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

See:
http://blogs.oracle.com/yuanlin/entry/concurrency_vs_parallelism_concurrent_programming
http://blogs.oracle.com/yuanlin/entry/more_on_concurrency_vs_parallelism

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

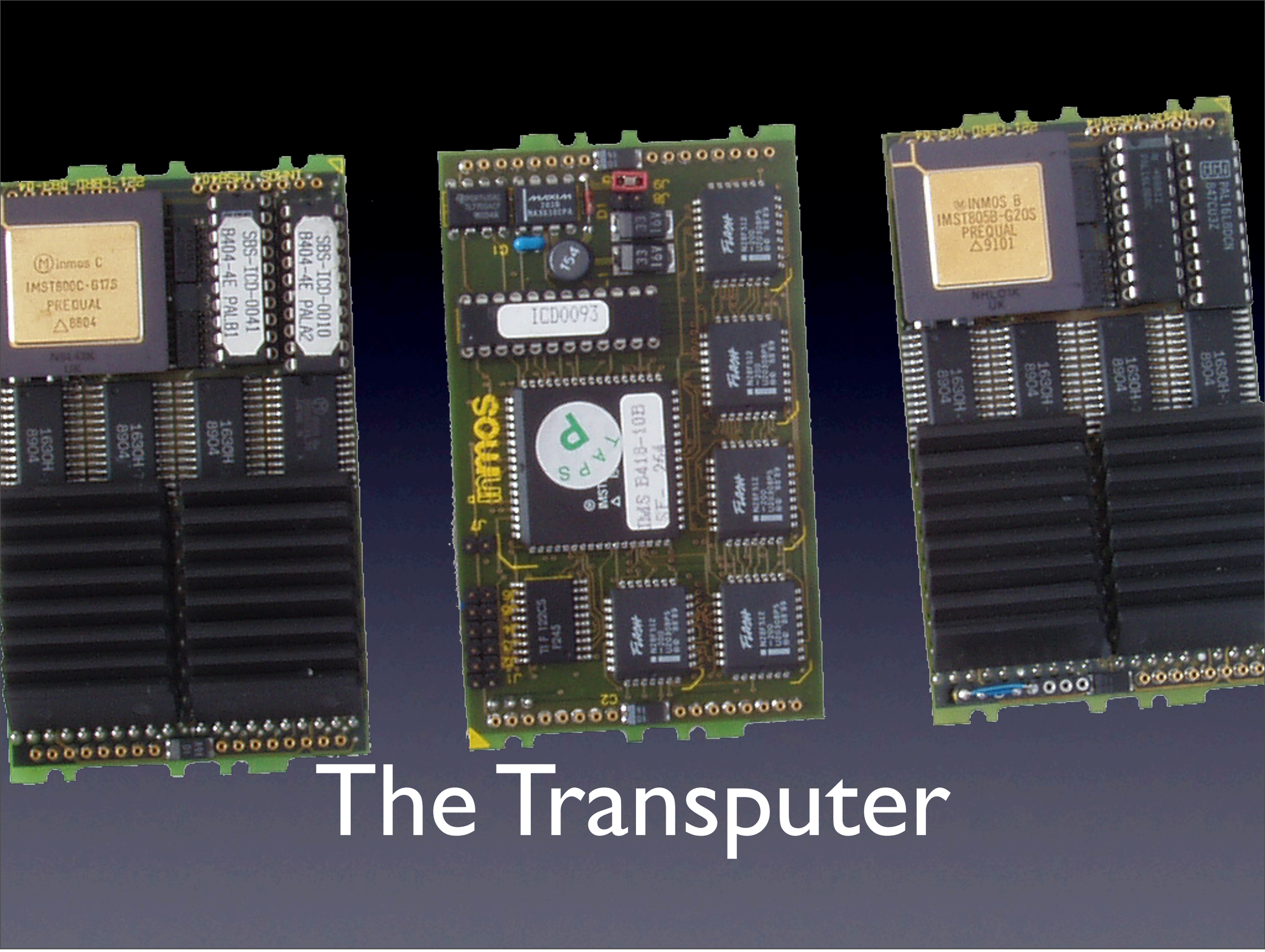
Concurrency depends on the developer and language while parallelism usually depends on the hardware (and software, system state, other runtime characteristics)

What isn’t this talk?

