UiT

THE ARCTIC
UNIVERSITY
OF NORWAY

# Communicating Sequential Processes (CSP)

John Markus Bjørndalen

# Multithreading?

- http://java.sun.com/products/jfc/tsc/articles/threads/threads1.html
  ” If you can get away with it, avoid using threads. Threads can be difficult to use, and they make programs harder to debug. In general, they just aren't necessary for strictly GUI work, such as updating component properties.”
- Other interesting quotes:
  http://www.cs.kent.ac.uk/projects/ofa/co538/anonqa/a0-2005.html

# Message passing

- Deadlocks?
- Buffering?
- Coordination of sending and receiving?
- Simple?

# How about object orientation?

```
class Foo:
    def __init__(self, next_fun):
        self.val = 42
        self.next_fun = next_fun

    def fun(self, v):
        print "Self.val =", self.val, \
          "adding", v
        self.val += v
        print "  -  val now", self.val
        self.next_fun()
        print "  -  val still", self.val
        print "  - done"
```

# How about object orientation?

```
class Foo:
    def __init__(self, next_fun):
        self.val = 42
        self.next_fun = next_fun

    def fun(self, v):
        print "Self.val =", self.val, \
          "adding", v
        self.val += v
        print "  -  val now", self.val
        self.next_fun()
        print "  -  val still", self.val
        print "  - done"
```

```
Execution 1:
Self.val = 42 adding 1
  -  val now 43
     This is a function
  -  val still 43
  - done
Self.val = 43 adding 1
  -  val now 44
     This is a function
  -  val still 44
  - done
```

# How about object orientation?

```
class Foo:
    def __init__(self, next_fun):
        self.val = 42
        self.next_fun = next_fun

    def fun(self, v):
        print "Self.val =", self.val, \
          "adding", v
        self.val += v
        print "  -  val now", self.val
        self.next_fun()
        print "  -  val still", self.val
        print "  - done"
```

```
Execution 1:
Self.val = 42 adding 1
  -  val now 43
     This is a function
  -  val still 43
  - done
Self.val = 43 adding 1
  -  val now 44
     This is a function
  -  val still 44
  - done
----------------------------------------
Execution 2:
Self.val = 42 adding 1
  -  val now 43
     This is another function
  -  val still 85
  - done
Self.val = 85 adding 1
  -  val now 86
     This is another function
  -  val still 128
  - done
```

How is this possible?

# How about object orientation?

```
class Foo:
    def __init__(self, next_fun):
        self.val = 42
        self.next_fun = next_fun

    def fun(self, v):
        print "Self.val =", self.val, \
          "adding", v
        self.val += v
        print "  -  val now", self.val
        self.next_fun()
        print "  -  val still", self.val
        print "  - done"
```

```
def fun_a():
    print "     This is a function"

def fun_b():
    print "     This is another function"
    f.val += 42


f = Foo(fun_a)
for i in range(2):
    f.fun(1)

print "-------------------------------------"

f = Foo(fun_b)
for i in range(2):
    f.fun(1)
```

# CSP

- “Communicating Sequential Processes (CSP) is a precise mathematical theory of concurrency that can be used to build multithreaded applications that are guaranteed to be free of the common problems of concurrency and (perhaps more importantly) *can be proven* to be so.“

  Abhijit Belapurkar, IBM Developerworks, 21 Jun 2005

  www.ibm.com/developerworks/java/library/j-csp1.html

# CSP

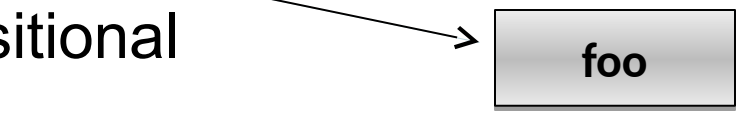
- Tony Hoare, 1978, CSP paper
- Observations
  - The action of assignment is familiar and well understood
    - Any change of internal state of a machine executing a program can be modeled as an assignment
  - Operations of input and output, affecting external environment, are not nearly so well understood
- ”This paper suggests that input and output are basic primitives of programming and that parallel composition of communicating sequential processes is a fundamental program structuring method. When combined with a development of Dijkstra’s guarded command, these concepts are suprisingly versatile.”

