High-Performance Web Applications in Haskell

Gregory Collins
Google Switzerland

QCon, London, UK

Friday, March 11, 2011

A little about me

- My academic background was in type systems and functional programming, mostly in Standard ML.
- Past 2–3 years: 90% of my spare-time hacking has been in Haskell.
- I'm one of the lead programmers of the "Snap Framework" (http://snapframework.com/), a web server/programming library written in Haskell.
- I work at Google Zürich as a Site Reliability Engineer.
  - *When Google breaks, we fix it.*
  - *Google is hiring!*

What’s this talk about?

My assertion: Haskell is a really good choice for web programming.

Back in 1995, we knew something that I don't think our competitors understood, and few understand even now: when you're writing software that only has to run on your own servers, you can use any language you want. …

Our hypothesis was that if we wrote our software in Lisp, we'd be able to get features done faster than our competitors, and also to do things in our software that they couldn't do. And because Lisp was so high-level, we wouldn't need a big development team, so our costs would be lower.

— Paul Graham, Beating the Averages

In a recent talk I said something that upset a lot of people: that you could get smarter programmers to work on a Python project than you could to work on a Java project.

I didn't mean by this that Java programmers are dumb. I meant that Python programmers are smart. It's a lot of work to learn a new programming language. And people don't learn Python because it will get them a job; they learn it because they genuinely like to program and aren't satisfied with the languages they already know.

— Paul Graham, The Python Paradox

In a recent talk I said something that upset a lot of people: that you could get smarter programmers to work on a ~~Python~~ ***Haskell*** *project than you could to work on a Java project.*

I didn't mean by this that Java programmers are dumb. I meant that ~~Python~~ ***Haskell*** *programmers are smart. It's a lot of work to learn a new programming language. And people don't learn ~~Python~~* ***Haskell*** *because it will get them a job; they learn it because they genuinely like to program and aren't satisfied with the languages they already know.*

— ~~Paul Graham~~ ***Me***

Why build web applications in Haskell?

- Expressiveness
- Correctness
- Performance

Expressiveness

- Closures and higher-order functions are awesome:

```
map :: (a -> b) -> [a] -> [b]

map toUpper "hello, world!" == "HELLO, WORLD!"

foldl' (+) 0 [1..10] == 55
```

- In Haskell, a little bit of typing can go a really long way.
- Higher-order functions allow you to abstract over common structural idioms — many (most?) Gang of Four “design patterns” are absolutely trivial for us

Expressiveness

Never write code like this again:

```
Iterator<Foo> it = l1.iterator();
ArrayList<Foo> l2 = new ArrayList<Foo>();
while (it.hasNext()) {
    l2.add(foo(it.next()));
}
```

Instead:

```
l2 = map foo l1
```

Correctness

Haskell helps you write correct programs in several ways:

- Strong static typing
- Pure functions
- Awesome testing tools

Strong static typing

- *Static* typing: the type of every value and expression is known at compile time, before any code is executed.
- *Strong* typing: the type system guarantees that a program cannot have certain kinds of errors.
    - *No null pointer exceptions*
    - *No segmentation violations*
    - *E.g. you can use the type system to ensure things like “HTML strings are always properly escaped”.*

Pure Functions

Biggest win for correctness: *pure functions*. (The Clojure guys have this figured out too!)

- Side effects are like inputs/outputs from functions which are hidden from the programmer.
- In Haskell, by default the output of a function depends **only** on the inputs the programmer explicitly provides.
- Pure functions have the following amazing property: *given the same inputs, a pure function will always produce the same output.*

Pure Functions (cont’d)

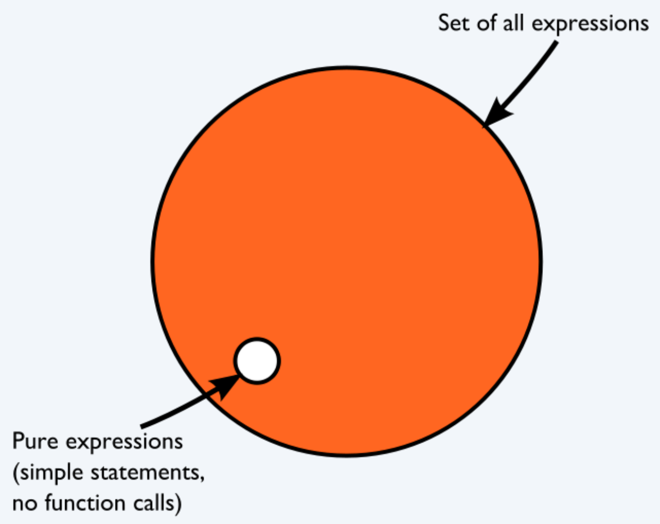
In most languages, any function you call could have arbitrary, unknowable side-effects: it could change state, write to disk, fire the missiles, etc.

Purity makes code more predictable and easier to test, by eliminating whole classes of potential errors.

In concurrent code, pure functions can never cause deadlocks or interfere with each other in any way. **Pure functions are always thread-safe!**

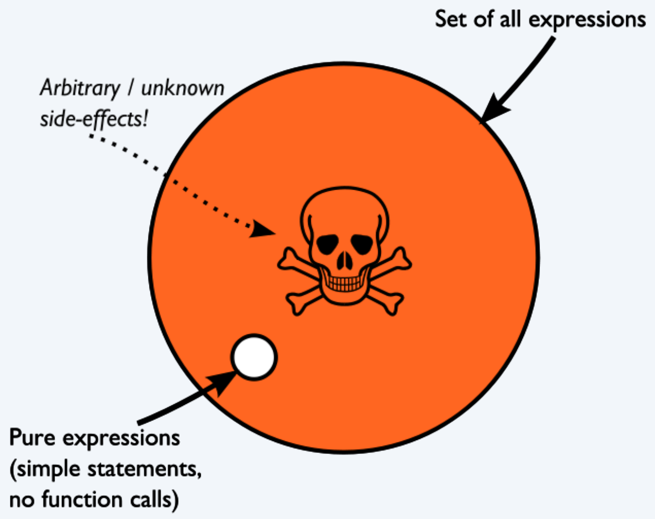
Pure Functions (cont'd)

Consider the Venn diagram of all the possible programs you could write in a typical programming language:



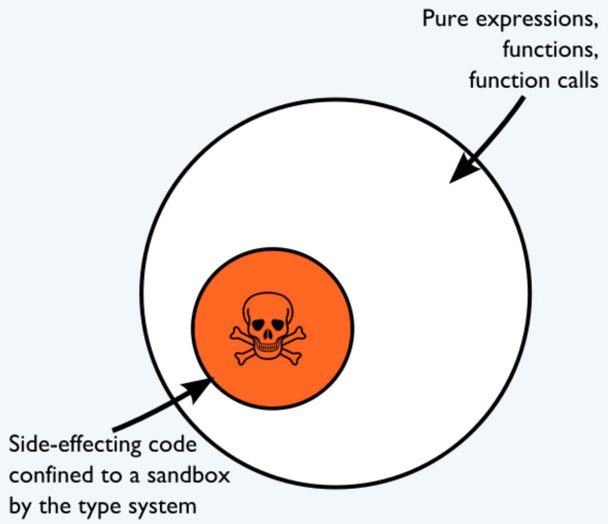
Pure Functions (cont'd)

Consider the Venn diagram of all the possible programs you could write in a typical programming language:


Set of all expressions
Arbitrary / unknown side-effects!
Pure expressions (simple statements, no function calls)

Pure Functions (cont'd)

In Haskell, the Venn diagram is reversed: code is pure by default, and functions which are potentially side-effecting are clearly marked as such by the type system.



