Unified *Concurrent* Runtime for D

DConf 2018
Dmitry Olshansky

Unify what?

In essence it's all about fibers

...and threads that schedule them

Fiber is a cooperative user-mode thread

D's fibers are scheduled *manually*

Hence we have a multitude of custom schedulers

Vibe.d is a popular one

Fibers primer

```
struct Node {
    int payload;
    Node* left, right;
}

void main(){
    int current;
    auto tree = new Node(3, new Node(2, new Node(1)), new Node(4));
    void postorder(Node* n) {
        if (n.left) postorder(n.left);
        current = n.payload;
        Fiber.yield();
        if (n.right) postorder(n.right);
    }
    Fiber f = new Fiber(() => postorder(tree));
    for (;;) {
        f.call();
        if (f.state == Fiber.State.TERM) break;
        writeln(current);
    }
}
```

Fibers and I/O

Now with I/O it gets tricky - the thread would be blocked

```
// Sketch of TCP echo server
import std.socket;
void echo(Socket sock, ubyte[] buffer)
{
    ssize_t read, write;
    for(;;) {
        do {
            read = sock.read(buffer[]);
            if (sock.wouldHaveBlocked) Fiber.yield();
            else if (read <= 0) break;
        } while(read < 0);
        do {
            write = sock.write(buffer[0..read]);
            if (sock.wouldHaveBlocked) Fiber.yield();
            else if (read <= 0) break;
        } while(write < 0);
    }
}
```

Suboptimal w/o eventloop + somebody needs to 'call' fibers

Fibers and I/O

Vibe.d chooses to provide its own I/O API

```
import vibe.vibe;

void main()
{
        listenTCP(7, (conn) { conn.write(conn); });
        runApplication();
}
```

Nice and clean

Or is it?

Caveats

Using anything not Fiber-aware will block thread and ruin performance

There are zero C/C++/Rust libraries that are made to run on top of Vibe.d

std.net.curl, std.socket, std.process do *not* work with Fibers

std.concurrency was *extended* to work with custom scheduler

To Fiberize or not, that is the question

Poject Photon

Transparent fiber scheduler that works with *most* blocking C/C++/etc code

Multi-threaded with great performance out of the box

Look beyond I/O and 3rd party libraries: Events, Channels/Queues

Photon in action

Parallel download of files with std.net.curl, Photon edition

```
import std.net.curl, std.stdio, std.datetime.stopwatch;
import photon;

immutable urls = [ "https://...", ... ];

void main(){
    startloop(); // init Photon eventloop (to be done lazily soon)
    void spawnDownload(string url, string file) {
        spawn(() => download(url, file));
    }
    StopWatch sw;
    sw.start();
    foreach(url; urls) spawnDownload(url, url.split('/').back);
    runFibers(); // start schedulers and run fibers
    sw.stop();
    writefln("Done: %s ms", sw.peek.total!"msecs");
}
```

Photon in action

Parallel download of files with std.net.curl, fiber edition

12 files (~1Mb file)	3 files (~200Kb each)
Sequential: 16 sec	Sequential: 4.4 sec
Threads: 5.3 sec	Threads: 0.46 sec
Fibers: 5.9 sec	Fibers: 0.52 sec

The test is super noisy (easily ~30%) on the internet

We are in the ballpark of dedicated threads

...even when running photon on a single core

Transparent scheduler

Fiber.yield needs to happen in some 3rd party library

Most code calls libc syscall wrapper!

That is precisely the point of our "integration" ... on Posix-like OS

Shadowing libc syscalls

Put in a shared library linked *before* the libc

```
extern(C) ssize_t read(int fd, void *buf, size_t count) nothrow
{
    return universalSyscall!(SYS_READ, "READ", SyscallKind.read,
                             Fcntl.explicit, EWOULDBLOCK)
                            (fd, cast(size_t)buf, count);
}
... // 6 or so more

ssize_t universalSyscall(size_t ident, string name, SyscallKind kind,
                         Fcntl fcntlStyle, ssize_t ERR, T...)
                        (int fd, T args) nothrow {
    if (currentFiber is null) {
        logf("%s PASSTHROUGH FD=%s", name, fd);
        return syscall(ident, fd, args).withErrorno;
    }
    else { ... MAGIC IS HERE }
}
```

Scheduler (Linux ver.)

Thread per "Core" (SMT etc.) with affinity mask

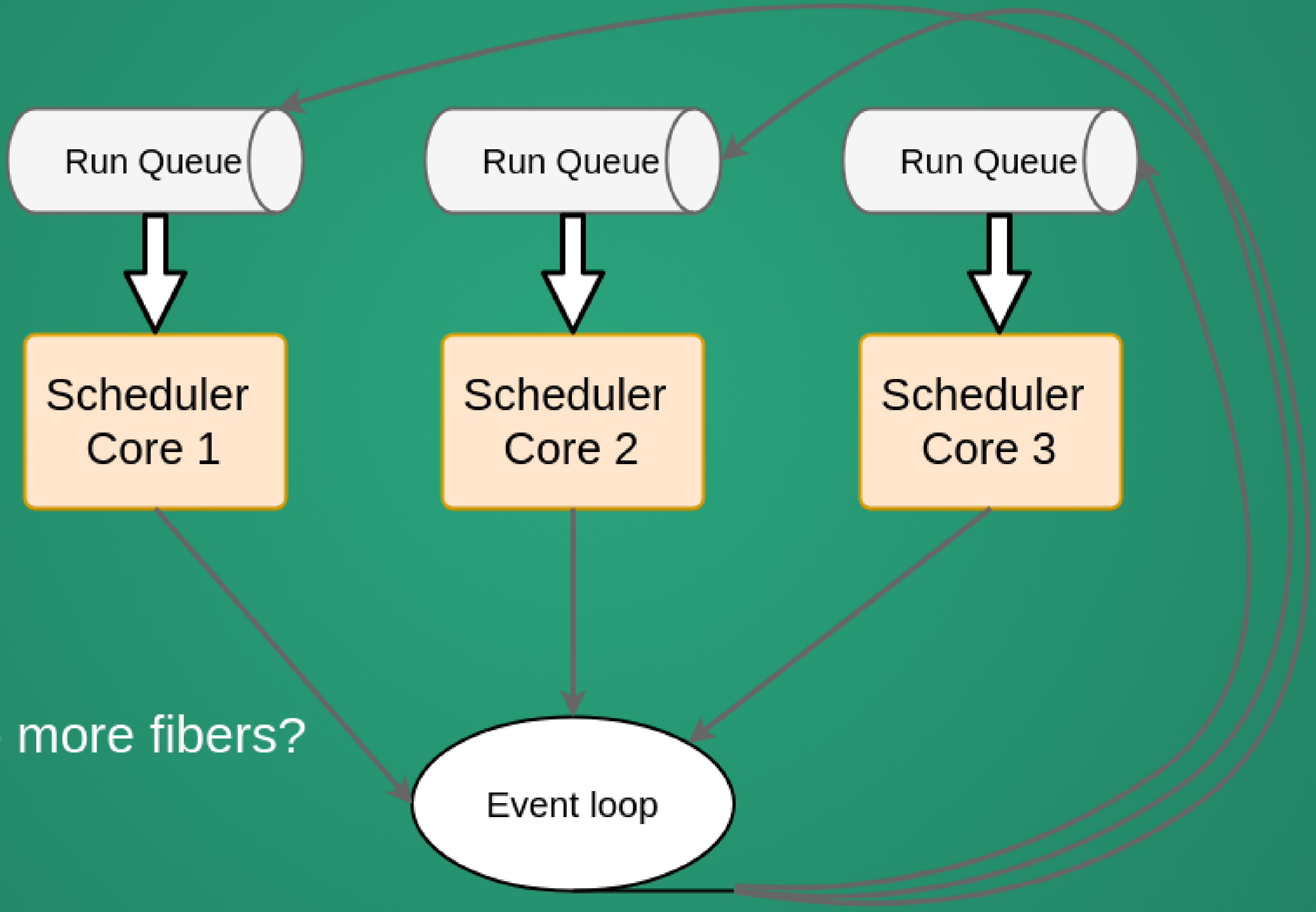
All I/O state is static and pre-mapped memory

"Cowboy" lock-free scheduler(s) <-> event loop interface

Fibers are pinned to thread based on load via "power of 2 random choices"

