Application Level Concurrency in Haskell: Combining Events and Threads

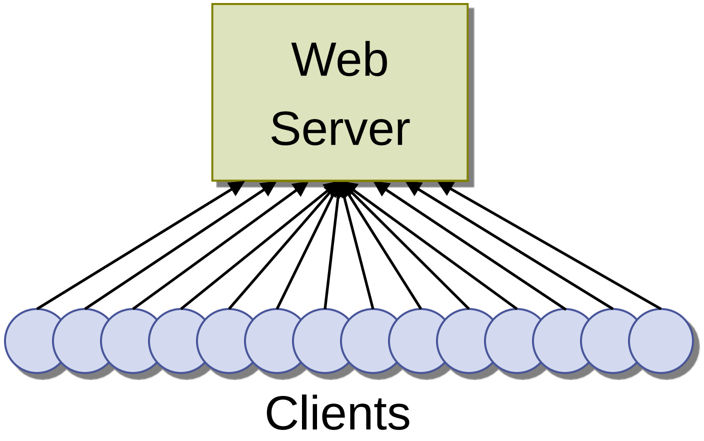
Steve Zdancewic
Peng Li

University of Pennsylvania

(Research presented at PLDI 2007)

Building Network Services

- Network Services:
  - Web servers, games, chat rooms, peer-to-peer systems, …



- Concurrency is necessary:
  - Mostly I/O-bound: many idle threads
  - "C10K problem"   10,000 clients on one server?

    [www.kegel.com/c10k.html]

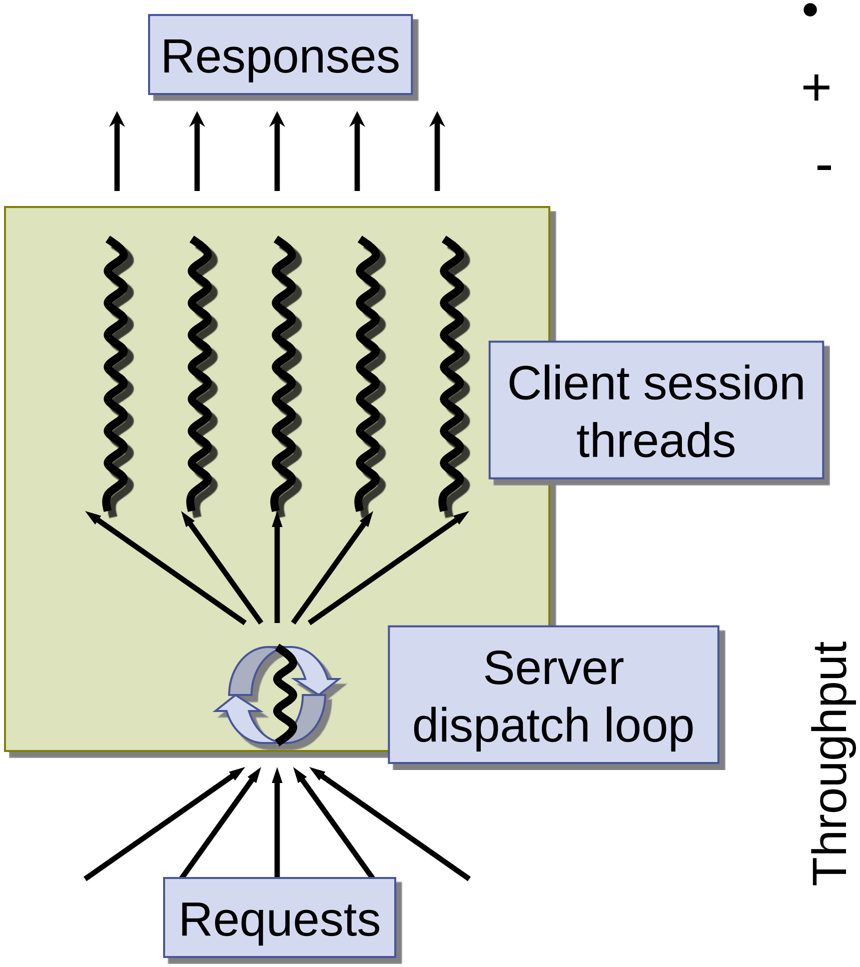
Threads
Simple programming model

vs.

Events
Scalability & Performance

- This talk: An attempt to reconcile these two approaches.

A Multithreaded Network Service



- One OS thread for each client

\+ Simple programming model

\- Less scalable:

inefficient memory usage,
context switching

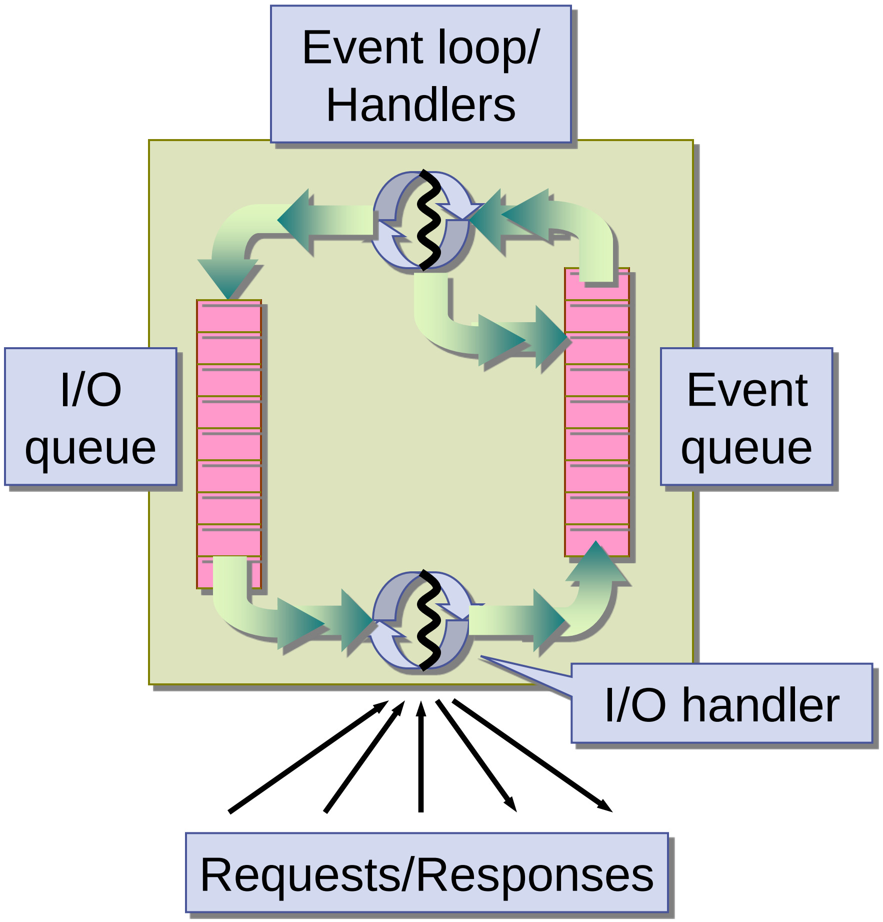
[Welsh et al. 2000]

Event-driven Network Service

• A few OS threads for handle all clients

- Complex programming model

+ More scalable:
efficient memory usage,
few context switches



Event loop/ Handlers

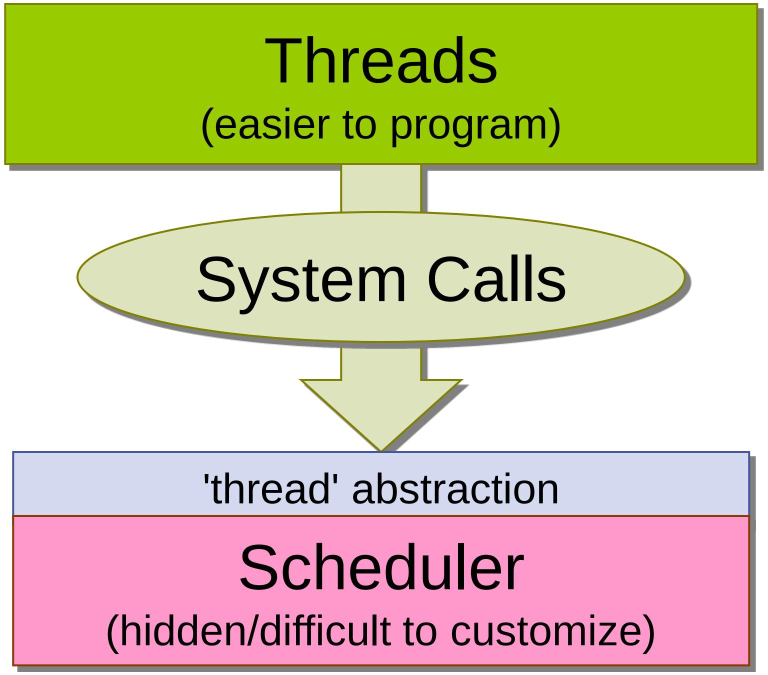
I/O queue

Event queue

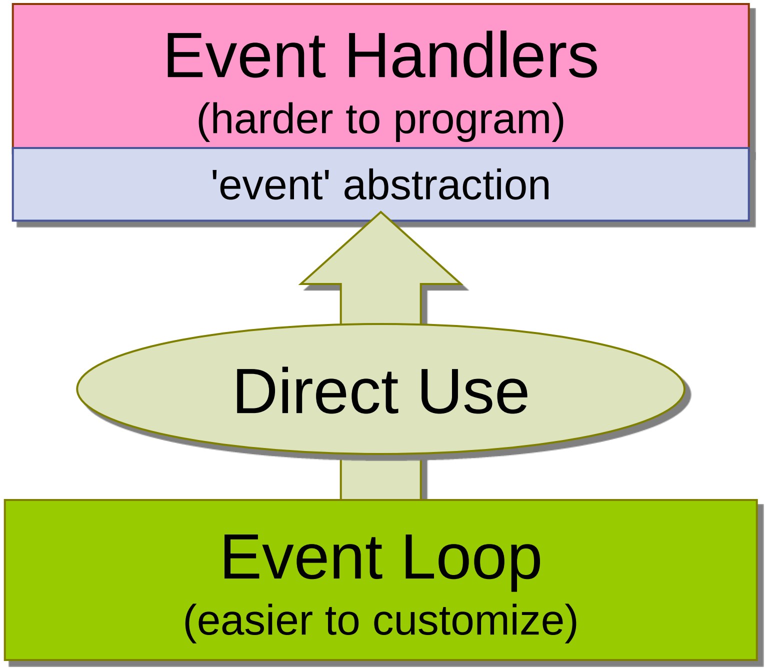
I/O handler

Requests/Responses

Threads


Threads
(easier to program)
System Calls
'thread' abstraction
Scheduler
(hidden/difficult to customize)

Events


Event Handlers
(harder to program)
'event' abstraction
Direct Use
Event Loop
(easier to customize)

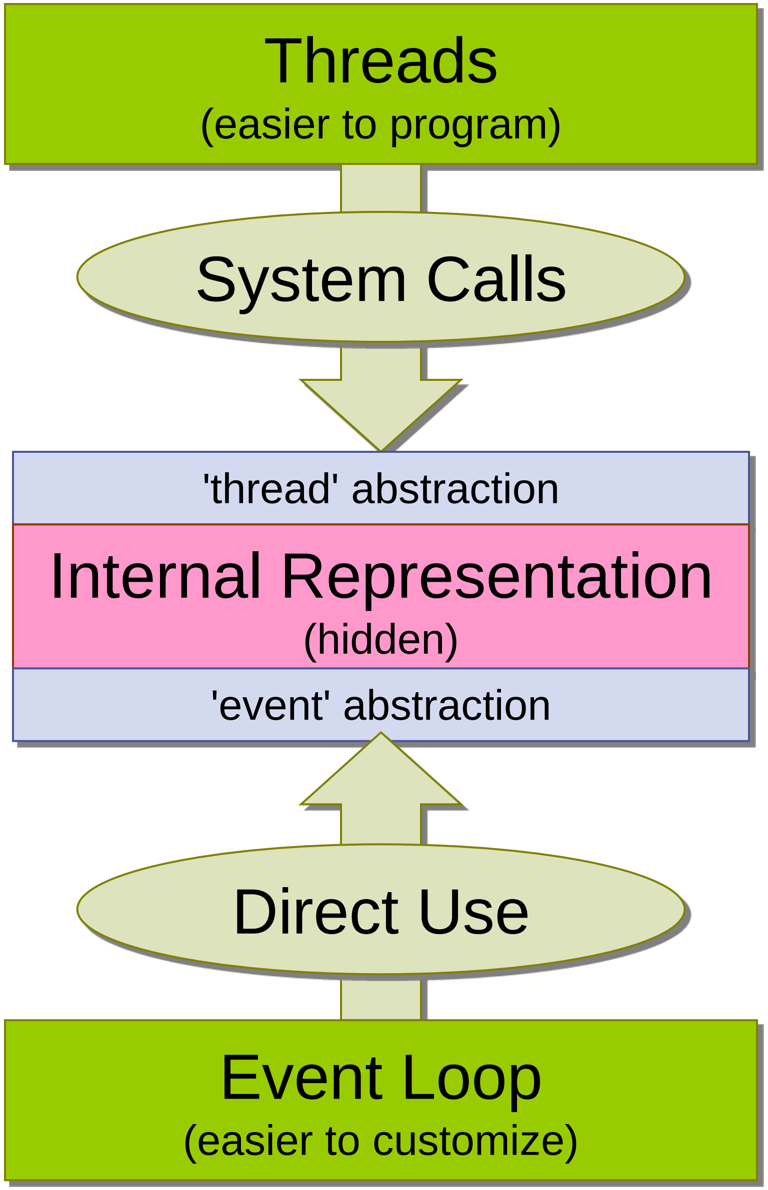
Thread Continuation	~	Event Handler
Thread Scheduler	~	Event Loop
Exported System Function	~	Event
Blocking Call	~	Send / Await reply

“On the duality of operating system structures” [Lauer&Needham 1978]

Spectrum of Solutions

- Flash Web Server [Pai, Druschel, Zwaenepoel 1999]
- SEDA: Staged event-driven architecture [Welsh, Culler, Brewer, 2001]
- Capriccio: Scalable threads [von Behren, et al. 2003]
- User-level threads / co-routines / continuations / etc.
  [Wand 1980], [Shivers 1997], [Claessen 1999],[Fisher & Reppy 2002],…
- Libraries/Compiler support for event-driven programs
  - Python's "Twisted" Package [twistedmatrix.com]
  - Automatic stack management in C++ [Adya, et al. 2002]
- Domain-specific languages
  - Erlang
  - Flux [Burns, et al 2006]
- …

Best of Both Worlds?