Testing tools: why QuickCheck is the best thing since sliced bread

QuickCheck is kind of a “killer app” for Haskell.

- Programmers tend to be lazy when writing tests, and often only test for the cases they’re expecting to see.
- QuickCheck allows you to write propositional invariants about your code, and then QuickCheck will fuzz-test your invariant against a set of randomly-generated inputs.
  - *If it finds an input which breaks your invariant, it can quite often **shrink the testcase** to find a minimal example.*

QuickCheck, cont’d

In Haskell, we have a function called “take”, which takes the first N elements of a list:

```haskell
take :: Int -> [a] -> [a]
```

A couple of invariants we might want to test here:

- ∀ l . ∀ n | n >= 0 && length(l) >= n . length(take n l) == n
- ∀ l . ∀ n . isPrefixOf (take n l) n

QuickCheck, cont’d

Let’s write a (broken) implementation of take:

```
myTake n _      | n <= 1 =  []  -- "1" should be "0" here
myTake _ []             =  []
myTake n (x:xs)         =  x : myTake (n-1) xs
```

You can easily express those invariants using QuickCheck:

```
prop_length (l,n) = length l >= n && n >= 0 ==>
                    length (myTake n l) == n

prop_prefix (l,n) = myTake n l `isPrefixOf` l
```

QuickCheck, cont’d

QuickCheck easily finds the error and gives you a minimal failing testcase:

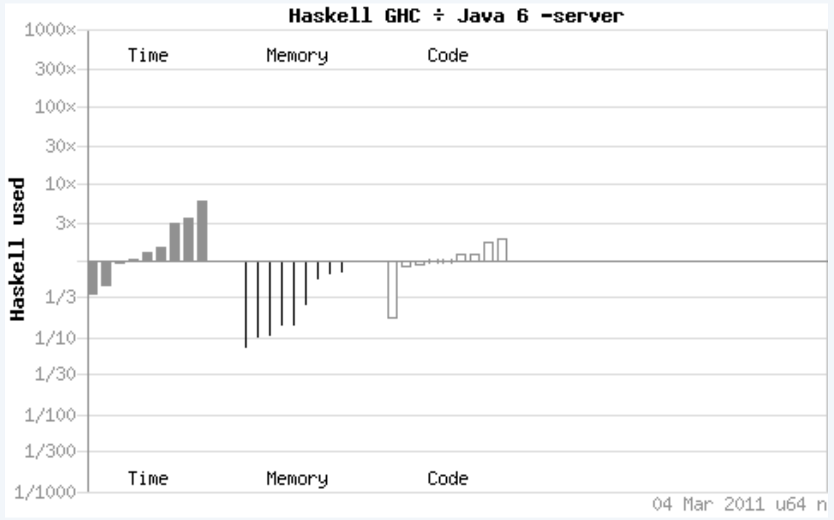
```
> quickCheck prop_length


*** Failed! Falsifiable (after 6 tests and 4 shrinks):
([()],1)
```

Performance

Haskell has an efficient native-code compiler, with performance competitive with languages like Java and Go.

- Compared to Java, the average Haskell program uses significantly less memory while being only slightly slower on average. (With an asterisk).
- We get this even though Haskell is a much higher-level language.
- Haskell does concurrency *really well*, out of the box.

Performance (cont'd)



(source: http://shootout.alioth.debian.org/u64/benchmark.php?test=all&lang=ghc)

Performance (cont’d)

When speed is absolutely critical or you want to get closer to the “bare metal”, Haskell’s “Foreign Function Interface” lets you easily drop down to C:

```
foreign import ccall unsafe "unistd.h read" c_read
    :: CInt -> Ptr a -> CSize -> IO (CSize)
foreign import ccall unsafe "unistd.h write" c_write
    :: CInt -> Ptr a -> CSize -> IO (CSize)
```

This is a piece of cake compared to Python’s FFI or JNI.

Scalability in web applications

Serving HTTP responses is an inherently concurrent problem.

- Servers communicating with multiple clients simultaneously.
- Servers should be able to scale well to large numbers of simultaneous clients.
- Highly parallelizable: typically, throw more CPUs at the problem and things go faster.

Concurrency models for web servers

1. Separate processes (forking or pre-forking). Every connection served by a separate OS process, and no OS process serves more than one request at once. Blocking I/O.
2. OS-level threads. Every connection served by a separate OS thread, one process serves many requests at once. Blocking I/O.
3. Event-driven. Server has one or more “event loops”, each of which runs in a single thread, handling N active connections. Uses OS-level multiplexing (epoll() or kqueue()) to get notifications for sockets which are ready to be read or written to. Non-blocking I/O.

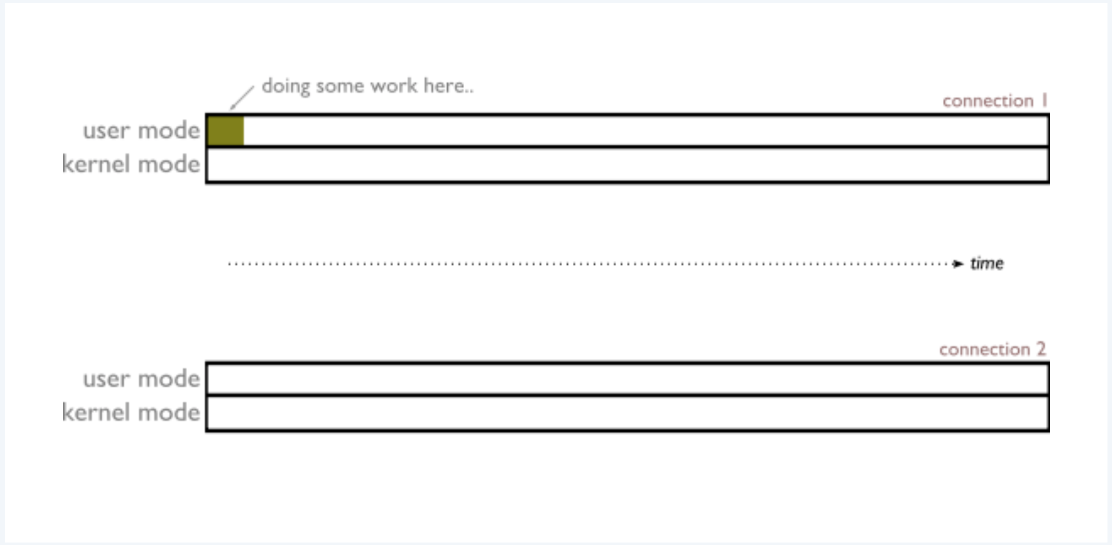
Separate processes

Everyone learns this pattern in Unix 101:

```
int s = accept(...);
pid_t pid = fork();
if (pid == 0) {
  /* handle request */
} else /* ... */
```

