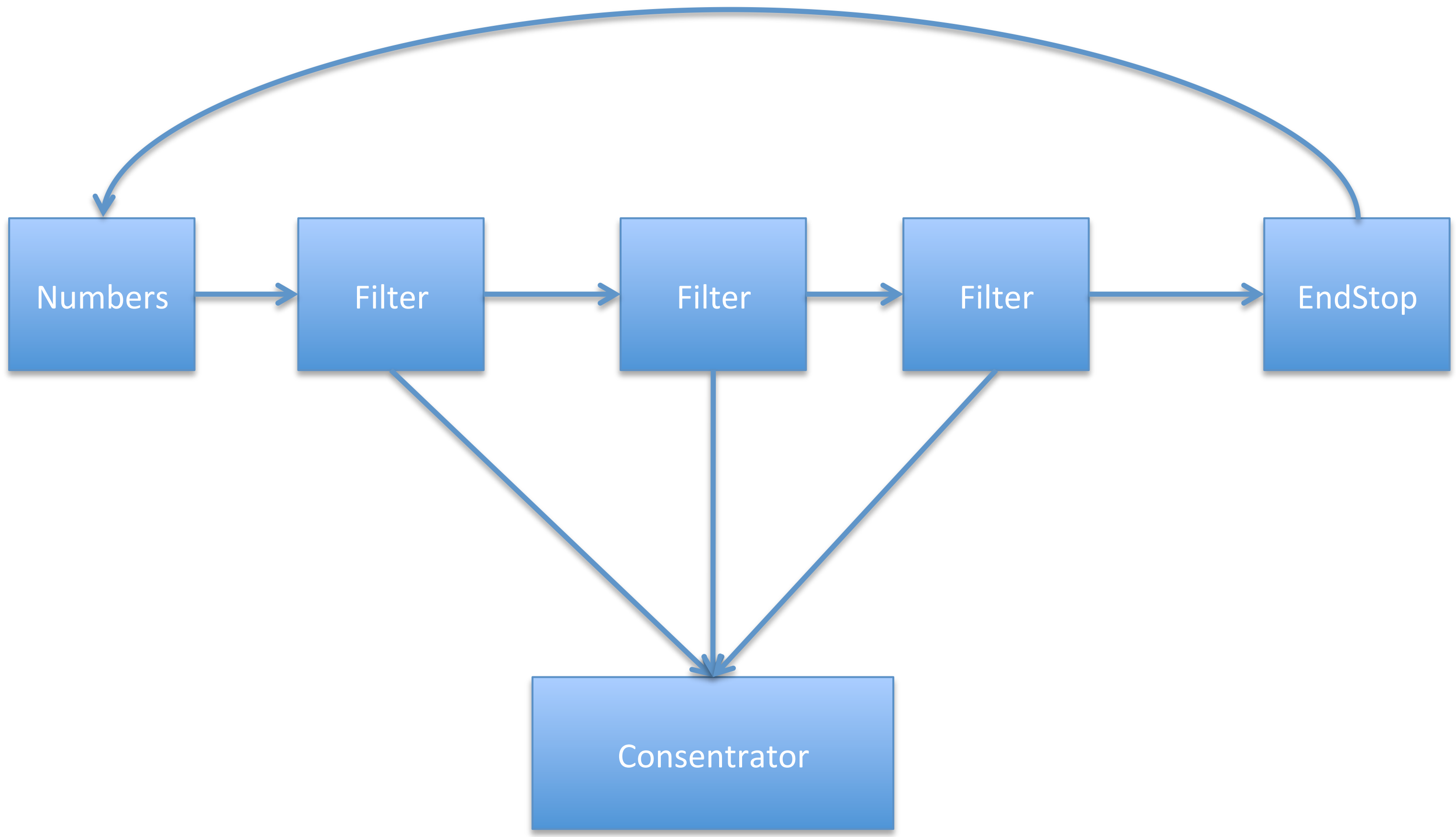
```
INT N.PROC:
SEQ
  Ch ? N.PROC
  PAR P = 0 FOR N.PROC
    par.process
```

```
INT N.PROC:
VAL INT MAX IS 100:
SEQ
  Ch ? N.PROC
  IF N.PROC > MAX
    STOP
  PAR P = 0 FOR N.PROC
    par.process
```

Sieve in Occam

- Generate Prime numbers
  - With a lot of processes ☺

Sieve in Occam



Numbers

```
PROC Numbers(CHAN OF INT in, out)
  INT i:
  SEQ
    i:=2
    WHILE i <> EndToken
      PRI ALT
        in ? i
          SKIP
        TRUE & SKIP
          SEQ
            out ! i
            i := i+1
:
```

EndStop

```
PROC EndSTOP(CHAN OF INT in, out)
  INT temp:
  SEQ
    in ? temp
    PAR
      out ! EndToken
      WHILE temp <> EndToken
        in ? temp
:
```

Filter

```
PROC Filter(CHAN OF INT left, right, down)
  INT p,q:
  SEQ
    left ? p
    q:=1
    PAR
      down ! p
      WHILE q <> EndToken
        SEQ
          left ? q
          IF
            q = EndToken
              SKIP
            (q\p)<>0
              right ! q
            TRUE
              SKIP
  right ! EndToken
:
```

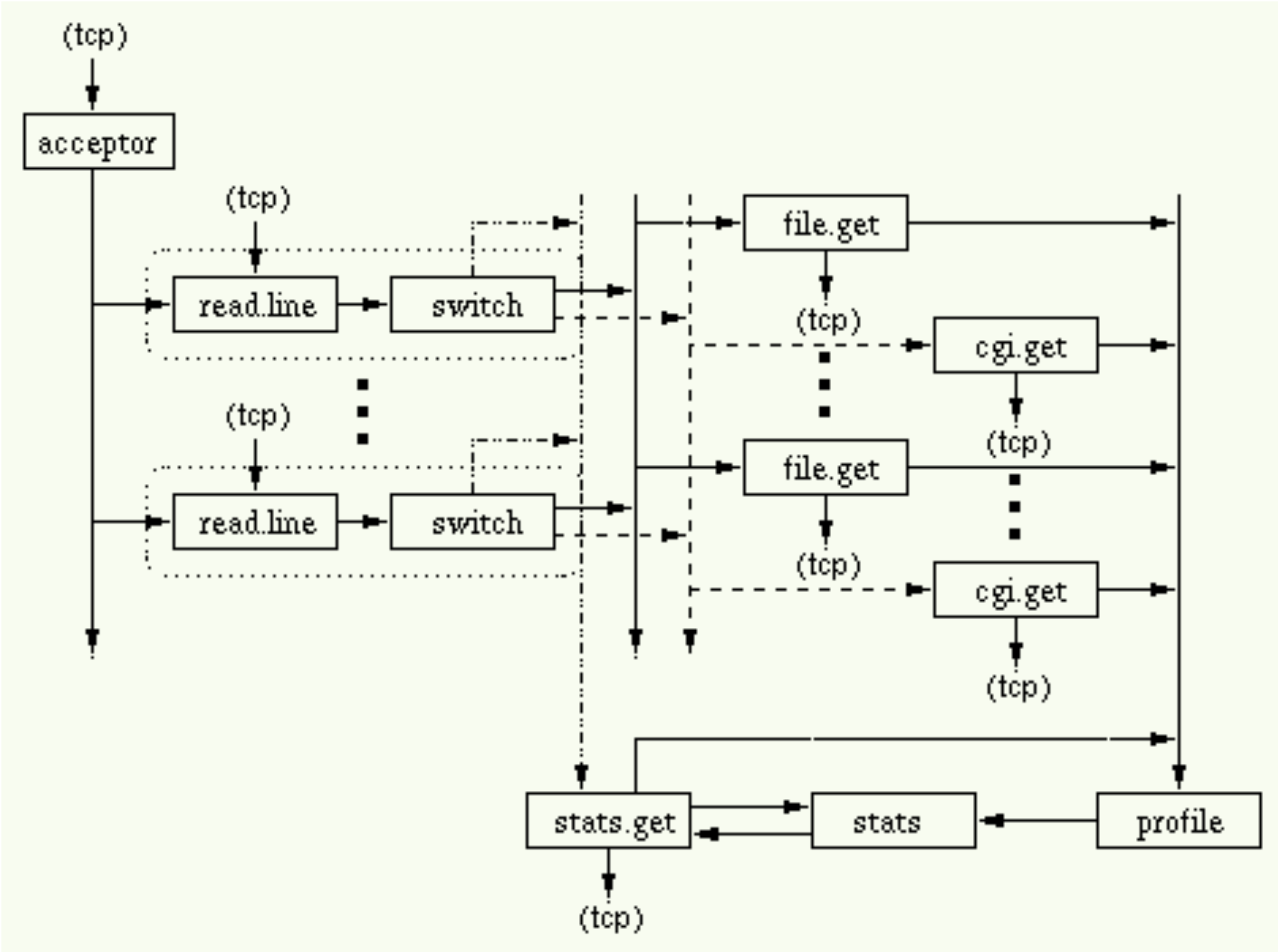
Concentrator

```
PROC Concentrator([]CHAN OF INT in,
                    CHAN OF INT out)
  INT p:
  SEQ i = 0 FOR SIZE in
    SEQ
      in [i] ? p
      out ! p
:
```

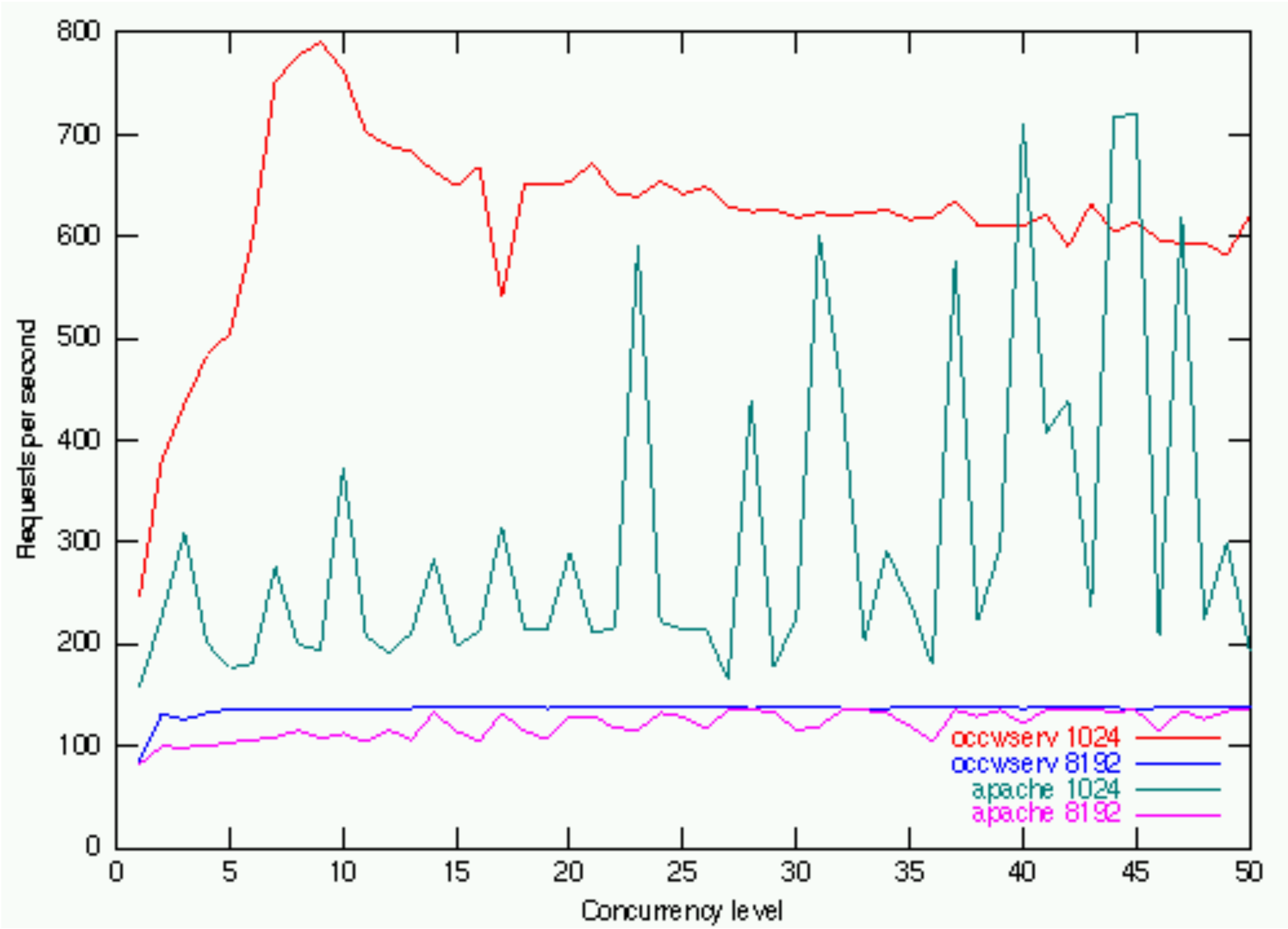
Sieve

```
VAL INT N is 30:
PROC Generate(CHAN OF INT Primes)
  VAL INT EndToken IS 0:
  PROC Numbers(CHAN OF INT in, out)
  PROC EndSTOP(CHAN OF INT in, out)
  PROC Filter(CHAN OF INT, left, right, down)
  PROC Concentrator([]CHAN OF INT in,
                    CHAN OF INT out)
  [N+1]CHAN OF INT InterFilter:
  [N]CHAN OF INT PC:
  CHAN INT OK.STOP:
  PAR
    Numbers(OK.STOP, InterFilter[0])
    PAR i = 0 FOR N
      Filter(InterFilter[i], InterFilter[i+1], PC[i])
    EndStop(InterFilter[N], OK.STOP)
    Concentrator(PC, Primes)
:
```

http://wotug.ukc.ac.uk/ocweb/



ocweb Performance



Occam for Linux

- KRoC
  - http://www.cs.ukc.ac.uk/projects/ofa/kroc/
- Easy to use

Occam OS

- KRoC runtime library ported to be an operating system in its own right RMoX

Java

CSP for Java (JCSP)

- A ***process*** is an object of a class implementing the ***CSProcess*** interface:

```
interface CSProcess {
  public void run();
}
```

- The *behaviour* of the process is determined by the body given to the `run()` method in the implementing class.