- Client Code: Threads
  - One thread ↔ One client
  - Familiar programming model
  - Blocking I/O
- Internal Representation
  - Hidden from the programmer
  - Automatic transformation from thread abstractions to events
- Scheduling:
  - Event driven
  - Customizable
  - Non-blocking I/O
- Application level:
  - Threads & scheduler implemented in high-level language

This Work: Network Services in Haskell

- Claim: High-level programming languages can simplify programming of network services while yielding good scalability and performance.

- Demonstrated using Haskell [www.haskell.org]
  - Pure: strong, expressive type system that isolates effects
  - Lazy: computations are performed 'on demand'
  - Functional: first-class functions

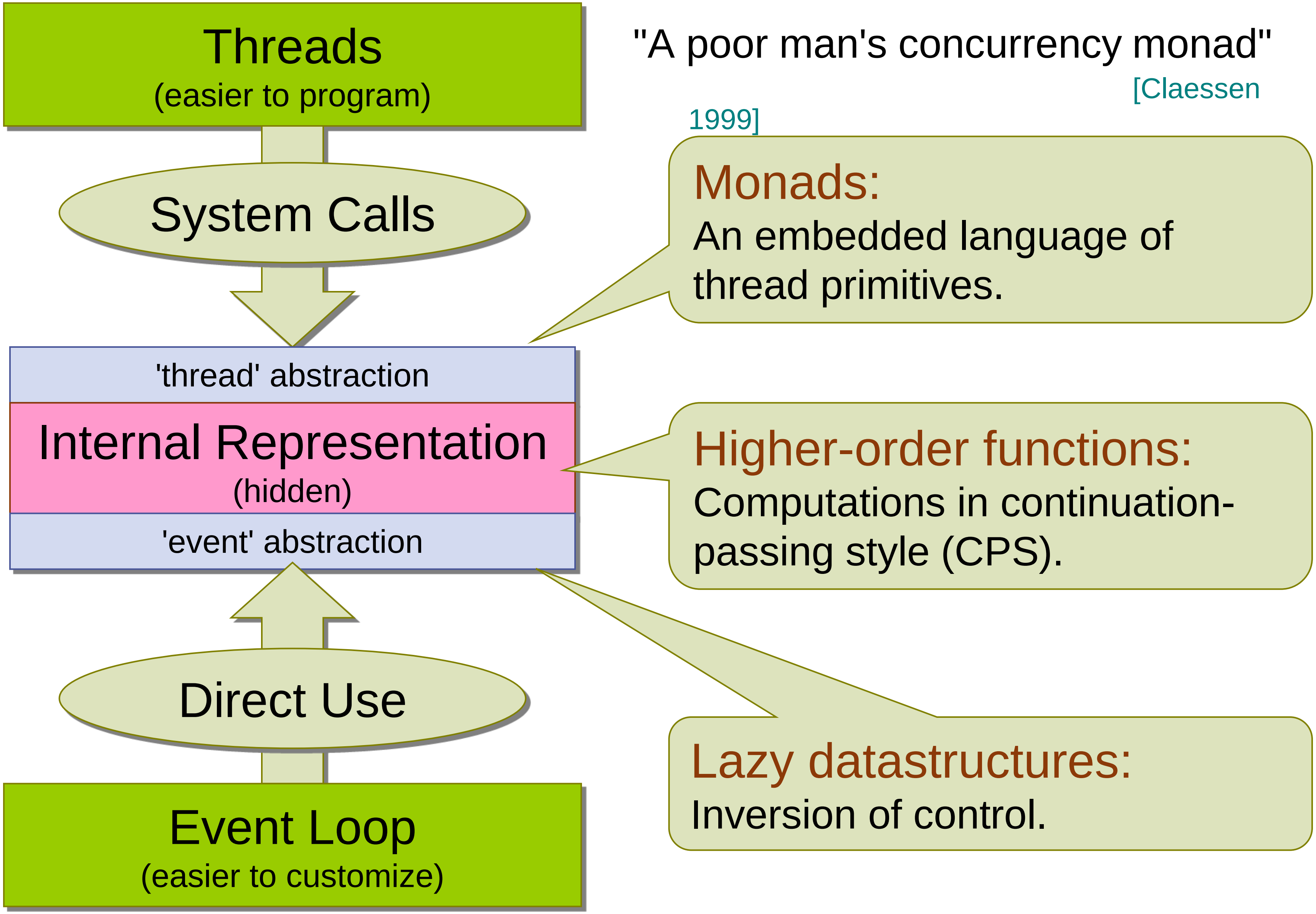
Outline

- Application-level cooperative concurrency in Haskell
  - Thread programming and Traces
  - Schedulers and Event processing
  - CPS translation and monads

- Examples
- Performance

- Future Directions & Conclusions

Implementing this Hybrid Model



Threads
(easier to program)

System Calls

'thread' abstraction

Internal Representation
(hidden)

'event' abstraction

Direct Use

Event Loop
(easier to customize)

"A poor man's concurrency monad" [Claessen 1999]

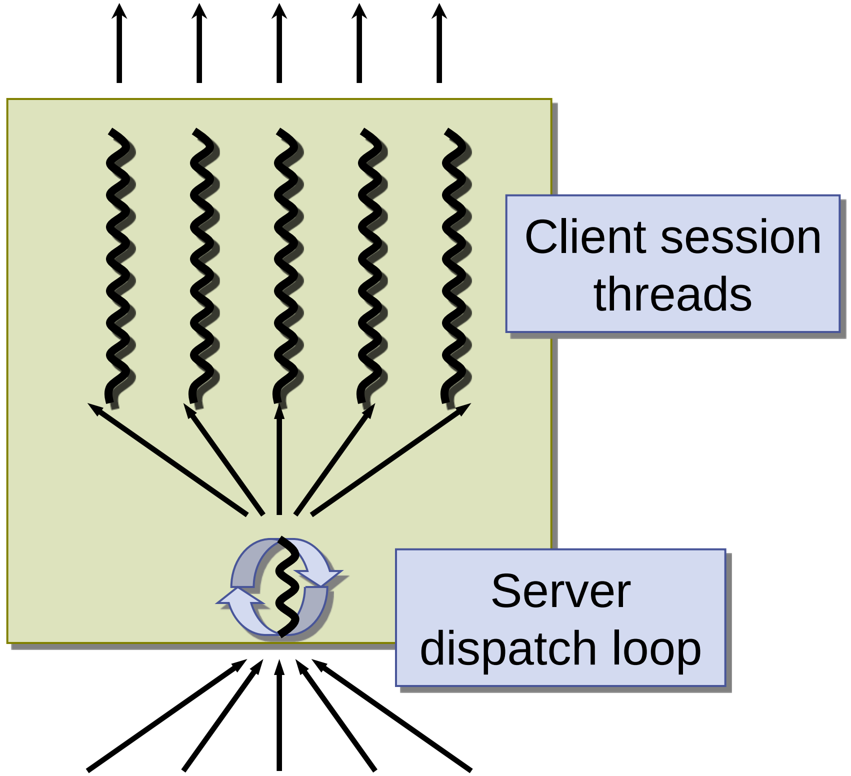
Monads:
An embedded language of thread primitives.

Higher-order functions:
Computations in continuation-passing style (CPS).

Lazy datastructures:
Inversion of control.

Example Server Code

```
server s = do {
  sock <- sock_accept s;
  sys_fork (session sock);
  server s;
}

session sock = do {
  n <- sys_nbio (read …);
  …<code>…
  sys_wait sock EPOLL_READ;
  …<code>…
  sys_nbio (write …);
  sys_ret;
}
```

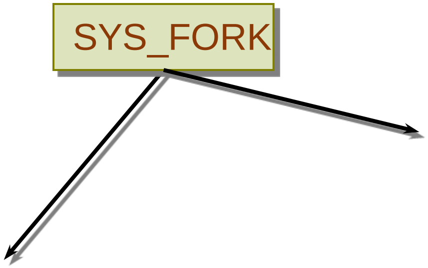


Client session threads

Server dispatch loop

Thread Code: Producing a Trace

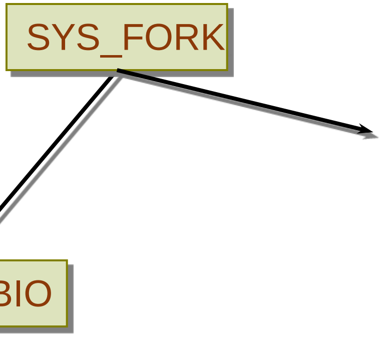
```
server s = do {
  sock <- sock_accept s;
  sys_fork (session sock);
  server s;
}

session sock = do {
  n <- sys_nbio (read …);
  …<code>…
  sys_wait sock EPOLL_READ;
  …<code>…
  sys_nbio (write …);
  sys_ret;
}
```



SYS_FORK

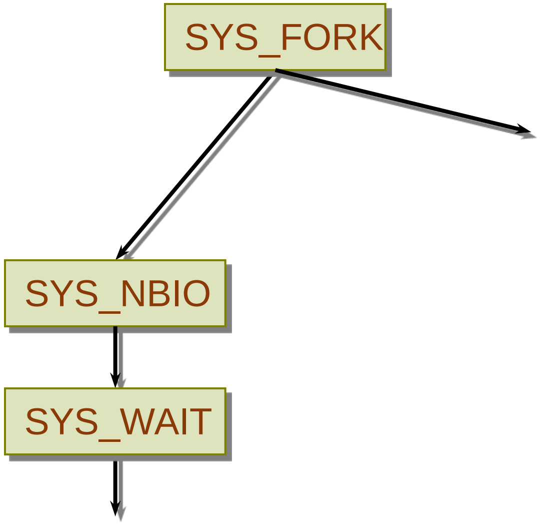
Thread Code: Producing a Trace

```
server s = do {
  sock <- sock_accept s;
  sys_fork (session sock);
  server s;
}

session sock = do {
  n <- sys_nbio (read …);
  …<code>…
  sys_wait sock EPOLL_READ;
  …<code>…
  sys_nbio (write …);
  sys_ret;
}
```



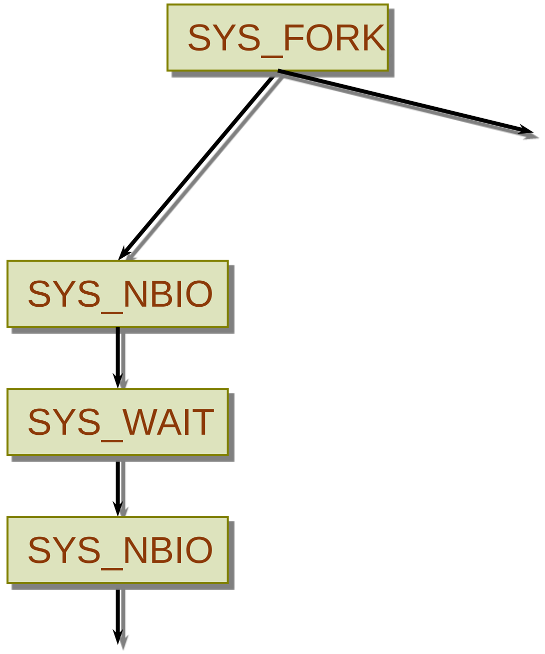
Thread Code: Producing a Trace

```
server s = do {
  sock <- sock_accept s;
  sys_fork (session sock);
  server s;
}

session sock = do {
  n <- sys_nbio (read …);
  …<code>…
  sys_wait sock EPOLL_READ;
  …<code>…
  sys_nbio (write …);
  sys_ret;
}
```



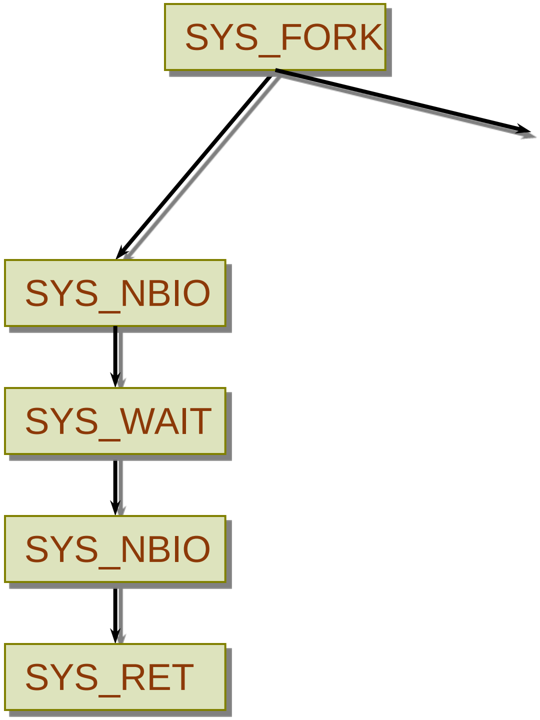
Thread Code: Producing a Trace

```
server s = do {
  sock <- sock_accept s;
  sys_fork (session sock);
  server s;
}

session sock = do {
  n <- sys_nbio (read …);
  …<code>…
  sys_wait sock EPOLL_READ;
  …<code>…
  sys_nbio (write …);
  sys_ret;
}
```



Thread Code: Producing a Trace

```
server s = do {
  sock <- sock_accept s;
  sys_fork (session sock);
  server s;
}

session sock = do {
  n <- sys_nbio (read …);
  …<code>…
  sys_wait sock EPOLL_READ;
  …<code>…
  sys_nbio (write …);
  sys_ret;
}
```



SYS_FORK

SYS_NBIO

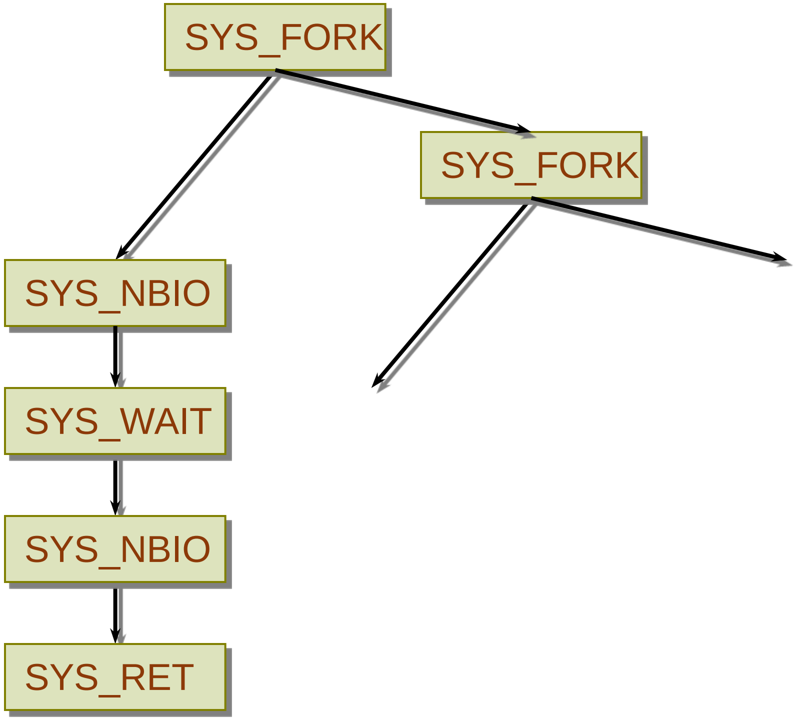
SYS_WAIT

SYS_NBIO

SYS_RET

Thread Code: Producing a Trace

```
server s = do {
  sock <- sock_accept s;
  sys_fork (session sock);
  server s;
}

session sock = do {
  n <- sys_nbio (read …);
  …<code>…
  sys_wait sock EPOLL_READ;
  …<code>…
  sys_nbio (write …);
  sys_ret;
}
```