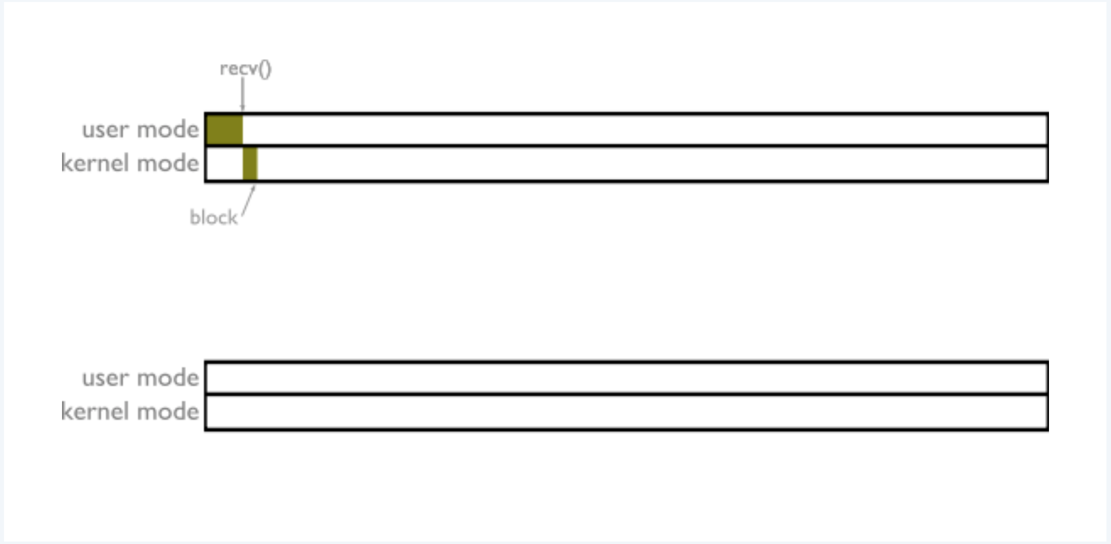
Separate processes: diagram

In the process model (and threading looks similar), we devote one OS thread per connection:



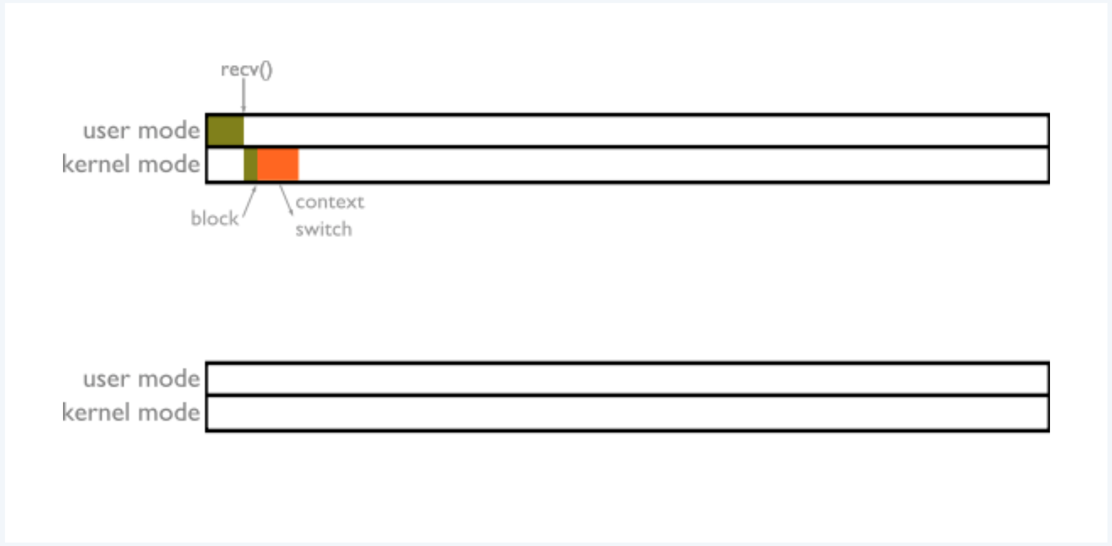
Separate processes: diagram

In the process model (and threading looks similar), we devote one OS thread per connection:



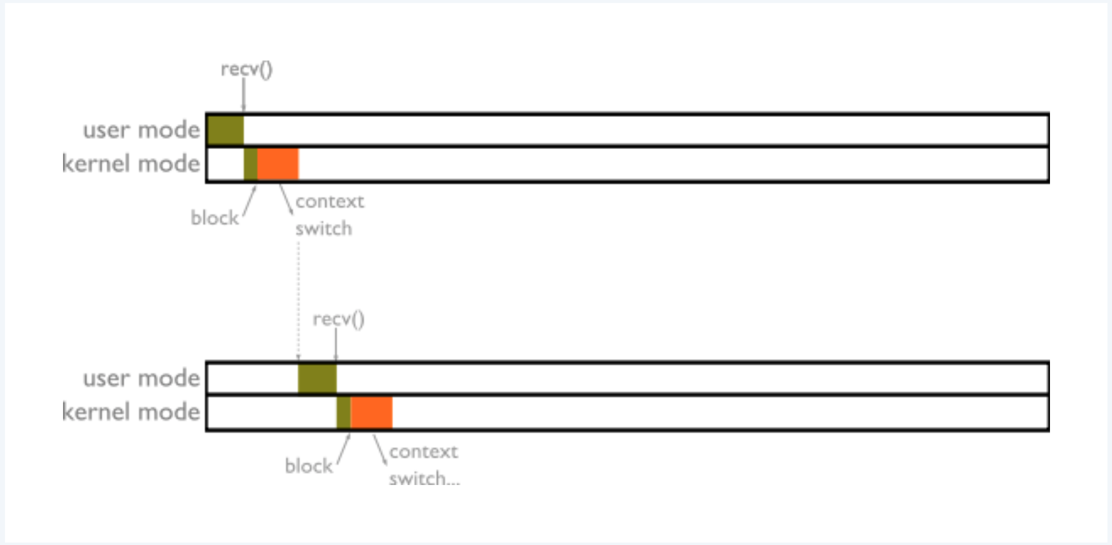
Separate processes: diagram

In the process model (and threading looks similar), we devote one OS thread per connection:



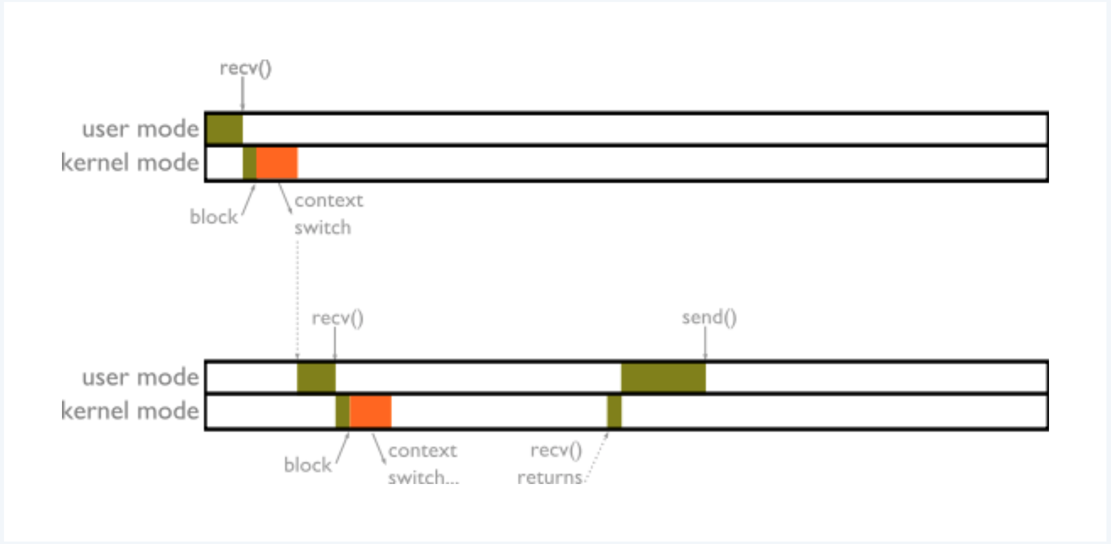
Separate processes: diagram

In the process model (and threading looks similar), we devote one OS thread per connection:



