The History of Concurrency

by Michael Schurter 2011-6-23

What is concurrency?

- Two tasks are concurrent if they may be executed in indeterminate order.
  - HTTP GET Requests
  - Writing data to disk & over the network
  - Making a sandwich and tea

What is parallelism?

- Parallelism is the act of performing multiple tasks simultaneously
  - Serving 2 HTTP requests
  - Writing to disk and the network
  - Making a sandwich with one hand and tea with the other

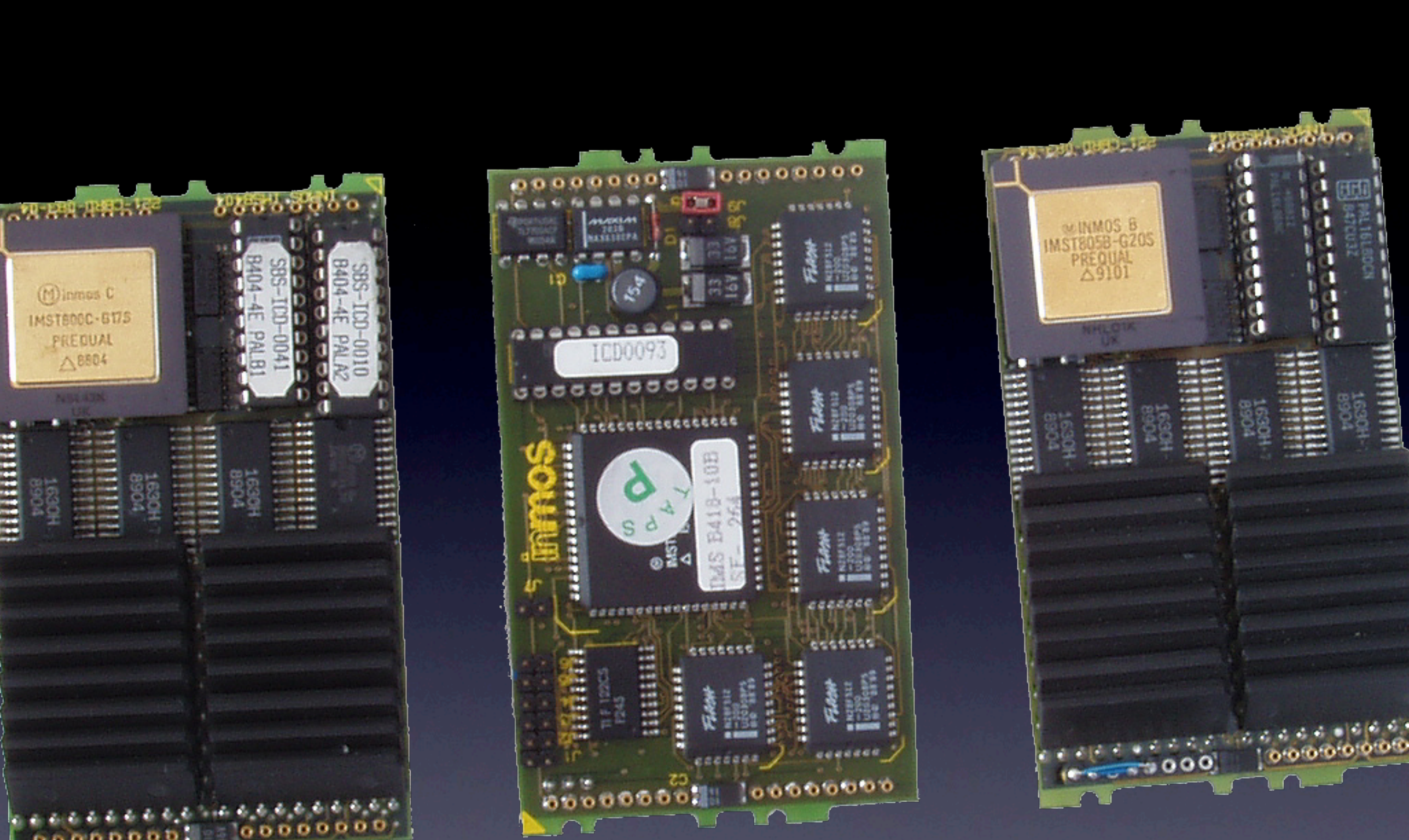
Concurrency vs Parallelism

- Concurrency is a property of a program
- Parallelism is a runtime state
- Not everything is concurrent is parallel
- Anything that is not concurrent but run in parallel is a bug

What isn’t this talk?