SYS_FORK

SYS_FORK

SYS_NBIO

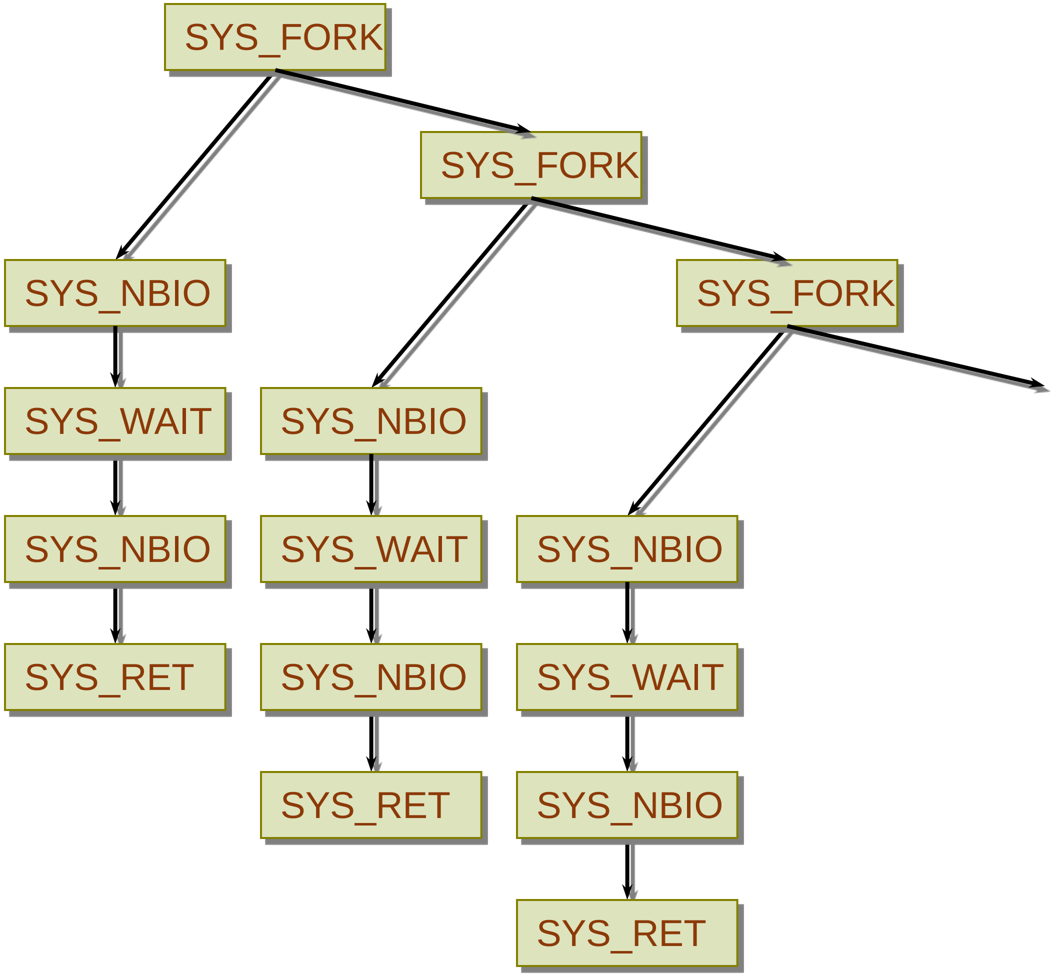
SYS_WAIT

SYS_NBIO

SYS_RET

Thread Code: Producing a Trace

```
server s = do {
  sock <- sock_accept s;
  sys_fork (session sock);
  server s;
}

session sock = do {
  n <- sys_nbio (read …);
  …<code>…
  sys_wait sock EPOLL_READ;
  …<code>…
  sys_nbio (write …);
  sys_ret;
}
```



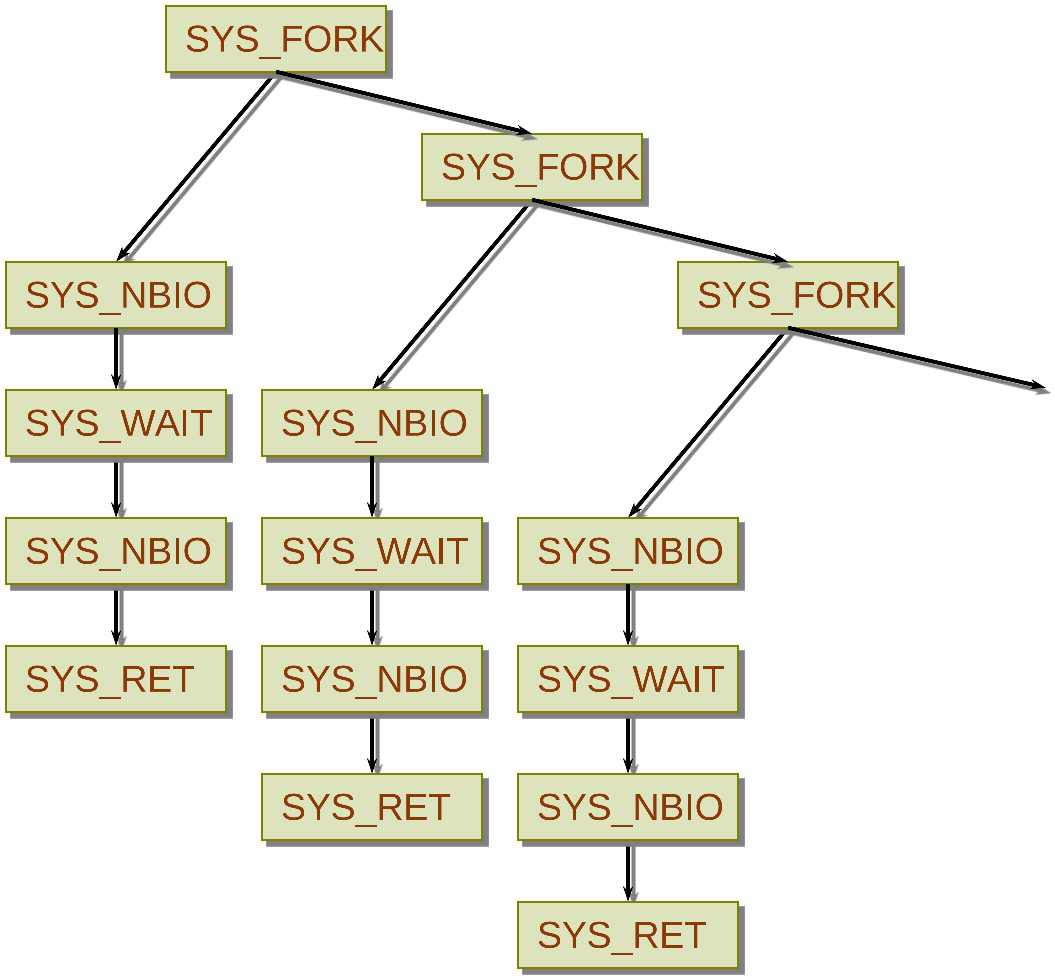
Trace Datatype

- Reflect the trace of system calls as a datastructure:

```
-- A lazy tree of system calls/events
data Trace =
  SYS_RET
| SYS_FORK Trace Trace
| SYS_YIELD Trace
| SYS_NBIO (IO Trace)
| SYS_WAIT Socket EPOLL_Event Trace
| …
```

- Key use of Haskell's laziness:
  - Trace datatype represents potentially infinite trees
  - Nodes of the tree are computed only when needed
- Strong, expressive types:
  - IO Trace is the type of computations that do some I/O and then produce a trace.
- Nodes provide the event abstraction

Event-driven Scheduler Code

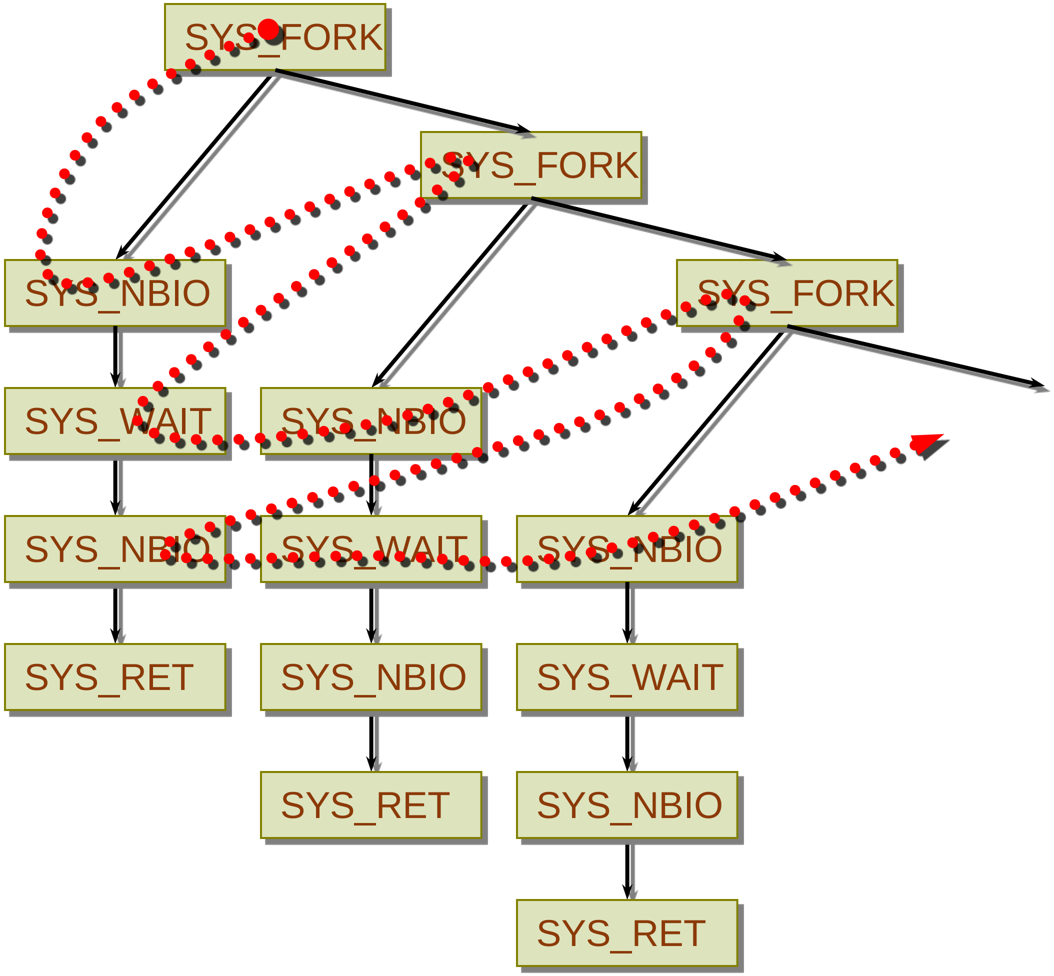


Scheduling the events

=

traversing the tree!

Event-driven Scheduler Code



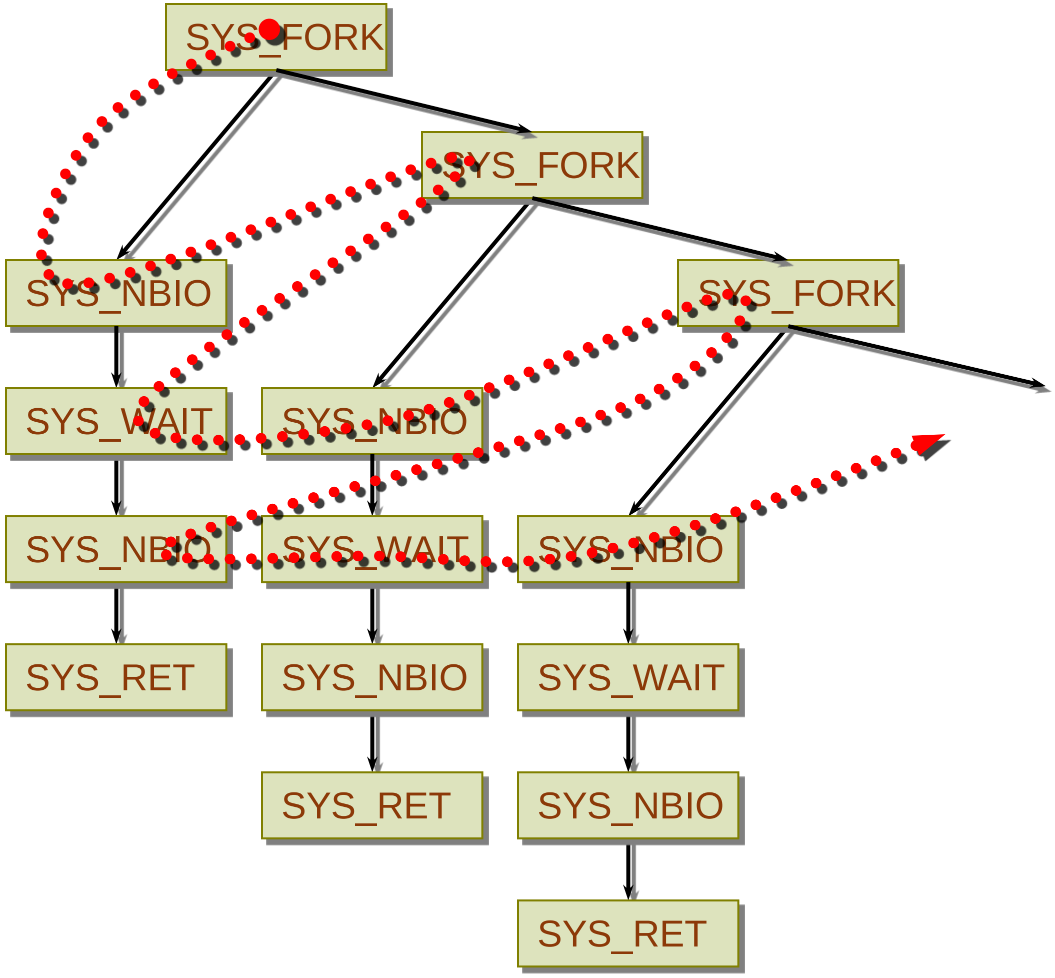
Scheduling the events

=

traversing the tree!