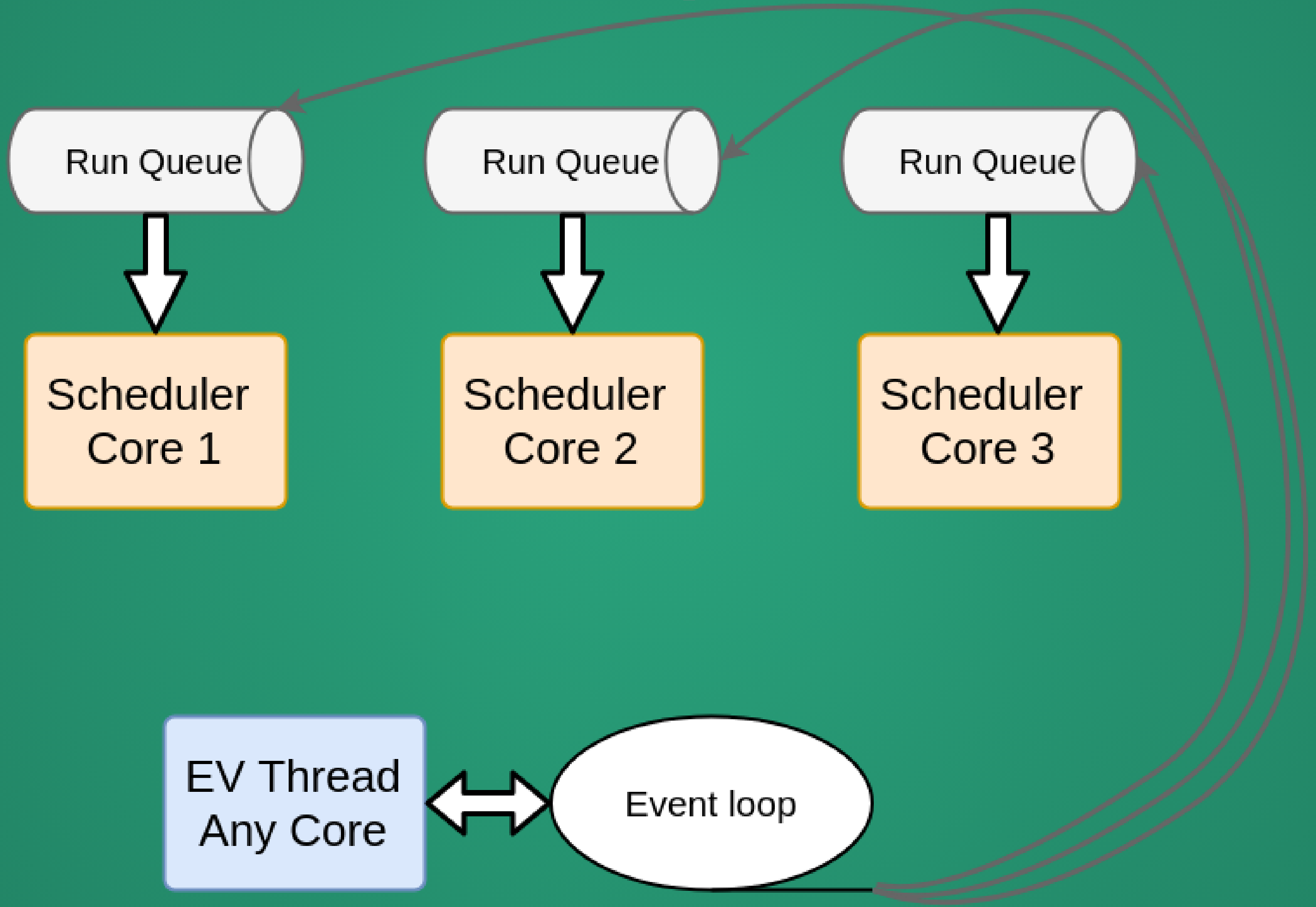
Some things are "Research code" quality

Scheduler in picture (v2)

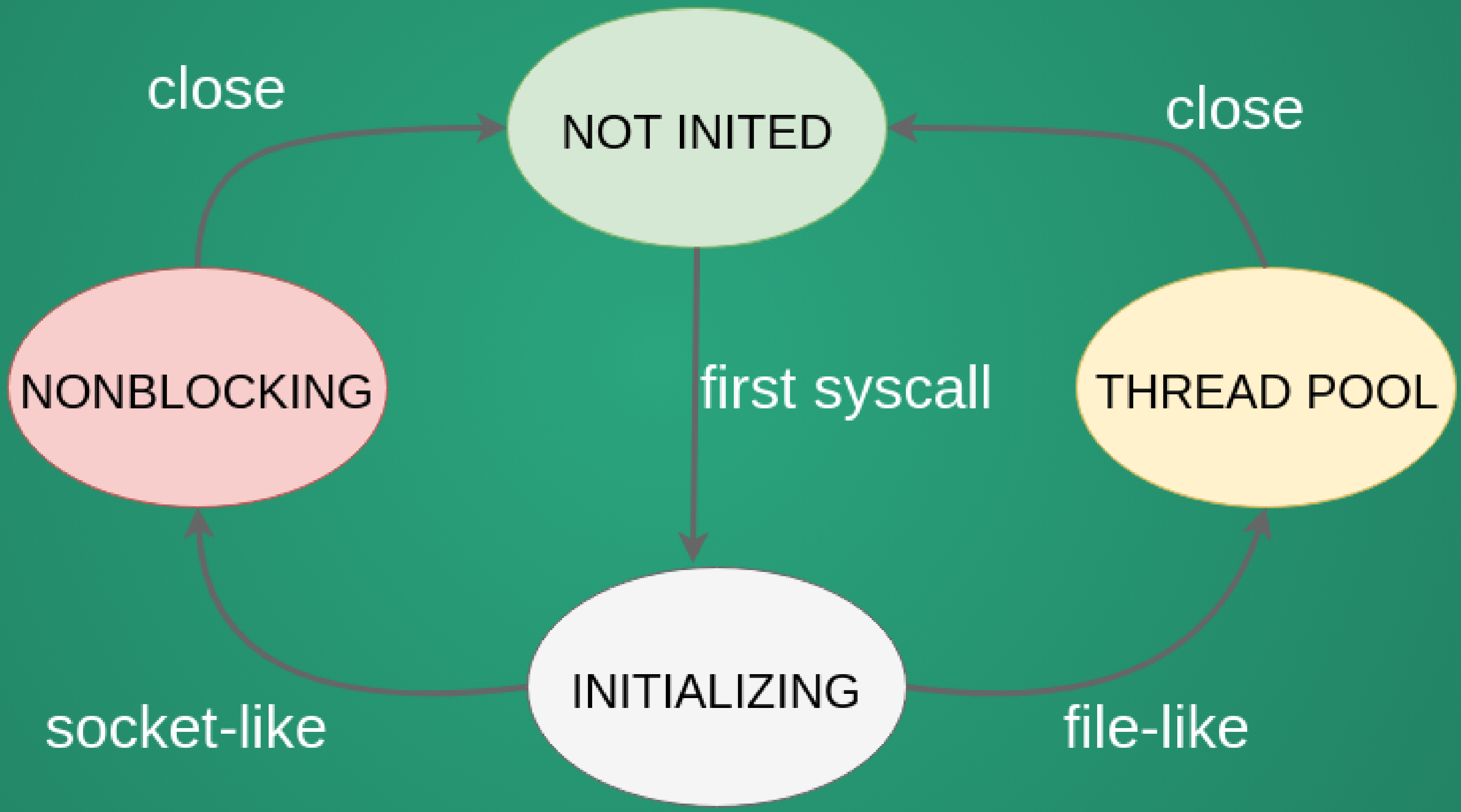


No more fibers?