JCSP Process Structure

```
class Example implements CSProcess {

   ...  private shared synchronisation objects
        (channels etc.)
   ...  private state information

   ...  public constructors
   ...  public accessors(gets)/mutators(sets)
        (only to be used when not running)

   ...  private support methods (part of a run)
   ...  public void run() (process starts here)

}
```

Two Sets of Channel Classes (and Interfaces)

`Object` channels

- carrying (references to) arbitrary Java objects

`int` channels

- carrying Java **`int`**s

Channel Interfaces and Classes

- Channel interfaces are what the processes see. Processes only need to care what kind of data they carry (**`int`**s or **`Object`**s) and whether the channels are for output, input or ***ALTing*** (i.e. *choice*) input.

- It will be the network builder's concern to choose the actual channel **classes** to use when connecting processes together.

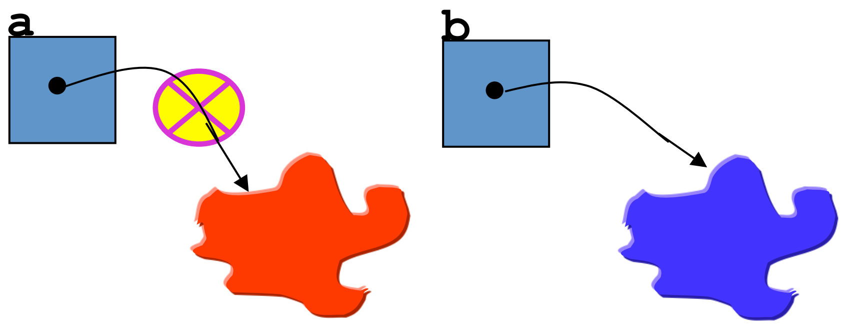
`int` Channels

- The int channels are convenient and secure.
- For completeness, **JCSP** should provide channels for carrying all of the Java primitive data-types. These would be trivial to add. So far, there has been no pressing need.

`Object` Aliasing - Danger !!

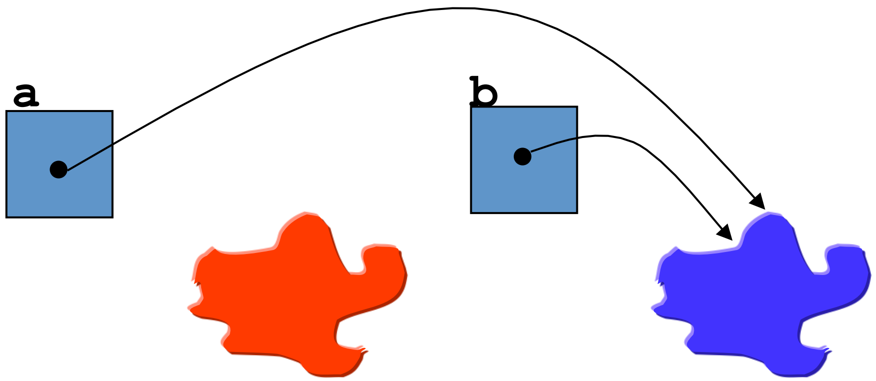
Java objects are referenced through variable names.

`Thing a = ..., b = ...;`



`a` and **`b`** are now ***aliases*** for the same object!

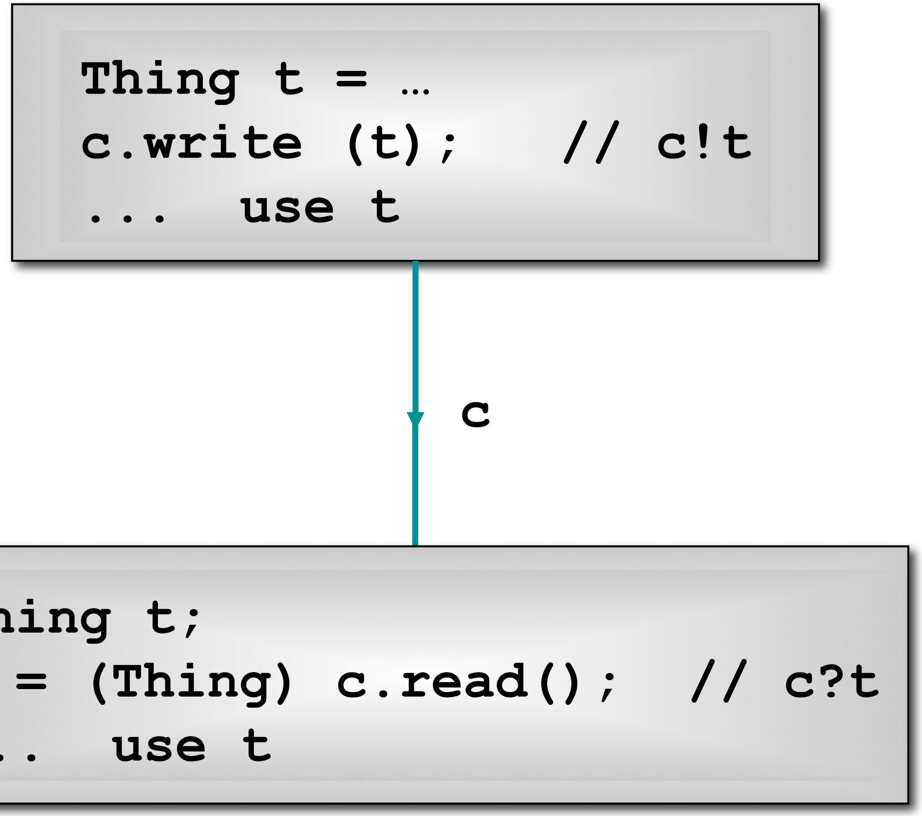
`a = b;`



`Object` Channels - Danger !!

- **`Object`** channels expose a danger

- Channel communication only communicates the **`Object`** reference.



`Object` Channels - Danger !!

- After the communication, each process has a reference (in its variable **`t`**) to the ***same*** object.

- If ***one*** of these processes modifies that object (in its … use **`t`**), the ***other*** one had better forget about it!

```
Thing t = …
c.write (t);    // c!t
...  use t
```

`c`

```
Thing t;
t = (Thing) c.read();  // c?t
...  use t
```

`Object` Channels - Danger !!

- Otherwise, the parallel usage rule is violated and we will be at the mercy of ***when*** the processes get scheduled for execution - a **RACE HAZARD**!

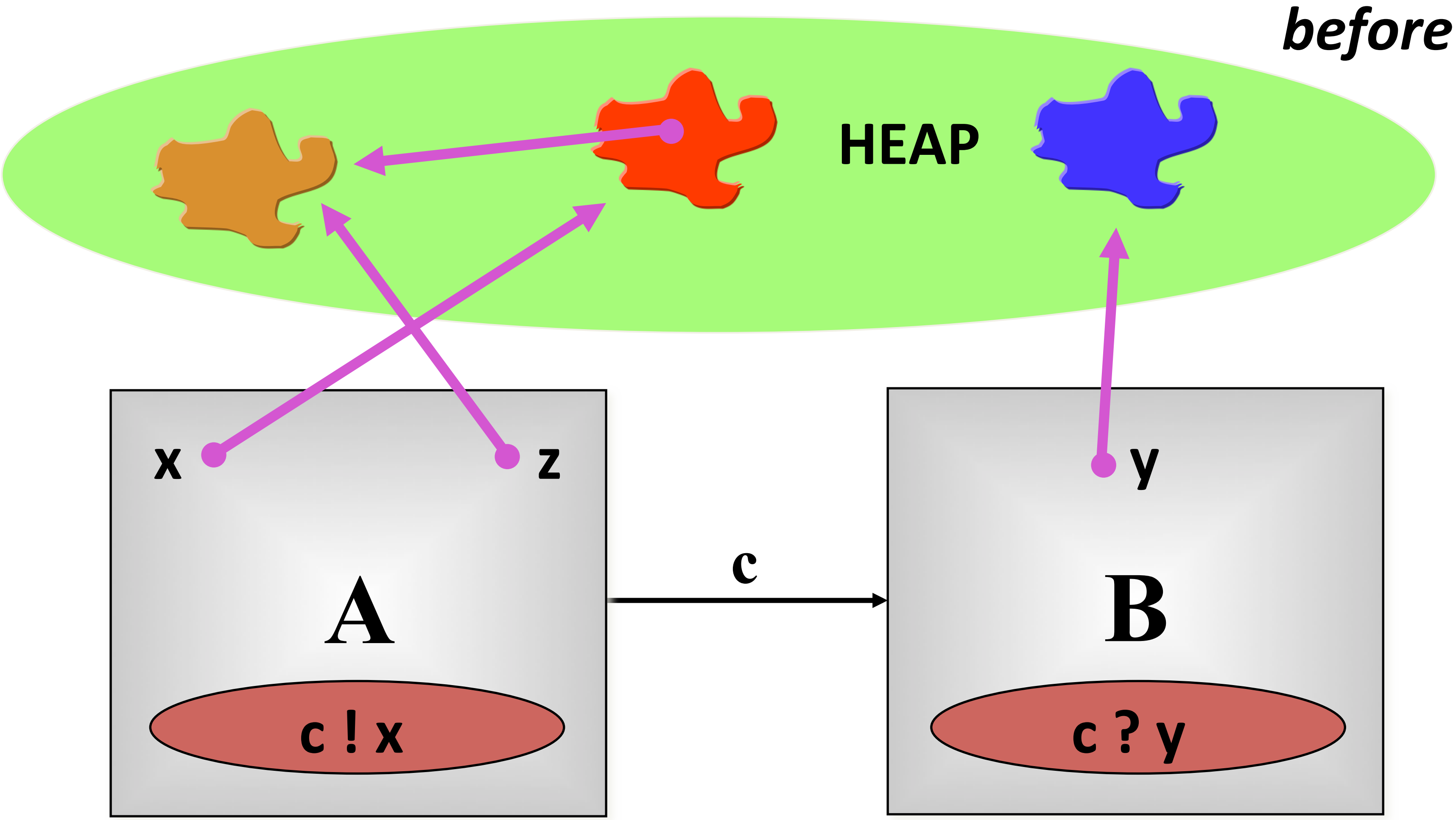

```
Thing t = …
c.write (t);    // c!t
...   use t
```

c

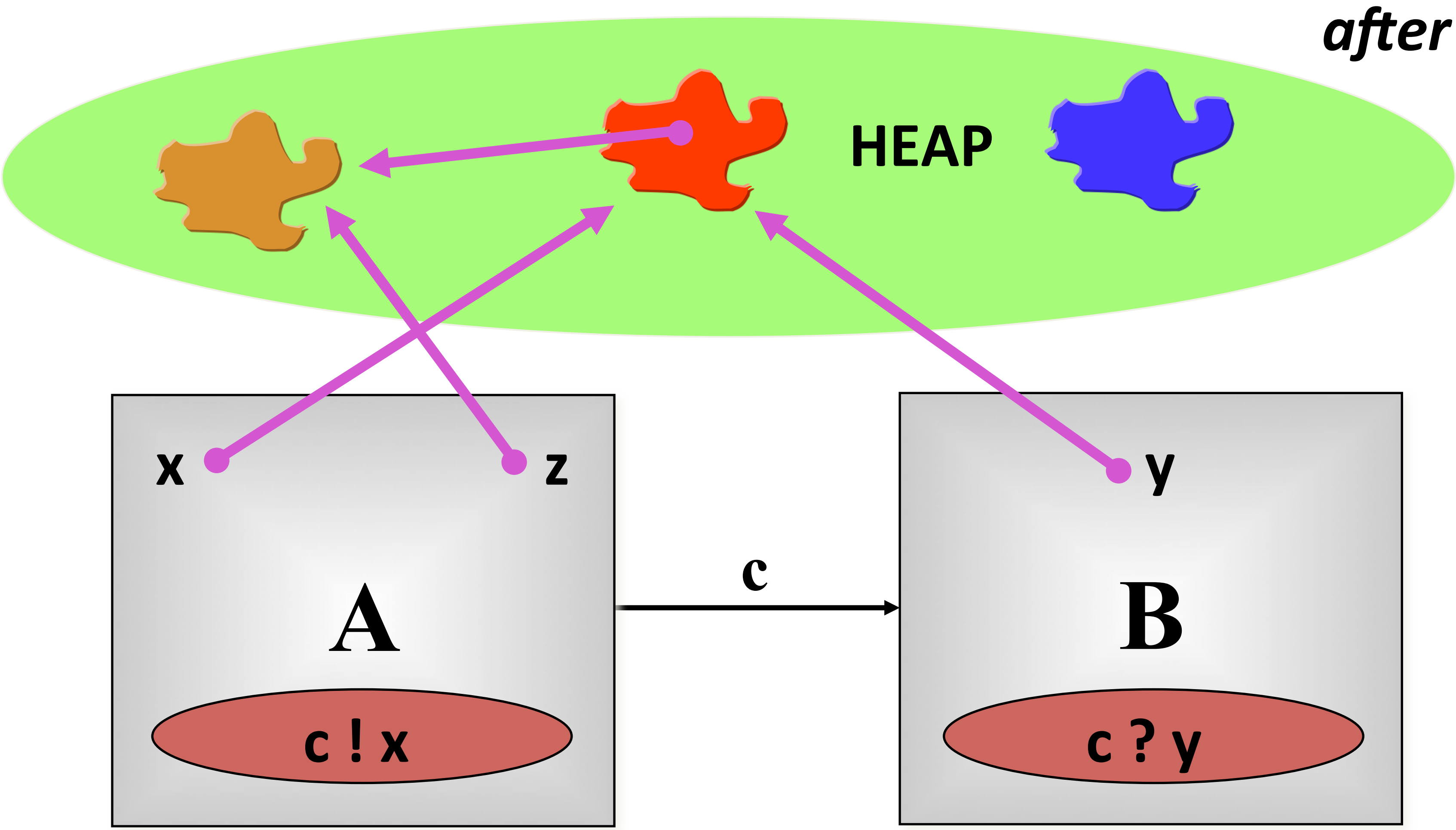
```
Thing t;
t = (Thing) c.read();   // c?t
...   use t
```

- We have design patterns to prevent this.