Example: Round robin is just breadth-first traversal.

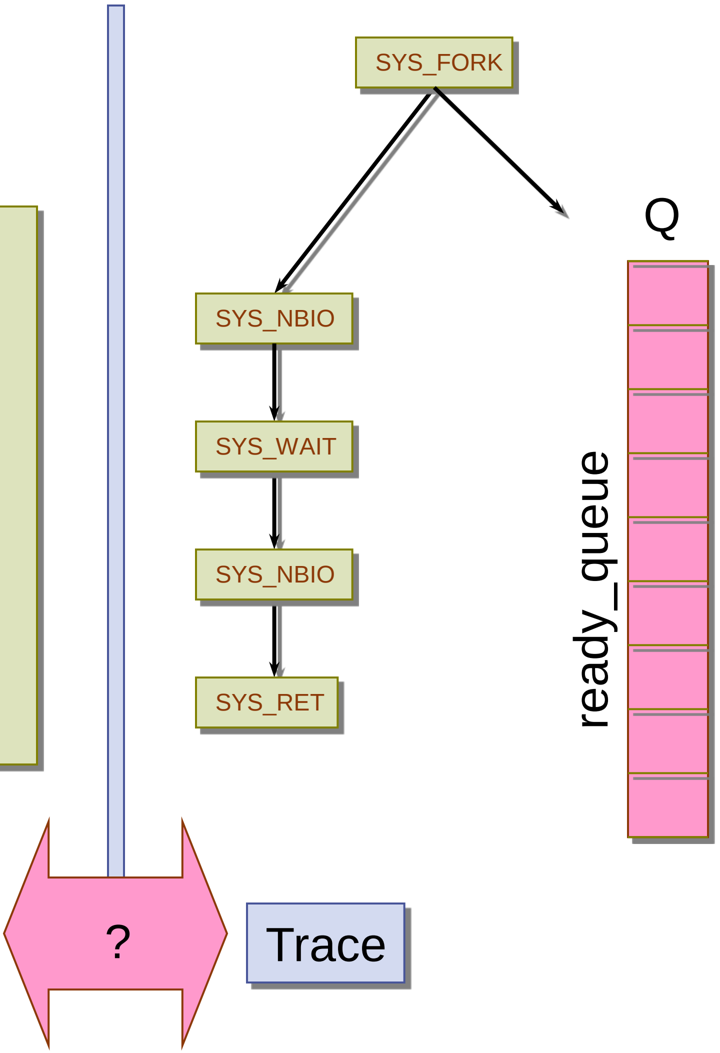
Event-driven Scheduler Code



```
worker_main Q = do {
  trace <- fetch_thread Q;
  execute trace Q;
  worker_main Q;
}

execute trace Q =
  case trace of
    SYS_RET -> return()

    SYS_FORK t1 t2 ->
      do { add_thread t1 Q;
           add_thread t2 Q;
         }

    SYS_NBIO cmd ->
      do { t <- cmd;
           add_thread t Q
         }

    …
```

Threads to Events?

```
server s = do {
  sock <- sock_accept s;
  sys_fork (session sock);
  server s;
}

session sock = do {
  n <- sys_nbio (read …);
  …<code>…
  sys_wait sock EPOLL_READ;
  …<code>…
  sys_nbio (write …);
  sys_ret;
}
```



```
worker_main Q = do {
  trace <- fetch_thread Q;
  execute trace Q;
  worker_main Q;
}

execute trace Q =
  case trace of
    SYS_RET -> return()

    SYS_FORK t1 t2 ->
      do { add_thread t1 Q;
           add_thread t2 Q;
         }

    SYS_NBIO cmd ->
      do { t <- cmd;
           add_thread t Q
         }

    …
```

Threaded code

Trace

Event Loop

Monads in Haskell

- A monad is a datatype that describes programs written in a domain-specific "embedded" sublanguage:
  - Each monad provides some primitive commands
  - Users can create "embedded programs" in the monad by composition

```
-- Monad interface for a parameterized datatype M (excerpt)
class Monad M where
  return :: a -> M a                    -- return a value
  (>>=)  :: M a -> (a -> M b) -> M b      -- sequential composition
```

- Example: IO monad

```
-- Example primitive IO operations:
hGetChar :: Handle -> IO Char
hPutChar :: Handle -> Char -> IO ()
```

```
double :: Handle -> IO ()
double h = do {
  x <- hGetChar h;
  hPutChar h x;
  hPutChar h x;
}
```

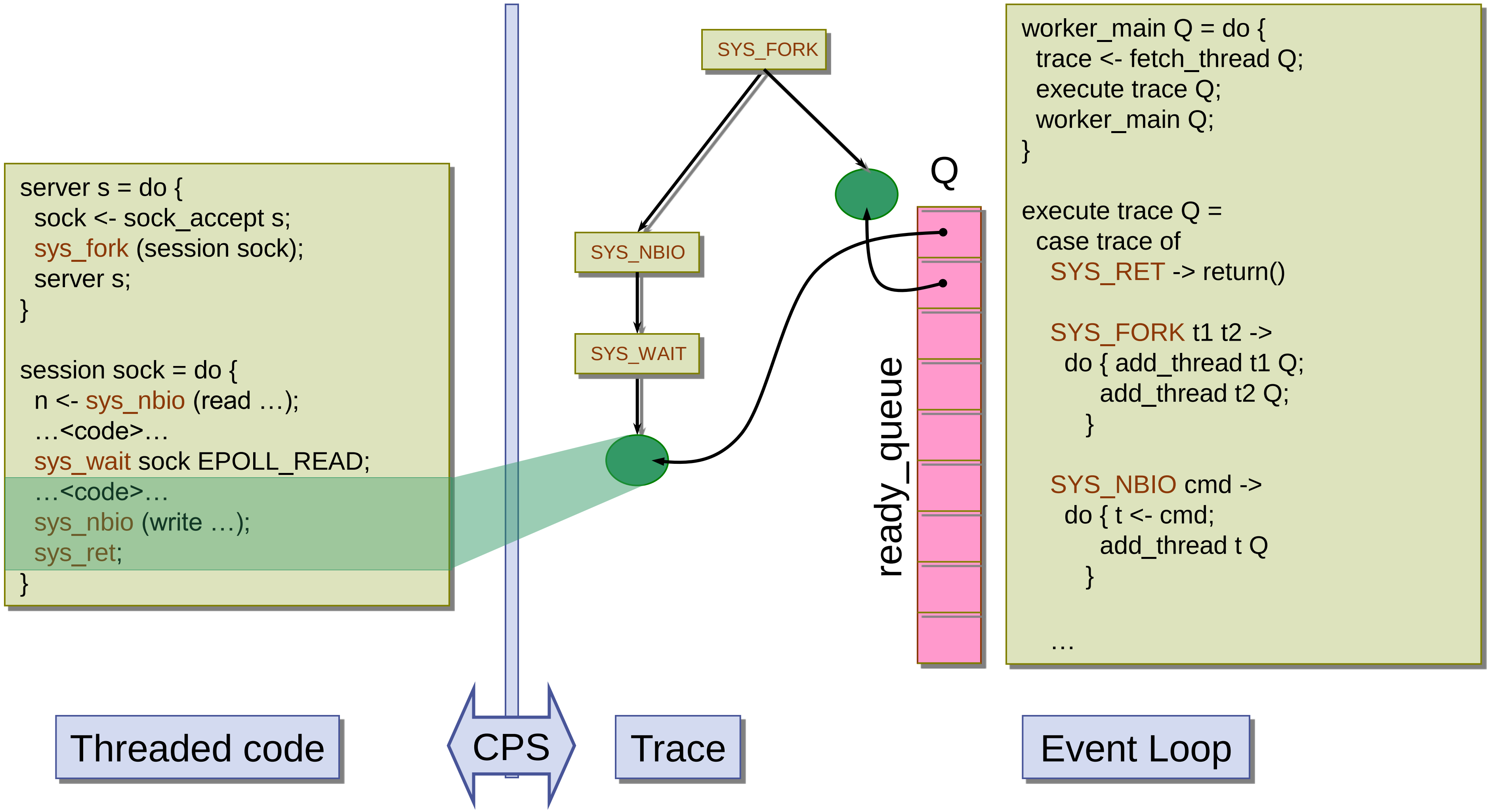
```
double :: Handle -> IO ()
double h =
  hGetChar h   >>= (\x ->
  hPutChar h x >>= (\_ ->
  hPutChar h x >>= (\_ ->
  return ()
)))
```

CPS Conversion, Monadically

- A continuation is just a function that produces a Trace
- The datatype of CPS computations makes the continuation explicit
- All of this is hidden in a library

```
-- CPS Monad
newtype CPS a = CPS ((a -> Trace) -> Trace)
class Monad CPS where
  return x      = CPS (\c -> c x)
  (CPS g) >>= f = CPS (\c -> g (\x -> let CPS h = f x in h c))

-- Complete a trace by putting SYS_RET at the leaves
build_trace :: CPS a -> Trace
build_trace (CPS f) = f (\c -> SYS_RET)

-- CPS primitive commands:
sys_ret     = CPS (\c -> SYS_RET)
sys_fork f  = CPS (\c -> SYS_FORK (build_trace f) (c ()))
sys_yield   = CPS (\c -> SYS_YIELD (c ()))
sys_nbio f  = CPS (\c -> SYS_NBIO (do x <- f; return (c x)))
…
```

Inversion of Control



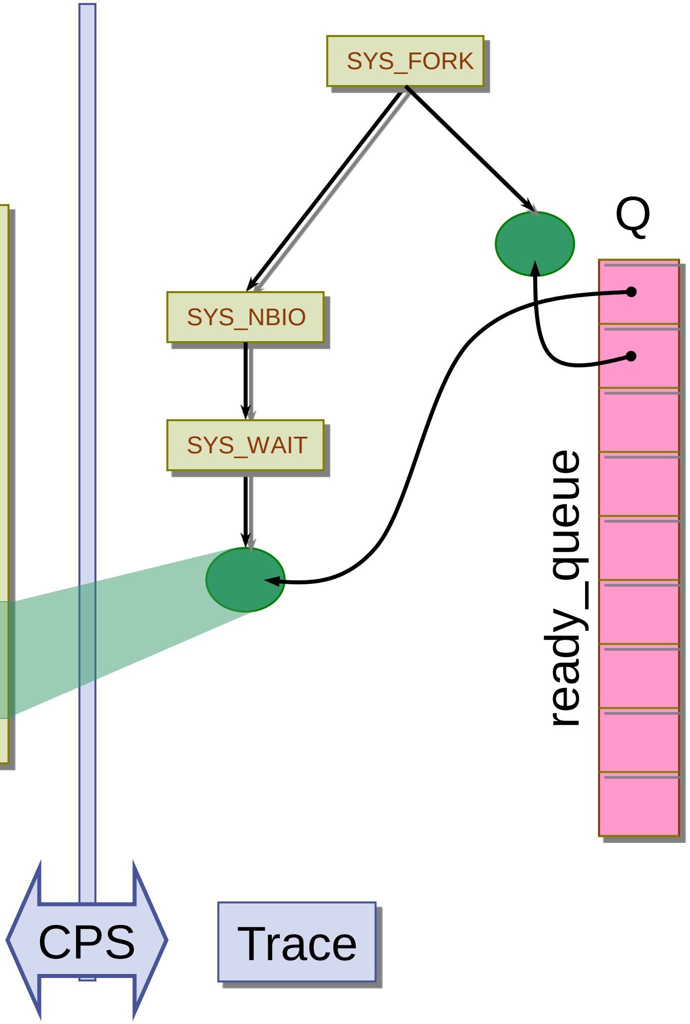
```
server s = do {
  sock <- sock_accept s;
  sys_fork (session sock);
  server s;
}

session sock = do {
  n <- sys_nbio (read …);
  …<code>…
  sys_wait sock EPOLL_READ;
  …<code>…
  sys_nbio (write …);
  sys_ret;
}
```

```
worker_main Q = do {
  trace <- fetch_thread Q;
  execute trace Q;
  worker_main Q;
}

execute trace Q =
  case trace of
    SYS_RET -> return()

    SYS_FORK t1 t2 ->
      do { add_thread t1 Q;
           add_thread t2 Q;
      }

    SYS_NBIO cmd ->
      do { t <- cmd;
           add_thread t Q
      }

    …
```

Inversion of Control

```
server s = do {
  sock <- sock_accept s;
  sys_fork (session sock);
  server s;
}

session sock = do {
  n <- sys_nbio (read …);
  …<code>…
  sys_wait sock EPOLL_READ;
  …<code>…
  sys_nbio (write …);
  sys_ret;
}
```

Threaded code

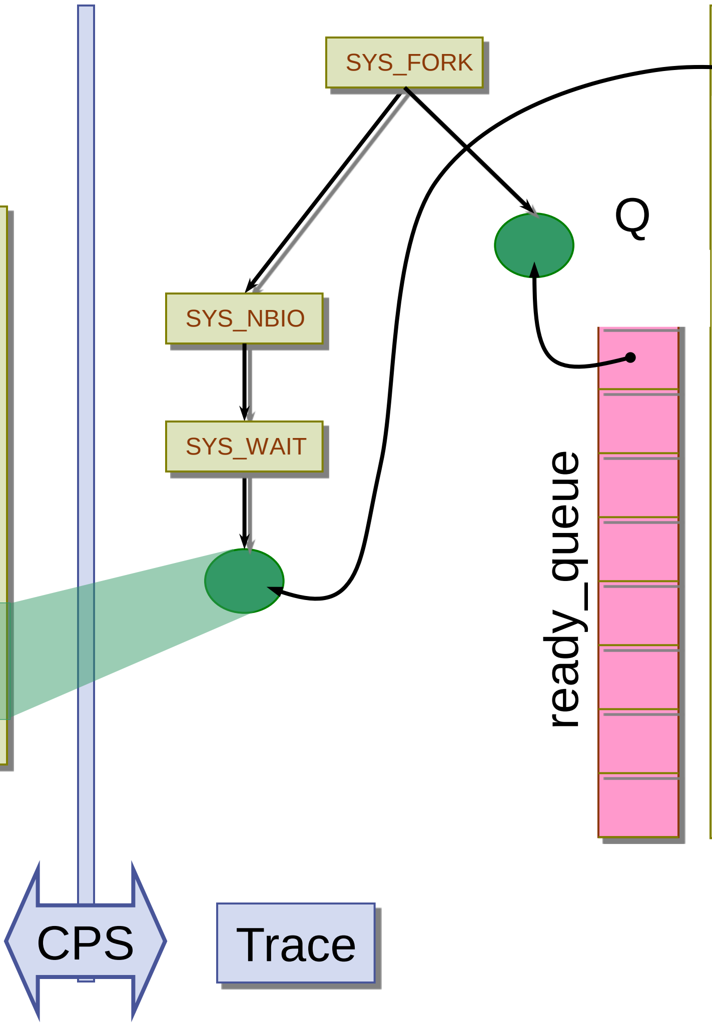


```
worker_main Q = do {
  trace <- fetch_thread Q;
  execute trace Q;
  worker_main Q;
}

execute trace Q =
  case trace of
    SYS_RET -> return()

    SYS_FORK t1 t2 ->
      do { add_thread t1 Q;
           add_thread t2 Q;
         }

    SYS_NBIO cmd ->
      do { t <- cmd;
           add_thread t Q
         }

    …
```