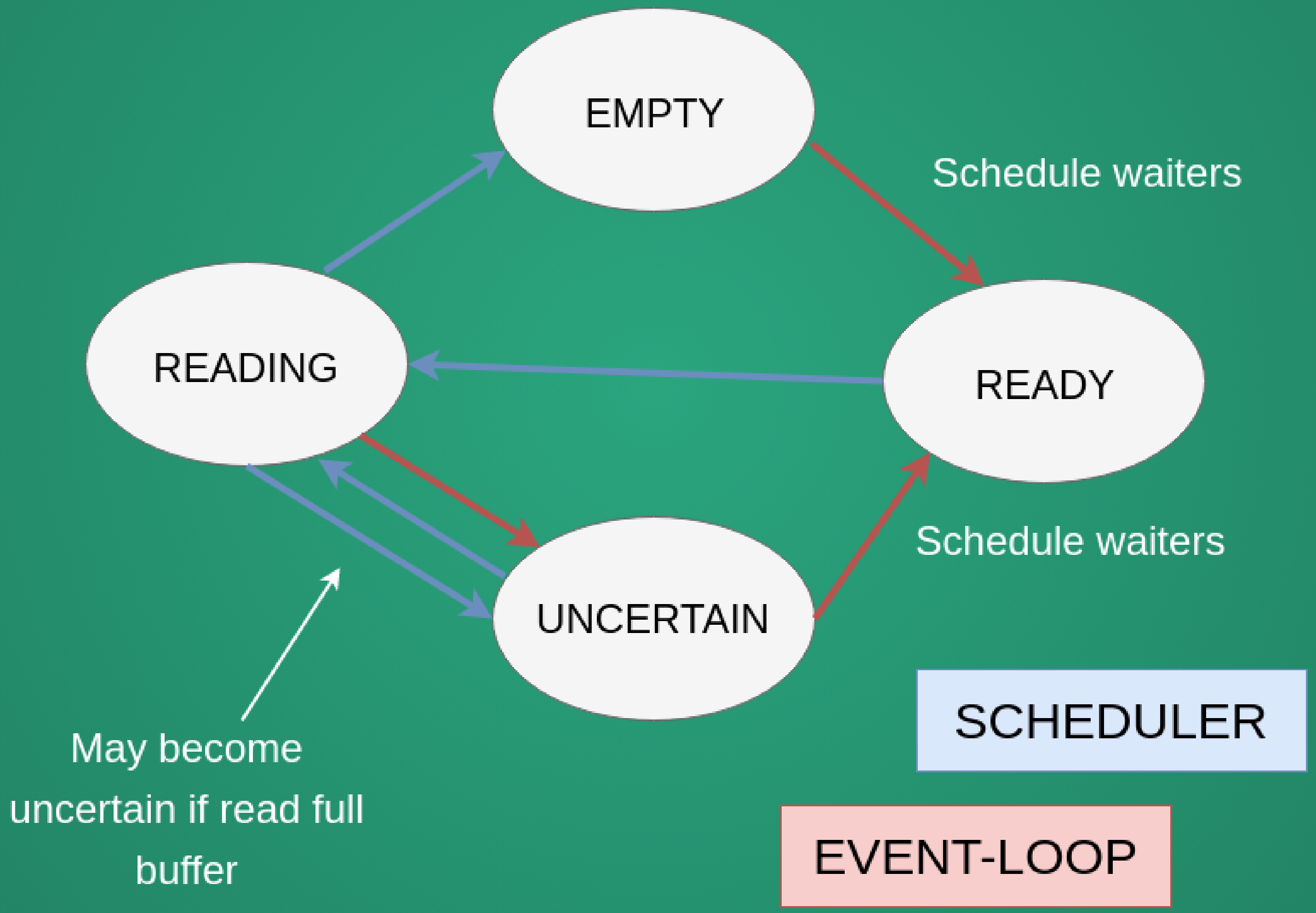
Scheduler in picture (v3?)