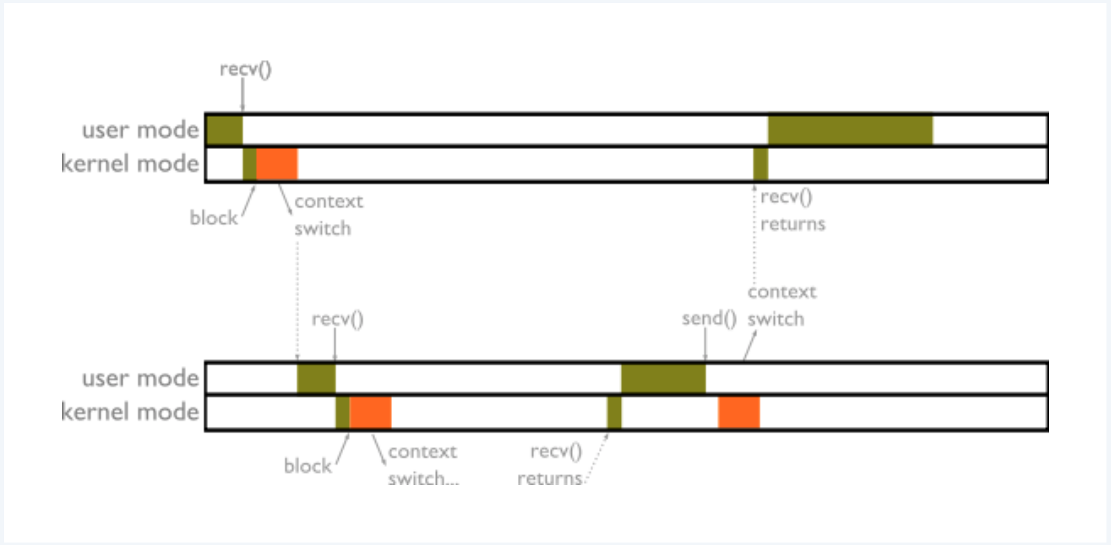
Separate processes: diagram

In the process model (and threading looks similar), we devote one OS thread per connection:



Separate processes: diagram

In the process model (and threading looks similar), we devote one OS thread per connection:



Separate processes: advantages

- simple programming model
- process isolation. No shared mutable state $\Rightarrow$ less opportunity for concurrency problems (deadlocks, etc)

Separate processes: disadvantages

- fork() is expensive, and even with pre-forking you have to fork more servers under heavy concurrent load
- per-process memory overhead is high
- context switches between processes are pretty expensive (O(1µs) for the context switch, plus CPU cache is likely to be trashed)
- if you want to communicate between processes (or share state) you need IPC

Threading

Instead of fork(), we create a lightweight thread:

```
int s = accept(...);
int tid = pthread_create(...);
...
```

Threading model: advantages

- Simple to implement with blocking I/O — like the per-process model, each request is handled in a logically-separate execution context.
- Lightweight threads have less context-switch (no need to remap MMU tables) and memory overhead (process descriptor tables, etc) than full processes.

Threading model: disadvantages

- switching between threads still involves a context switch in/out of kernel protected mode
- although lighter than processes, threads still introduce a high per-connection overhead (each thread requires its own stack, O(2MB))

Event-driven model

When events are received, the event dispatcher passes control to a callback function that does some work. When callbacks need to read or write more they re-register themselves with the event dispatcher.

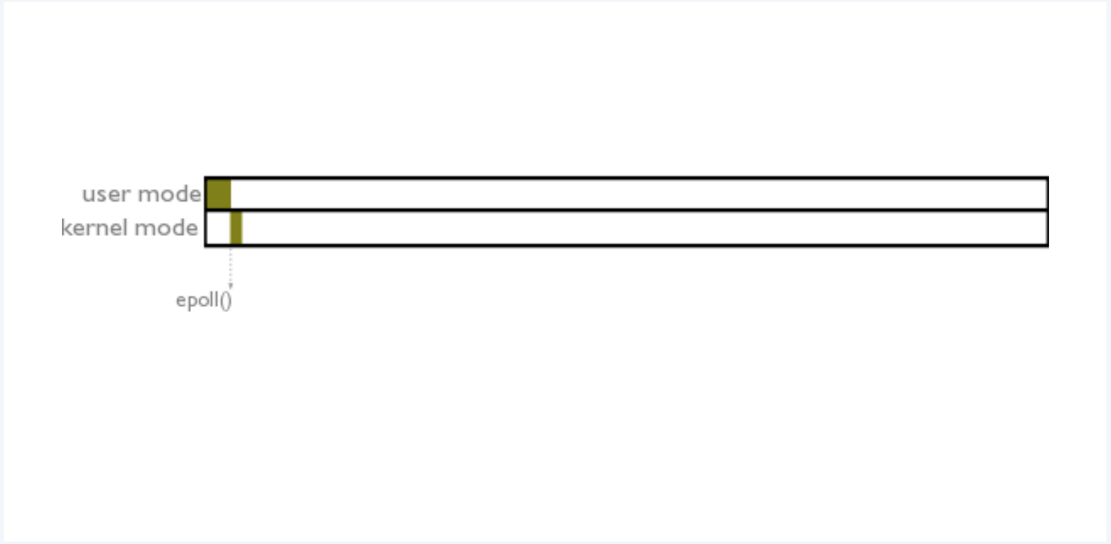
Event-driven model

Here’s an example from node.js (with a nod to Stefan):

```
function read(callback) {
    request.addListener("response", function (response) {
      var responseBody = "";
      response.setBodyEncoding("utf8");
      response.addListener("data", function(chunk) {
        responseBody += chunk;
      });
      response.addListener("end", function() {
        callback(responseBody);
      })
    });
    request.close();
}
```