This is not a hardware talk

except...



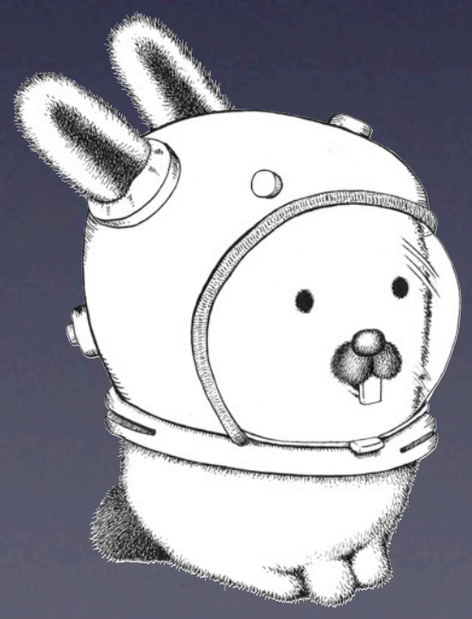
The Transputer

This is not a computer science lecture.

General purpose languages only.
I'm no Dijkstra.

This is not a talk about distributed computing

Although it will probably come up.



In the beginning

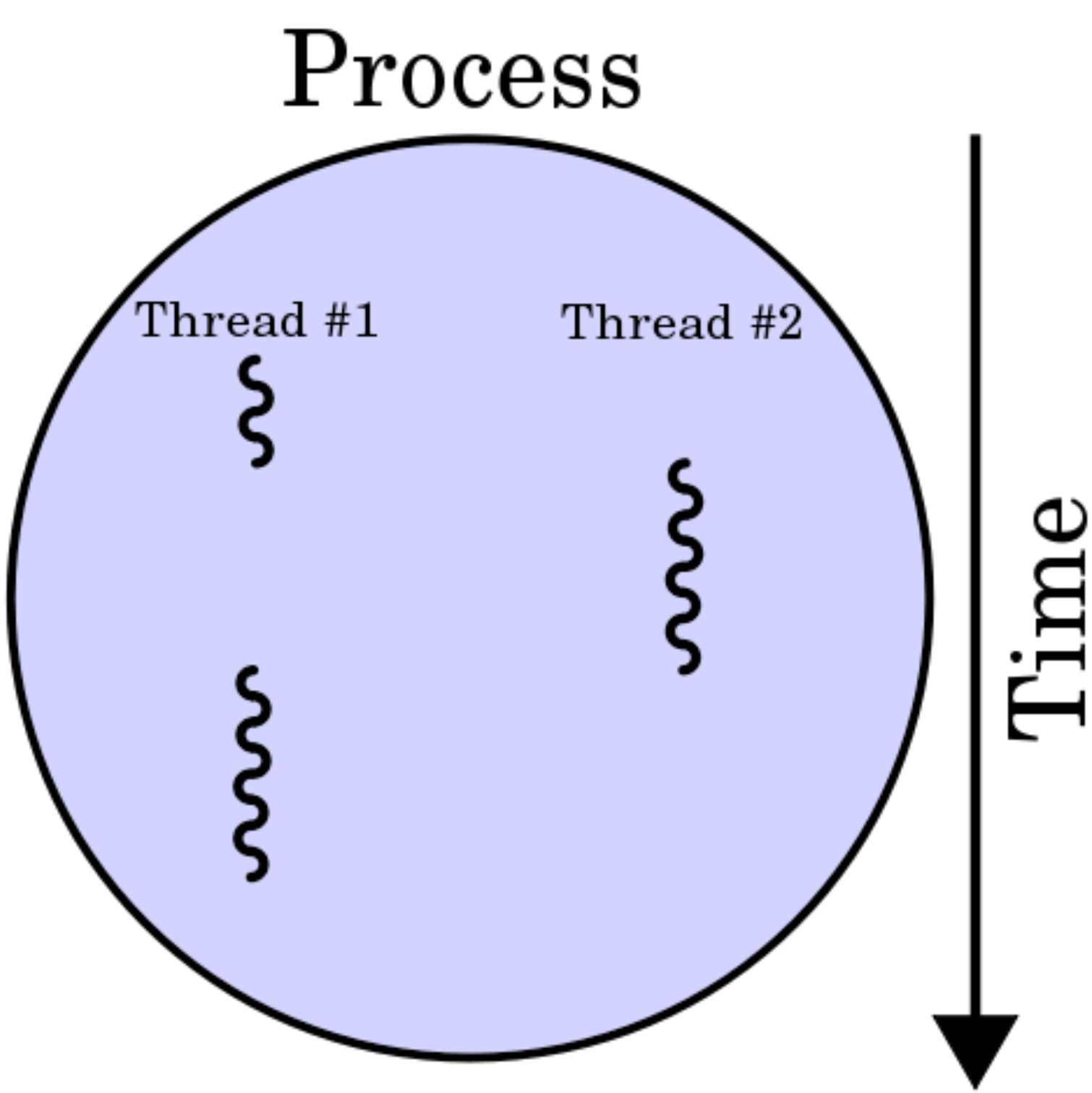
Multiprogramming via time-slicing appears in late 50s