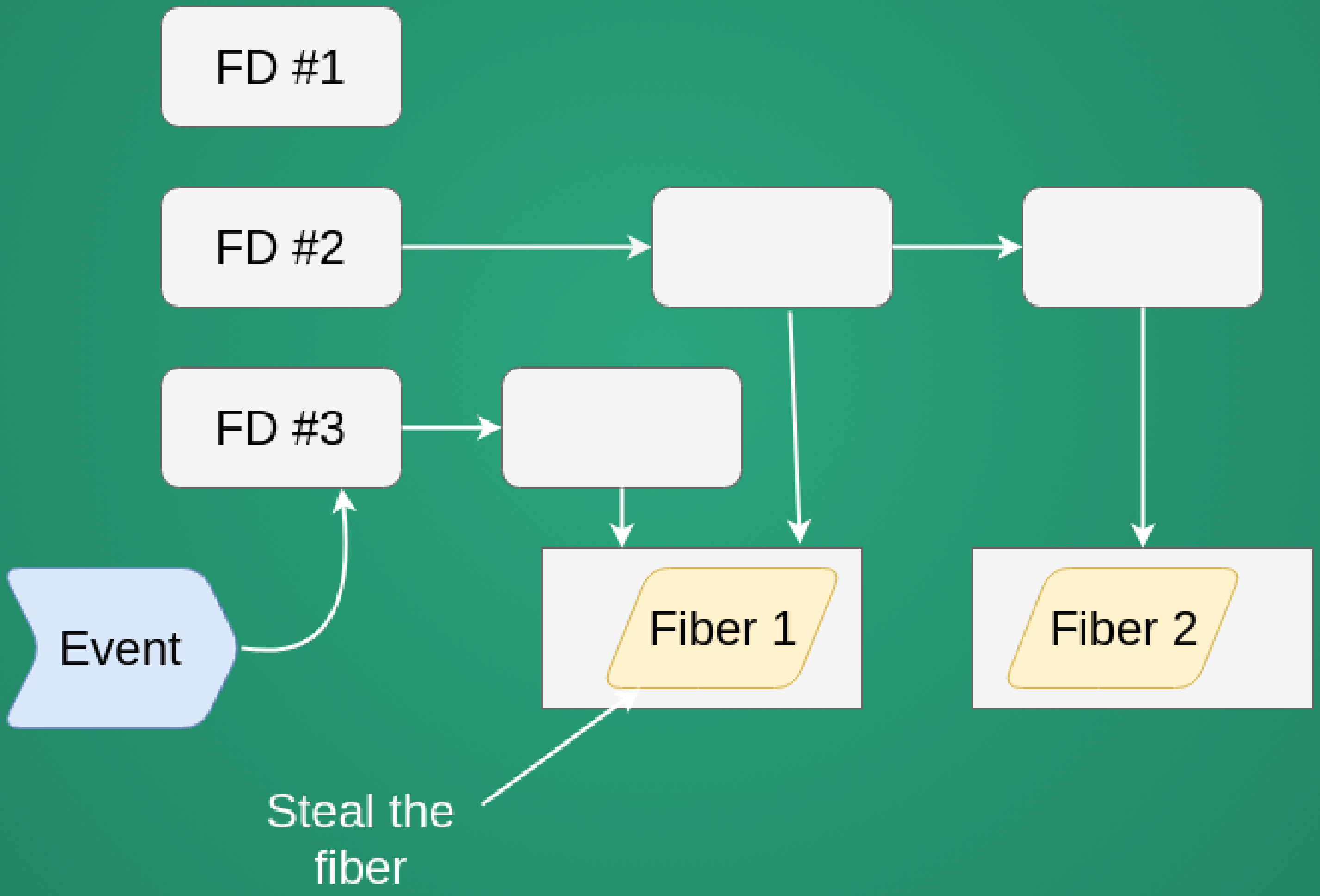
Descriptor state



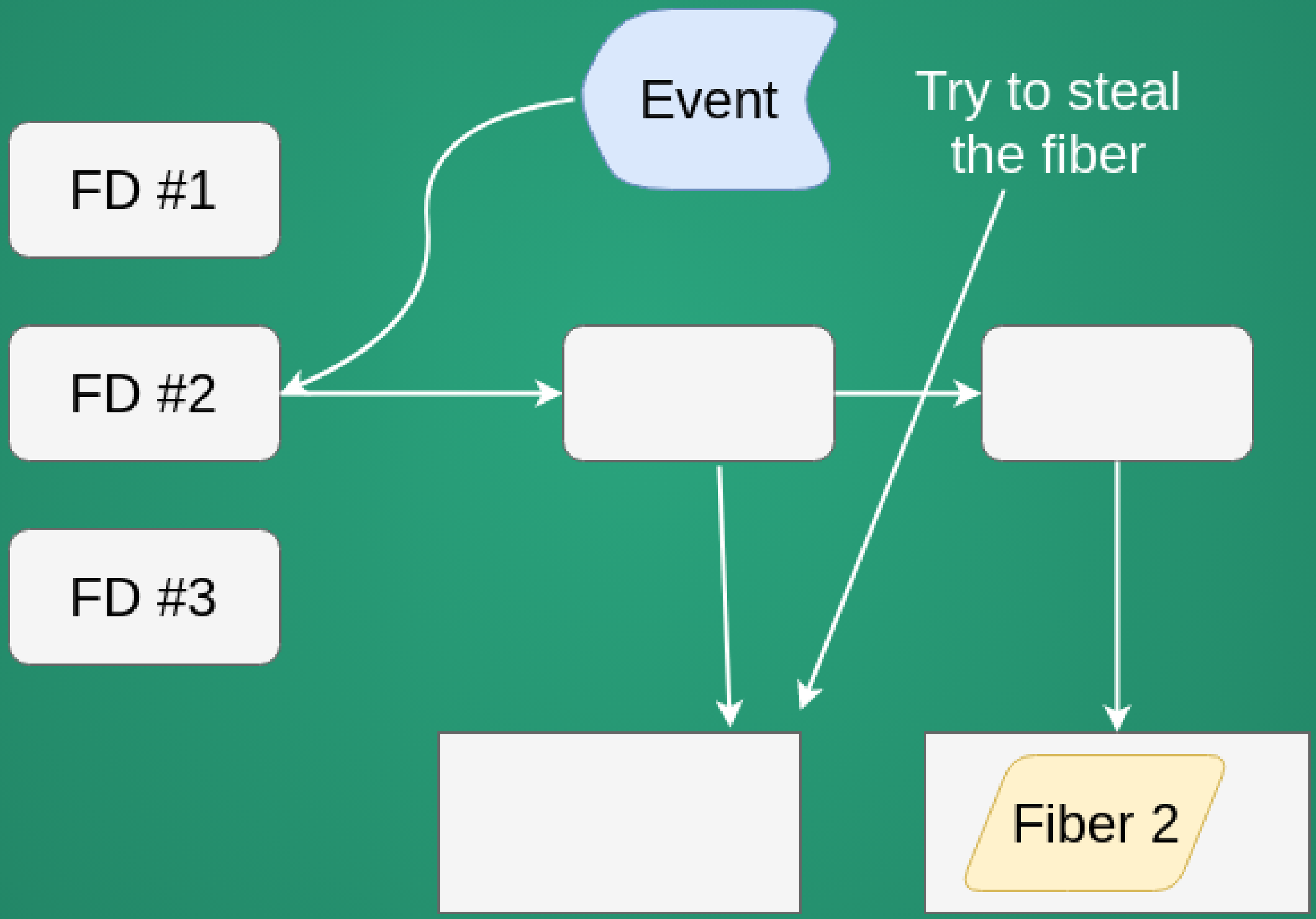
Descriptor read state



Implementing `poll`



Implementing `poll` (2)



What about Windows?

Same technique would work...

but there is no "small" libc

WinAPI is 10,000+ API entry points

Yet there is User Mode Scheduling subsystem

Which is kernel-assisted support for user-mode threads (of sorts...)

UMS in 5 seconds

Under User-Mode we get to create real threads:

1. *Never* scheduled by the NT *kernel scheduler*

2. To run them we pick a set of normal dedicated (per core typically) *scheduler threads and run them explicitly*

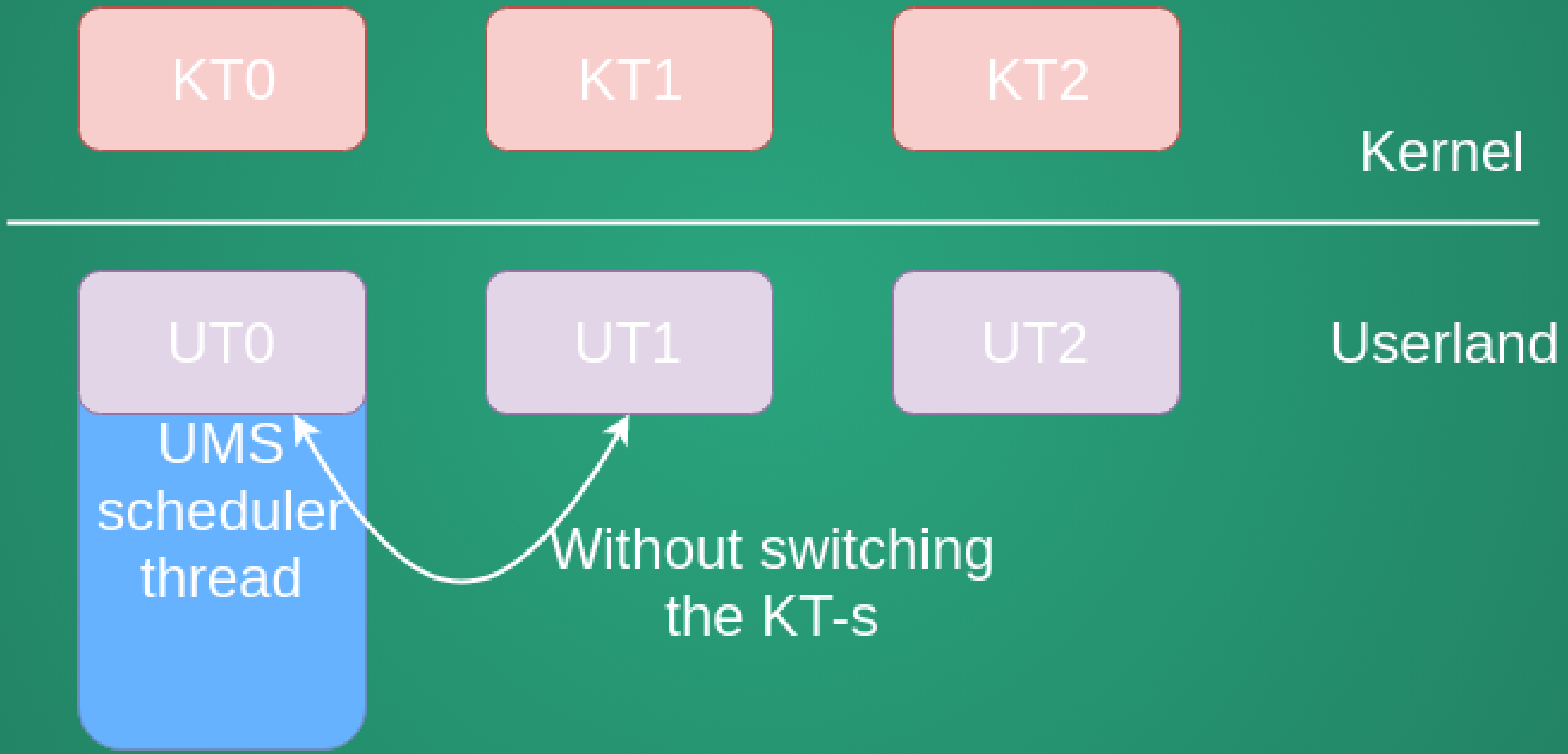
Better then a fiber in that:

1. An OS will automatically switch to scheduler should the current thread about to block

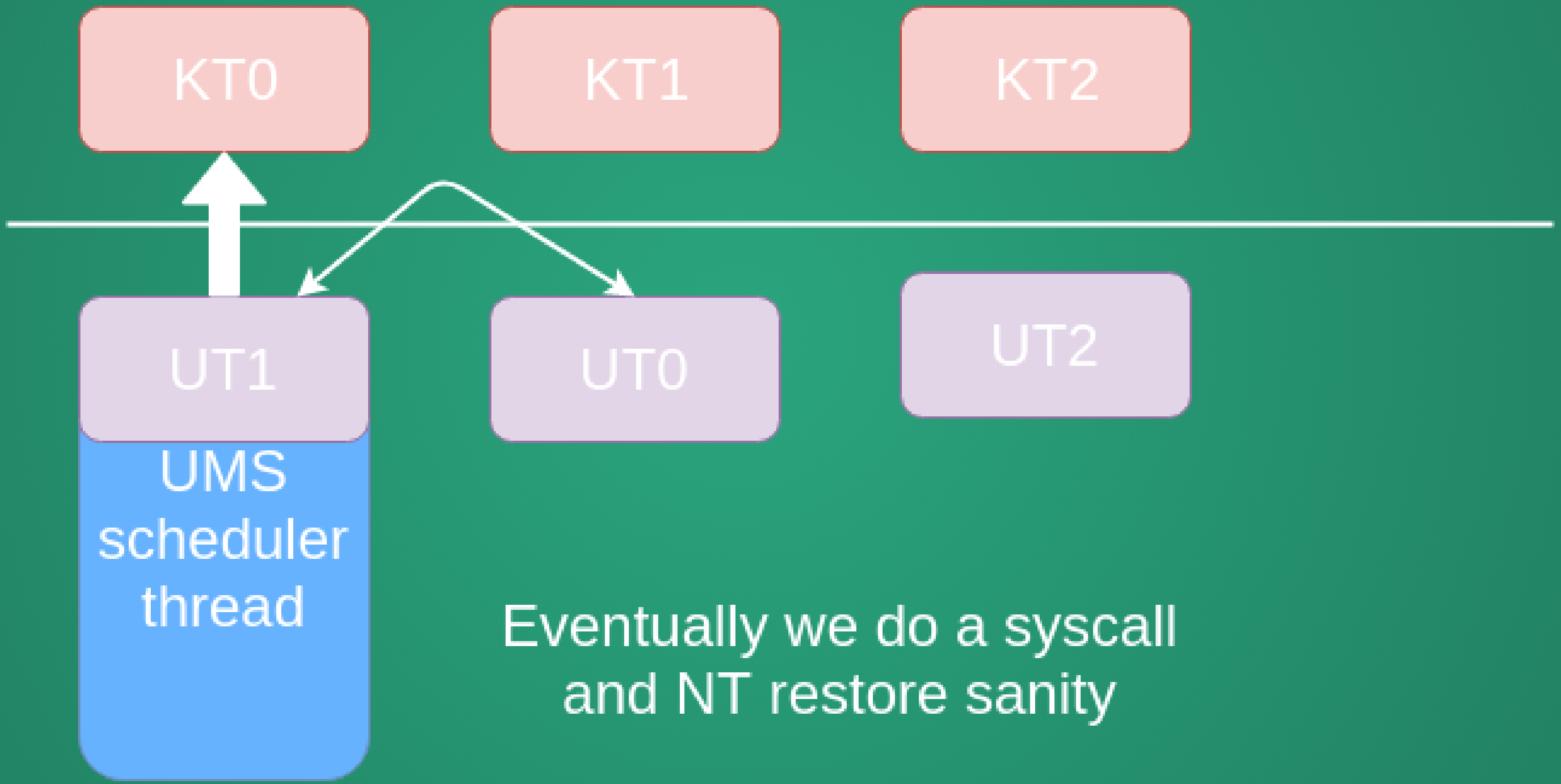
2. Any blocking event such as pagefault

3. Each thread has 1:1 user:kernel mapping, TLS and other goodies

UMS in pictures

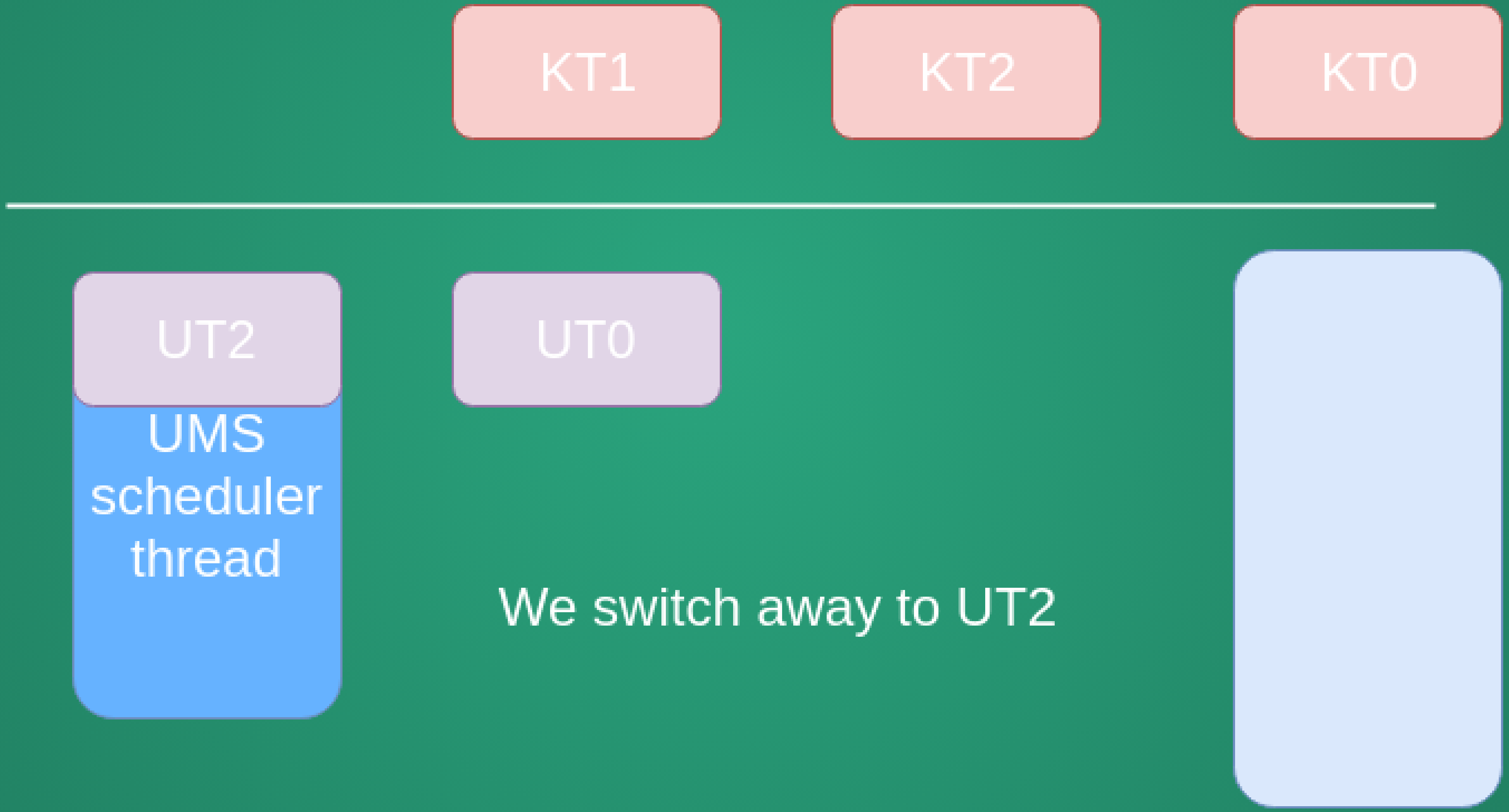


UMS in pictures

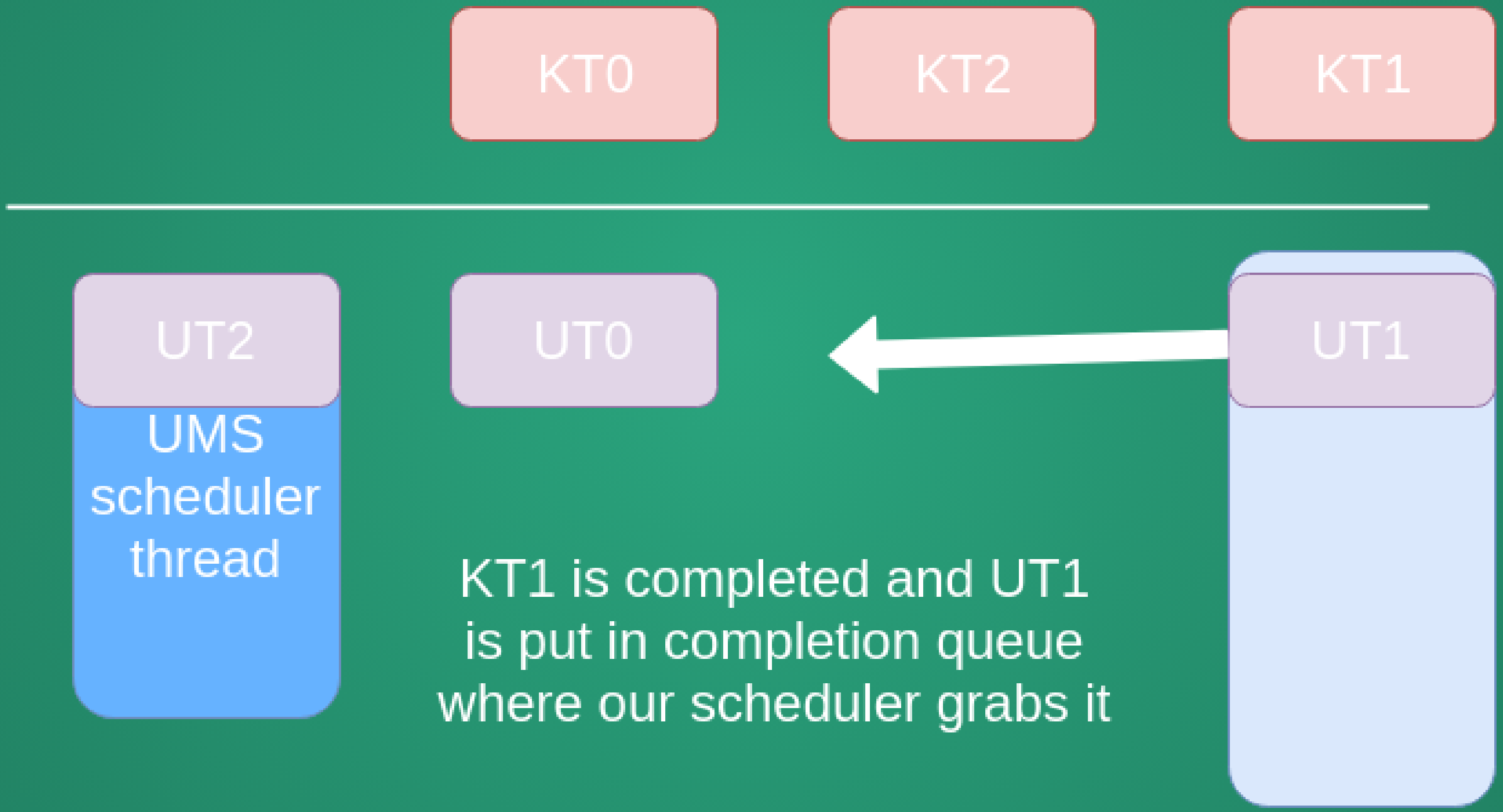


Eventually we do a syscall and NT restore sanity

UMS in pictures



UMS in pictures



KT1 is completed and UT1 is put in completion queue where our scheduler grabs it

It's not all roses

On subject, there is

- 1 interview with Windows Kernel Architect
- 2 examples with comments like "meh, doesn't improve performance for simple test"
- There *was* nice video at the BUILD conference on Windows 8, can't find it anymore

The real kicker:

Not my code nor any of 2 examples actually run on VirtualBox VM

Enter Alexander Ionescu

"A few months ago, as part of looking through the changes in Windows 10 Anniversary Update for the Windows Internals 7th Edition book, I noticed that the kernel began enforcing usage of the CR4[FSGSBASE] feature ... in order to allow usage of User Mode Scheduling (UMS)."

Freaking great find, Alex!

VirtualBox doesn't propagate that CPU flag

For future poor souls seeking to enable UMS:

http://www.alex-ionescu.com/?p=340

Closing notes on UMS

It is still more efficient to just use normal threads if we do not do async I/O with completion ports

Photon overrides parts of WinSock like on Linux

to use Overlapped I/O with Completion Ports

There is a nice option to use RIO sockets, should be super fast but trickier to get right

UMS thread is ~100x slower to *spawn* then Fiber

The whole thing is x64 only and Ivy Bridge or later CPU

Case study: HTTP server

Fast HTTP is not the main point of Photon

Using thin-wrapped Node.js (Nginx) HTTP parser

Using plain std.socket to prove the point