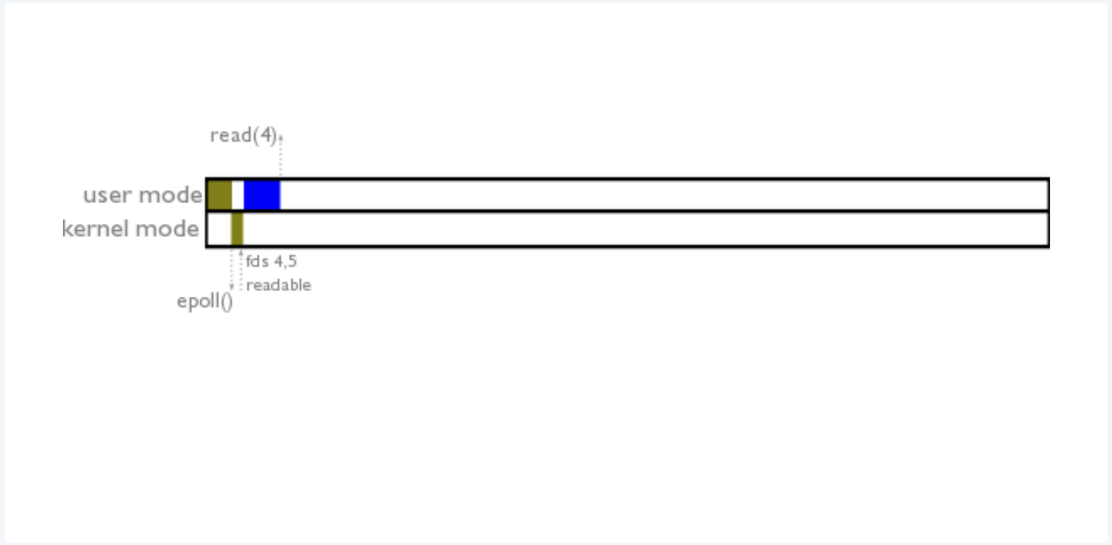
Event-driven model: diagram

In the event-driven model, we run an **event loop** on a single OS thread, and multiplex multiple connections on top of it:



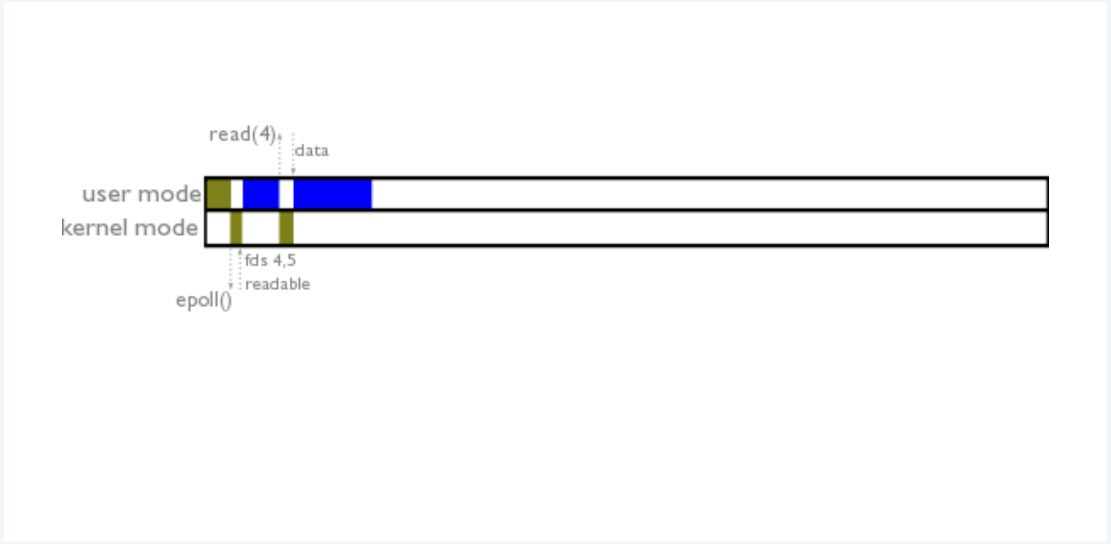
Event-driven model: diagram

In the event-driven model, we run an **event loop** on a single OS thread, and multiplex multiple connections on top of it:



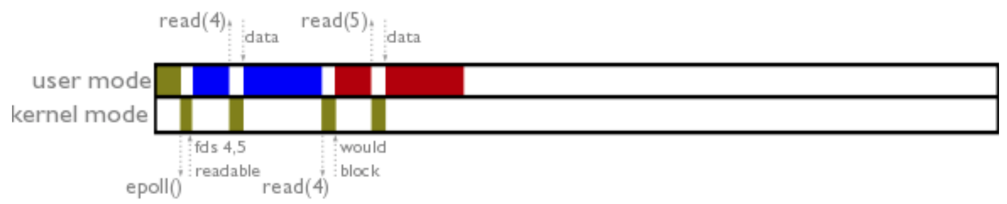
Event-driven model: diagram

In the event-driven model, we run an **event loop** on a single OS thread, and multiplex multiple connections on top of it:



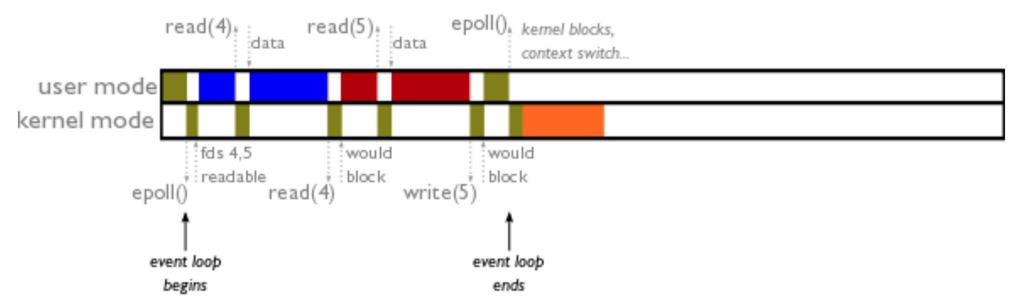
Event-driven model: diagram

In the event-driven model, we run an **event loop** on a single OS thread, and multiplex multiple connections on top of it:



Event-driven model: diagram

In the event-driven model, we run an **event loop** on a single OS thread, and multiplex multiple connections on top of it:



Event-driven programming: advantages

- Probably the most efficient model, especially when the server must handle large numbers of idle connections.
  - *Per-connection overhead is very low: a little bit of per-connection state, no stack.*
  - *Idle connections consume very few resources; modern syscalls like $epoll()$ and $kqueue()$ are $O(k)$ in the number of* ***active*** *connections, and usually $k << n$.*

Event-driven programming: advantages

- Key numbers here:
  - *Kernel context switch: O(1–4 µs) — max 250k–1M/s*
  - *Processor ring switch from user to kernel mode: O(50 ns)*
- If you want a highly-scalable networked server, using event-driven I/O under the hood is (almost) a must.
  - *All of the web server throughput champions (nginx, lighttpd, Cherokee, etc) use this model.*

Event-driven programming: disadvantages

- Callback functions must be careful to **never** issue blocking calls, as this will stall the entire event loop.
  - *Corollary: every C library you interface with must be adapted somehow to work asynchronously.*
- If you’re not careful….

Event-driven programming: disadvantages (cont’d)

- Nonlinearity of control flow becomes increasingly difficult to deal with as the complexity of the code increases.

Haskell’s threading model

Haskell uses **green threads**, which are lightweight user-space threads scheduled by the GHC runtime system.

- each thread takes up only a few kB for its stack and other context.
- M threads are multiplexed by the runtime system onto N OS threads.
- ignoring preemption, switching between Haskell threads does not involve a kernel context switch!