Everything happened in the 60s & 70s

- 1960 - Idea of continuations in Algol 60
- 1963 - Coroutines (implemented in Simula 67)
- 1964 - Time-slicing (preemptive multitasking; in DTSS)
- 1965 - Semaphores (implemented in Algol 68)
- 1968 - Algol 68 builds in parallelism (sema & par)
- 1973 - Unix Pipes (coprocesses)
- 1973 - Actor model
- 1974 - Monitors (in Concurrent Pascal)
- 1975 - Continuation-passing style (for Scheme)
- 1978 - Concurrent Sequential Processes (Transputer)

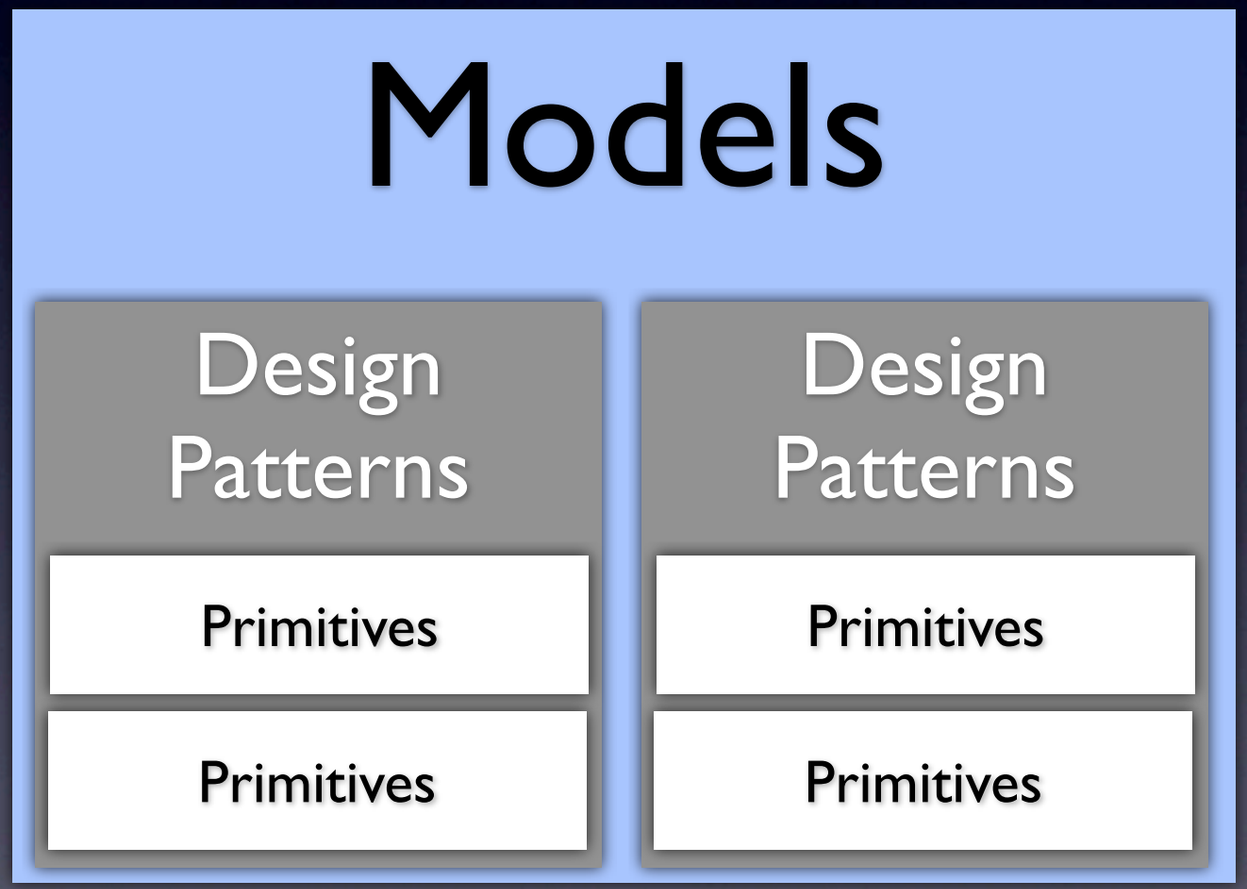
Threads are...