Reference Semantics

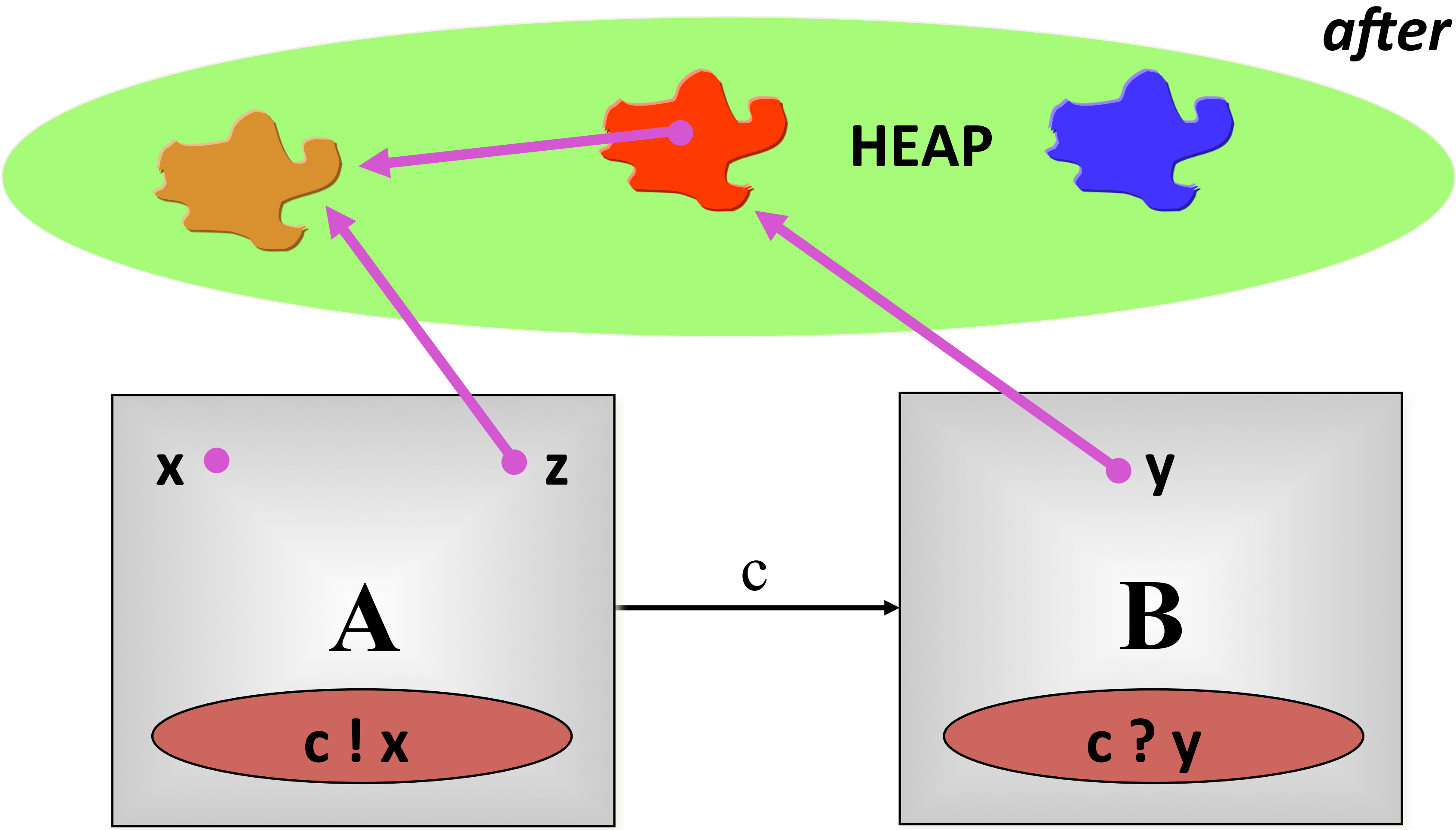


Reference Semantics



Red and brown objects are parallel compromised!

Reference Semantics



Even if the source variable is nulled, brown is done for!!

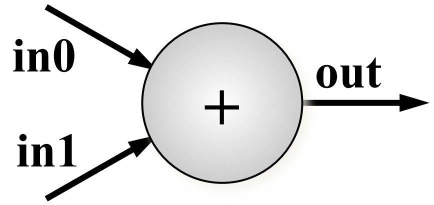

```
class SuccInt implements CSProcess {

  private final ChannelInputInt in;
  private final ChannelOutputInt out;

  public SuccInt (ChannelInputInt in,
                  ChannelOutputInt out) {
    this.in = in;
    this.out = out;
  }

  public void run () {
    while (true) {
      int n = in.read ();
      out.write (n + 1);
    }
  }

}
```



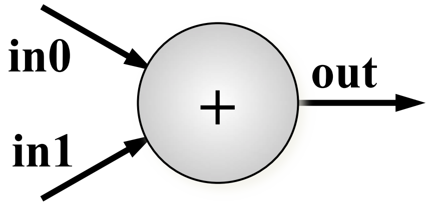

```
class PlusInt implements CSProcess {

  private final ChannelInputInt in0;
  private final ChannelInputInt in1;
  private final ChannelOutputInt out;


  public PlusInt (ChannelInputInt in0,
                  ChannelInputInt in1,
                  ChannelOutputInt out) {
    this.in0 = in0;
    this.in1 = in1;
    this.out = out;
  }

  ...  public void run ()

}
```




```
class PlusInt implements CSProcess {

  ...  private final channels (in0, in1, out)

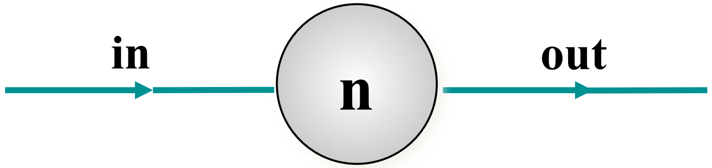
  ...  public PlusInt (ChannelInputInt in0, ...)

  public void run () {
    while (true) {
      int n0 = in0.read ();
      int n1 = in1.read ();
      out.write (n0 + n1);
    }
  }

}
```

serial ordering

Note: the inputs really need to be done in parallel - later!




```
class PrefixInt implements CSProcess {

  private final int n;
  private final ChannelInputInt in;
  private final ChannelOutputInt out;

  public PrefixInt (int n, ChannelInputInt in,
                    ChannelOutputInt out) {
    this.n = n;
    this.in = in;
    this.out = out;
  }

  public void run () {
    out.write (n);
    new IdInt (in, out).run ();
  }

}
```

Process Networks

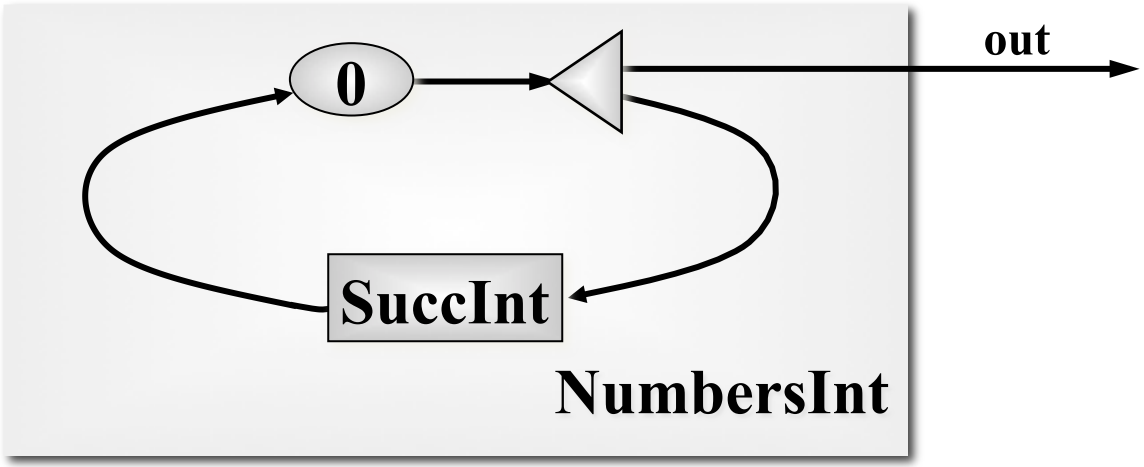
- We now want to be able to take instances of these ***processes*** (or components) and connect them together to form a network.
- The resulting network will itself be a ***process***.
- To do this, we need to construct some real wires - these are instances of the ***channel*** classes.
- We also need a way to compose everything together - the **`Parallel`** constructor.

Parallel

- ***Parallel*** is a **`CSProcess`** whose constructor takes an array of **`CSProcess`**es.

- Its ***run()*** method is the parallel composition of its given **`CSProcess`**es.
- The semantics is the same as for the CSP **||**.

- The ***run()*** terminates when and only when all of its component processes have terminated.



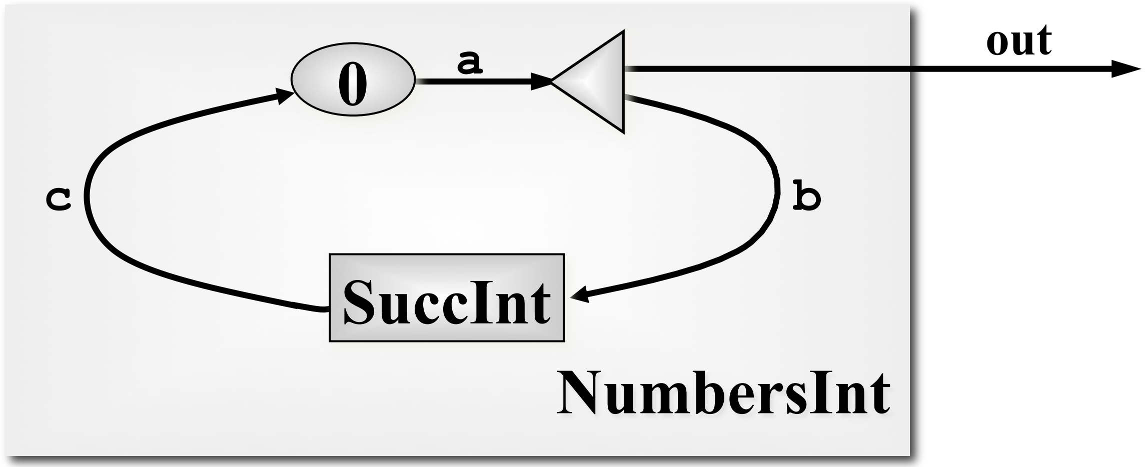

```
class NumbersInt implements CSProcess {

  private final ChannelOutputInt out;

  public NumbersInt (ChannelOutputInt out) {
    this.out = out;
  }

  ...  public void run ()

}
```

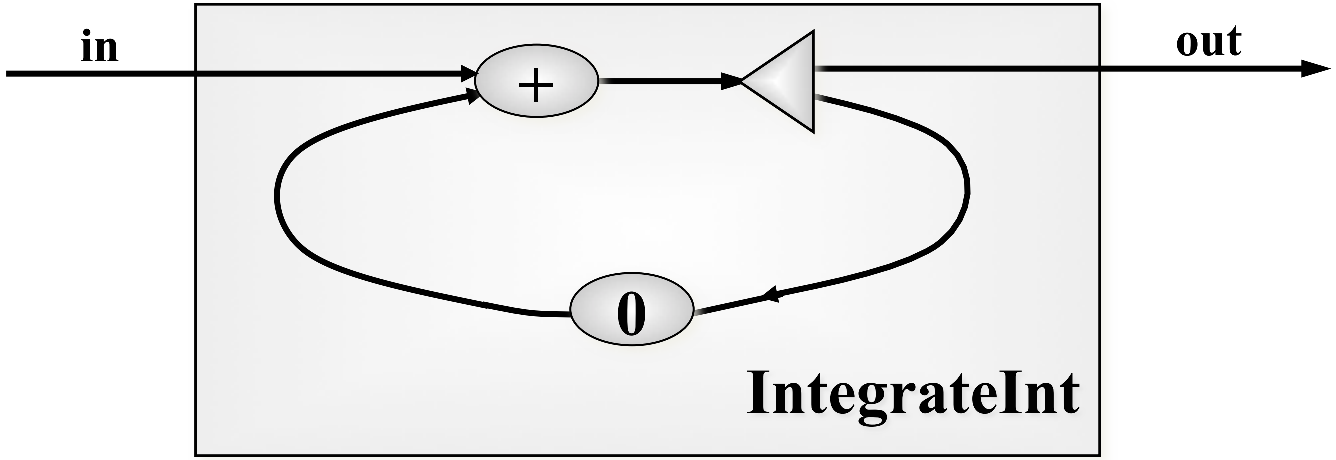



```
public void run () {

  One2OneChannelInt a = new One2OneChannelInt ();
  One2OneChannelInt b = new One2OneChannelInt ();
  One2OneChannelInt c = new One2OneChannelInt ();

  new Parallel (
    new CSProcess[] {
      new PrefixInt (0, c, a),
      new Delta2Int (a, out, b),
      new SuccInt (b, c)
    }
  ).run ();

}
```



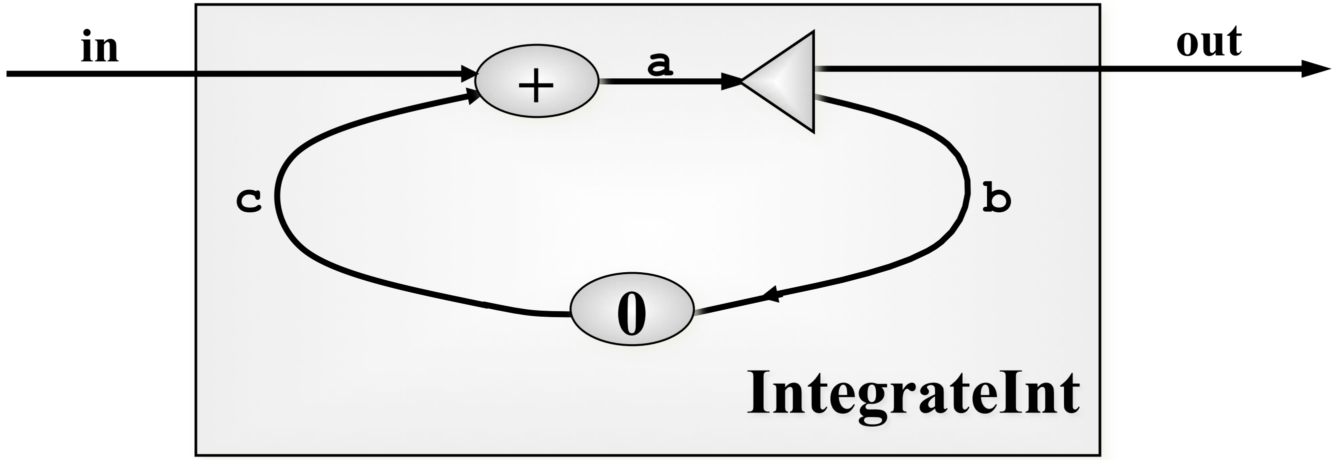

```
class IntegrateInt implements CSProcess {

  private final ChannelInputInt in;
  private final ChannelOutputInt out;

  public IntegrateInt (ChannelInputInt in,
                       ChannelOutputInt out) {
    this.in = in;
    this.out = out;
  }

  ...  public void run ()

}
```

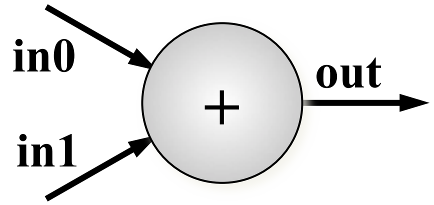



```
public void run () {

  One2OneChannelInt a = new One2OneChannelInt ();
  One2OneChannelInt b = new One2OneChannelInt ();
  One2OneChannelInt c = new One2OneChannelInt ();

  new Parallel (
    new CSProcess[] {
      new PlusInt (in, c, a),
      new Delta2Int (a, out, b),
      new PrefixInt (0, b, c)
    }
  ).run ();

}
```




```
class PlusInt implements CSProcess {

  ...  private final channels (in0, in1, out)

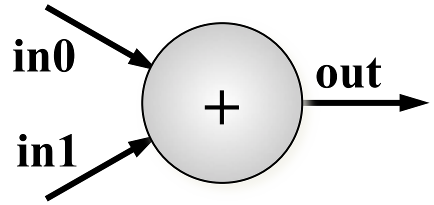
  ...  public PlusInt (ChannelInputInt in0, ...)

  public void run () {
    while (true) {
      int n0 = in0.read ();
      int n1 = in1.read ();
      out.write (n0 + n1);
    }
  }

}
```

Change this!