Haskell’s threading model (cont’d)

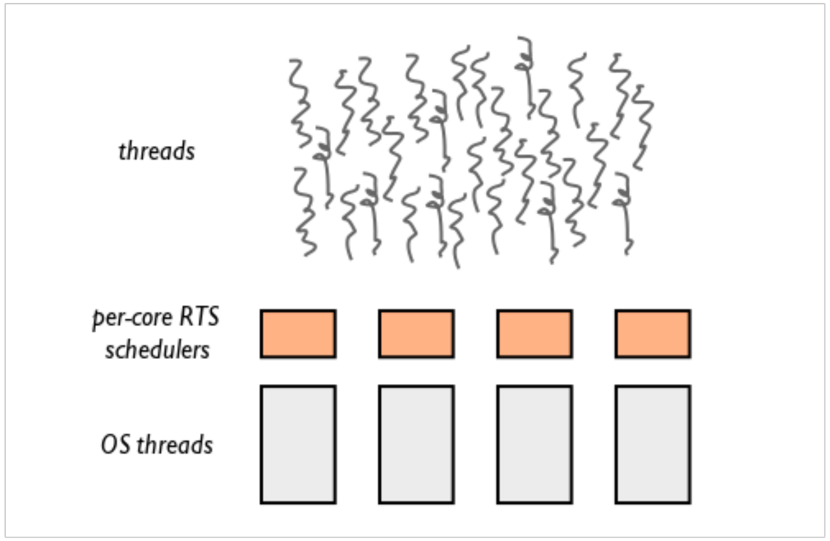
- Haskell threads are fast:
  - *thread-ring benchmark: 7.3M context switches per second, per-core (32-bit, Q6600), 4.2M context switches per second on x64.*
  - *this is because there is no process context switch or processor mode switch when scheduling the threads.*
- Entire Haskell runtime system uses non-blocking OS calls and uses userspace locks (and green thread scheduling) to get blocking behaviour.

Haskell’s threading model (cont’d)

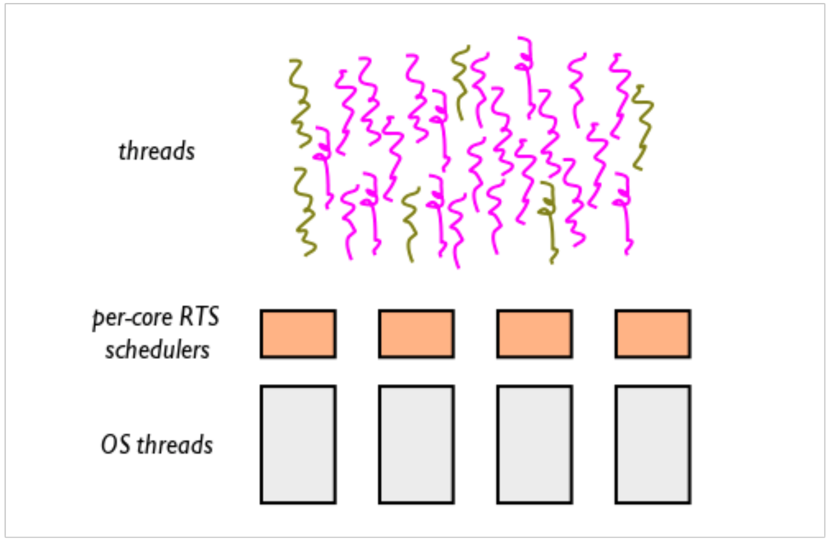
So what happens when a Haskell thread wants to recv() some data?

- thread makes a non-blocking call to recv()
- if the data is there, great!
- otherwise, we get EWOULDBLOCK and the thread:
    1. *schedules an interest in reading on the socket file descriptor with the runtime system*
    2. *waits on a lock which the RTS will twiddle when epoll() says the socket is readable.*

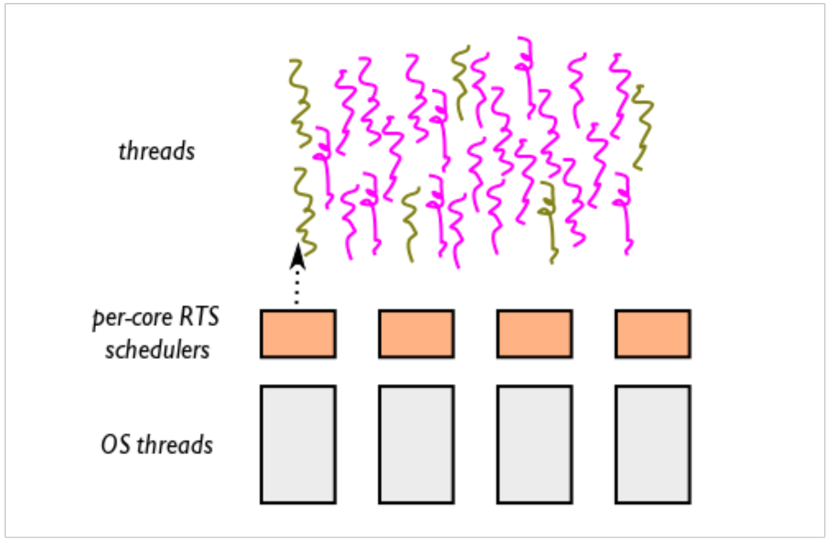
Haskell threading model (cont’d)



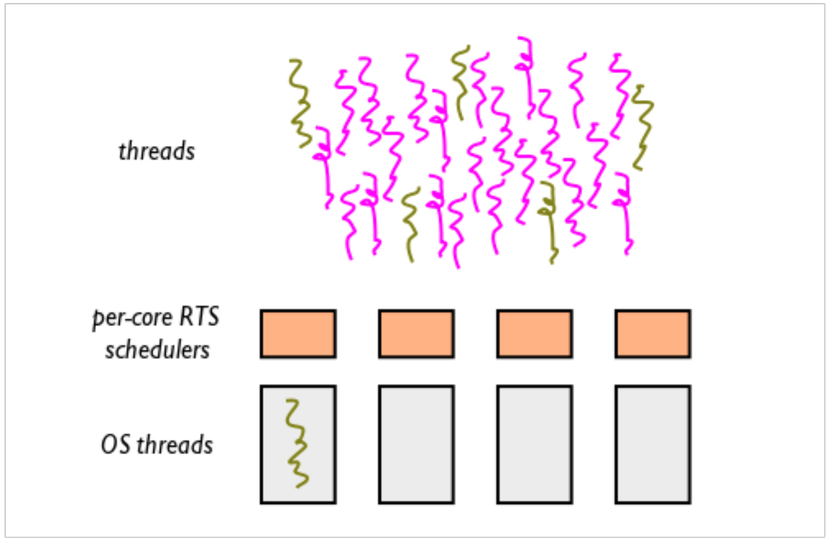
Haskell threading model (cont'd)



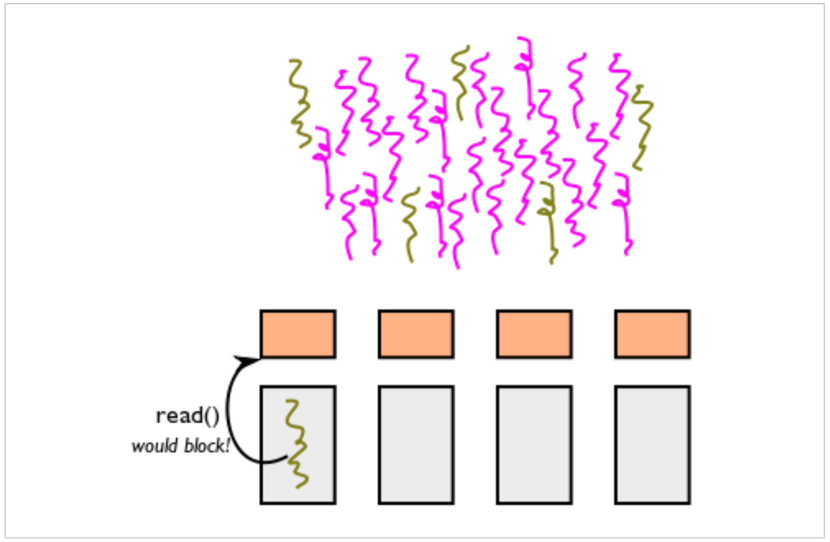
Haskell threading model (cont’d)