- Lightweight processes (concrete)
- Smallest unit of proccessing (abstract)



Threads: How do they work?

Models & Primitives



Two Models of Multitasking

Preemptive	**Cooperative**
Threads	Coroutines Continuations
Processes	Fibers Greenthreads

Cooperative Model

- Continuations
  - The ability to save and resume execution state
  - Can be implemented with closures or classes less easily/fully
- Coroutines (1963)
  - Subroutines that allow yielding and resuming control uses continuations
  - Generators are a type of coroutine
    - Tricky to use as pure coroutines
- Issues:
  - ***One uncooperative thread causes a deadlock***
  - Language support varies; tough to fully hack in later

Coroutines in Scheme

```
(let* ((yin
        ((lambda (cc) (display "@") cc)
         (call-with-current-continuation
          (lambda (c) c))))
       (yang
        ((lambda (cc) (display "*") cc)
         (call-with-current-continuation
          (lambda (c) c))) )
  (yin yang))
```

Generators in Python

```
def yin():
    while 1:
yield"yin"

def yang():
    while 1:
yield"yang"

yin_generator = yin()
yang_generator = yang()

for _ in range(10):
print yin_generator.next()
print yang_generator.next()
```

Generators in Python 2

```
from itertools import izip

def g(word):
for _ in range(10):
yield word

for yin, yang in izip(g("yin"), g("yang")):
    print yin
print yang
```

Preemptive Model

- Threads & processes
- Core feature of Unix from the beginning
- Windows lacked preemptive multitasking until 95 & NT
- Mac OS didn't gain it until OSX
- Core feature in AmigaOS circa 1985
- Native threads took a while in Unix:
  - Solaris didn't have native threads until 1993
  - Linux didn't until 2004 (2.6)

Processes in Python

```
import os

print"Hi! I'm %d" % os.getpid()

pid = os.fork()

if pid:
    print "I'm a parent!"
os.wait()
else:
print"And I'm %d, the child!" % os.getpid()
```

Threads in Python

```
import threading

classWorkerThread(threading.Thread):
def __init__(self, word):
        threading.Thread.__init__(self)
        self.word = word

def run(self):
for _ in range(10):
print self.word

yin = WorkerThread("yin")
yang = WorkerThread("yang")

yin.start()
yang.start()
yin.join()
yang.join()
```

Memory, Threads, & Processes

- Threads implicitly **share** a memory space
- Processes must use OS to either share memory regions or communicate (IPC)
  - eg mmap, shared memory, Unix pipes
  - Bash 4.0 includes a *coproc* for interacting with processes cooperatively
  - &| in ksh (from 1988)

Two Models of Interaction

Shared Memory	**Message Passing**
Synchronization is key	Communication is key
Coordination: Semaphores Mutexes Monitors	"Do not communicate by sharing memory; instead, share memory by communicating." (Go)

Cooperative Models

Speaking of message passing...