# CSP core functionality

- Processes
  - Compositional
- Channels
- Parallel
- Alternative (Alt)
- Guards

# CSP core functionality

- **Processes**
  - Compositional
- Channels
- Parallel
- Alternative (Alt)
- Guards



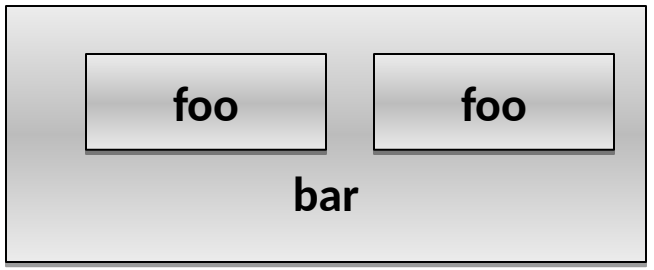

PyCSP:

```
@process
def foo():
    pass
```

- Processes don't share state (no global variables)
- Each process is a small sequential program
- Typical granularity: expression or function

# CSP core functionality

- Processes
  - **Compositional**
- Channels
- Parallel
- Alternative (Alt)
- Guards




PyCSP:

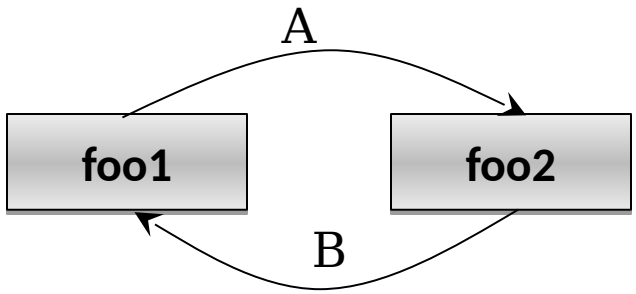
```
@process
def foo():
    pass

@process
def bar():
    Parallel(foo(),
             foo())
```

Each process in a parallel construct runs in parallel with the others.

Also: Sequence()

# CSP core functionality

- Processes
  - **Compositional**
- Channels
- Parallel
- Alternative (Alt)
- Guards




PyCSP:

```
@process
def foo1(cin, cout):
    cout(42)
    while 1:
        v = cin()
        cout(v+1)

@process
def foo2(cin, cout):
    while 1:
        v = cin()
        print "Got", v
        cout(v+1)

A = One2OneChannel()
B = One2OneChannel()
Parallel(foo1(A.read, B.write),
         foo2(B.read, A.write))
```

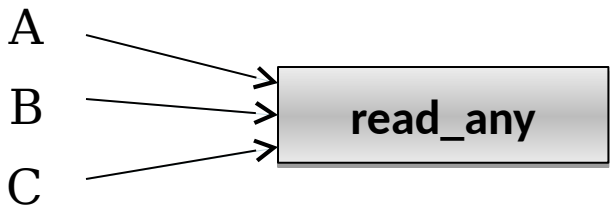
Synchronous reads and writes on Channels

CSP:
A?v (read from A)
B!v (write to B)

Similar to assign across processes.

# CSP core functionality

- Processes
  - **Compositional**
- Channels
- Parallel
- Alternative (Alt)
- Guards



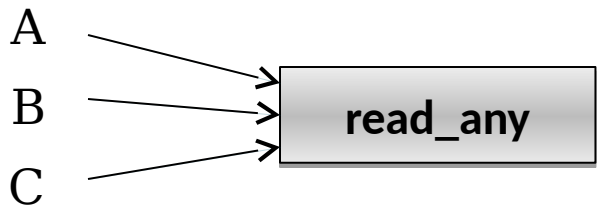

PyCSP:

```
@process
def read_any(A,B,C):
    alt = Alternative(A,B,C)
    while 1:
        ret = alt.select()
        print "Got value", ret()


A = One2OneChannel()
B = One2OneChannel()
C = One2OneChannel()

Parallel(read_any(A.read, B.read, C.read),
         …)
```

NB: in this case, input channels are used as *input guards*.