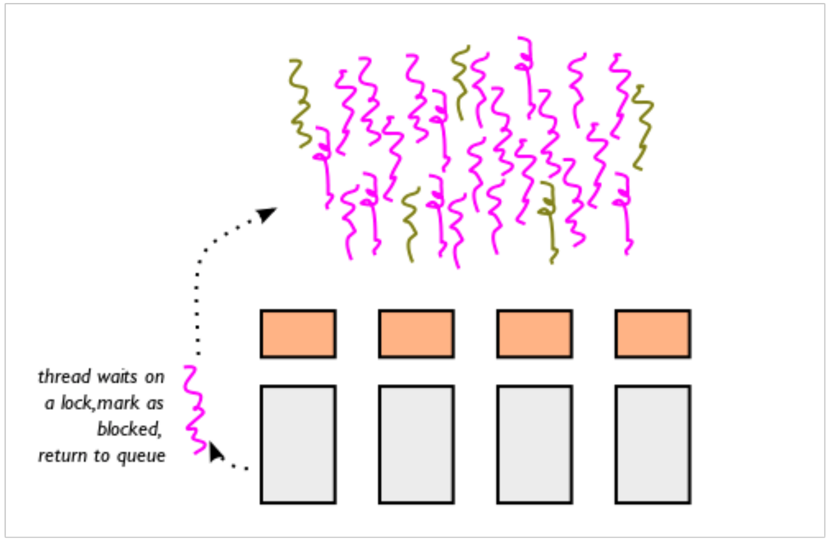
Haskell threading model (cont'd)



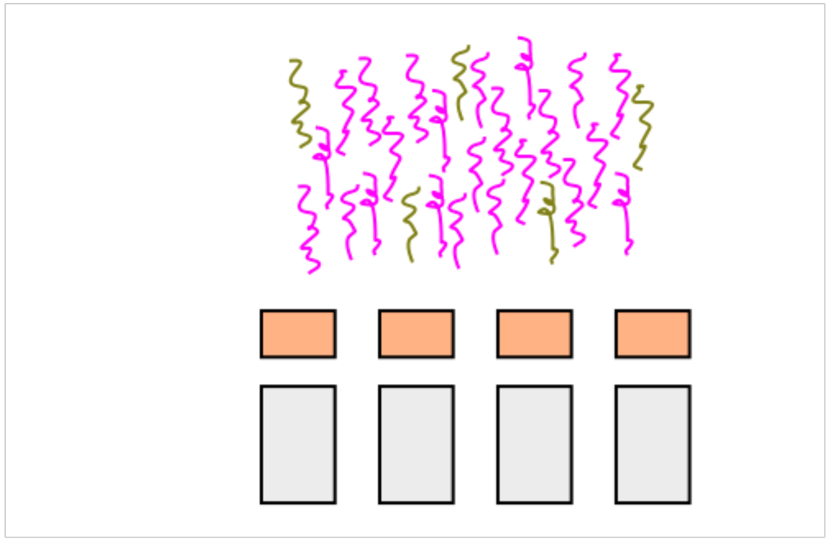
Haskell threading model (cont’d)



Haskell threading model (cont’d)

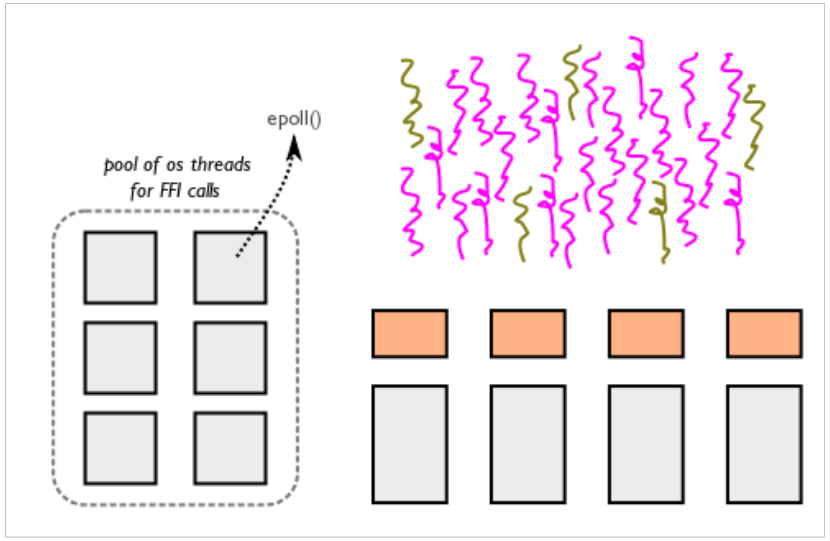


Haskell threading model (cont’d)



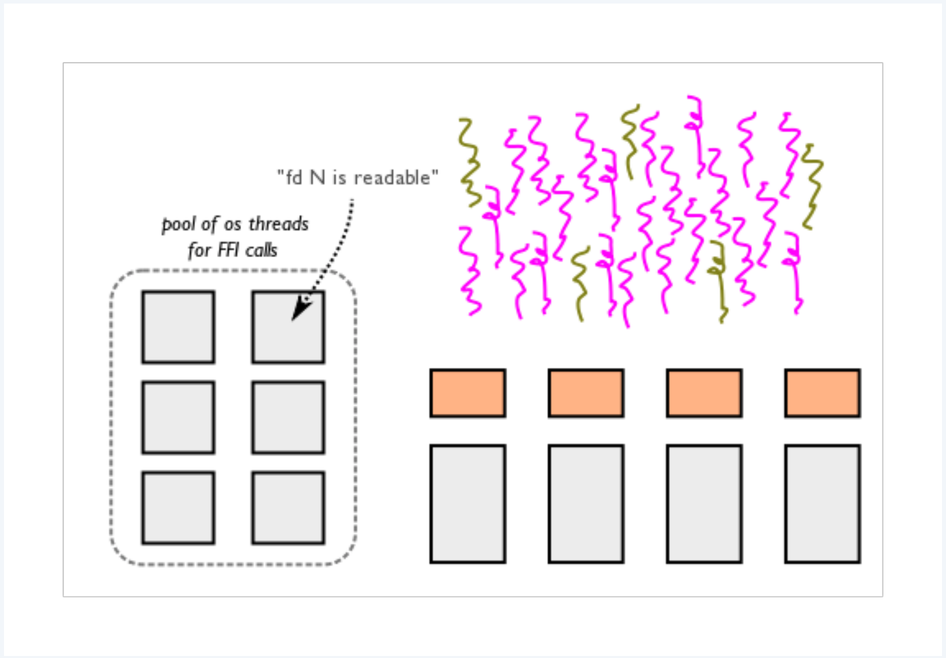
Haskell threading model (cont'd)

What about foreign calls that block?