Note: the inputs really need to be done in parallel - now!



This process does one input and terminates.

```
public void run () {

  ProcessReadInt readIn0 = new ProcessReadInt (in0);
  ProcessReadInt readIn1 = new ProcessReadInt (in1);

  CSProcess parRead =
    new Parallel (new CSProcess[] {readIn0, readIn1});

  while (true) {
    parRead.run ();
    out.write (readIn0.value + readIn1.value);
  }

}
```

Note: the inputs are now done in parallel.

Implementation Note

- A **JCSP** **`Parallel`** object runs its first (n-1) components in *separate* Java threads and its last component in *its own* thread of control.
- When a **`Parallel.run()`** terminates, the **`Parallel`** object parks all its threads for reuse in case the **`Parallel`** is run again.
- So processes like **`PlusInt`** incur the overhead of Java thread creation *only during its first cycle*.
- That's why we named the **`parRead`** process before loop entry, rather than constructing it anonymously each time within the loop.

Termination

- Termination in JCSP is done exactly as in C++CSP which will be covered next
  - Since C++CSP introduced the mechanism it will be covered there

Assignment 1

FIG. 7.

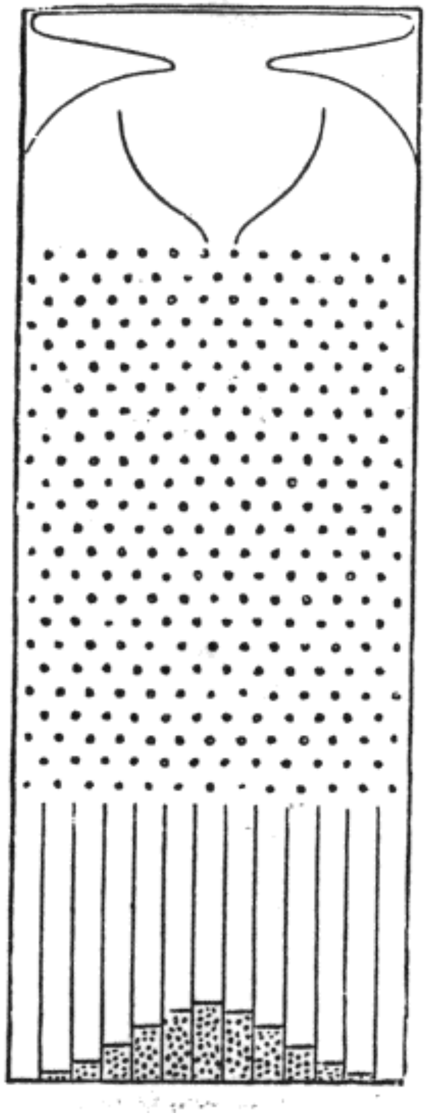


FIG. 8.

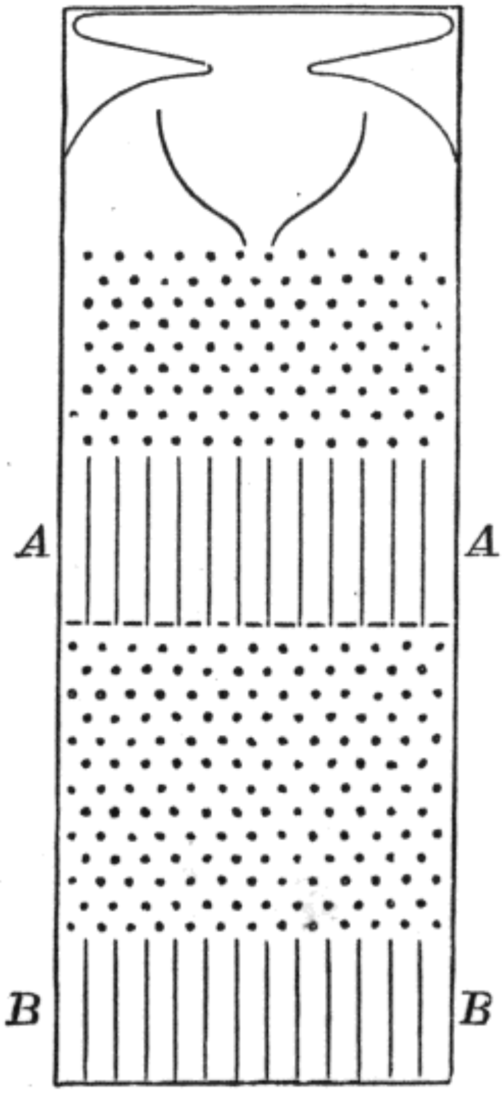
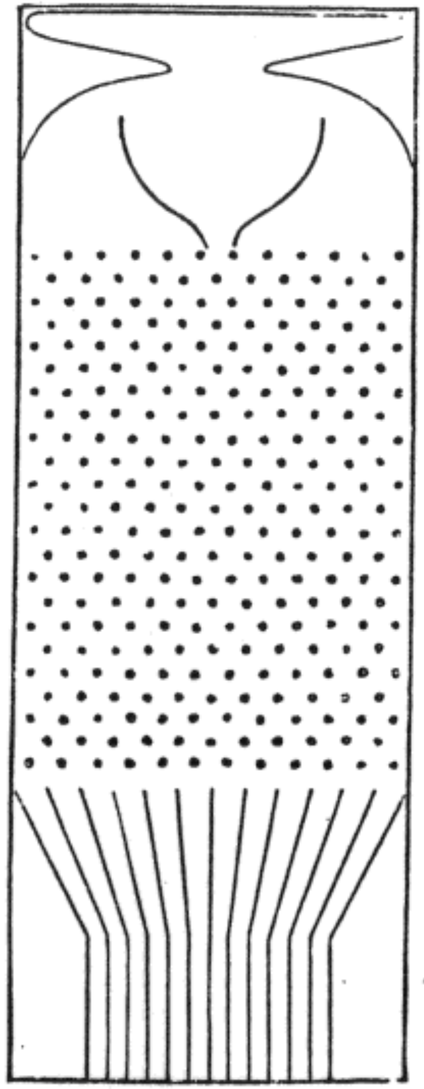


FIG. 9.

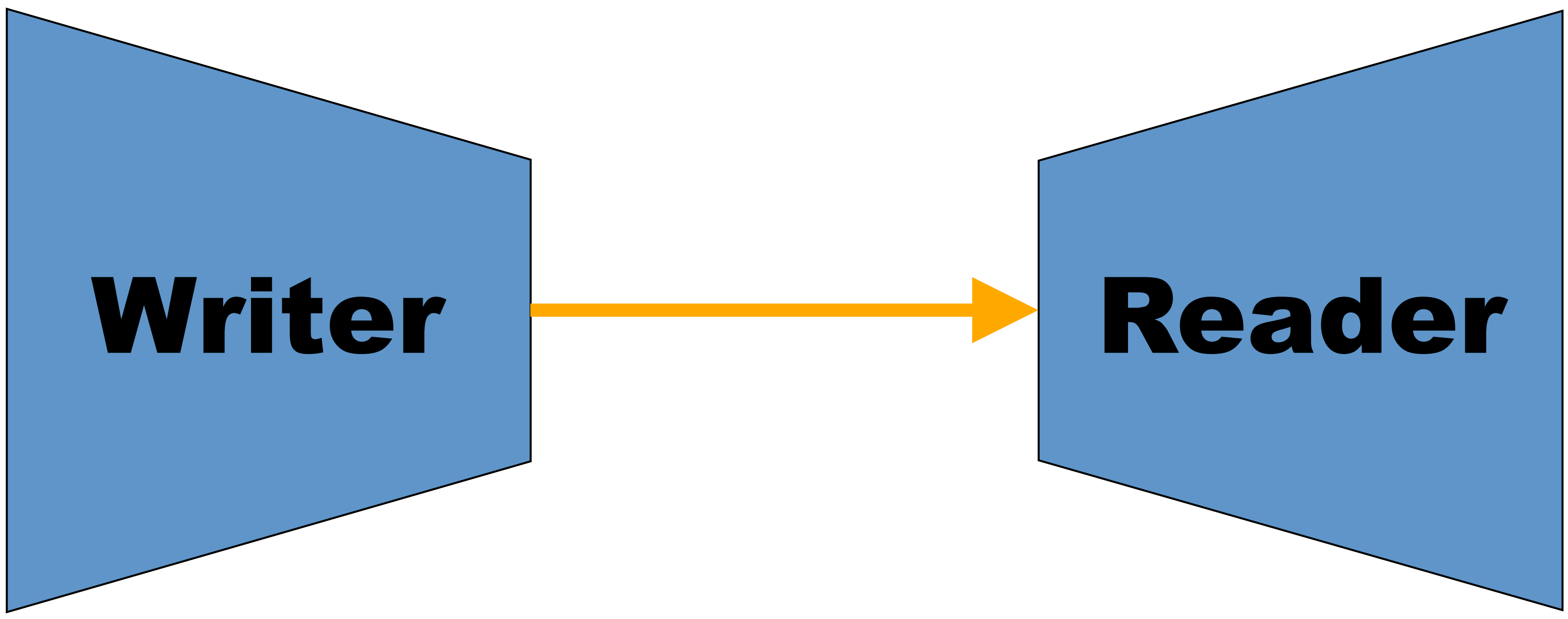


C++

C++ Library Overview

- A process is a subclass of `CSProcess`, implementing the `run` method
- Channels are templated, so they can communicate any type
- Communication is done using channel end objects, not the channel itself

Our First Processes



Our First Processes

```
class Writer
  :  public CSProcess
{
private:
  Chanout<int> out;
protected:
  void run();
public:
  Writer(
    const Chanout<int>& o
  );
};
```

`Chanout<int>` is the writing end of a channel of integers

Our First Processes

```
class Reader
  :  public CSProcess
{
private:
  Chanin<int> in;
protected:
  void run();
public:
  Reader(
    const Chanin<int>& i
  );
};
```

`Chanin<int>` is the reading end of a channel of integers

Our First Processes

```
void Writer::run() {
  int i;
  for (i = 0;i < 100;i++) {
    out << i;
  }
}

void Reader::run() {
  int i,n;
  for (i = 0;i < 100;i++) {
    in >> n;
  }
}
```

Input and output are done with simple operators, as in *occam*

Our First Processes

```
void function()
{
  One2OneChannel<int> channel;

  Parallel(
    new Writer(channel.writer()),
    new Reader(channel.reader()),
  NULL);
}

void main() {
  Start_CSP();
  function();
  End_CSP();
}
```

Where appropriate, JCSP's API is copied

Use `writer`/`reader` calls to get channel ends

`Parallel` takes a `NULL`-terminated list of process pointers

The `Start_CSP`/`End_CSP` functions must be called before/after the library is used