# CSP core functionality

- Processes
  - **Compositional**
- Channels
- Parallel
- Alternative (Alt)
- Guards



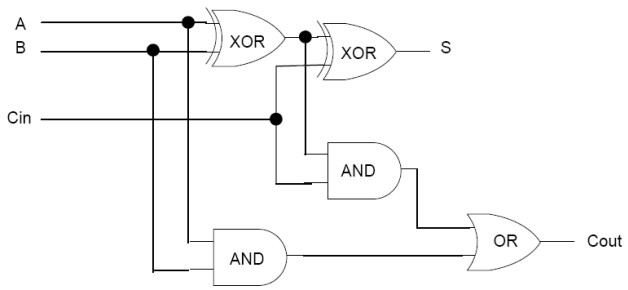

PyCSP:

```
@process
def read_any(A,B,C):
    timeout = Guards.Timer(10)
    alt = Alternative(A,B,C, timeout)
    while 1:
        ret = alt.select()
        if ret == timeout:
            print "Timed out"
        else:
            print "Got value", ret()
```

Adding a timeout guard

```
A = One2OneChannel()
B = One2OneChannel()
C = One2OneChannel()

Parallel(read_any(A.read, B.read, C.read),
         …)
```

# Example: Circuit Design

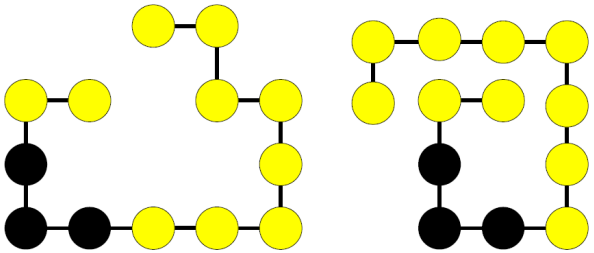


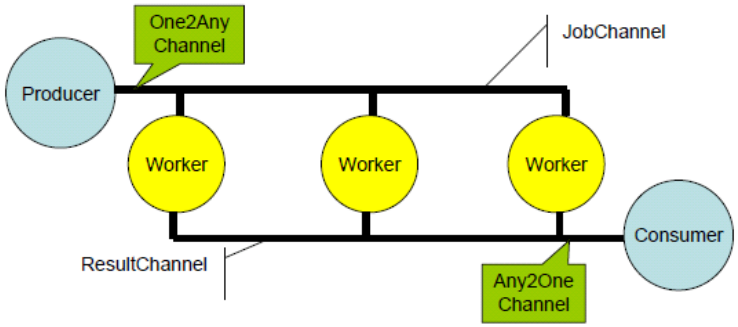

```
1  def Adder(A, B, Cin, S, Cout):
2      Aa = One2OneChannel()
3      Ab = One2OneChannel()
4      Ba = One2OneChannel()
5      Bb = One2OneChannel()
6      Ca = One2OneChannel()
7      Cb = One2OneChannel()
8      i1 = One2OneChannel()
9      i1a = One2OneChannel()
10     i1b = One2OneChannel()
11     i2 = One2OneChannel()
12     i3 = One2OneChannel()
13
14     Parallel(Process(delta, A.read, Aa.write, Ab.write),
15              Process(delta, B.read, Ba.write, Bb.write),
16              Process(delta, Cin.read, Ca.write, Cb.write),
17              Process(delta, i1.read, i1a.write, i1b.write),
18              Process(XOR, Aa.read, Ba.read, i1.write),
19              Process(XOR, i1a.read, Ca.read, S.write),
20              Process(AND, Ab.read, Bb.read, i2.write),
21              Process(AND, i1b.read, Cb.read, i3.write),
22              Process(OR, i2.read, i3.read, Cout.write))
```