Event Loop

Inversion of Control

```
server s = do {
  sock <- sock_accept s;
  sys_fork (session sock);
  server s;
}

session sock = do {
  n <- sys_nbio (read …);
  …<code>…
  sys_wait sock EPOLL_READ;
  …<code>…
  sys_nbio (write …);
  sys_ret;
}
```

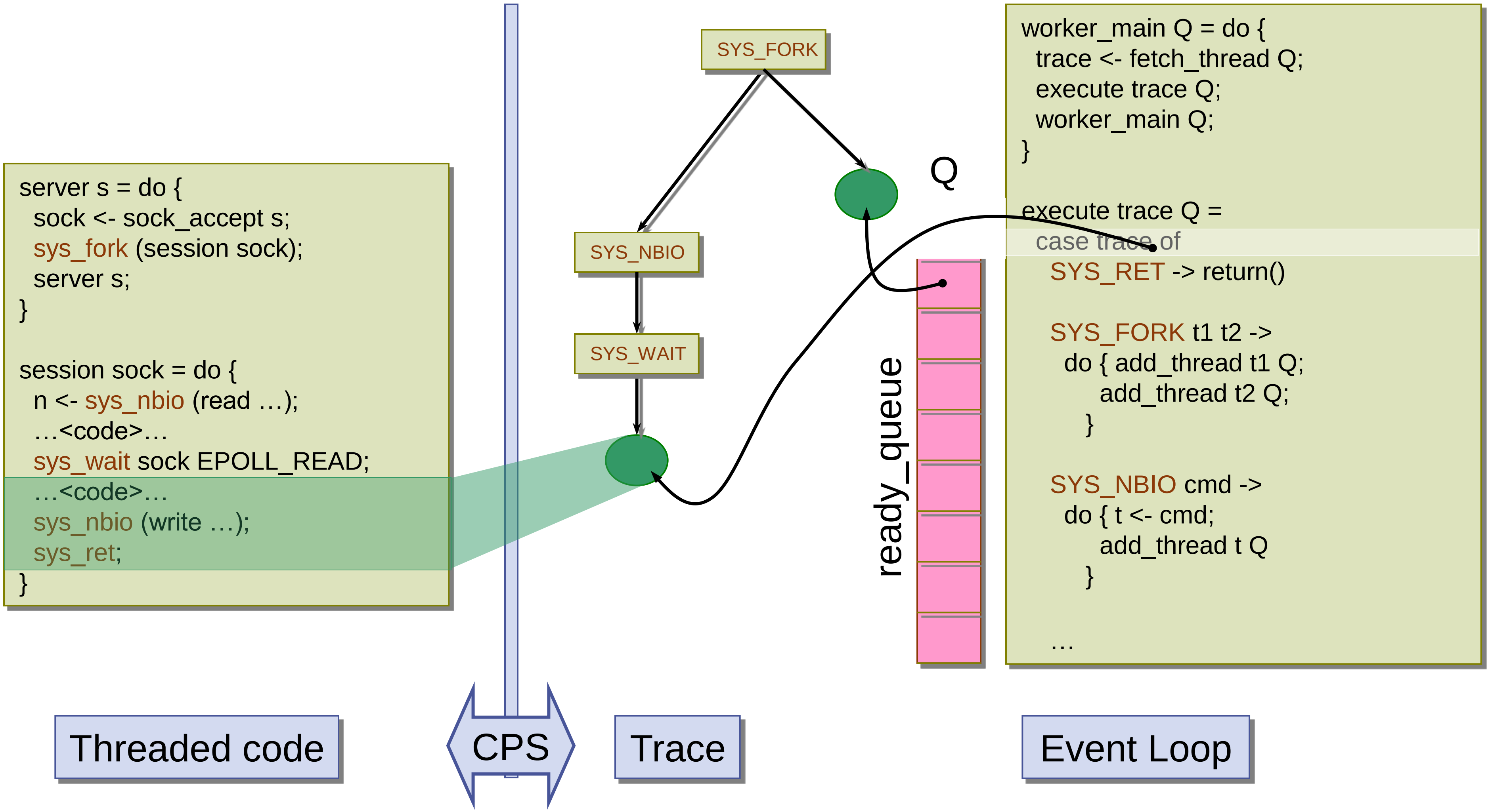


```
worker_main Q = do {
  trace <- fetch_thread Q;
  execute trace Q;
  worker_main Q;
}

execute trace Q =
  case trace of
    SYS_RET -> return()

    SYS_FORK t1 t2 ->
      do { add_thread t1 Q;
           add_thread t2 Q;
         }

    SYS_NBIO cmd ->
      do { t <- cmd;
           add_thread t Q
         }

    …
```

Threaded code

Event Loop

Inversion of Control



```
server s = do {
  sock <- sock_accept s;
  sys_fork (session sock);
  server s;
}

session sock = do {
  n <- sys_nbio (read …);
  …<code>…
  sys_wait sock EPOLL_READ;
  …<code>…
  sys_nbio (write …);
  sys_ret;
}
```

```
worker_main Q = do {
  trace <- fetch_thread Q;
  execute trace Q;
  worker_main Q;
}

execute trace Q =
  case trace of
    SYS_RET -> return()

    SYS_FORK t1 t2 ->
     do { add_thread t1 Q;
          add_thread t2 Q;
        }

    SYS_NBIO cmd ->
     do { t <- cmd;
          add_thread t Q
        }

    …
```

Threaded code

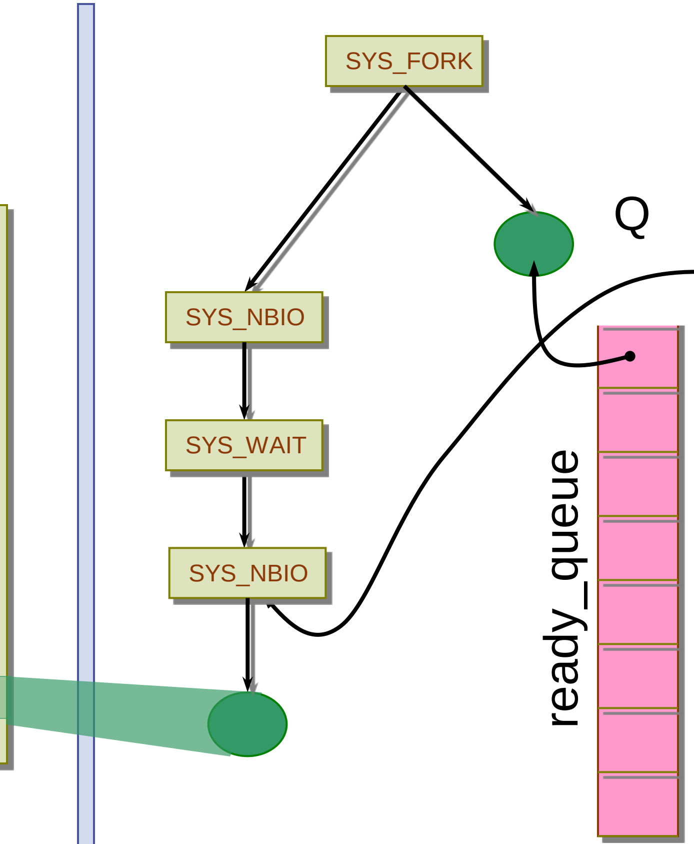
CPS

Trace

Event Loop

Inversion of Control

```
server s = do {
  sock <- sock_accept s;
  sys_fork (session sock);
  server s;
}

session sock = do {
  n <- sys_nbio (read …);
  …<code>…
  sys_wait sock EPOLL_READ;
  …<code>…
  sys_nbio (write …);
  sys_ret;
}
```



```
worker_main Q = do {
  trace <- fetch_thread Q;
  execute trace Q;
  worker_main Q;
}

execute trace Q =
  case trace of
    SYS_RET -> return()

    SYS_FORK t1 t2 ->
      do { add_thread t1 Q;
           add_thread t2 Q;
      }

    SYS_NBIO cmd ->
      do { t <- cmd;
           add_thread t Q
      }

    …
```

Threaded code

CPS

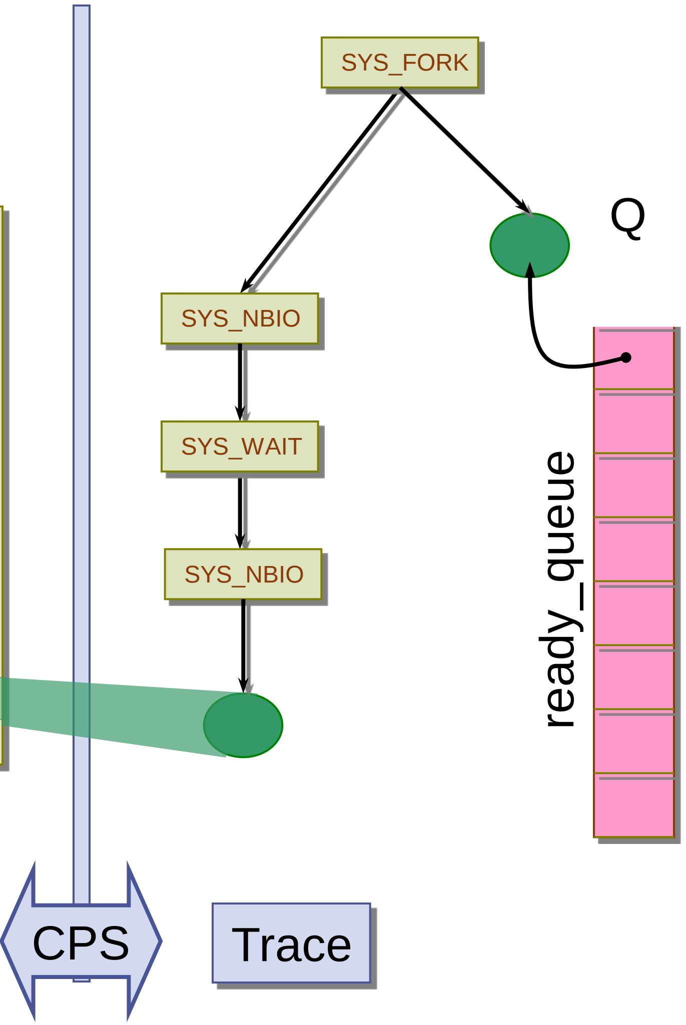
Trace

Event Loop

Inversion of Control

```
server s = do {
  sock <- sock_accept s;
  sys_fork (session sock);
  server s;
}

session sock = do {
  n <- sys_nbio (read …);
  …<code>…
  sys_wait sock EPOLL_READ;
  …<code>…
  sys_nbio (write …);
  sys_ret;
}
```

Threaded code



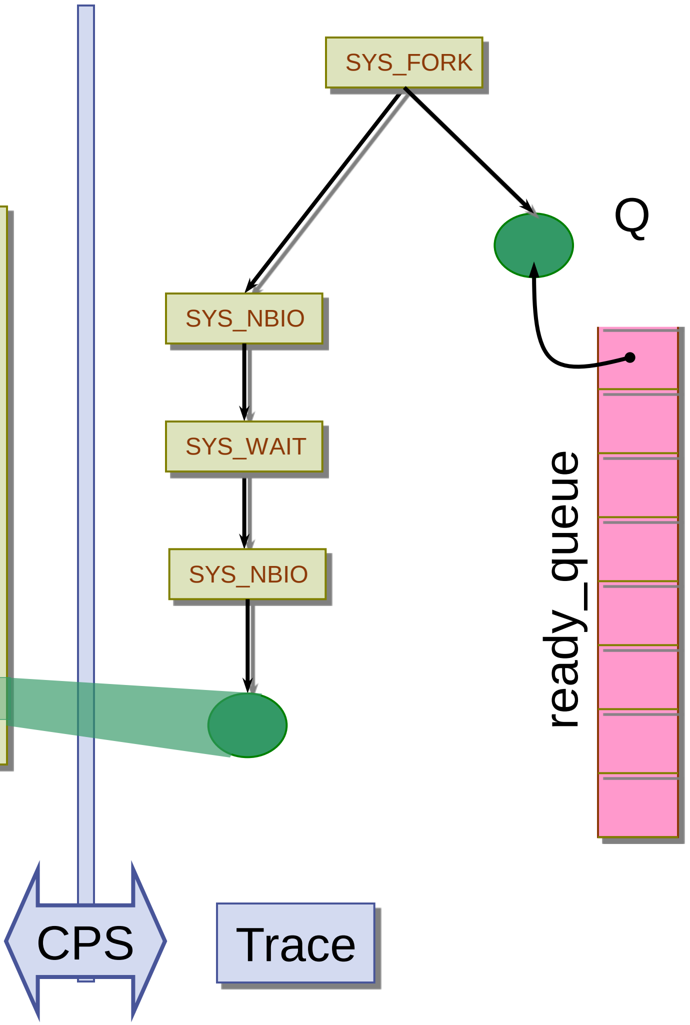
```
worker_main Q = do {
  trace <- fetch_thread Q;
  execute trace Q;
  worker_main Q;
}

execute trace Q =
  case trace of
    SYS_RET -> return()

    SYS_FORK t1 t2 ->
      do { add_thread t1 Q;
           add_thread t2 Q;
         }

    SYS_NBIO cmd ->
      do { t <- cmd;
           add_thread t Q
         }

    …
```

Event Loop

Inversion of Control

```
server s = do {
  sock <- sock_accept s;
  sys_fork (session sock);
  server s;
}

session sock = do {
  n <- sys_nbio (read …);
  …<code>…
  sys_wait sock EPOLL_READ;
  …<code>…
  sys_nbio (write …);
  sys_ret;
}
```