This is not a hardware talk

except...

General purpose computers are only going to get more cores (more parallel).
Even phones going multicore.
Used to just be specialty systems.



The Transputer

Thursday, June 23, 2011

Theoretical groundwork for CSP laid in 1978, First implementation in 1983.
System on a chip; Horizontally scalable CPUs; Expensive; Didn't run Unix
http://en.wikipedia.org/wiki/File:Transputer_Standardmodule_IMSB404_IMSB418_73.jpg

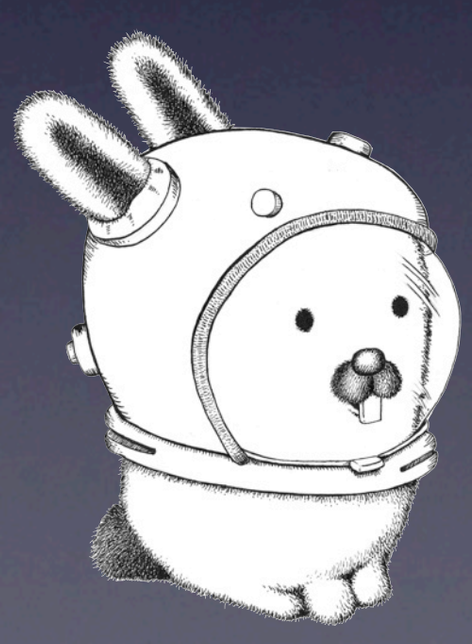
This is not a computer science lecture.

General purpose languages only.
I'm no Dijkstra.

No: Chu spaces, tuplespaces, hypercomputing

This is not a talk about distributed computing

Although it will probably come up.



Because being able to distribute work is critical for most cases where concurrency is critical.

Glenda from: http://cm.bell-labs.com/plan9/glenda.html

In the beginning

Multiprogramming via time-slicing appears in late 50s

Multiprogramming refers to the ability of a computer to run multiple programs concurrently
Pioneered by John McCarthy of Lisp fame in 1957
The Dartmouth Time Sharing system was the first commercial system and is available online via emulator: http://dtss.dartmouth.edu/
http://en.wikipedia.org/wiki/Dartmouth_Time_Sharing_System

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)