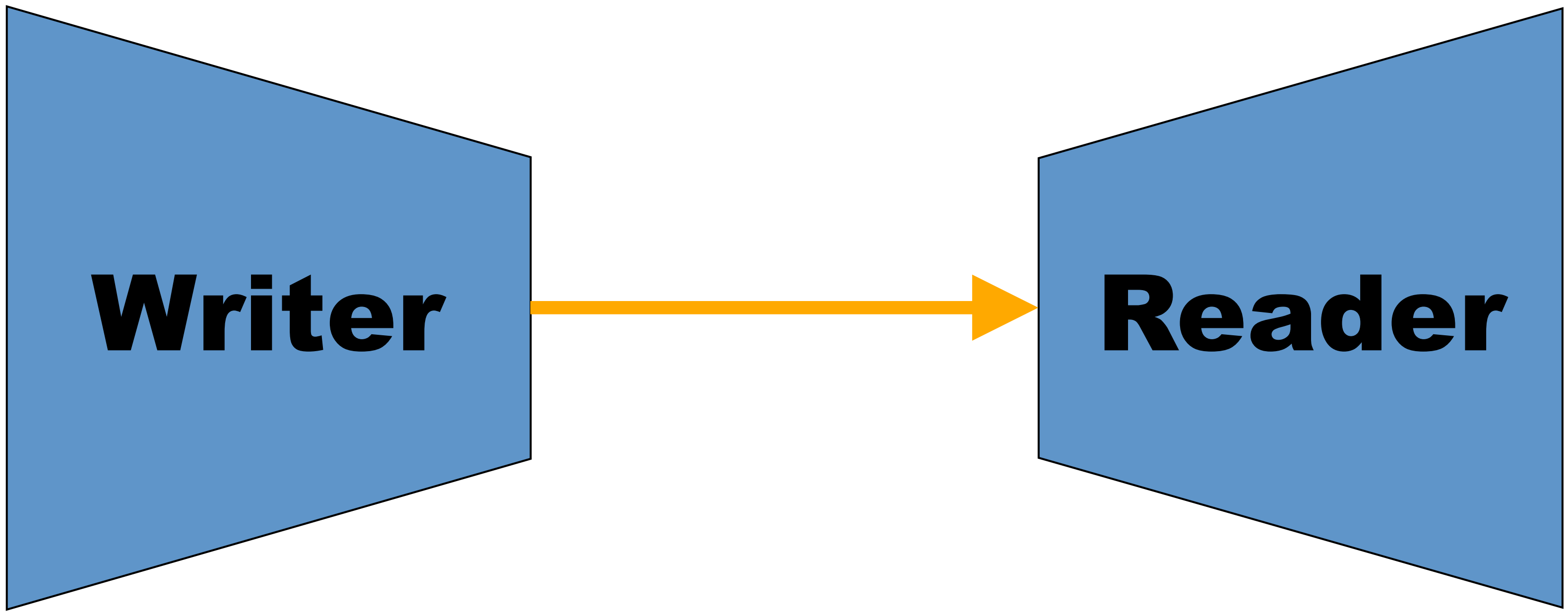
Our First Processes

```
void Writer::run() {
  int i;
  for (i = 0;i < 100;i++) {
    out << i;
  }
}

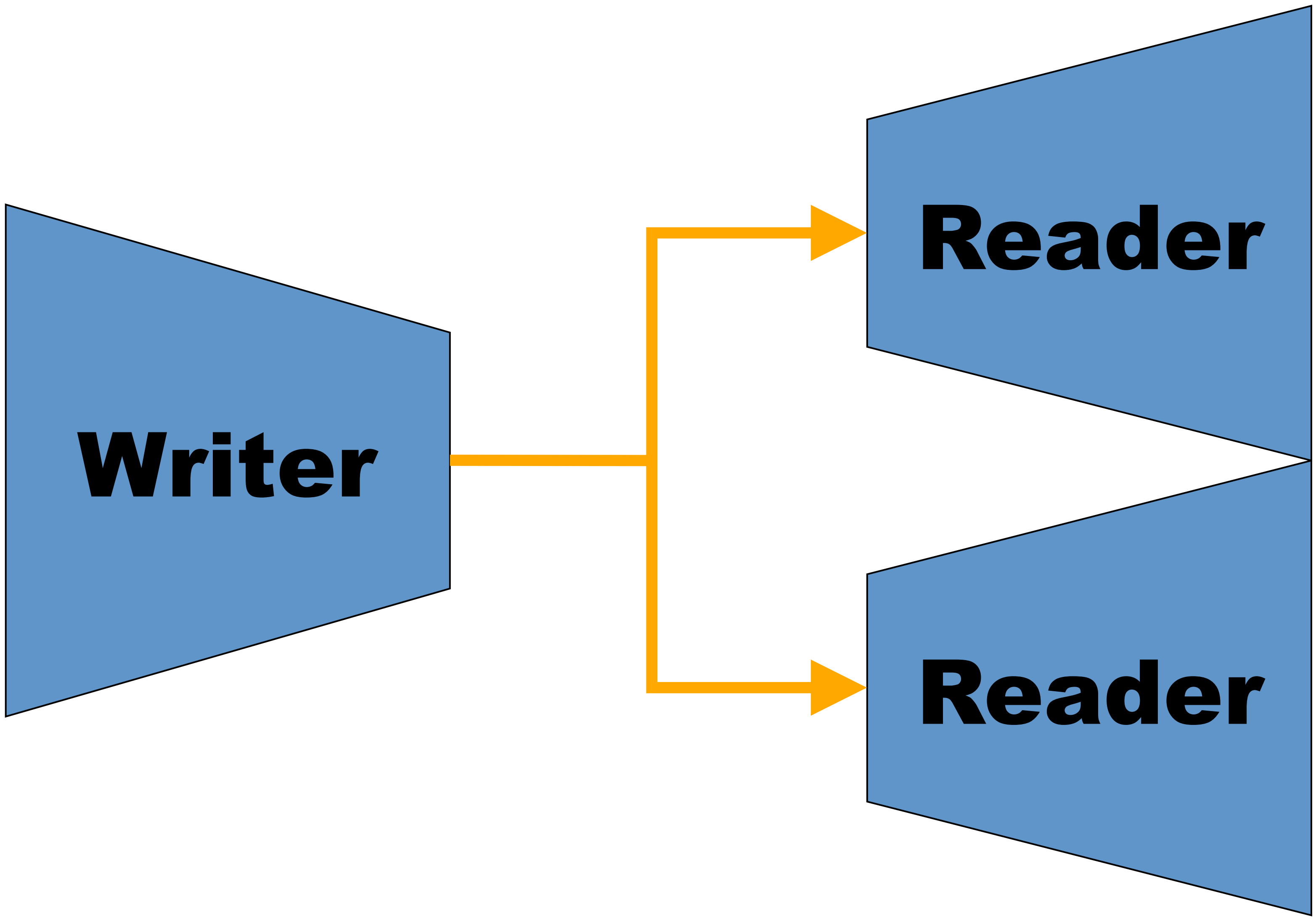
void Reader::run() {
  int i,n;
  for (i = 0;i < 100;i++) {
    in >> n;
  }
}
```

The `for` loop is inflexible - maybe there is another way? ....

Shared Channels



Shared Channels



Shared Channels

```
One2OneChannel<int> channel;

Parallel(
  new Writer(channel.writer()),
  new Reader(channel.reader()),
NULL);
```

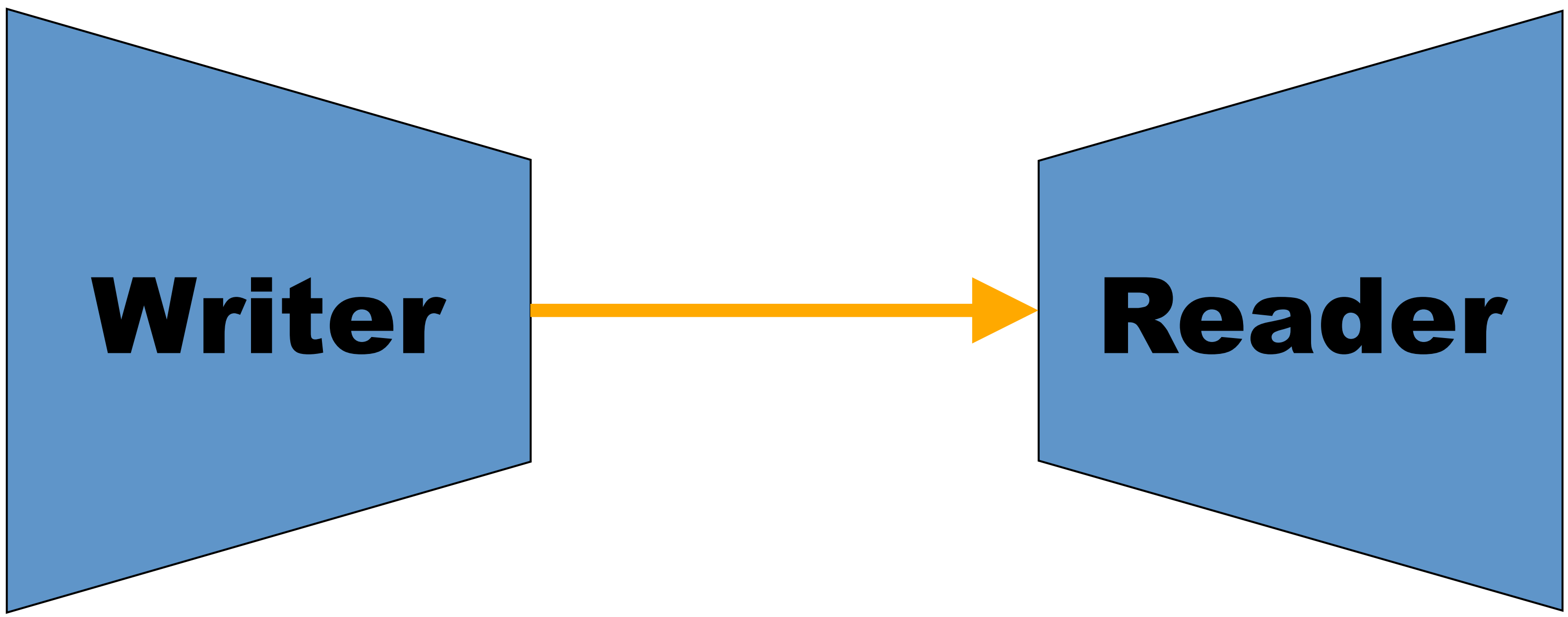
```
One2AnyChannel<int> channel;

Parallel(
  new Writer(channel.writer()),
  new Reader(channel.reader()),
  new Reader(channel.reader()),
NULL);
```

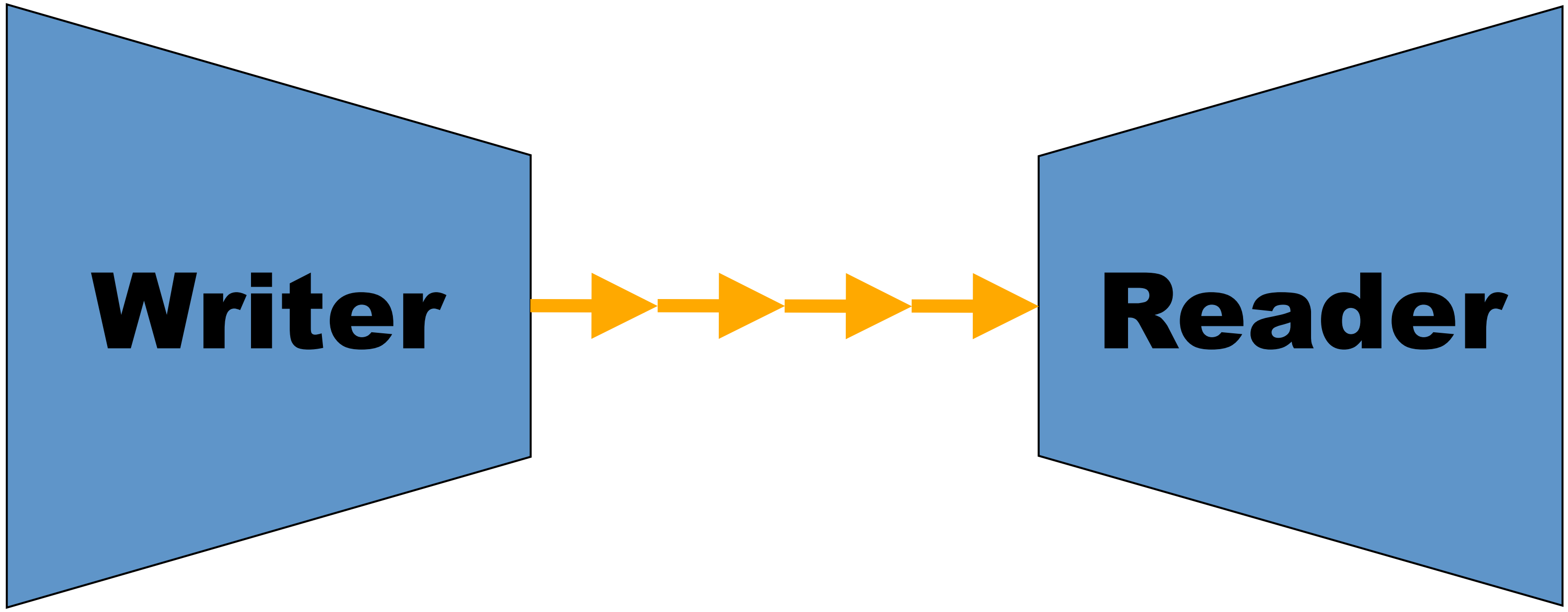
Change the channel type and add an extra reader - simple as that!

Buffered Channels

Buffered Channels



Buffered Channels



Buffered Channels

```
One2OneChannel<int> channel;

Parallel(
  new Writer(channel.writer()),
  new Reader(channel.reader()),
NULL);
```

```
One2OneChannelX<int>
  channel( Buffer<int>(4) );

Parallel(
  new Writer(channel.writer()),
  new Reader(channel.reader()),
NULL);
```

Note the added X!

Pass in the buffer object you want to use for buffering - like JCSP, C++CSP takes a copy of it rather than use the original

Parallel Communications

```
void Delta::run() {
  int n;
  ParallelComm pc(
    outA.parOut(&n),
    outB.parOut(&n),
  NULL);

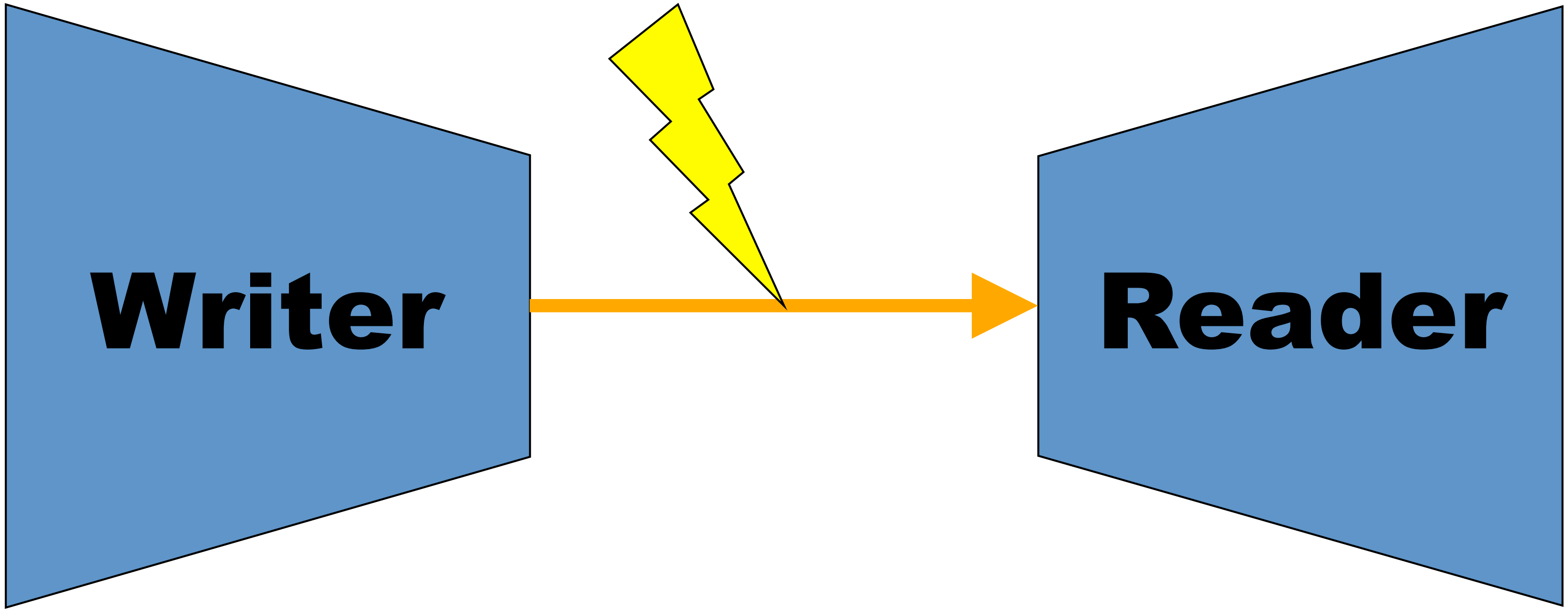
  while (true) {
    in >> n;
    pc.communicate();
  }
}
```