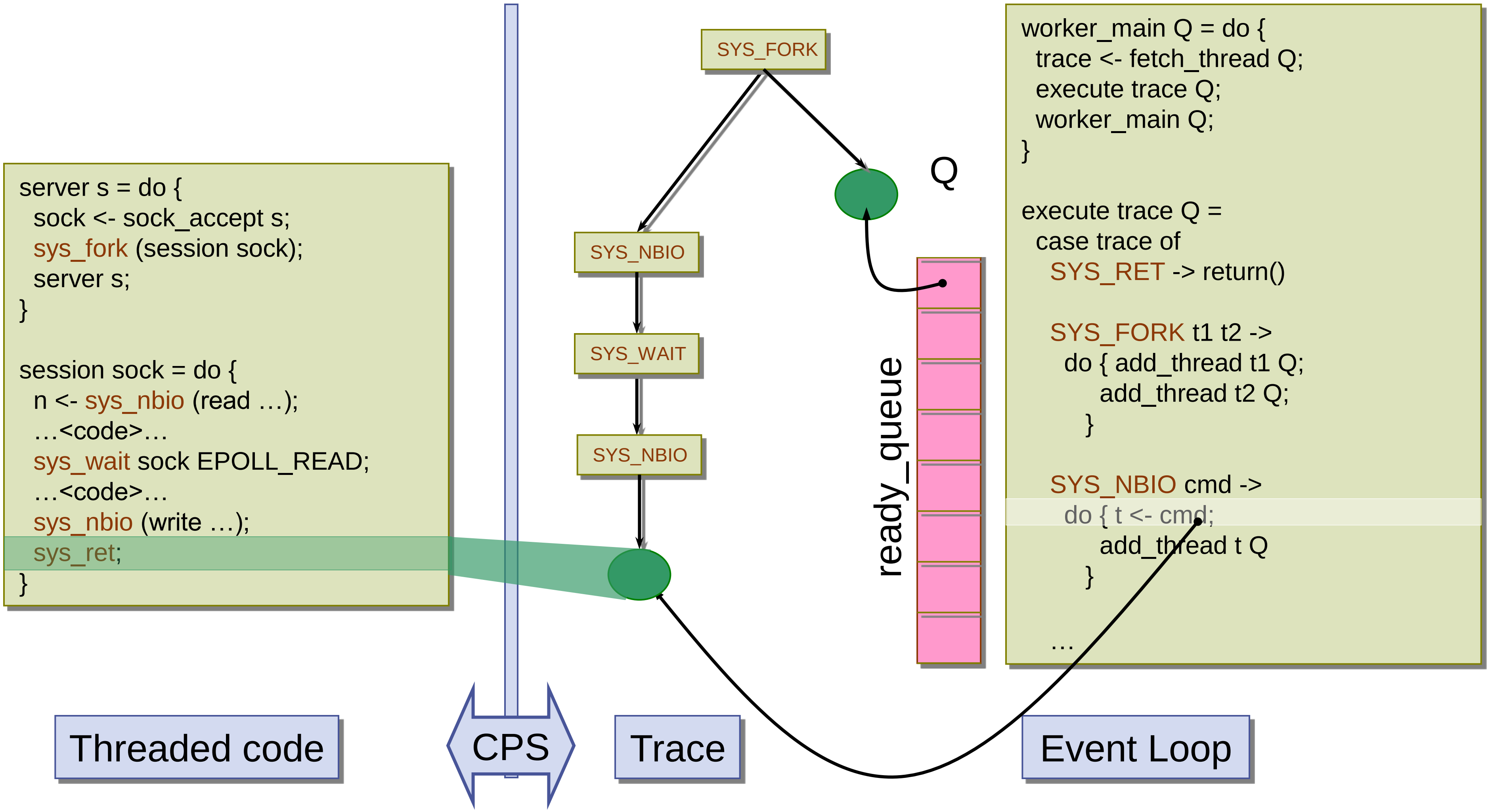
Threaded code



```
worker_main Q = do {
  trace <- fetch_thread Q;
  execute trace Q;
  worker_main Q;
}

execute trace Q =
  case trace of
    SYS_RET -> return()

    SYS_FORK t1 t2 ->
      do { add_thread t1 Q;
           add_thread t2 Q;
         }

    SYS_NBIO cmd ->
      do { t <- cmd;
           add_thread t Q
         }

    …
```

Event Loop

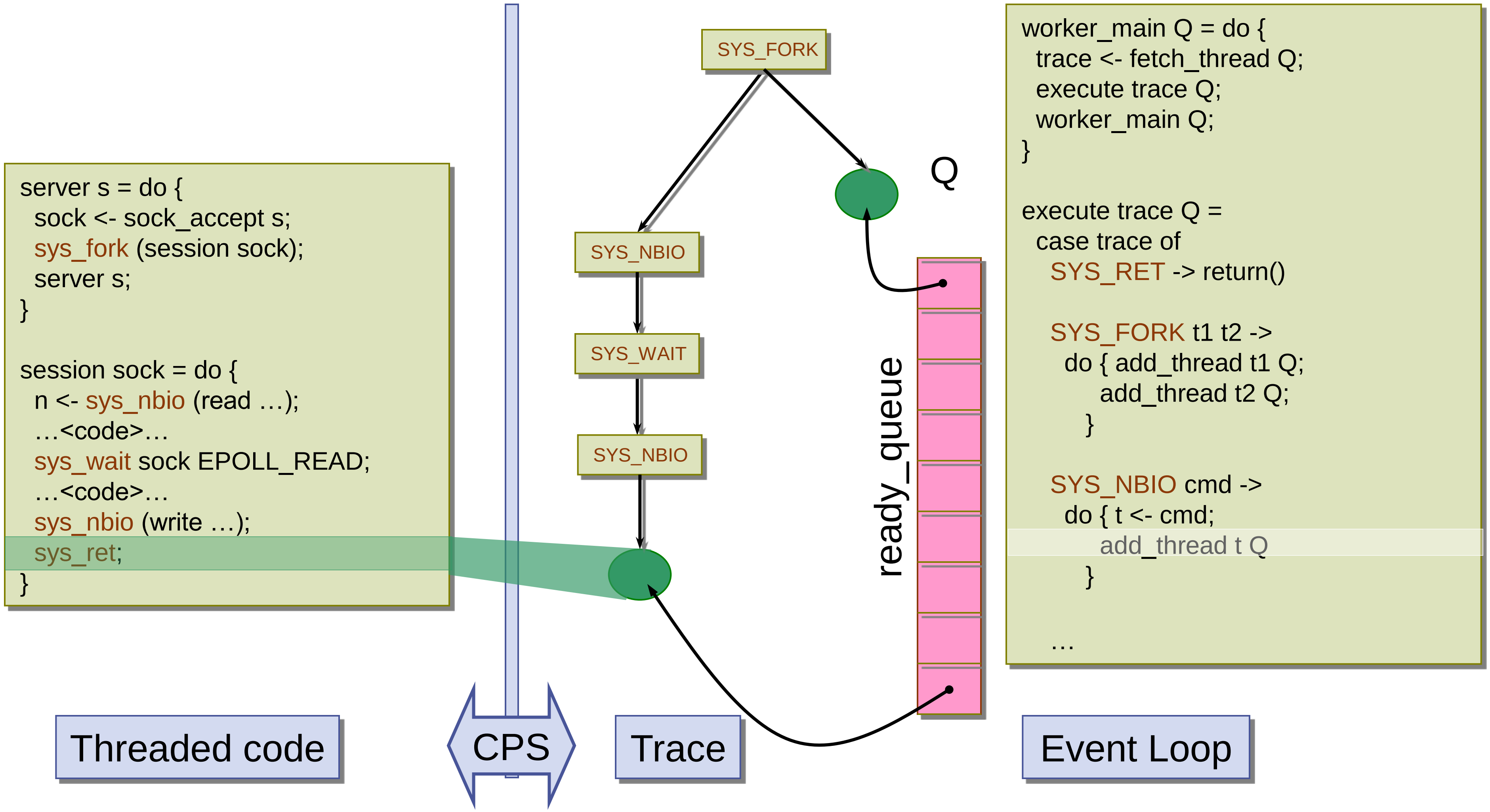
Inversion of Control



```
server s = do {
  sock <- sock_accept s;
  sys_fork (session sock);
  server s;
}

session sock = do {
  n <- sys_nbio (read …);
  …<code>…
  sys_wait sock EPOLL_READ;
  …<code>…
  sys_nbio (write …);
  sys_ret;
}
```

```
worker_main Q = do {
  trace <- fetch_thread Q;
  execute trace Q;
  worker_main Q;
}

execute trace Q =
  case trace of
    SYS_RET -> return()

    SYS_FORK t1 t2 ->
      do { add_thread t1 Q;
           add_thread t2 Q;
         }

    SYS_NBIO cmd ->
      do { t <- cmd;
           add_thread t Q
         }

    …
```

Inversion of Control



```
server s = do {
  sock <- sock_accept s;
  sys_fork (session sock);
  server s;
}

session sock = do {
  n <- sys_nbio (read …);
  …<code>…
  sys_wait sock EPOLL_READ;
  …<code>…
  sys_nbio (write …);
  sys_ret;
}
```

```
worker_main Q = do {
  trace <- fetch_thread Q;
  execute trace Q;
  worker_main Q;
}

execute trace Q =
  case trace of
    SYS_RET -> return()

    SYS_FORK t1 t2 ->
      do { add_thread t1 Q;
           add_thread t2 Q;
         }

    SYS_NBIO cmd ->
      do { t <- cmd;
           add_thread t Q
         }

    …
```

Outline

- Application-level cooperative concurrency in Haskell
  - Thread programming and Traces
  - Schedulers and Event processing
  - CPS translation and monads
- Examples
- Performance
- Future Directions & Conclusions

Example Threaded Code

```
-- Sends a file over a socket
send_file sock filename =
  do {  fd <- open_file filename;
        buf <- alloc_aligned_memory buffer_size;
        sys_catch (
          copy_data fd sock buf 0;
        ) \exception -> do {
             file_close fd;
             sys_throw exception;
           } -- so the caller can catch it again
        file_close fd;
    }

-- Copy data from file descriptor to socket until EOF
copy_data fd sock buf offset =
  do { num_read <- file_read fd offset buf;
       if (num_read == 0) then return () else
         do { sock_send sock buf num_read;
              copy_data fd sock buf (offset + num_read);
            }
    }
```

Nested function call

Exception handler

System call

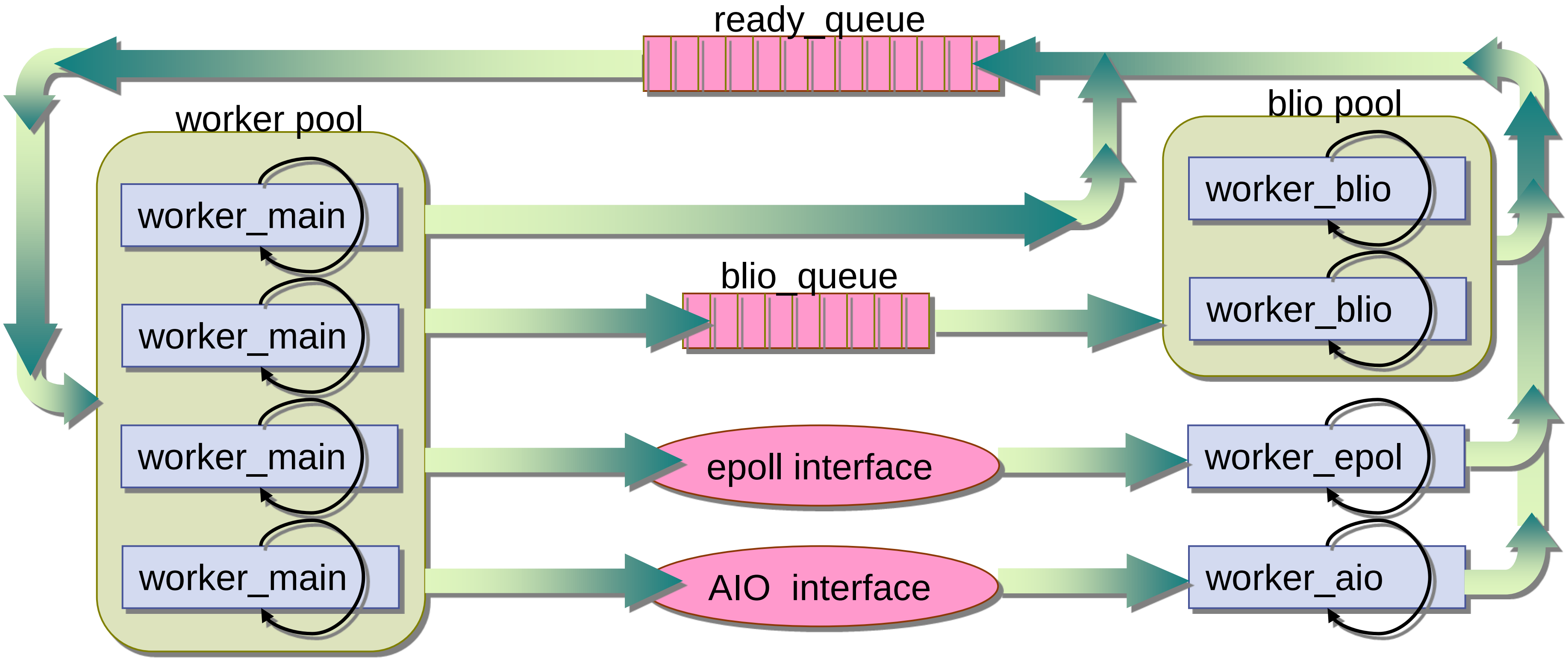
Conditional branch

Function call to I/O lib

Recursion

Example Event System Architecture

- Each event loop ("worker") runs in an OS thread - synchronized using queues
- Example configuration:
  - Worker pool for CPU-intensive computations
  - Worker pool for blocking IO operations
  - Dedicated worker threads for monitoring epoll & AIO events



Scheduler Implementation

- Epoll: high-performance "select()" on Linux
- AIO: asynchronous file I/O
- Wrap underlying C functions using Haskell's FFI