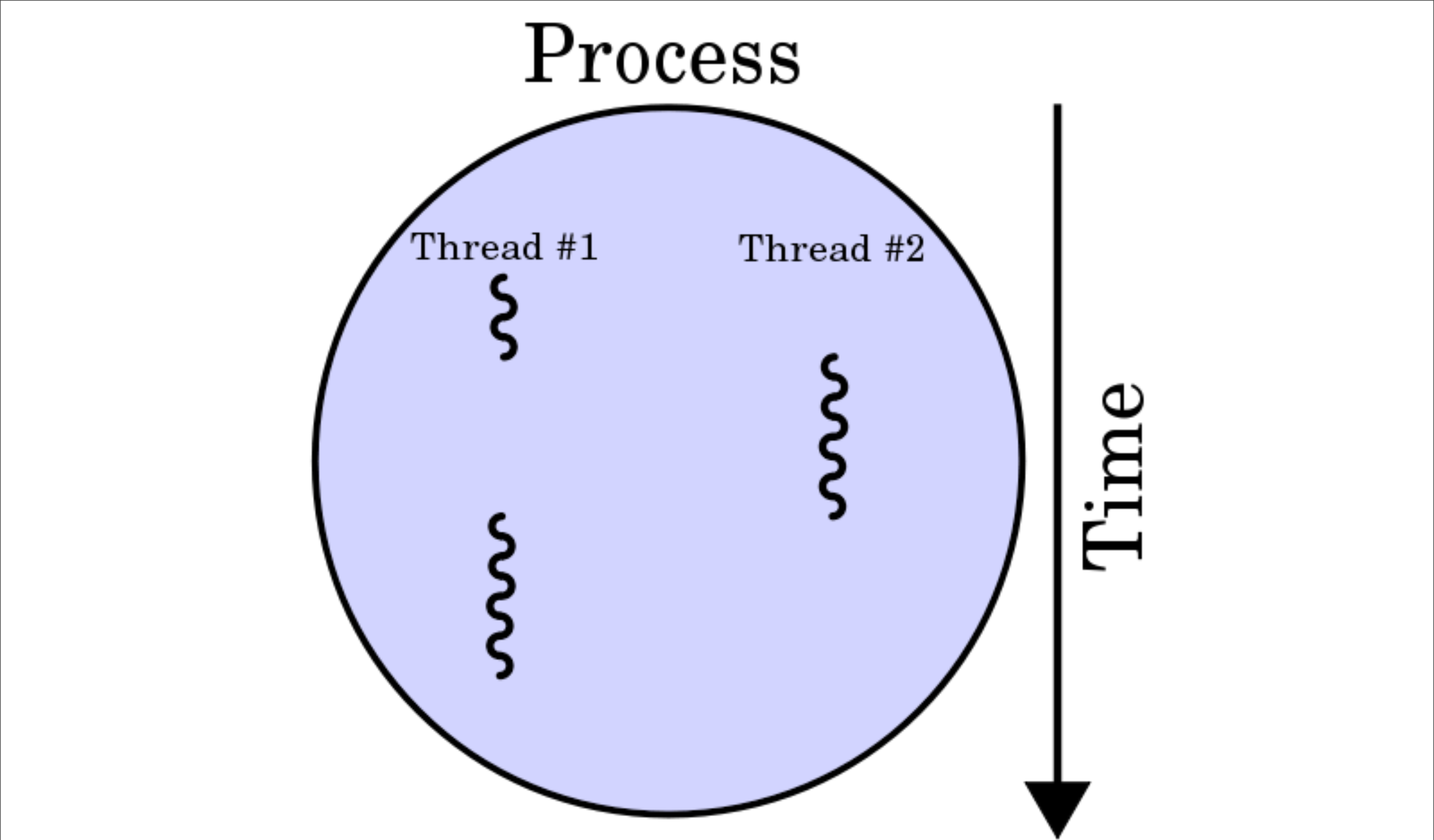
Where are threads?

Threads are...

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

Thursday, June 23, 2011

Lightweight processes *or* similar primitive concurrency construct
Generically: a thread of execution -



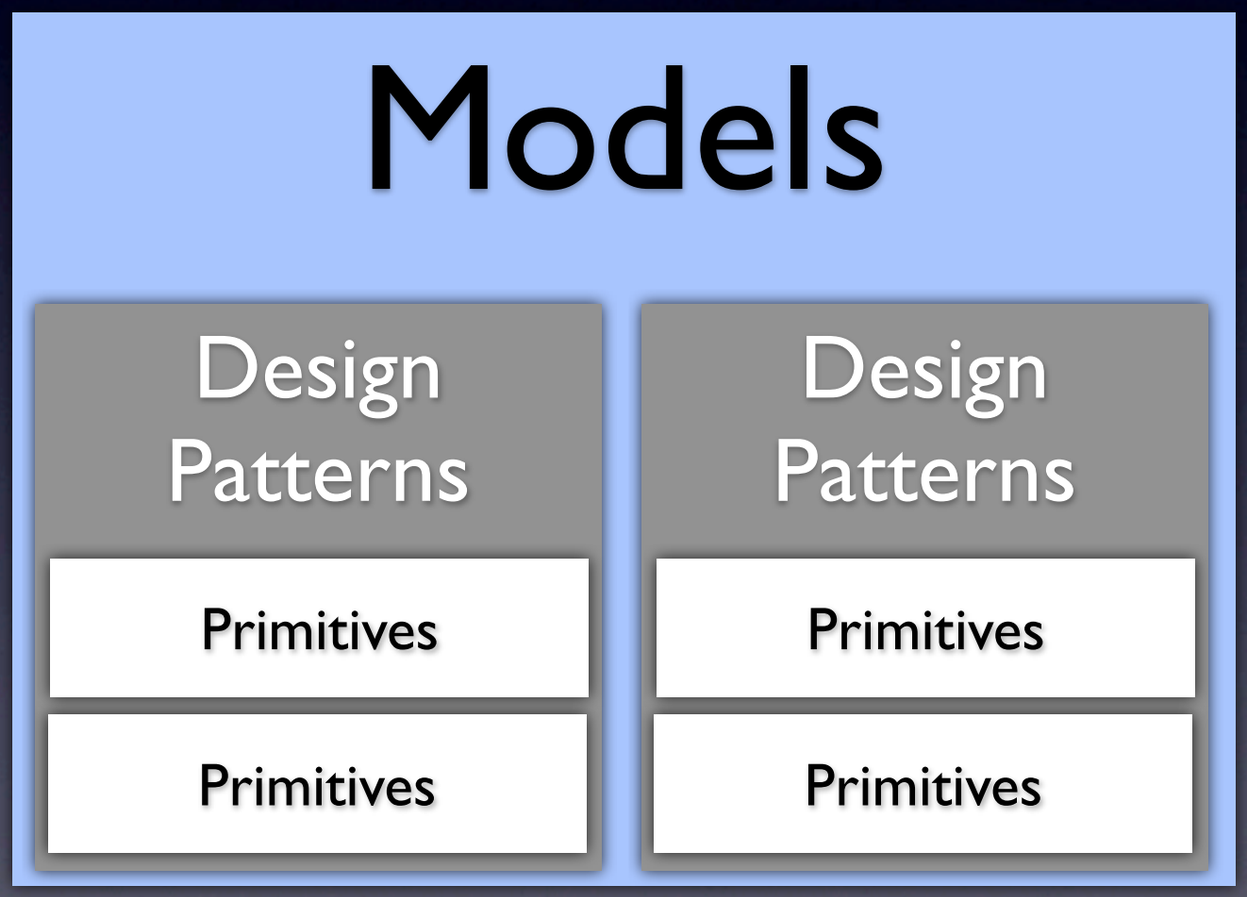
Threads: How do they work?