Like other library calls, the `ParallelComm` constructor takes a `NULL`-terminated list of `ParCommItems`

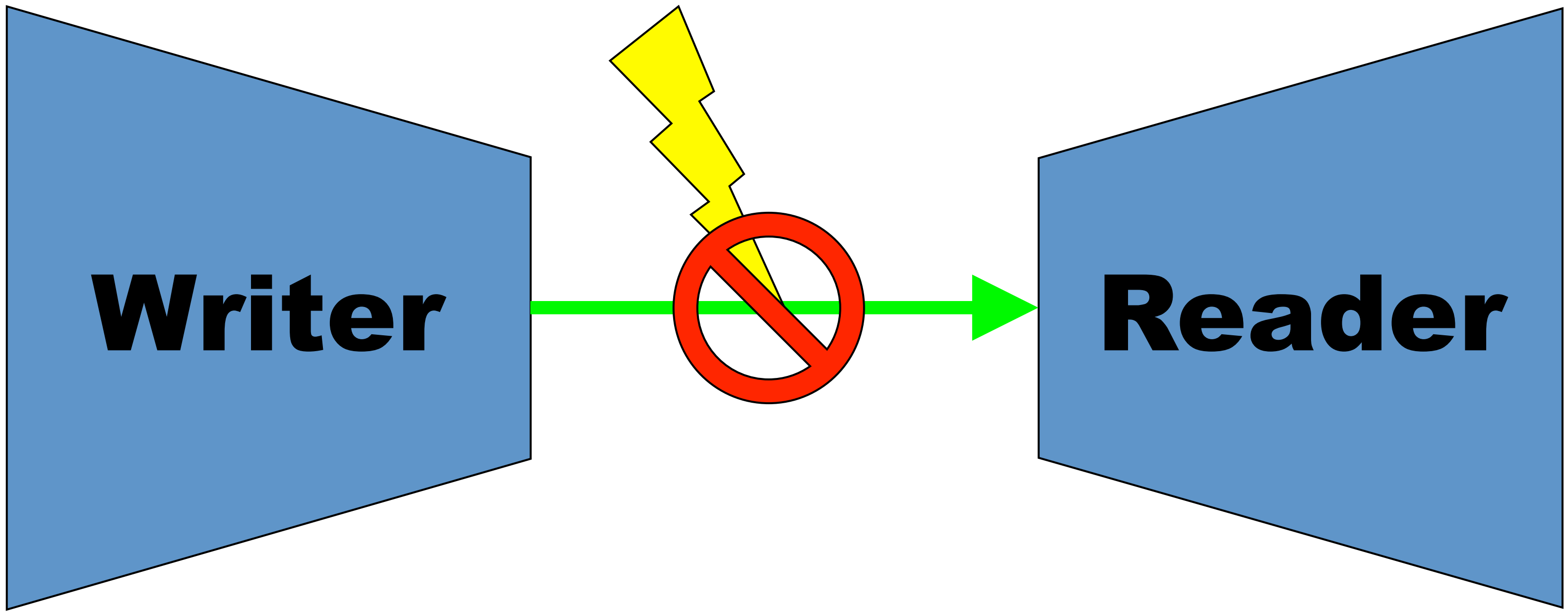
`Chanin` and `Chanout` have `parIn` and `parOut` calls respectively to get `ParCommItem` objects. They take the destination/source of the communication respectively as an argument

This `communicate` call performs all the communications associated with the `ParCommItems` passed to the constructor of the `ParallelComm`

Poisoning



Poisoning



Poisoning

```
void Writer::run() {
  int i;
  for (i = 0;i < 100;i++) {
    out << i;
  }
}
```

```
void Writer::run() {
  int i;
  for (i = 0;i < 100;i++) {
    out << i;
  }
  out.poison();
}
```

This channel will now be poisoned (forever - no antidote available!).

Any attempts to use the channel will cause a `PoisonException` to be thrown.

<u>Except:</u> poisoning an already-poisoned channel has no effect (no exception is thrown).

Poisoning

```
void Reader::run() {
  int i,n;
  for (i = 0;i < 100;i++) {
    in >> n;
  }
}
```

```
void Reader::run() {
  try {
    int n;
    while (true) {
      in >> n;
    }
  }
  catch (PoisonException e) {
  }
}
```

The `for` loop can now become a `while (true)` loop

Poisoning

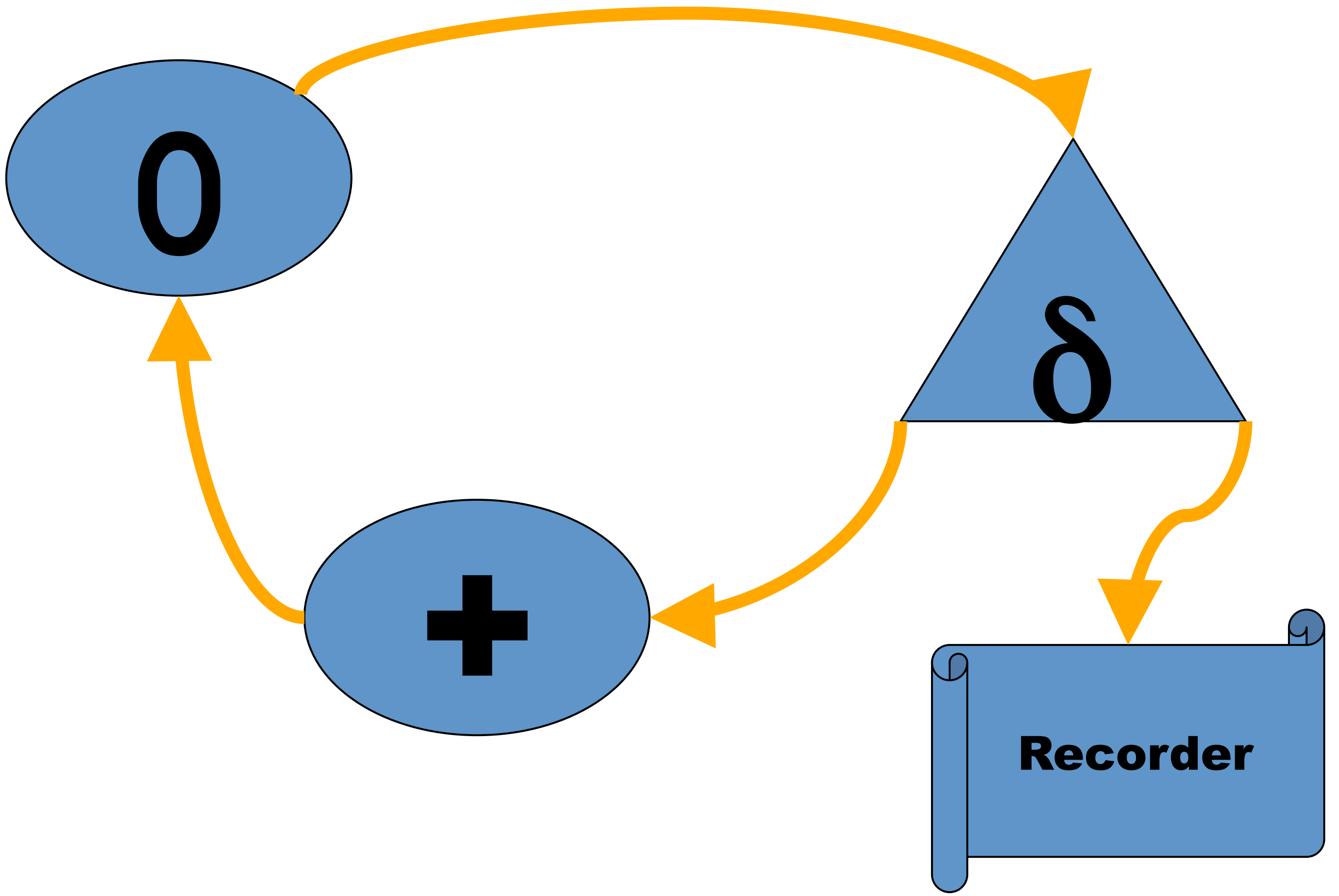
```
void Reader::run() {
  int i,n;
  for (i = 0;i < 100;i++) {
    in >> n;
  }
}
```

```
void Reader::run() {
  try {
    int n;
    while (true) {
      in >> n;
    }
  }
  catch (PoisonException e) {
  }
}
```

The `fo`
become

When the channel has been poisoned, the next input will cause a `PoisonException` to be thrown, and it will be caught here

CommsTime



CommsTime

<u>CSP Framework</u>	<u>Time per iteration</u>

CommsTime

CSP Framework	Time per iteration
occam (KRoC 1.3.3)	1.3 microseconds

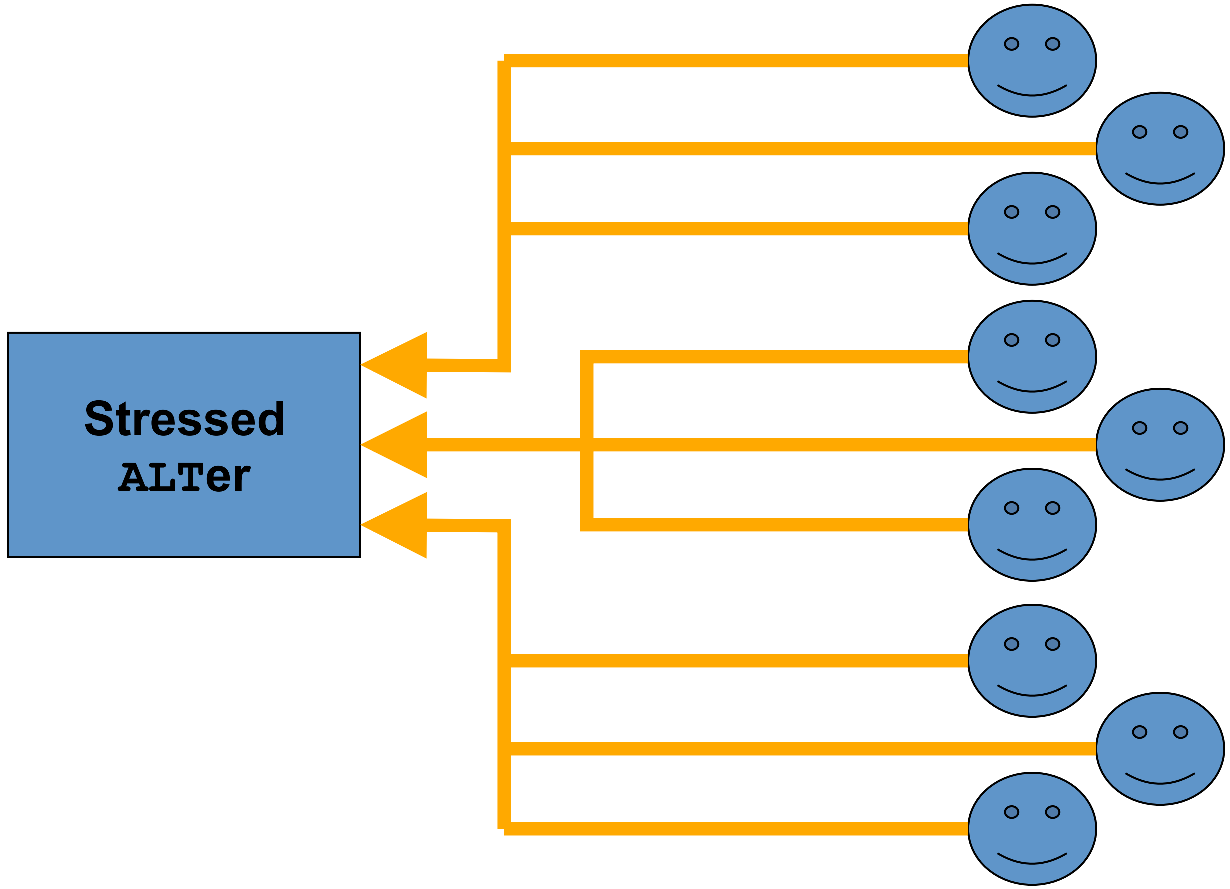
CommsTime

CSP Framework	Time per iteration
occam (KRoC 1.3.3)	1.3 microseconds
C++CSP	5 microseconds

CommsTime

CSP Framework	Time per iteration
occam (KRoC 1.3.3)	1.3 microseconds
C++CSP	5 microseconds
JCSP (JDK 1.4)	230 microseconds

Stressed ALT



Stressed `ALT`

CSP Framework	10 writers * 10 channels	100 writers * 20 channels	200 writers * 100 channels

Stressed `ALT`

CSP Framework	10 writers * 10 channels	100 writers * 20 channels	200 writers * 100 channels
occam (KRoC 1.3.3)	0.6	0.7	1

(All times are in microseconds)

Stressed `ALT`

CSP Framework	10 writers * 10 channels	100 writers * 20 channels	200 writers * 100 channels
occam (KRoC 1.3.3)	0.6	0.7	1
C++CSP	3	7	10

(All times are in microseconds)

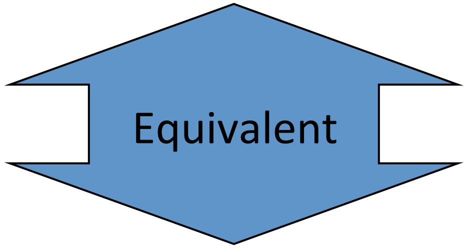
Stressed `ALT`

CSP Framework	10 writers * 10 channels	100 writers * 20 channels	200 writers * 100 channels
occam (KRoC 1.3.3)	0.6	0.7	1
C++CSP	3	7	10
JCSP (JDK 1.4)	130	200	-

(All times are in microseconds)

An Equivalence

```
One2OneChannel<int> channel;
Parallel(
  new Writer(channel.writer()),
  new Reader(channel.reader()),
NULL);
```




```
One2OneChannel<int> channel;
Barrier barrier(1);

spawnProcess(new Writer(channel.writer()),&barrier);
spawnProcess(new Reader(channel.reader()),&barrier);

barrier.sync();
```

Going Multi-Threaded

```
One2OneChannel<int> channel;
Barrier barrier(1);

spawnProcess(new Writer(channel.writer()),&barrier);
spawnProcess(new Reader(channel.reader()),&barrier);

barrier.sync();
```

```
InterThreadChannel<int> channel;
InterThreadBarrier barrier(1);

spawnAsNewThread(new Writer(channel.writer()),&barrier);
spawnAsNewThread(new Reader(channel.reader()),&barrier);

barrier.sync();
```

Python

Why PyCSP

- Internal research projects
  - Simple prototyping, especially in projects that already use Python
  - Want to use CSP from Python
- eScience
  - Python
    - Script and integration language
    - Prototyping
    - Easy to learn, readable code
    - Plenty of tools and libraries