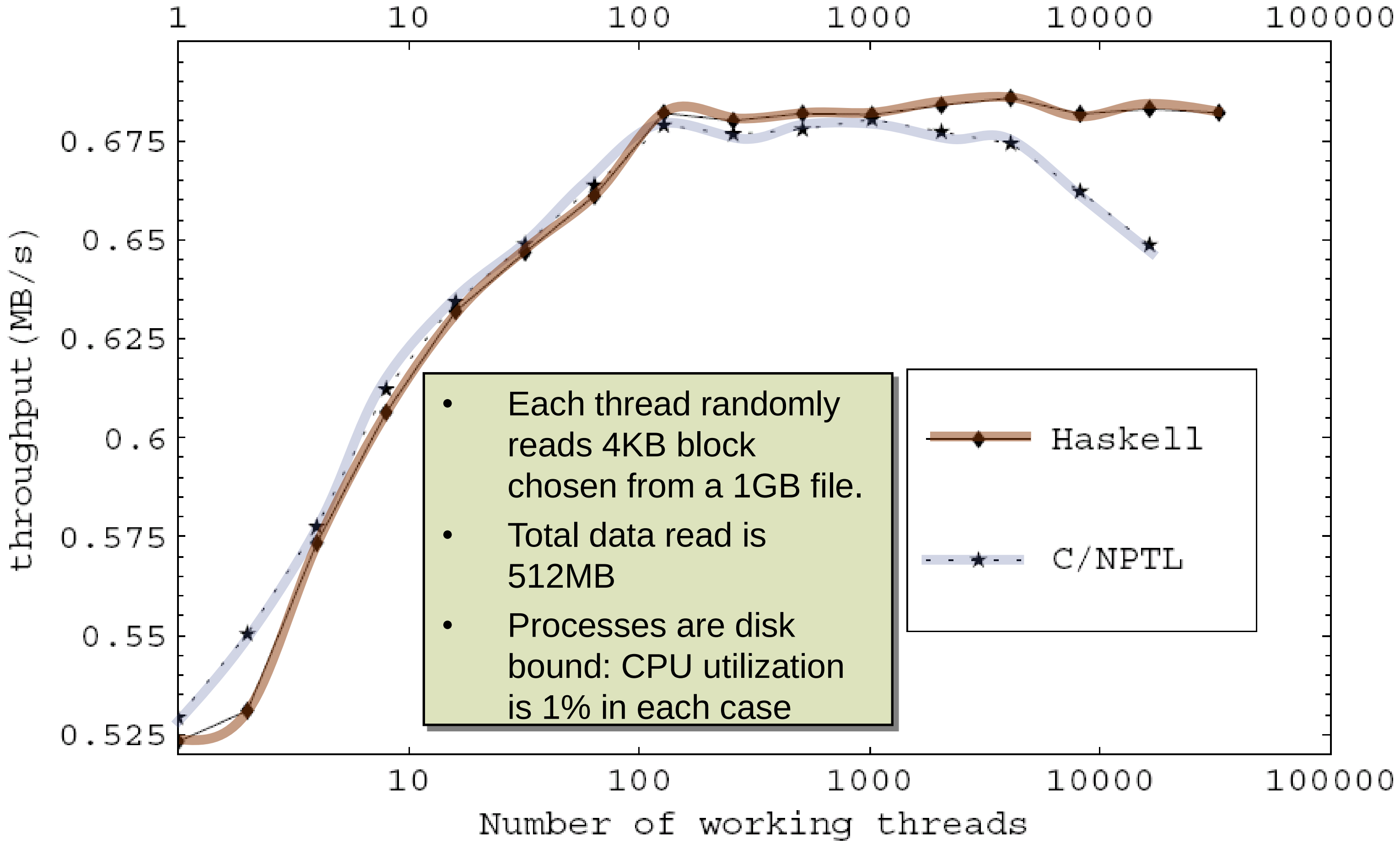
```
-- epoll event loop
worker_epoll sched = do {
  -- wait for some epoll events
  results <- epoll_wait;
  -- write each thread in results to the ready_queue
  mapM (add_thread (ready_queue sched)) results;
  worker_epll sched;
}
```

Performance?

- Implementation
  - Concurrent Haskell GHC 6.5 on Linux 2.6.15
- Haskell is a pure, lazy, functional language
  - Significantly slower than C
  - Uses garbage collection
- CPS threads are cheap and lightweight:
  - Everything is heap allocated (no thread-local stack)
  - Actual space usage depends on needed thread-local state
  - Memory footprint for a minimal thread is just 48 bytes
  - GC accounts for < 0.2% of the runtime in our experiments

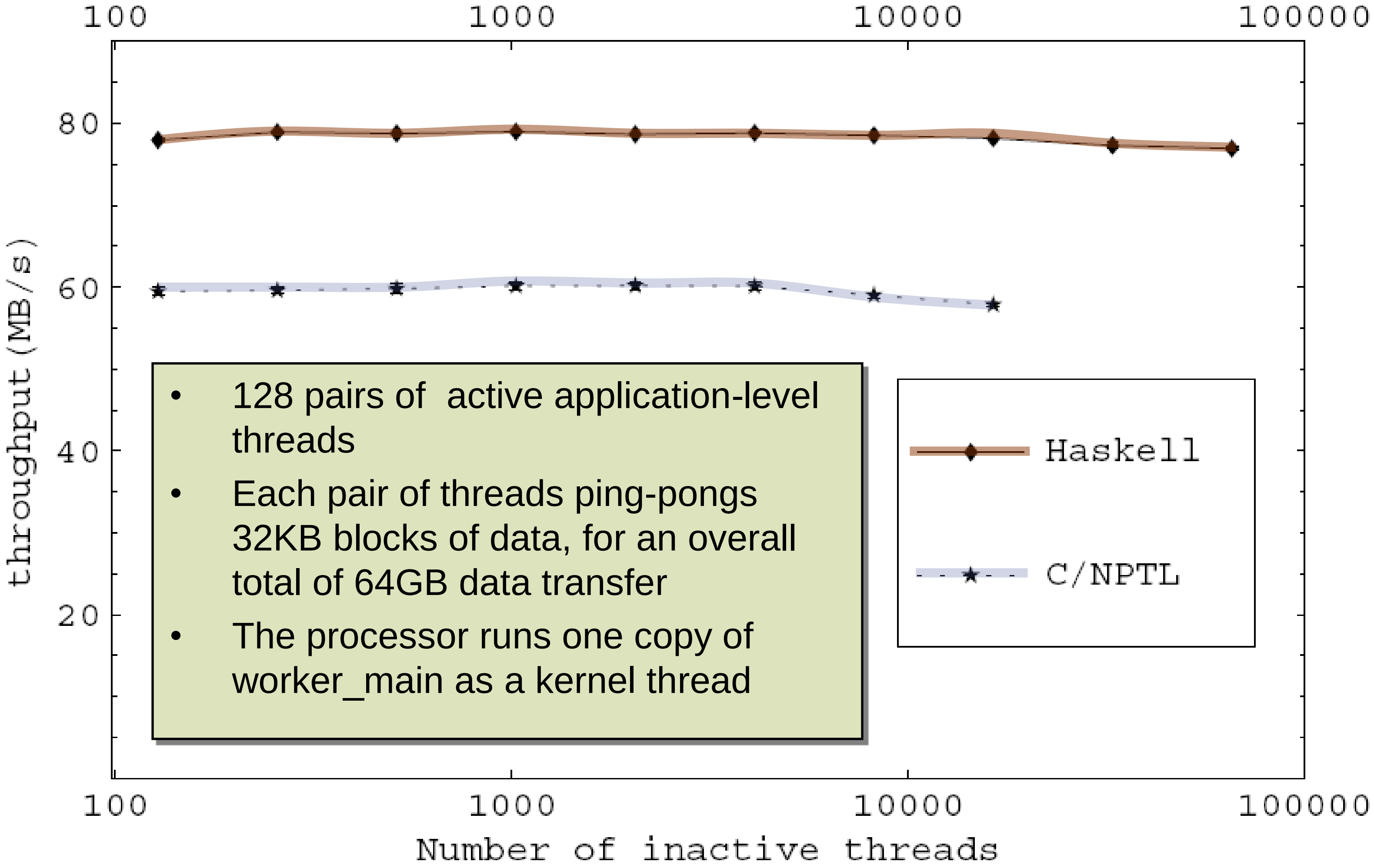
- Events are efficient:
  - Constant number of OS threads means less overhead
  - Event-driven scheduling makes it easy to use high-performance epoll and AIO interfaces

Disk head scheduling performance



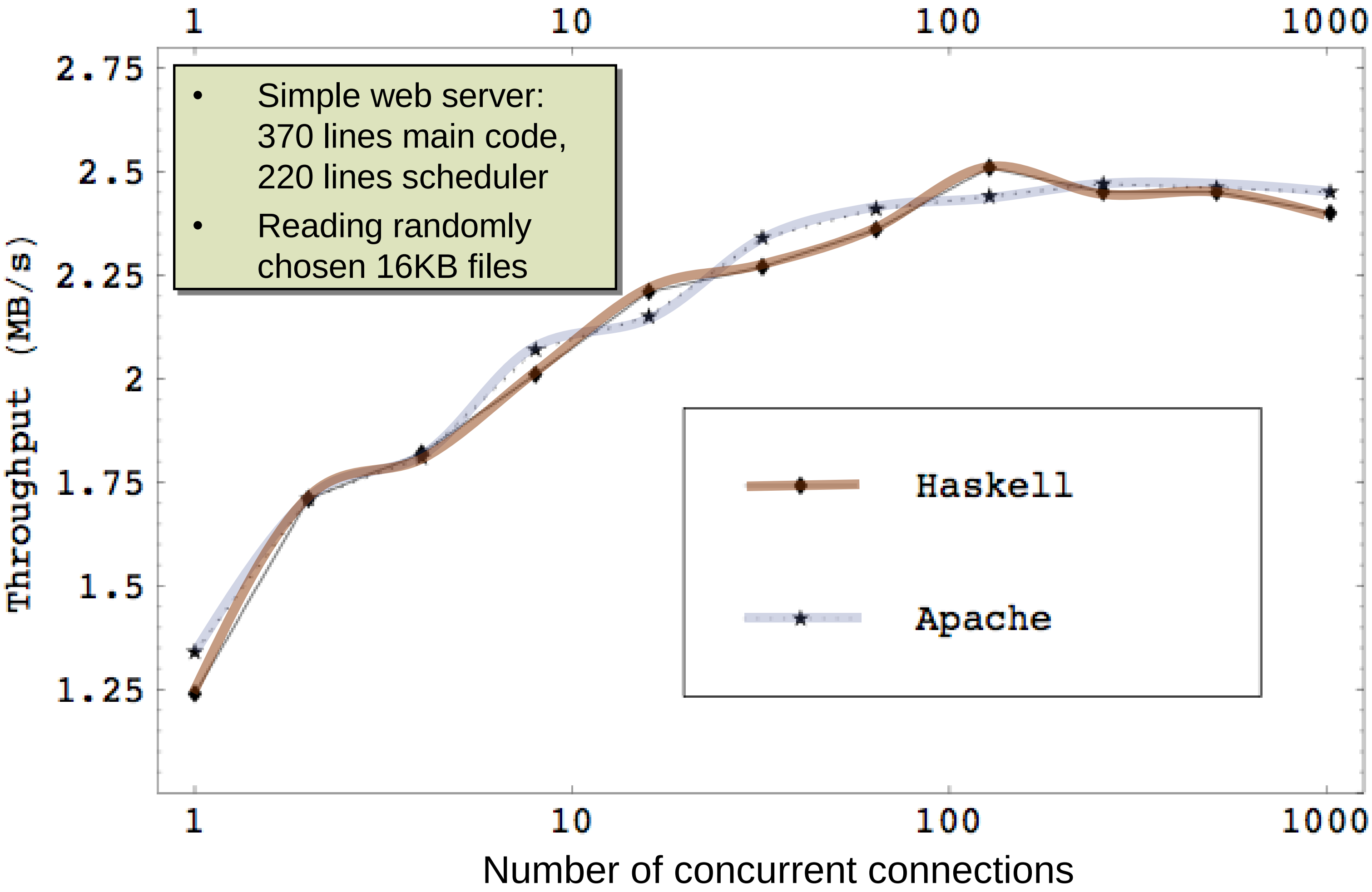
- Each thread randomly reads 4KB block chosen from a 1GB file.
- Total data read is 512MB
- Processes are disk bound: CPU utilization is 1% in each case

FIFO performance with idle threads



- 128 pairs of active application-level threads
- Each pair of threads ping-pongs 32KB blocks of data, for an overall total of 64GB data transfer
- The processor runs one copy of worker_main as a kernel thread

Test App: Web Server



- Simple web server: 370 lines main code, 220 lines scheduler
- Reading randomly chosen 16KB files

Qualitative Experience

- Implementing other features:
  - Exceptions
  - Timers
  - Mutexes, locks, and other synchronization mechanisms
- Easy to customize the schedulers
- Plugging in a user-level TCP stack (also in Haskell):
  - Defining/interpreting new system calls: 22 LOC
  - Event loop for incoming packets: 7 LOC
  - Event loop for timers: 9 LOC
  - Minimal changes elsewhere in the code

liveops℠
networking the world's work

www.liveops.com

- Presentation at CUFP 2007
  - Replaced Java-based servers with monadic CPS/event style servers written in Ocaml.
  - Supports 5,000 simultaneous users connecting via SSL
  - Sustains 700+ TPS (with bursts of 1,500 TPS) during peaks
  - Two major feature releases since initial deployment (mid-2006)

> "Although developed independently, this work is the same vein as (and, in some ways, validates) Peng Li and Steve Zdancewic's 'A Language-based Approach to Unifying Events and Threads'…"
>
> -- Chris Waterson CUFP 2007

Outline

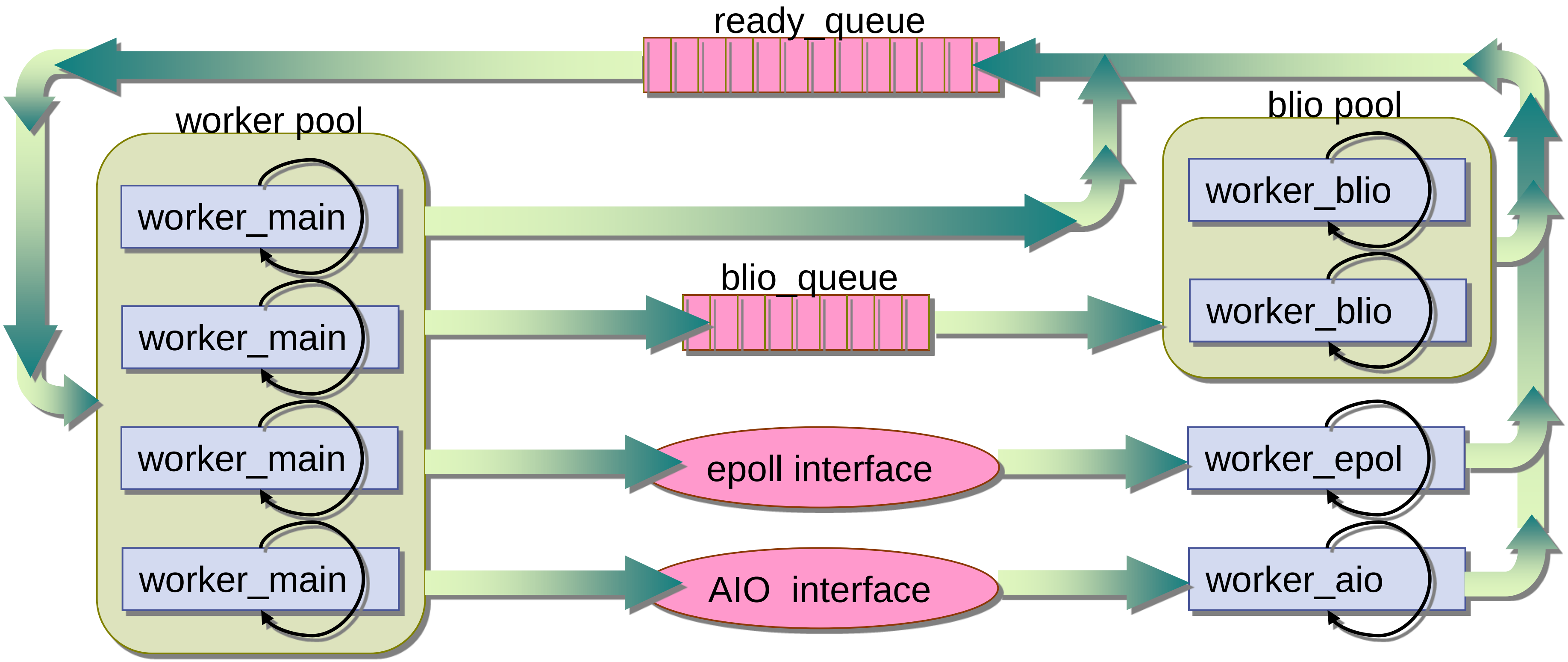
- Application-level cooperative concurrency in Haskell
  - Thread programming and Traces
  - Schedulers and Event processing
  - CPS translation and monads