Actor Model

- **Actors** are the concurrency primitive
- **No shared state** between Actors
- Messages are delivered *asynchronously* to mailboxes
- Often built upon underlying threads, locks, and buffers (channels, queues, etc)
- Inspired by Simula and Smalltalk

Erlang (1986)

Actor model in core language & runtime design

Erlang's Philosophy

- "The world is concurrent
  Things in the world don't share data
  Things communicate with messages
  Things fail

  Model this in a language" - Joe Armstrong

Erlang Example

```erlang
% Create a process and invoke function
ServerProcess = spawn(web, start_server, [Port, MaxConns]),

% Create a remote process and invoke function
RemoteProcess = spawn(RemoteNode, web, start_server, [Port, MaxConns]),

% Send a message to ServerProcess (asynchronously)
ServerProcess ! {pause, 10},

% Receive messages sent to this process
receive
        a_message -> do_something;
        {data, DataContent} -> handle(DataContent);
        {hello, Text} -> io:format("Got hello message: ~s", [Text]);
        {goodbye, Text} -> io:format("Got goodbye message: ~s", [Text])
end.
```

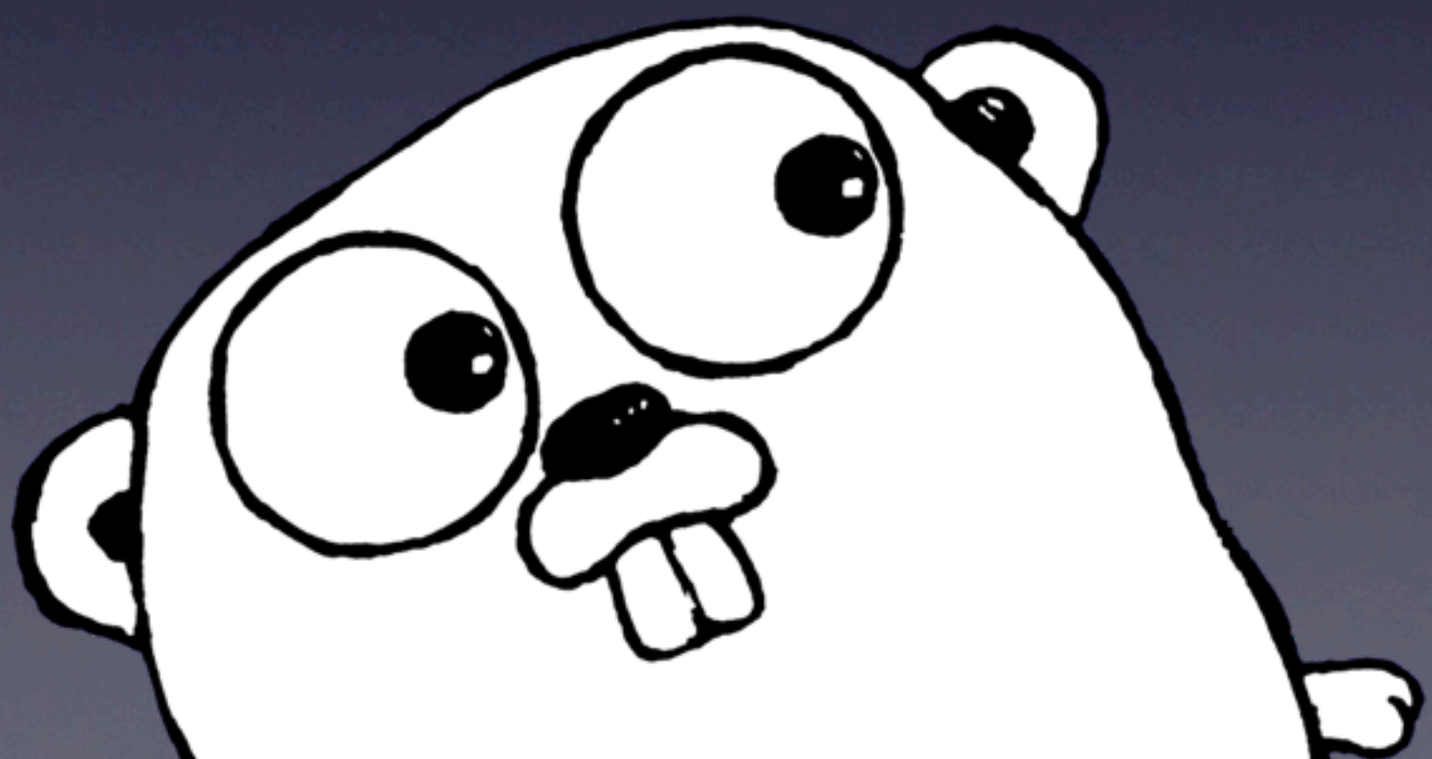
More Cooperation

Communicating Sequential Processes (CSP)
Remember the Transputer?

CSP

- Processes pass messages over channels
- Similar to Actors except:
  - Processes are anonymous, while Actors have identities
  - CSP communication is synchronous vs Actors async
  - CSP uses explicit channels while Actors have named destinations
- Influenced Limbo which influenced Go

Did someone say Go?



Go from Google

- Google's take on CSP from the gang that brought you Unix, C, Plan 9, etc.
- Released in 2009 to a mixed reception
- Goroutines are coroutine-like primitives that communicate via channels
- Goroutines map to OS threads managed by the language runtime

goroutines

```
ch := make(chan int)
go sum(hugeArray, ch)
// ... do something else for a while
result := <-ch  // block & retrieve result
```

What about preemptive models?

The Rise & Fall of Threading as a Model

- Thread primitives have matured in the past 15 years
- POSIX Threads standardized in 1995
- Java (synchronized) and later C# (locked) have threading primitives as language/VM features
- Just spinning up a thread became a tempting way to implement concurrent applications

Threading is ***not*** a model

- Threads are an important ***primitive***