NB: old PyCSP syntax

```
@process
def AND(cin1, cin2, cout):
    x1=x2=0
    alt = Alternative([cin1, cin2])
    while True:
        cout(x1 and x2)
        ret = alt.select()
        if ret == cin1:
            x1 = ret()
        else:
            x2 = ret()
```

# Example: Protein Folding






```
1  feeder = One2AnyChannel()
2  collector = Any2OneChannel()
3  done = Any2OneChannel()
4
5  Parallel( Process(producer, protein, map, place, feeder.write),
6            Process(worker, feeder.read, collector.write, done.write),
7            Process(worker, feeder.read, collector.write, done.write),
8            Process(worker, feeder.read, collector.write, done.write),
9            Process(worker, feeder.read, collector.write, done.write),
10           Process(worker, feeder.read, collector.write, done.write),
11           Process(barrier, 5, done.read, collector.write),
12           Process(consumer, collector.read) )
```

# Some Implementations

- Occam (and Occam-pi)
  - Originally used with the Transputer
  - Now with runtime environments and compilers for modern processors, embedded systems, mobile phones and robots
- JCSP
  - CSP through class library
- C++CSP
- CSP for .NET
- Common Lisp
- PyCSP

# PyCSP

- Still in development
  - Tromsø, København
- Python implementation
  - Currently of CSP.Core
    - Processes implemented using Python threads
    - Channels
    - Par
    - Alt
    - Guards
- https://code.google.com/p/pycsp/

# Other CSP-based languages?

- Actors and actor-based systems and languages (ex: Erlang, Scala(?))
- Go (a CSP-based language): http://golang.org/

# CSP in Go

- http://golang.org/doc/effective_go.html
  - Check the concurrency section
- http://golangtutorials.blogspot.no/2011/06/channels-in-go-range-and-select.html

# CSP in Go

- CSP process => goroutine

```
// Example 1:
go list.Sort()  // run list.Sort concurrently; don't wait for it.

// Example 2:
func Announce(message string, delay time.Duration) {
    go func() {
        time.Sleep(delay)
        fmt.Println(message)
    }()  // Note the parentheses - must call the function.
}
```

# CSP in Go

- CSP channels => chan

```
c := make(chan int)  // Allocate a channel.
// Start the sort in a goroutine; when it completes,
// signal on the channel.
go func() {
    list.Sort()
    c <- 1  // Send a signal; value does not matter.
}()
doSomethingForAWhile()
<-c   // Wait for sort to finish; discard sent value.
```

# CSP in Go

http://golangtutorials.blogspot.no/2011/06/channels-in-go-range-and-select.html

Channels as guards + a default (always true) guard.

```
func receiveCakeAndPack(strbry_cs chan string, choco_cs chan string) {
    …
   for {
        //if both channels are closed then we can stop
        if (strbry_closed && choco_closed) { return }
        fmt.Println("Waiting for a new cake ...")
        select {
        case cakeName, strbry_ok := <-strbry_cs:
            … do something …
        case cakeName, choco_ok := <-choco_cs:
            … do something …
        }
    }
}
The _ok values indicate whether the channel is closed (false).
```

# CSP Resources

- Tony Hoares CSP book (available as a pdf)
  http://www.usingcsp.com/
- Some CSP links and resources
  http://vl.fmnet.info/csp/
- WoTUG community
  http://www.wotug.org/
- JCSP homepage
  http://www.cs.kent.ac.uk/projects/ofa/jcsp/
- JCSP article at IBM developerworks
  http://www-128.ibm.com/developerworks/java/library/j-csp1.html
- Wikipedia
  http://en.wikipedia.org/wiki/Communicating_sequential_processes