- Examples
- Performance

- Future Directions & Conclusions

Multiprocessor Support

- Run one worker_main thread per CPU
- OS threads synchronized using Software Transactional Memory (STM)
- Use Haskell's STM monad
  - Application-level thread library can still implement mutexes or locks if they are more applicable
- Porting the implementation to use a multiprocessor was *very* easy

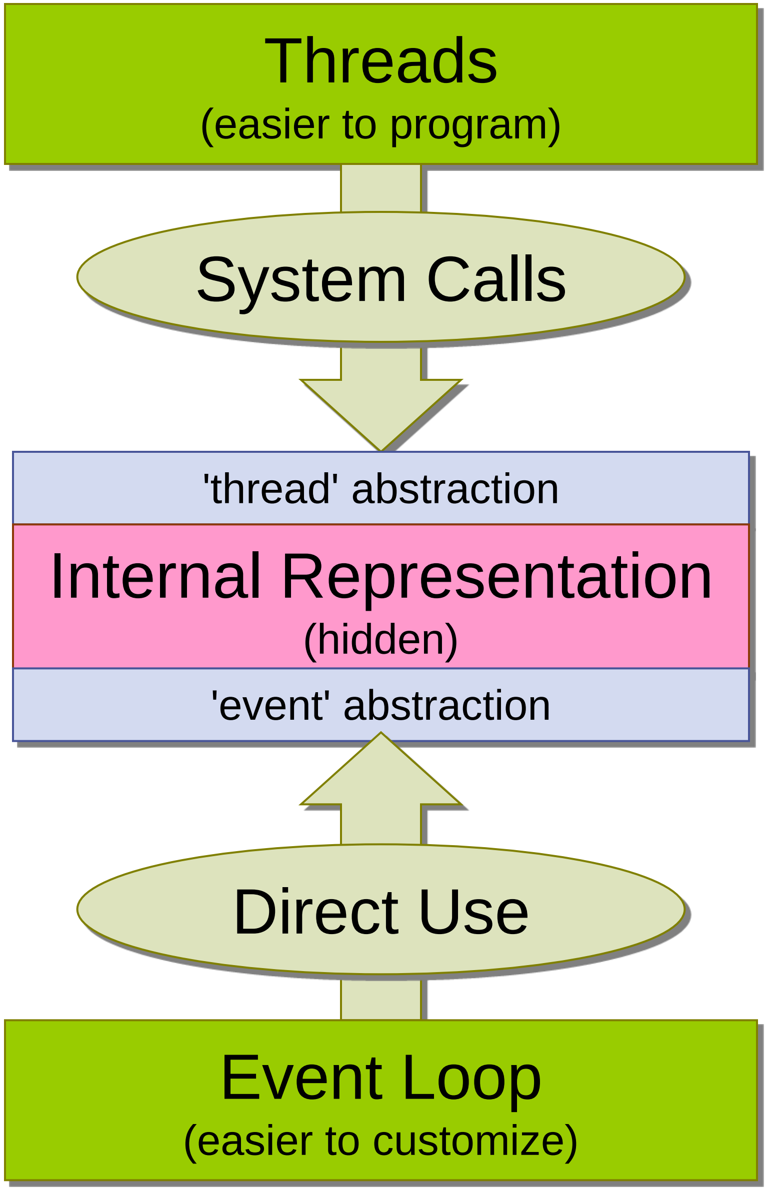


Future Directions

- More experience with STM and multiprocessors
- More experiments with custom schedulers
- Languages other than Haskell?
  - Ocaml, SML, Scheme, C#? (STM support may be harder)
- Provide different language support for concurrency?
  - Provide only minimal support for concurrency in the runtime itself
  - Move most scheduling into libraries
  - Provide good syntactic support for CPS
  - Integrate with STM?

- Peng Li and the GHC developers at MSR Cambridge
  - Proposed re-design of the Haskell runtime system

Conclusions

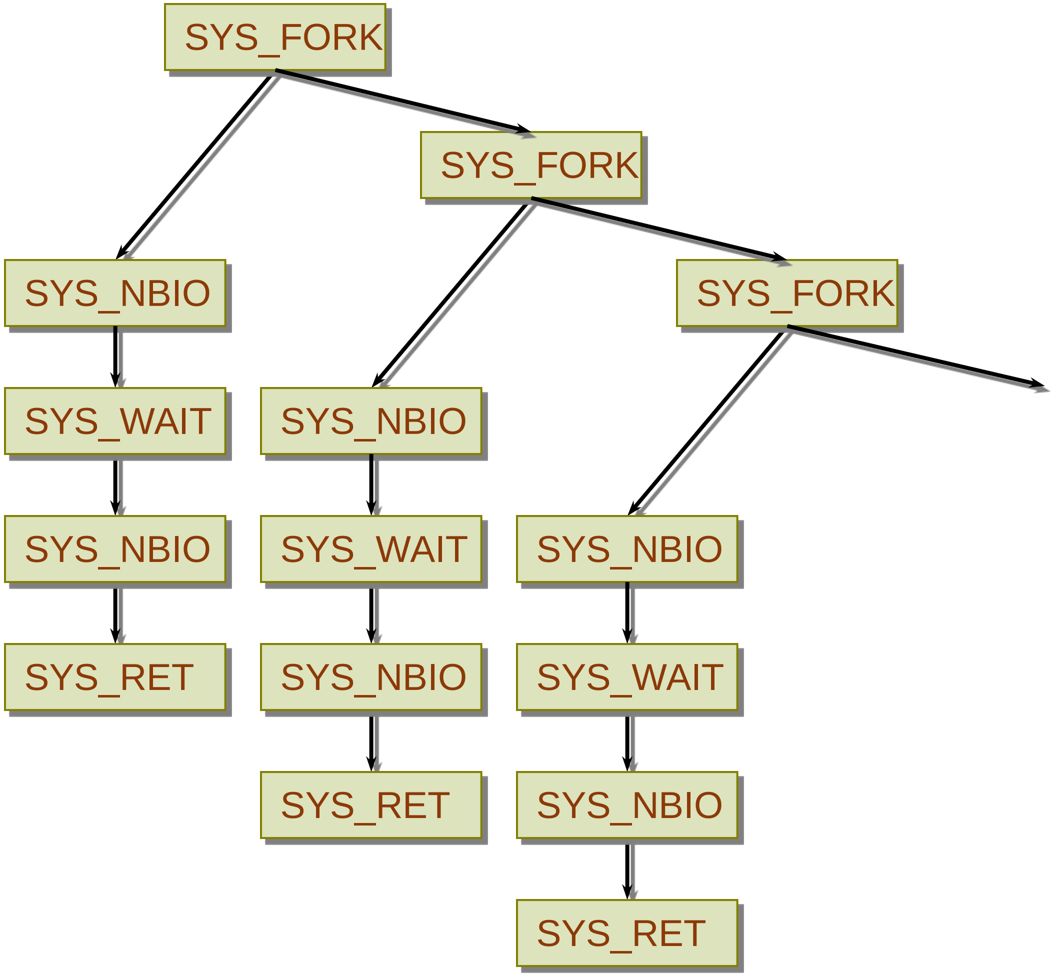


- We should strive to get the best of both worlds:
  - **Expressiveness** and **simplicity** of threads
  - **Scalability** and **flexibility** of event-driven systems
- Application-level concurrency in Haskell
  - CPS and explicit trace datastructure to represent events
  - Programmers write code in threaded style
  - Schedulers traverse the trace to drive the computation
- Haskell code can be found at:

  www.cis.upenn.edu/~lipeng

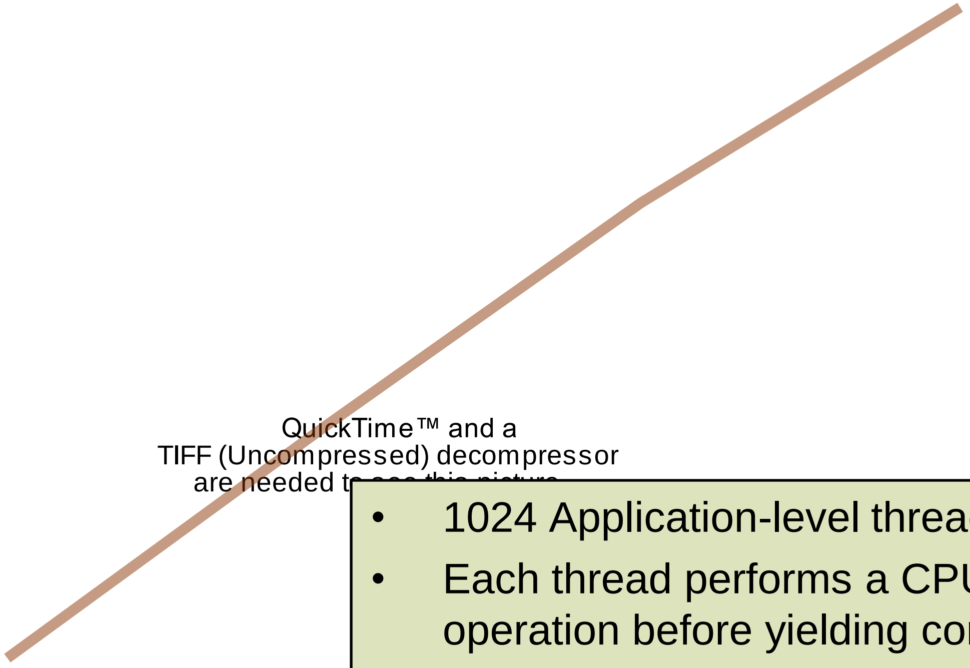
Thanks!

Event-driven Scheduler Code



```
worker_main Q = do {
  trace <- fetch_thread Q;
  execute trace Q;
  worker_main Q;
}

execute trace Q =
  case trace of
    SYS_RET -> return()

    SYS_FORK t1 t2 ->
      do { add_thread t1 Q;
           add_thread t2 Q;
         }

    SYS_NBIO cmd ->
      do { t <- cmd;
           add_thread t Q
         }

    …
```

Multiprocessor Speedups (Best Case)

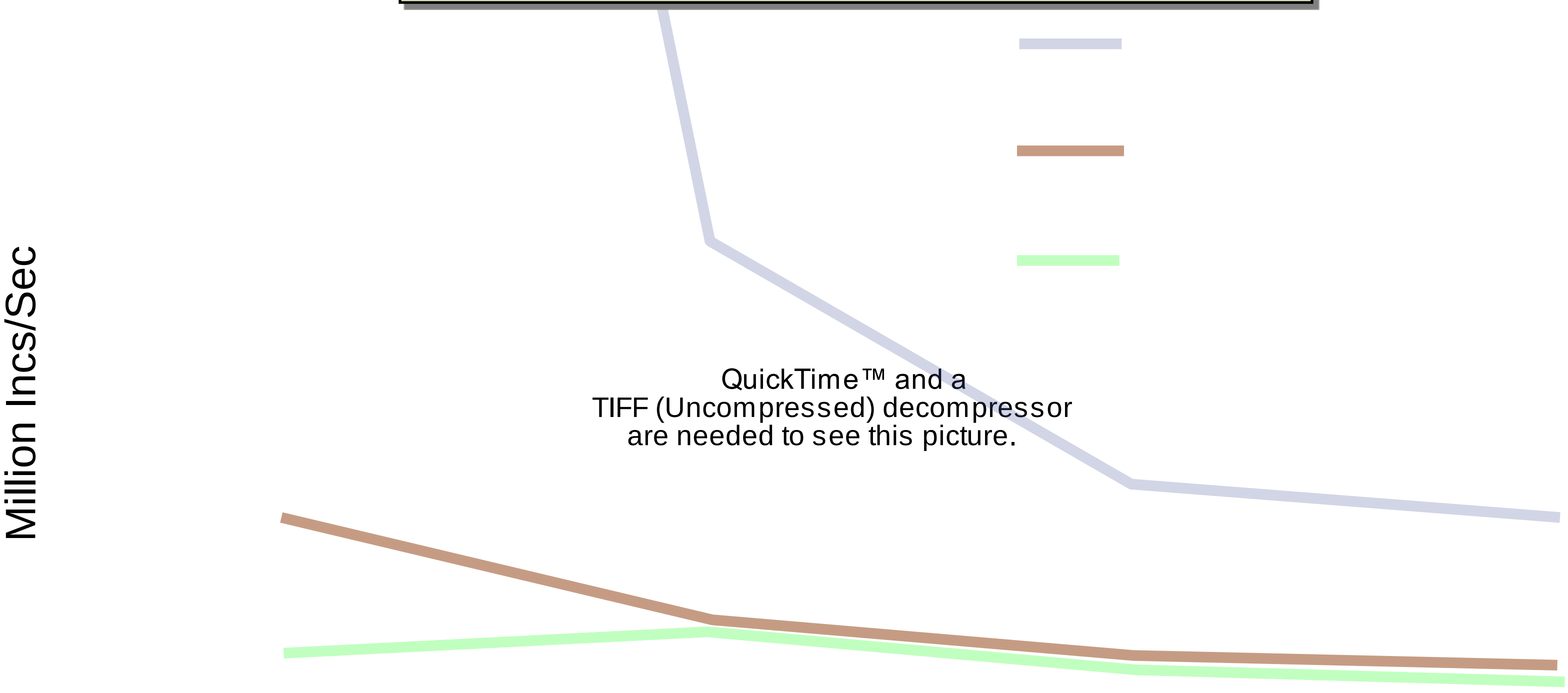


QuickTime™ and a
TIFF (Uncompressed) decompressor
are needed to see this picture.

- 1024 Application-level threads
- Each thread performs a CPU-intensive operation before yielding control
- Each processor runs one copy of worker_main as a kernel thread

STM Synchronization Overheads (Worst Case)

- 1 Application-level thread per processor.
- Each thread increments a shared integer: 3/4 of the transactions roll back.
- Each processor runs one copy of worker_main as a kernel thread.



QuickTime™ and a
TIFF (Uncompressed) decompressor
are needed to see this picture.