Some goals

- Simple, short, and readable source code
  - Should be easy to walk students through the code
- Pure python code
  - Portable implementation that does not depend on compiling extra libraries
- Reasonable performance

Processes

Processes are declared using a @process declarator

```
@process
def hello_world (msg ):
    print " Hello world , this is my message " + msg

 Parallel (
     source (),
     [ worker () for i in range (10)] ,
     sink ()
 )
```

Processes

Processes are declared using a @process declarator

```
@process
def hello_world (msg ):
    print " Hello world , this is my message " + msg

 Parallel (
     source (),
     [ worker () for i in range (10)],          10*worker()
     sink ()
 )
```

Channels

- In programming and in engineering the use of different channels makes sense
  - In science they become a nuisance
- Any process that has a given channel in its context may ask for a channel-end from that channel
  - Input or output end

Channels

- Channels are easily defined
  - `my_channel = Channel ()`
- Channel ends are obtained by requesting an input or output end
  - `my_reader = my_channel.reader()`
    - `my_reader = +my_channel`
  - `my_writer = my_channel.writer()`
    - `my_writer = -my_channel`

Choice

- Choices are now selected and executed in one step
  - More like Occam less like select()
- The execution part is either a (small) direct statement or a function
  - Declared with @choice
- Both input and output guards are supported

Choice

- Input guards are
  - <channel> : <guard>
- Output guards are
  - (<channel>, <value>) : <guard>

```
@choice
def print_result():
    print __channel_input

Alternation([
            {in : print_result()},
            {(out , value) : “value += 1”}
]).execute()
```

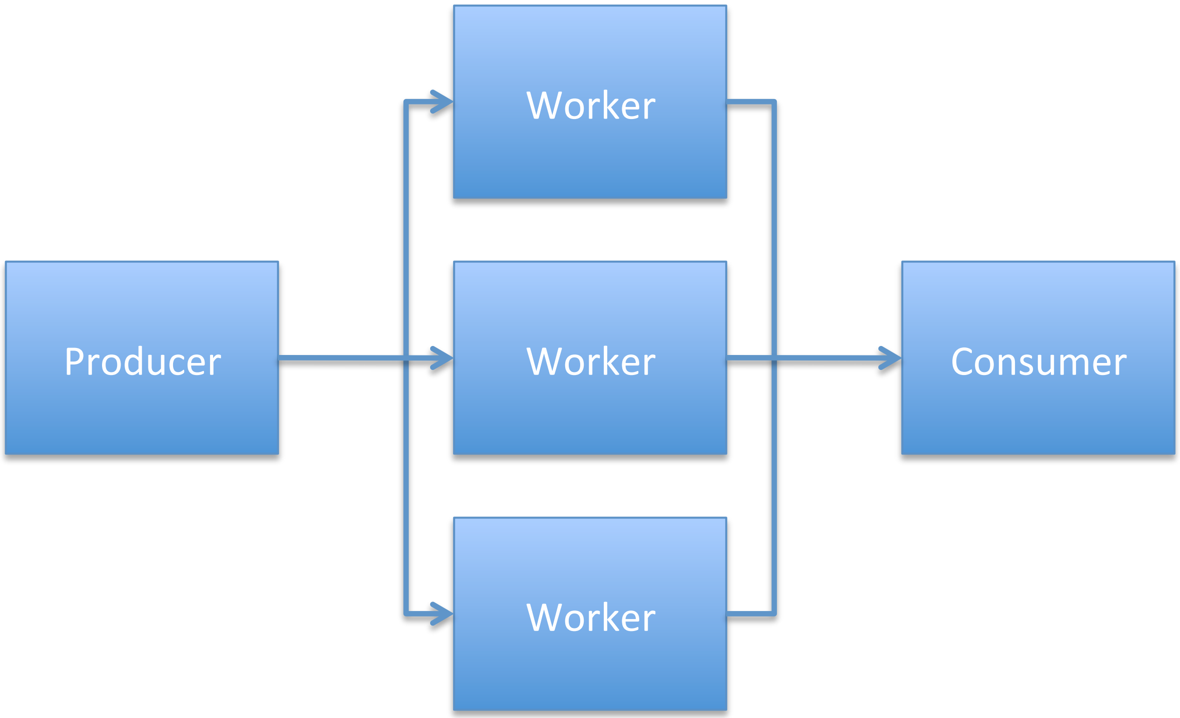
Prioritization

- Alternation support mixing prioritized and unprioritized guards
- An alternation is a set of lists
  - List order define priority
  - Within a list the elements are peer

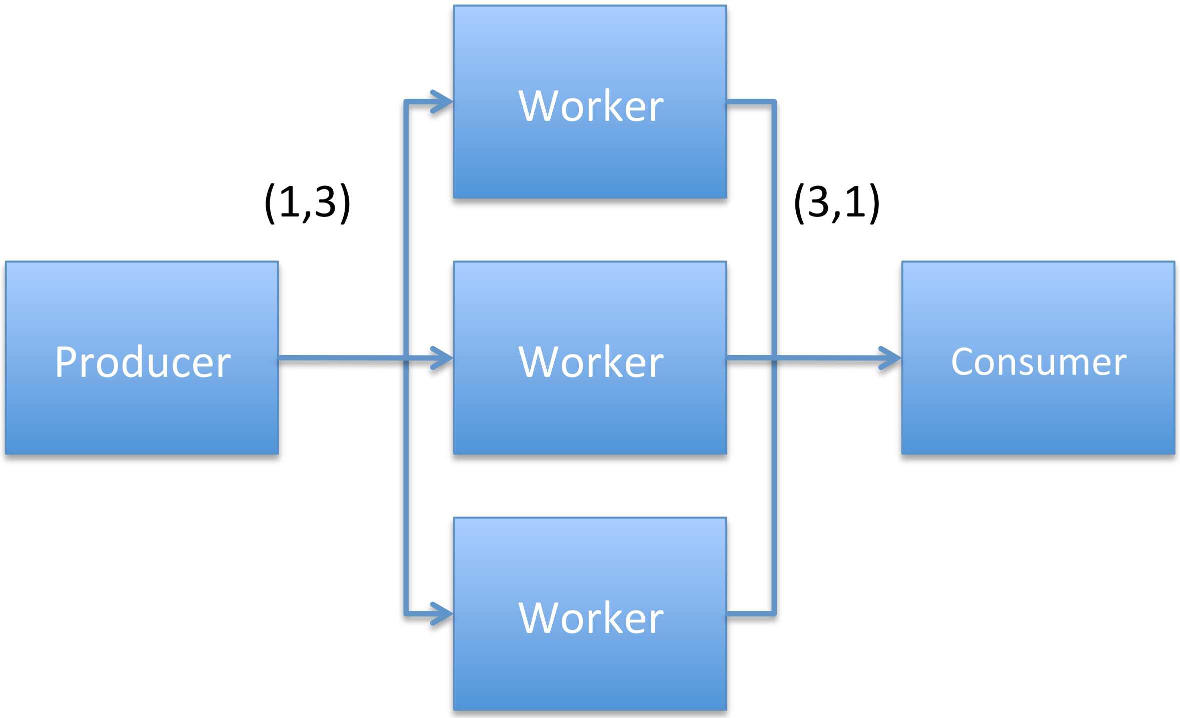
Controlled shutdown

- Rather than poisoning channels PyCSP also support reference counting
- When a channel end is created the count on that direction is increased
- A process can, where it would otherwise do a poison issue a retire
- When the reference count on a channel end reaches zero the whole channel enters a retired state