Thursday, June 23, 2011

Presenting: The most useless diagram ever.

http://en.wikipedia.org/wiki/File:Multithreaded_process.svg

Models & Primitives



Thursday, June 23, 2011

Models - Actor, CSP, etc
Design Patterns - Thread pools, Queues, Continuation passing, Callbacks
Primitives - Processes, Threads, Semaphores, Monitors, Coroutines, Continuations

Two Models of Multitasking

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

Multitasking - a generic term for concurrency
Time-slicing is an implementation detail of preemptive

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 (Melvin Conway, 1963)
"Subroutines are special cases of ... coroutines." —Donald Knuth.
Unix pipes are a kind of coroutine (coprocess really)

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
```


What if g blocked? What if g used a shared resource?

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)

Thursday, June 23, 2011

To understand how Linux could go so long without thread support you need to understand threads vs. processes.

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)

Thursday, June 23, 2011

Exception to #1: Thread local storage
Unix pipes were a huge advancement as the kernel runs the processes cooperatively and buffers the stream between

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)

Coordination = locks
What if the g() generator required a shared resource?
What if the threads required a shared resource?

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

Thursday, June 23, 2011

No guarantee messages are delivered
Actors can start more actors
High level model/abstraction; lots of variation in implementation

Erlang (1986)

Actor model in core language & runtime design

Implements Actor model using 1 userspace process per Actor and multiple OS threads for parallelism; Virtual machine wasn't parallelized until 2006!
Initially implemented in prolog; distributed

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

Thursday, June 23, 2011

Inspired by Modula building modules and processes on top of Pascal.
"[Erlang was designed to] raise the level of programming by using a language technology at a high abstraction level which provides built-in support required by the application domain."
- Bjarne Däcker
Fantastic reads:
http://ll2.ai.mit.edu/talks/armstrong.pdf
http://www.erlang.se/publications/bjarnelic.pdf

Erlang Example