Add in a few sensible performance optimisations

Peper it by a couple of speedhacks
(competion does it as well, so why not)

Case study: HTTP server

HTTP server in Photon runtime

```
void server() {
    Socket server = new TcpSocket();
    server.setOption(SocketOptionLevel.SOCKET, SocketOption.REUSEADDR, true);
    server.bind(new InternetAddress("0.0.0.0", 8080));
    server.listen(1000);
    void processClient(Socket client) {
        spawn(() => server_worker(client));
    }
    while(true) {
        try {
            Socket client = server.accept();
            processClient(client);
        }
        catch(Exception e) {
            writefln("Failure to accept %s", e);
        }
    }
}
void main() {
    startloop();
    spawn(() => server());
    runFibers();
}
```

Case study: HTTP server

HTTP server in plain D threads

```
void server() {
    Socket server = new TcpSocket();
    server.setOption(SocketOptionLevel.SOCKET, SocketOption.REUSEADDR, true);
    server.bind(new InternetAddress("0.0.0.0", 8080));
    server.listen(1000);
    void processClient(Socket client) {
        new Thread(() => server_worker(client)).start();
    }
    while(true) {
        try {
            Socket client = server.accept();
            processClient(client);
        }
        catch(Exception e) {
            writefln("Failure to accept %s", e);
        }
    }
}
void main() {
    new Thread(() => server()).start();
}
```

Case study: HTTP server

Common part - bare bones HTTP processor

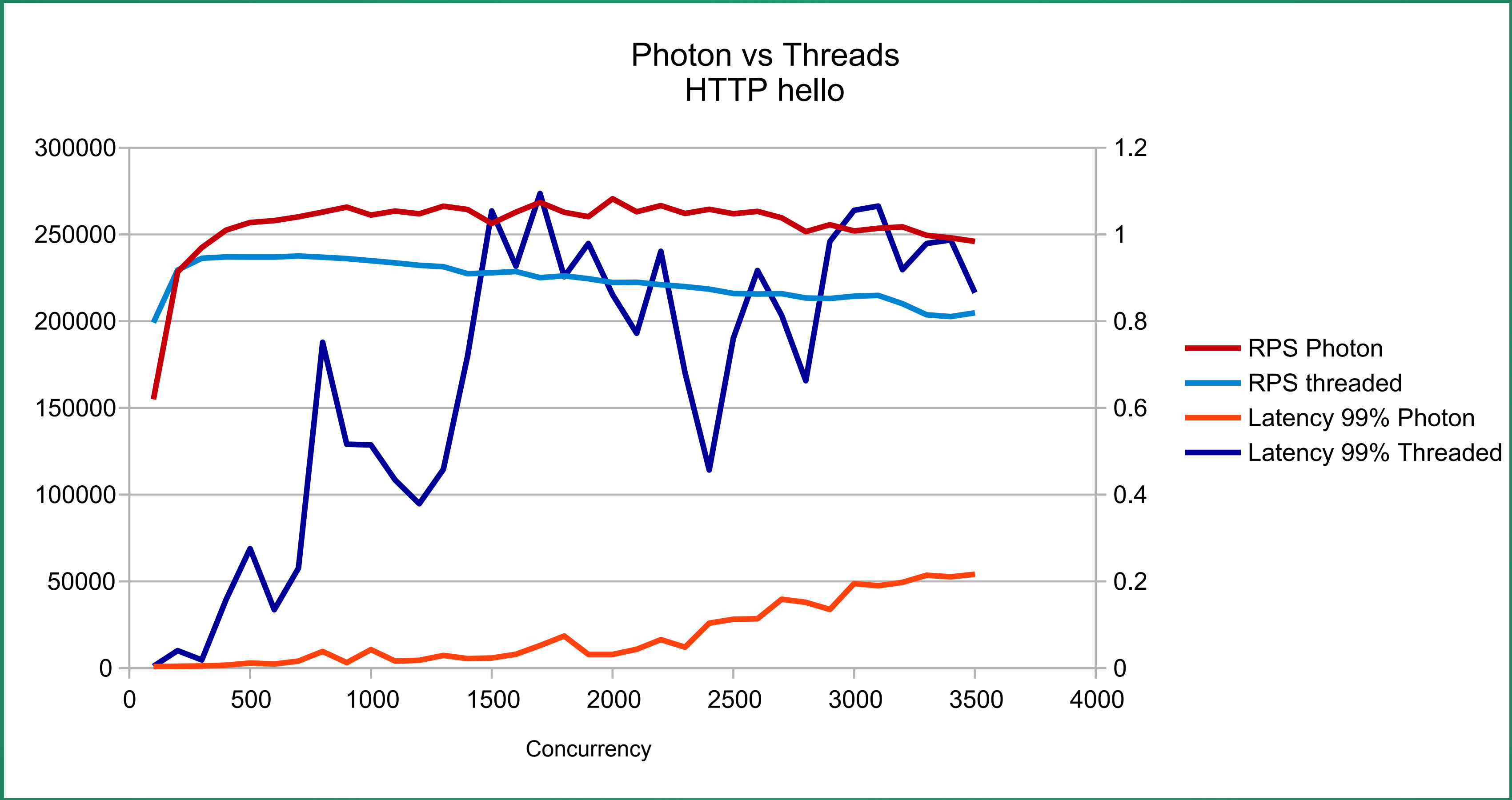
```
import utils.http_server;

class HelloWorldProcessor : HttpProcessor {
    HttpHeader[] headers = [HttpHeader("Content-Type", "text/plain; charset=utf-8

    this(Socket sock){ super(sock); }

    override void onComplete(HttpRequest req) {
        respondWith("Hello, world!", 200, headers);
    }
}
```