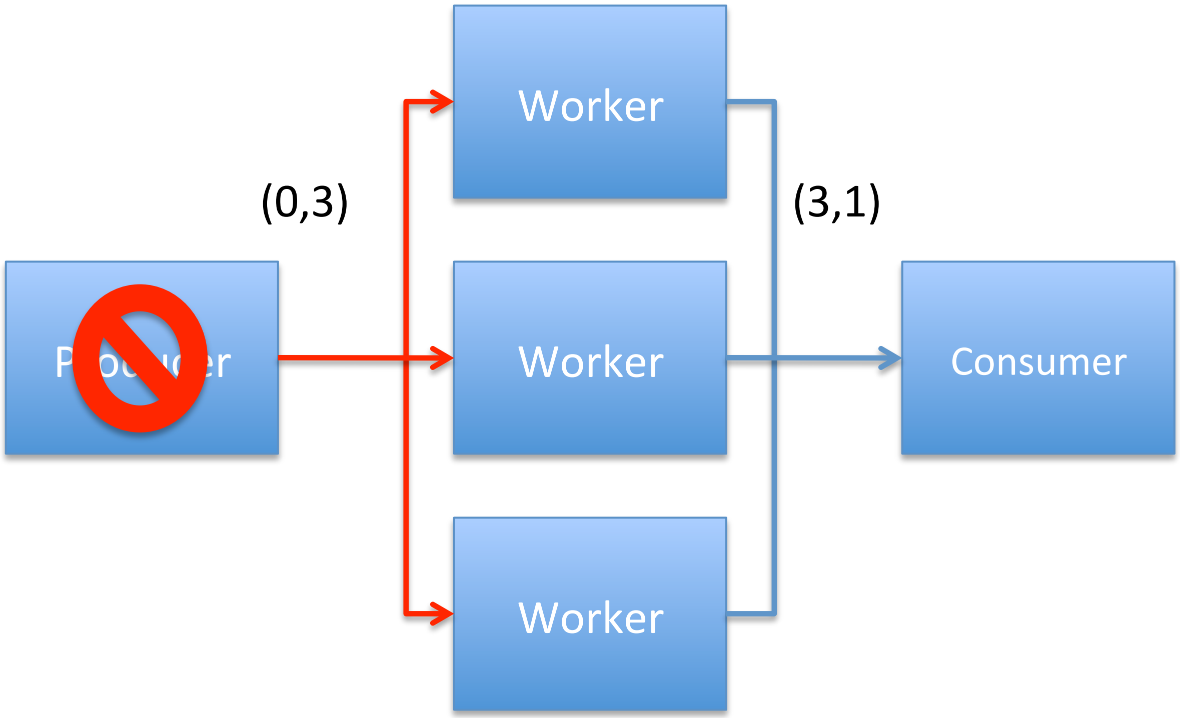
Controlled shutdown



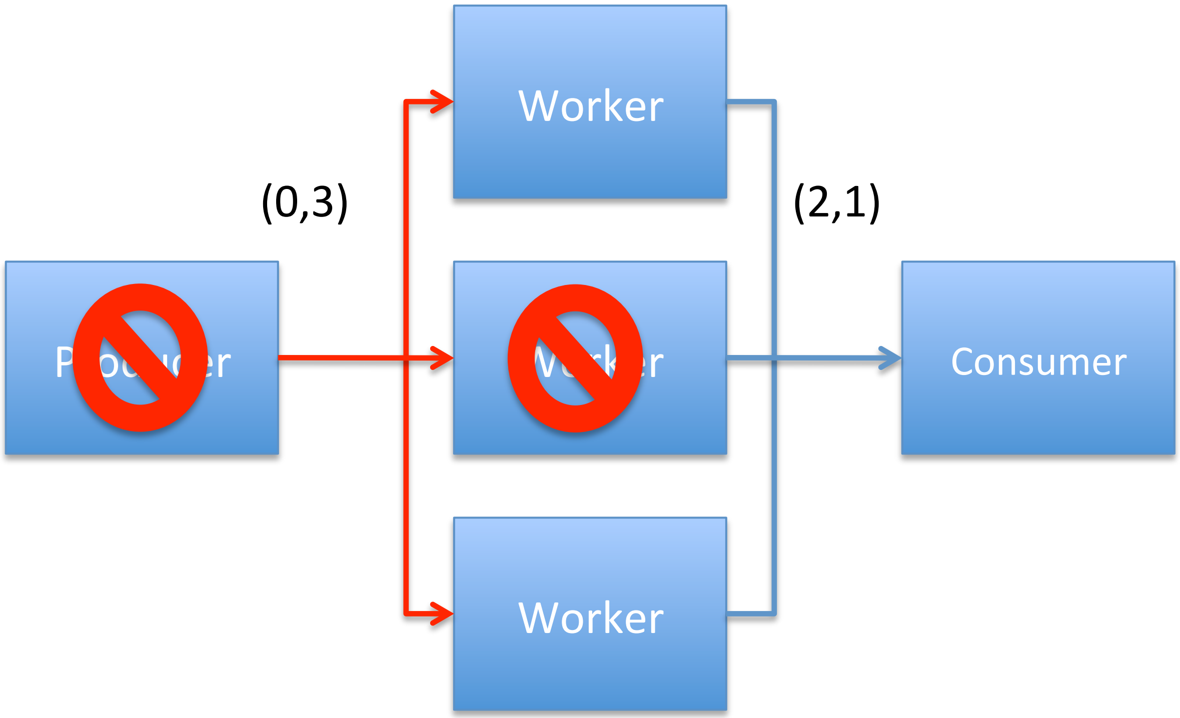
Controlled shutdown



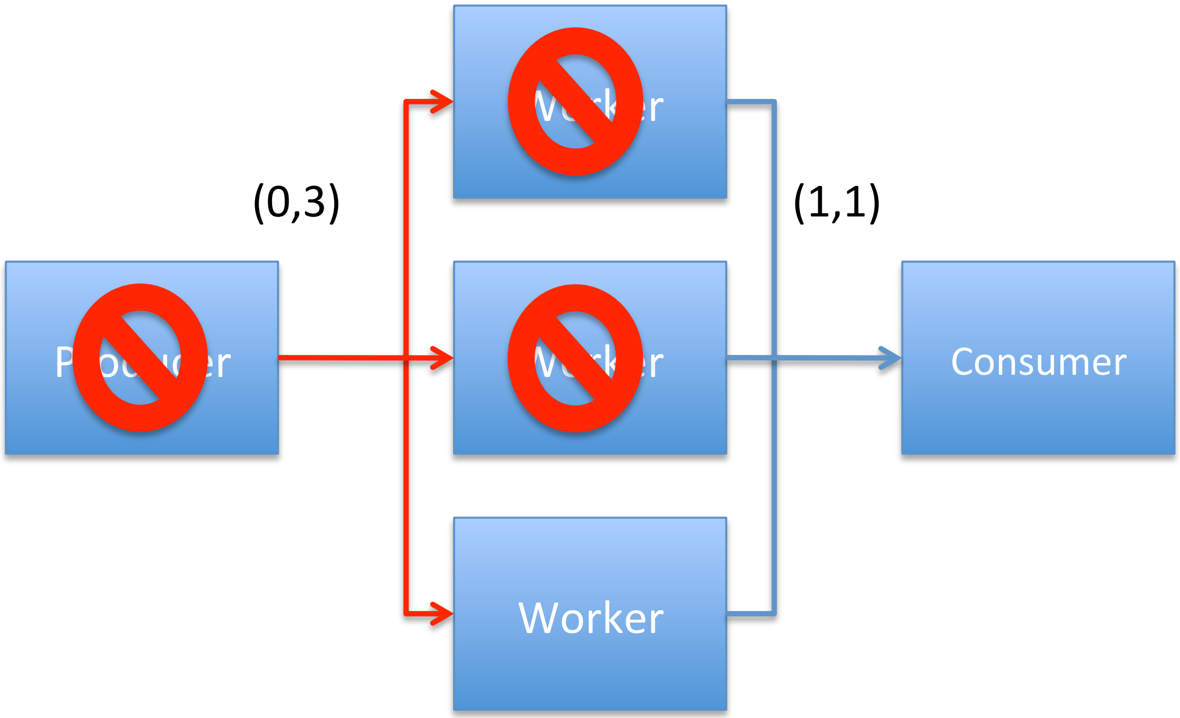
Controlled shutdown



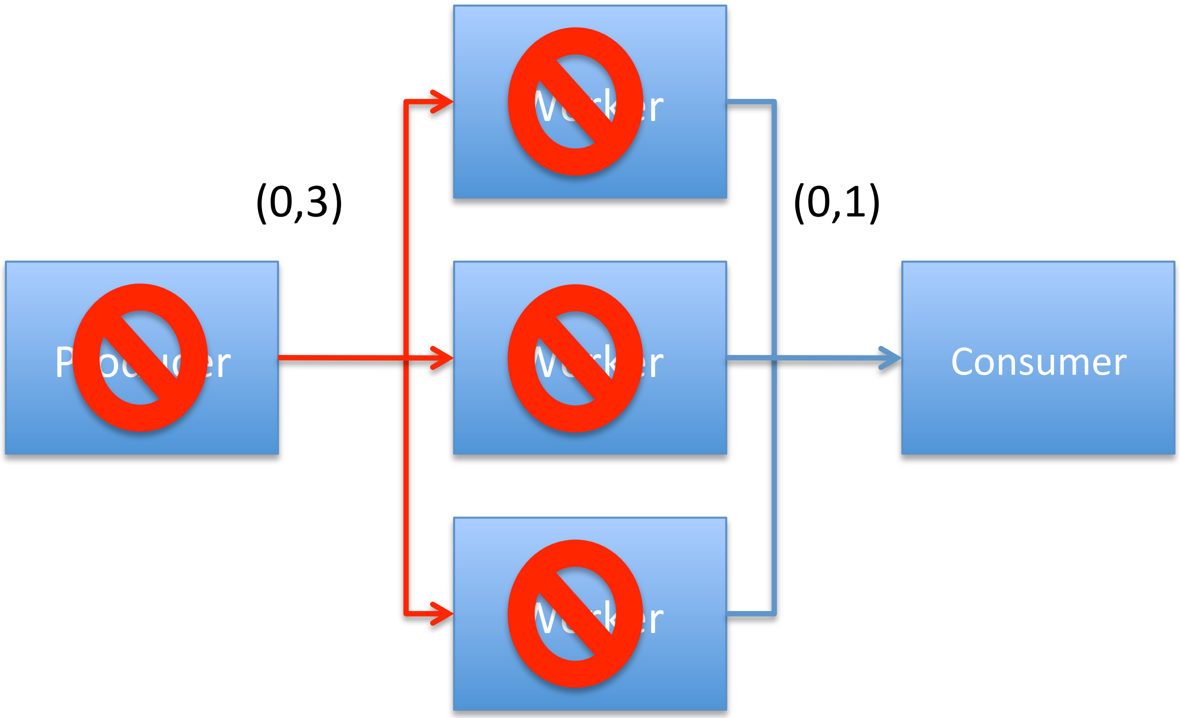
Controlled shutdown



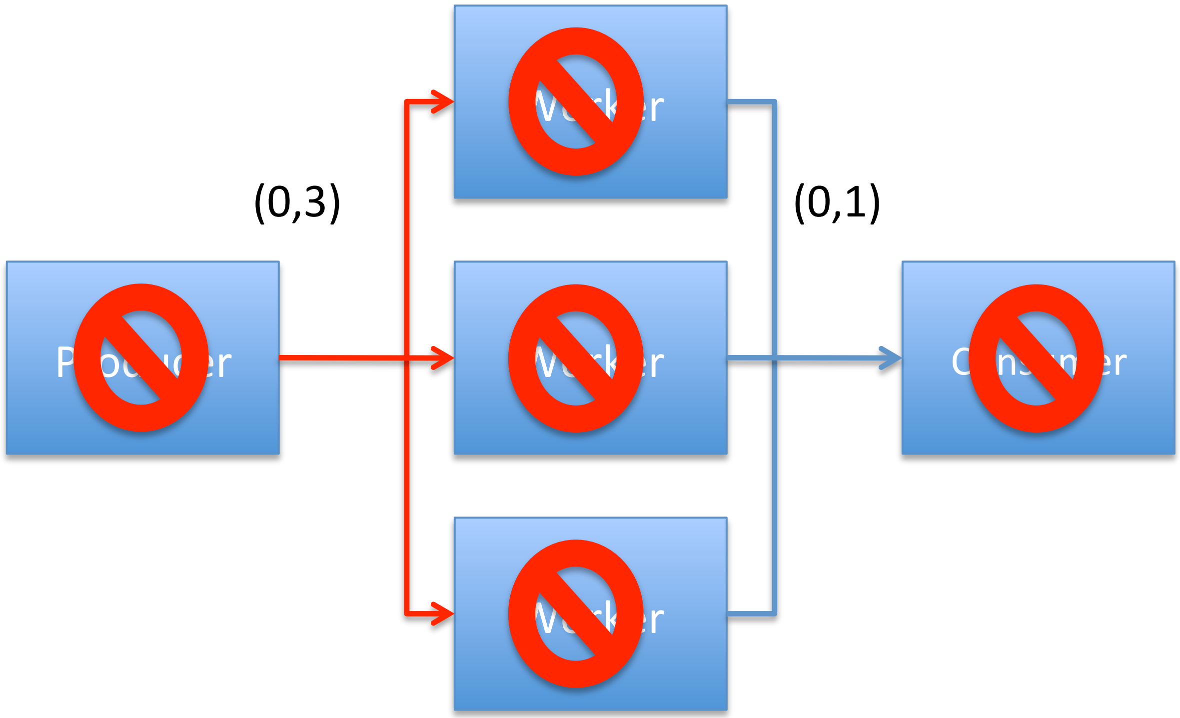
Controlled shutdown



Controlled shutdown



Controlled shutdown



LEGOs in PyCSP

```
@process
def id(inc, outc):
   while True:
      x = inc()
      outc(x)

@process
def succ(inc, outc):
   while True:
      x = inc()
      outc(x + 1)

@process
def prefix(n, inc, outc):
   outc(n)
   Sequence(id(inc, outc))
```

```
@process
def plus(inc0, inc1, outc):
   while True:
      x0, x1 = ParallelRead((inc0, inc1))
      outc(x0+x1)

@process
def delta(cin, cout1, cout2):
   while True:
      msg = cin()
      Alternation([{
         (cout1,msg):'cout2(msg)',
         (cout2,msg):'cout1(msg)'
         }]).execute()
```

Two extra bricks

```
@process
def shoot_and_die(outc):
    outc(42)
    poison(outc)

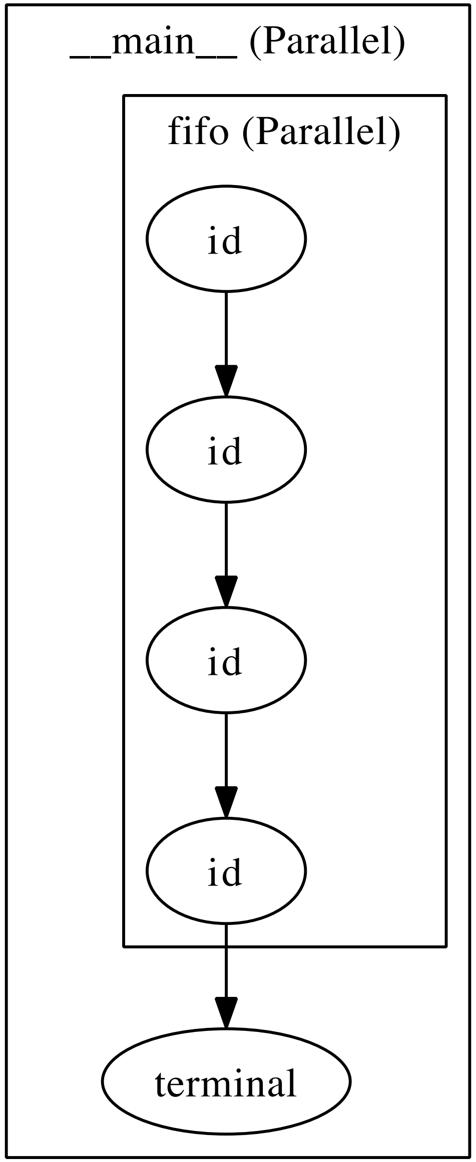
@process
def terminal(inc, terminal_signal=None):
    input = True
    while input != terminal_signal:
        input = inc()
        print input
    poison(inc)
```

Numbers

```
@process
def numbers(outc):
   a = Channel()
   b = Channel()
   c = Channel()

   Parallel(prefix(0, c.reader(), a.writer()),
           delta(a.reader(), outc, b.writer()),
           succ(b.reader(), c.writer()))
```

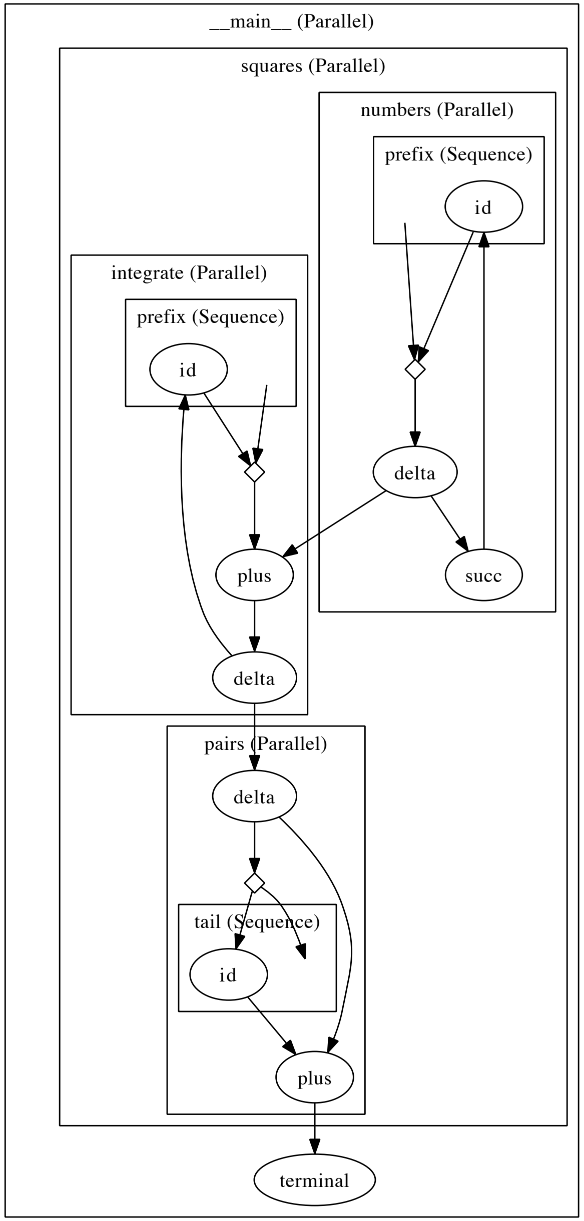
FIFO




```
@process
def fifo(n, inc, outc):
    channels = [Channel() * (n-1)]
    inputs = [c.reader() for c in channels]
    inputs.insert(0, inc)

    outputs = [c.writer() for c in channels]
    outputs.append(outc)

    Parallel([id(inputs[i], outputs[i]) for i in xrange(n)])
```



Squares

```
@process
def squares(outc):
    a = Channel()
    b = Channel()

    Parallel(numbers(a.writer()),
            integrate(a.reader(), b.writer()),
            pairs(b.reader(), outc))
```

Go

Largely borrowed fro Rob Pike

Hello World

```
package main
import "fmt"
func main() {
  fmt.Printf("Hello, World\n");
}
```

Compiles


THE #1 PROGRAMMER EXCUSE FOR LEGITIMATELY SLACKING OFF:
"MY CODE'S COMPILING."
HEY! GET BACK TO WORK!
COMPILING!
OH. CARRY ON.

The big picture

- Fundamentals:
  - Clean, concise syntax.
  - Lightweight type system.
  - No implicit conversions: keep things explicit.
  - Untyped unsized constants: no more **0x80ULL.**
  - Strict separation of interface and implementation.
- Run-time:
  - Garbage collection.
  - Strings, maps, communication channels.
  - Concurrency.
- Package model:
  - Explicit dependencies to enable faster builds.

Processes

- No parallel construct but asynchronous calls
  - GoStmt = "go" Expression
  - go Server()

Channels

- Channels are created using types as in Occam
  - c := make(chan int, 10)
- Channels are uniformly accesed
  - <-

Example

```
func generate(ch chan<- int) {
    for i := 2; ; i++ {
        ch <- i; // Send 'i' to channel 'ch'.
    }
}

func consume(ch chan<- int) {
    for i := 2; ; i++ {
        i <- ch; // Read 'i' from channel 'ch'.
    }
}
```

```go
// Copyright 2009 The Go Authors. All rights reserved.
// Use of this source code is governed by a BSD-style
// license that can be found in the LICENSE file.

package main

import "fmt"

// Send the sequence 2, 3, 4, ... to channel 'ch'.
func generate(ch chan int) {
   for i := 2; ; i++ {
      ch <- i // Send 'i' to channel 'ch'.
   }
}

// Copy the values from channel 'in' to channel 'out',
// removing those divisible by 'prime'.
func filter(in, out chan int, prime int) {
   for {
      i := <-in // Receive value of new variable 'i' from 'in'.
      if i%prime != 0 {
         out <- i // Send 'i' to channel 'out'.
      }
   }
}

// The prime sieve: Daisy-chain filter processes together
func main() {
   ch := make(chan int) // Create a new channel.
   go generate(ch)      // Start generate() as a goroutine.
   for {
      prime := <-ch
      fmt.Println(prime)
      ch1 := make(chan int)
      go filter(ch, ch1, prime)
      ch = ch1
   }
}
```