```
% 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.
```


Stolen shamelessly from http://en.wikipedia.org/wiki/Erlang_(programming_language)

More Cooperation

Communicating Sequential Processes (CSP)
Remember the Transputer?

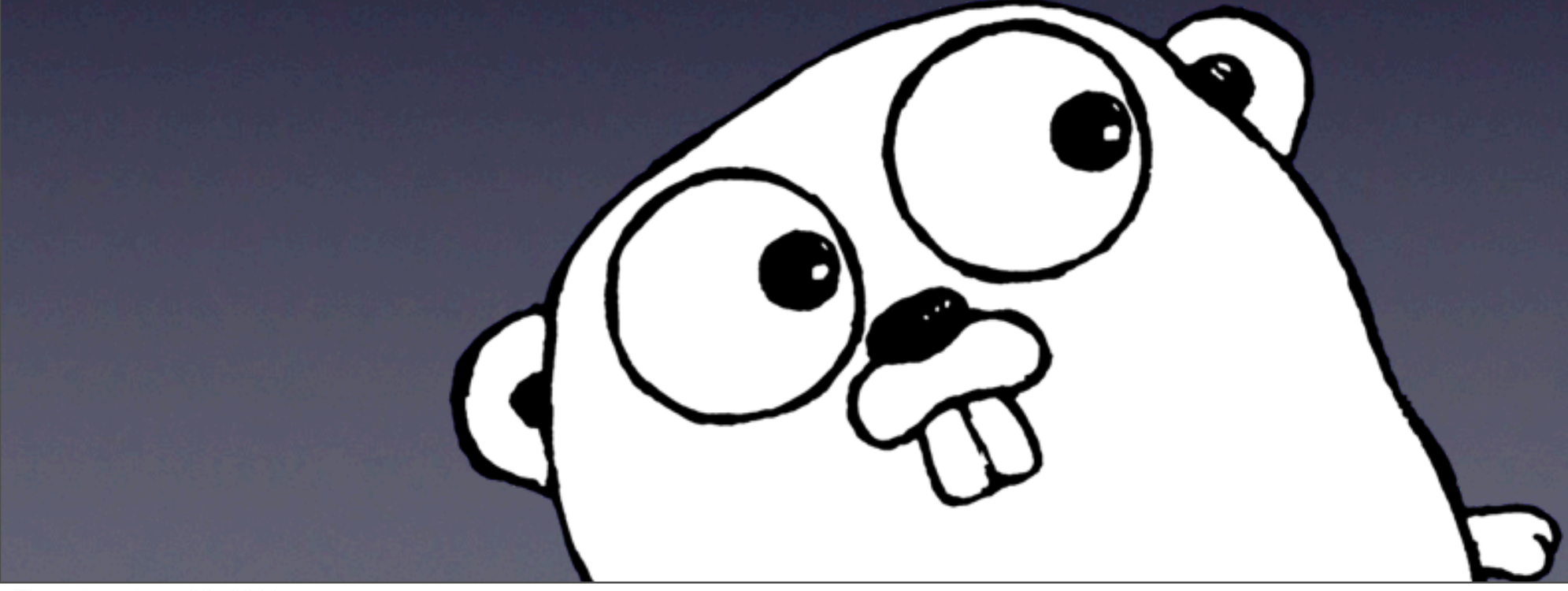
CSP

- 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

Thursday, June 23, 2011

Implemented in occam and the Transputer
Provable concurrency and flow
Lookup process calculi for nitty gritty

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

Thursday, June 23, 2011

Any function can be called as a goroutine (remember subroutines are a subset of coroutines? Google loves Knuth)

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

Thursday, June 23, 2011

From earlier slide: This is why threads were never "discovered" - the primitive/concept existed since the first multiprogramming systems and there was never a threading "model" until the 80s or 90s.

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.

C10k: http://www.kegel.com/c10k.html 2003
Shamelessly stolen from Joe Gregorio's PyCon talk
http://bitworking.org/news/2010/02/pycon

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

A great step away from threads as models to more formalized architecture.

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

Like database transactions
Support in all major languages via libraries; Clojure builds it in
Personal opinion: encourages wrong behavior (shared memory); still might be useful in certain cases.

DataFlow in Groovy

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

Stolen from: http://gpars.codehaus.org/Dataflow

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...

New greenlet implementation in Python might go cross-language

Seaside web framework

Smalltalk's continuations used to handle multiple web requests sequentially

Modern web framework for an old language (1969/72)
Smalltalk influenced the design of the actor model
Handle entire web sessions within a continuation

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)

Callbacks popular among GUI frameworks where deeply nested callbacks & concurrency are unlikely.
Nested closures are very difficult to test.
And yes, I'm jealous of how fast V8 is.

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!