Time to bench threads vs fibers

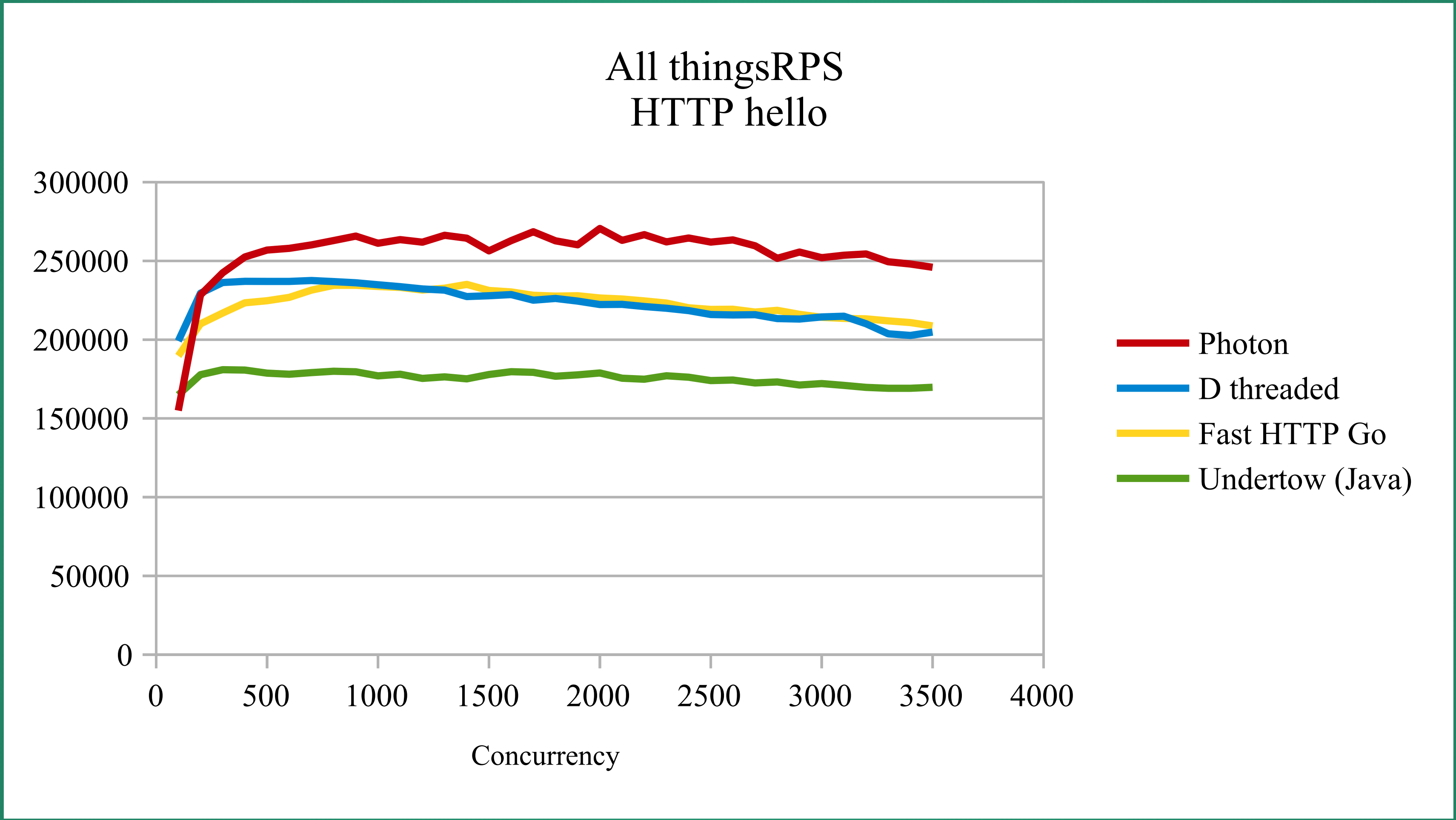
And a few of top guns from TechEmpower benchmark