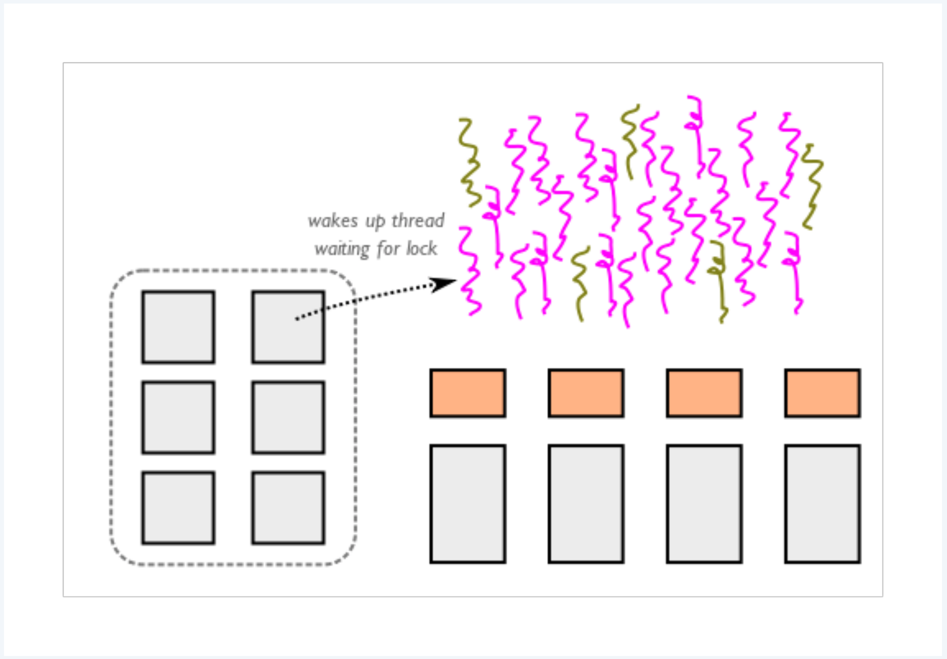
Haskell threading model (cont'd)

What about foreign calls that block?



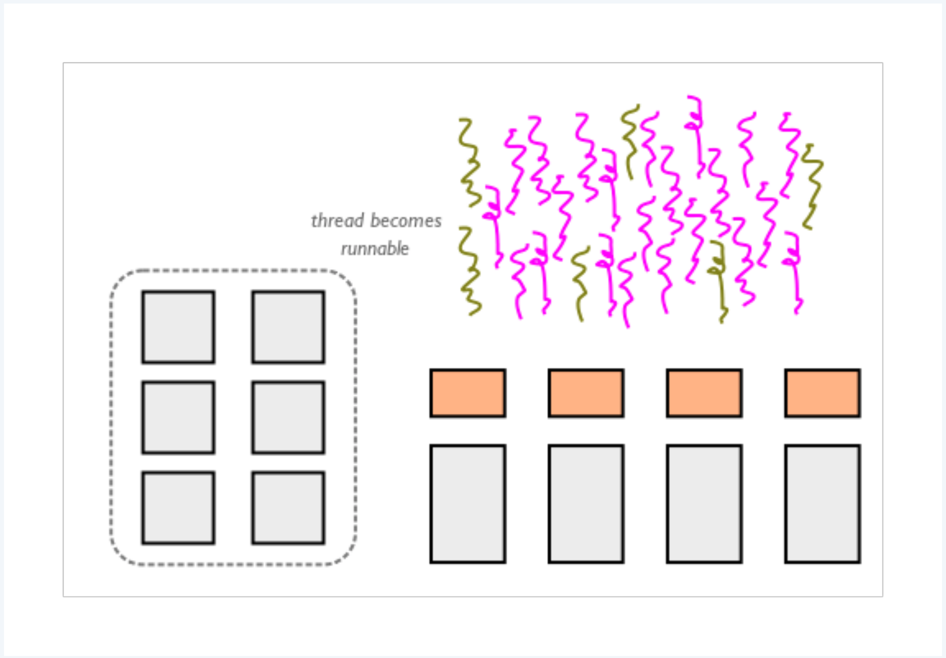
Haskell threading model (cont'd)

What about foreign calls that block?



Haskell threading model (cont’d)

What about foreign calls that block?



Haskell threading model: advantages

Haskell has a hybrid model; most I/O scheduling is event-driven (uses epoll() under the hood as of GHC 7), but for C calls that would block it **transparently** switches to threaded scheduling.

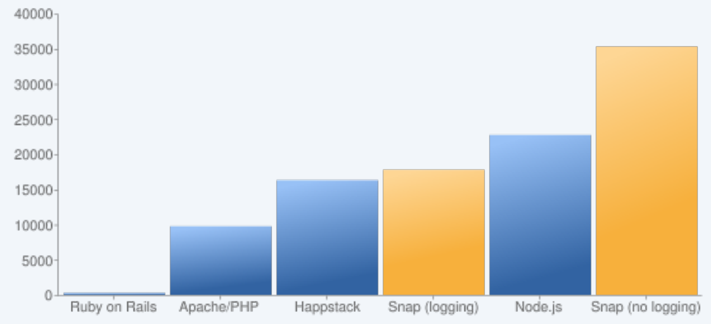
- we perform as if we were event-driven, but…
- no spaghetti callbacks.

Haskell web frameworks

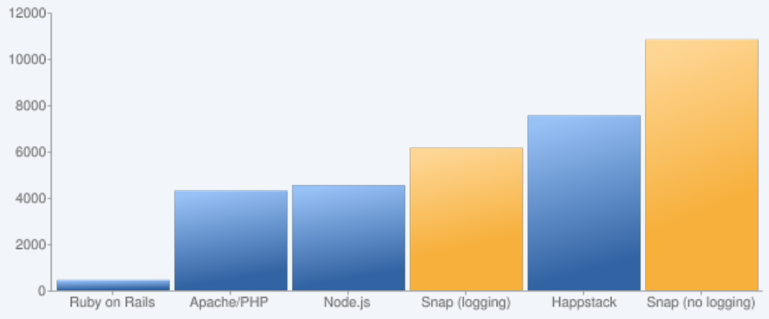
So what’s available in Haskell for doing web programming today?

- Happstack — http://happstack.com/
- Yesod — http://docs.yesodweb.com/
- Snap Framework — http://snapframework.com/

Snap Framework: unscientific benchmarks



(PONG benchmark, y axis is requests/second)



(serving a 40kB file, y axis is requests/second)

Haskell web frameworks (cont’d)

The guys who are working on Yesod just released a minimal web server called “Warp”.

- It’s 2–4X as fast as Snap on the pong benchmark!
- …so not only are we already fast, there’s headroom within the GHC runtime system to do even better.

So I can switch all my web programming to Haskell now?

- Haskell web community is young and is still getting going
- Many features web programmers have come to expect are skeletal or missing
- …but Haskell is currently an excellent choice for places where you need a targeted performance boost.
    - *API servers, compute-heavy workloads, hotspots*
    - *It’s close to as fast as C/C++/Java, but much much much nicer to program in.*
- We need hackers!

Thank you!

Gregory Collins
Google Switzerland

QCon, London, UK

Friday, March 11, 2011