- Thread Pools & related tools are important patterns
- Build concurrency models on top of threads
- C10k dealt with a symptom of the threads-as-a-model problem.

Staged Event Driven Architecture

- Presented in 2000; built in Java
- Actor-like model for threaded application development
  - Specifically Internet servers (high concurrency)
- Threads represent stages connected by queues
- Load can be balanced by back stages with thread pool

Akka (2009)

- Event driven
- Actor model with lightweight processes
- Software Transactional Memory
- Distributed features

New Primitives: STM

- Software Transactional Memory
- Attempt to ease the use of shared memory vs. locking
- Thread writes a log, validates it, and commits it
- Hardware theory in 1986; Software implementation in 1995
- Suffers a performance penalty

DataFlow in Groovy

- No race-conditions
- No live-locks
- Deterministic deadlocks
- Completely deterministic programs
- BEAUTIFUL code

Dataflow Example

```
importstatic
groovyx.gpars.dataflow.DataFlow.task
final def x = new DataFlowVariable()
final def y = new DataFlowVariable()
final def z = new DataFlowVariable()

task {
    z << x.val + y.val
    println "Result: ${z.val}"
}

task {
    x << 10
}

task {
    y << 5
}
```

Everybody wants new primitives & patterns

- Apple's Grand Central Dispatch
  - lightweight OS threadpool; new Obj-C syntax
- JavaScript's Web Workers (lightweight threadpool)
- .NET 4's Task Parallel Library (threadpool)
- new C# syntax for asynchronous tasks
- JDK 7 introduces a fork/join framework
  - lightweight threadpool executing map/reduce like tasks with an intelligent scheduler
- Newish languages like Io (actor based concurrency)
- And new life for some older ones...

Seaside web framework

Smalltalk's continuations used to handle multiple web requests sequentially

Node.js

- Concurrency via closures & callbacks
- Single frame continuations
- **Use it if you like JavaScript**
- Twisted in Python (2002)
- Event Machine in Ruby (2006)
- Scheme (1975)

In Conclusion

- Concurrency primitives haven't changed in a long time
- Concurrency models improve very slowly
- 10 years from theory to mainstream implementation for fork/join in Java
- No single primitive, model, or buzzword is a panacea
- **Need more frameworks**
  - Models too abstract
  - Primitives too hard to get right
  - Cross-platform remoting please!