And I'm taking DMD to this gunfight

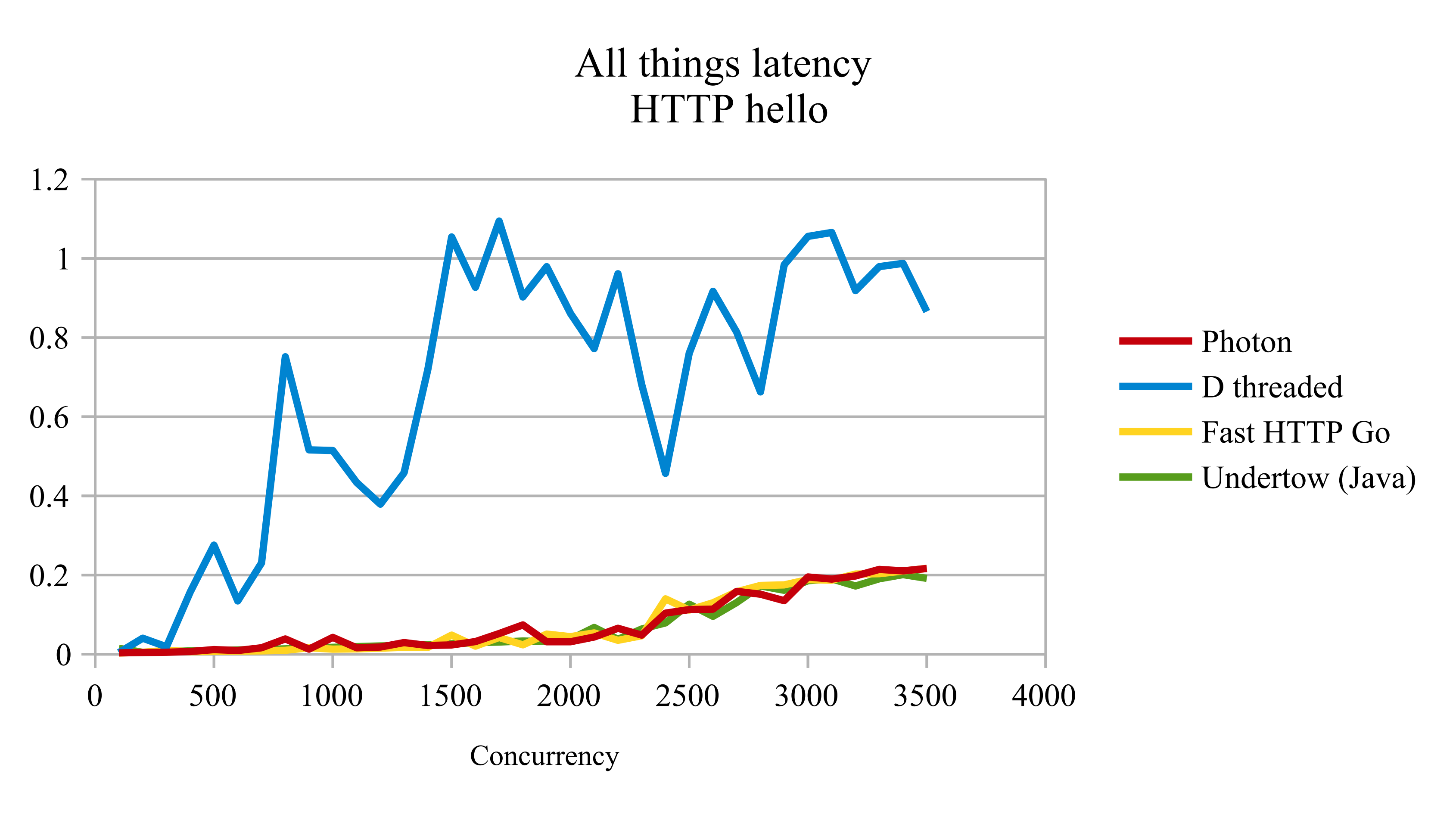
HTTP server benchmark



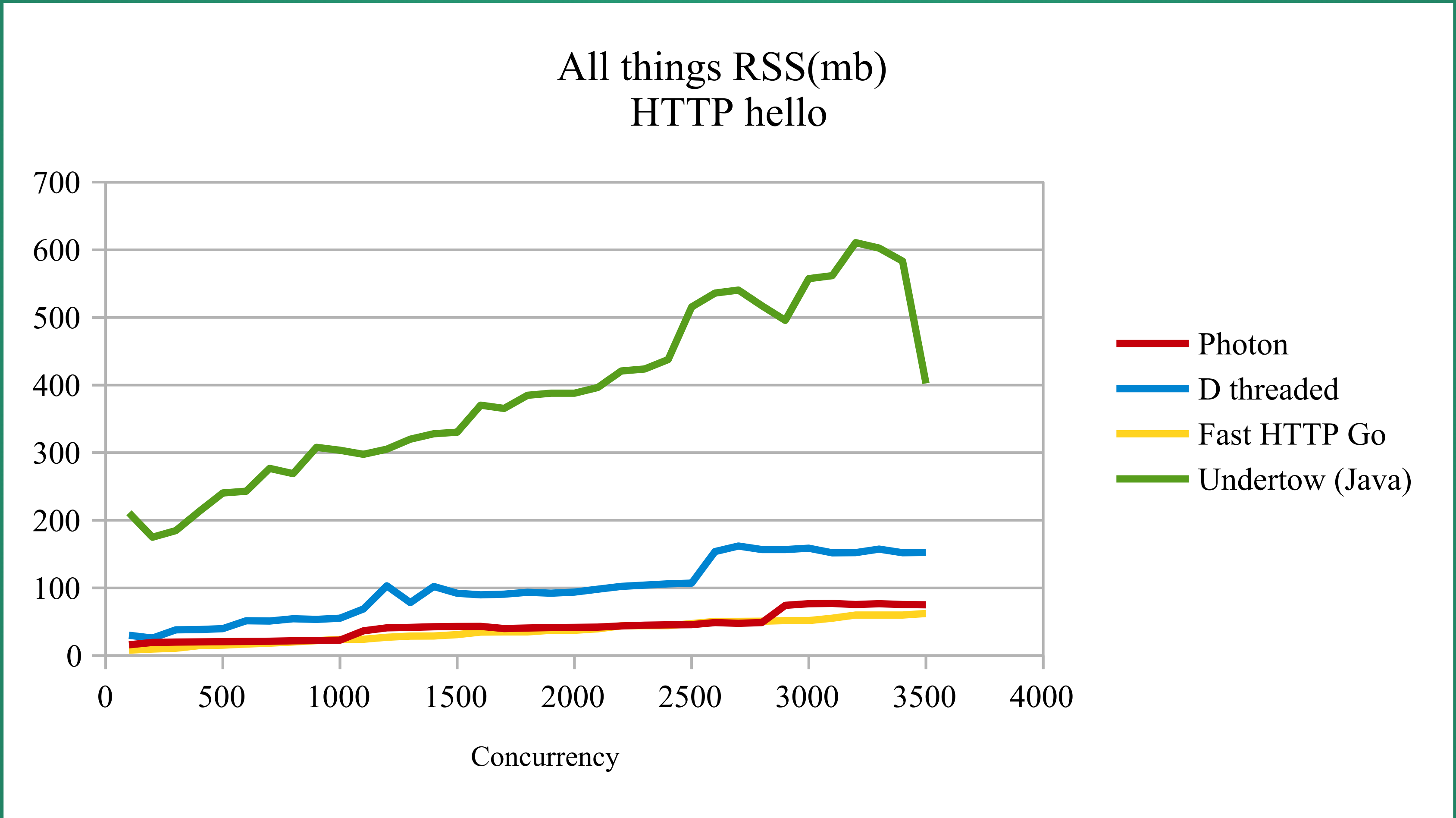
HTTP server benchmark



HTTP server benchmark

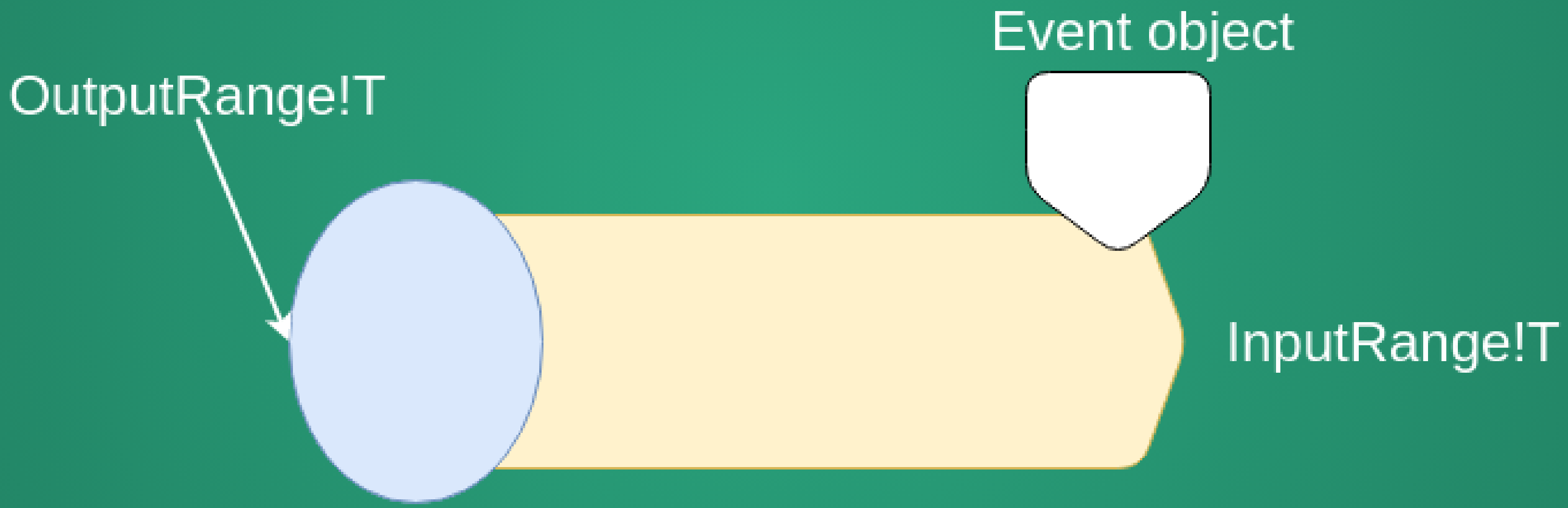


HTTP server benchmark



Beyond I/O

Channels or queues got popular with Go



Just label the obvious. Works for MPSC queue

Event, channel, source

```
interface Event {
   bool ready();
   void reset(); // typically called by await in Event-loop
}

size_t await(R)(R events)
if(isEvent!ElementType!R)) {
... event loop / scheduler magic
}
```

Channel is an OutputRange

Source is an InputRange

Event-aware algorithms

```
struct Map(R) {
  R _range;

  bool ready() { _range.ready: }

  auto event(){ return _range.event; }

}

struct Zip(RS...) {
  RS _ranges;

  bool ready() {
      return _ranges.reduce!(true)((a,b) => a && b.ready);
  }

  auto event(){
      return _ranges.map!(x => x.event);
  }
}
```

Bright future

Most of the ground work is done on:

basic I/O, primitive scheduling of Fibers

transparent integration of C libraries shown to work

Time to capitalize and expand:

- define composable multiplexing patterns
- "Executors" with Photon's current scheduler as an option
- Vibe.d on Photon (?)
- expand to cover more syscall surface

Help wanted!

Current platforms - Windows x64 and Linux x64

Even if you are not thrilled to hack on
Windows User Mode Something...

Lots of simple work to test things out!

32-bit Linux, FreeBSD and MacOS

BetterC Fibers would be an awesome addition

https://github.com/DmitryOlshansky/photon