Fast Enough

Cliff Moon - King of the Buttheads

Derp Derp Derp

I Had A Bunch Of Slow Erlang

That I ported to C

Now I Have Two Problems

Performance Tuning

- You absolutely must be measuring.
- What are you measuring?
- How are you measuring it?
- What is the goal for your metrics?

What Matters To Your App?

- Batch processing - Throughput
- Online transaction processing - 99.9% latency

The Stages Of Profiling

- Identify bottlenecks
- Optimize algorithms
- Port critical sections to C
- Repeat until satisfied

Erlang VM Concurrency

- 1 scheduler and run queue per core
- Async thread pool
- Driver managed threads

Porting To C

- NIF - Native Implemented Functions
- Linked In Drivers
- C Nodes

All About NIF

Runtime Characteristics Of NIF

- Extremely low overhead
- Must be re-entrant
- No setup or teardown
- Explicitly tied to a module version

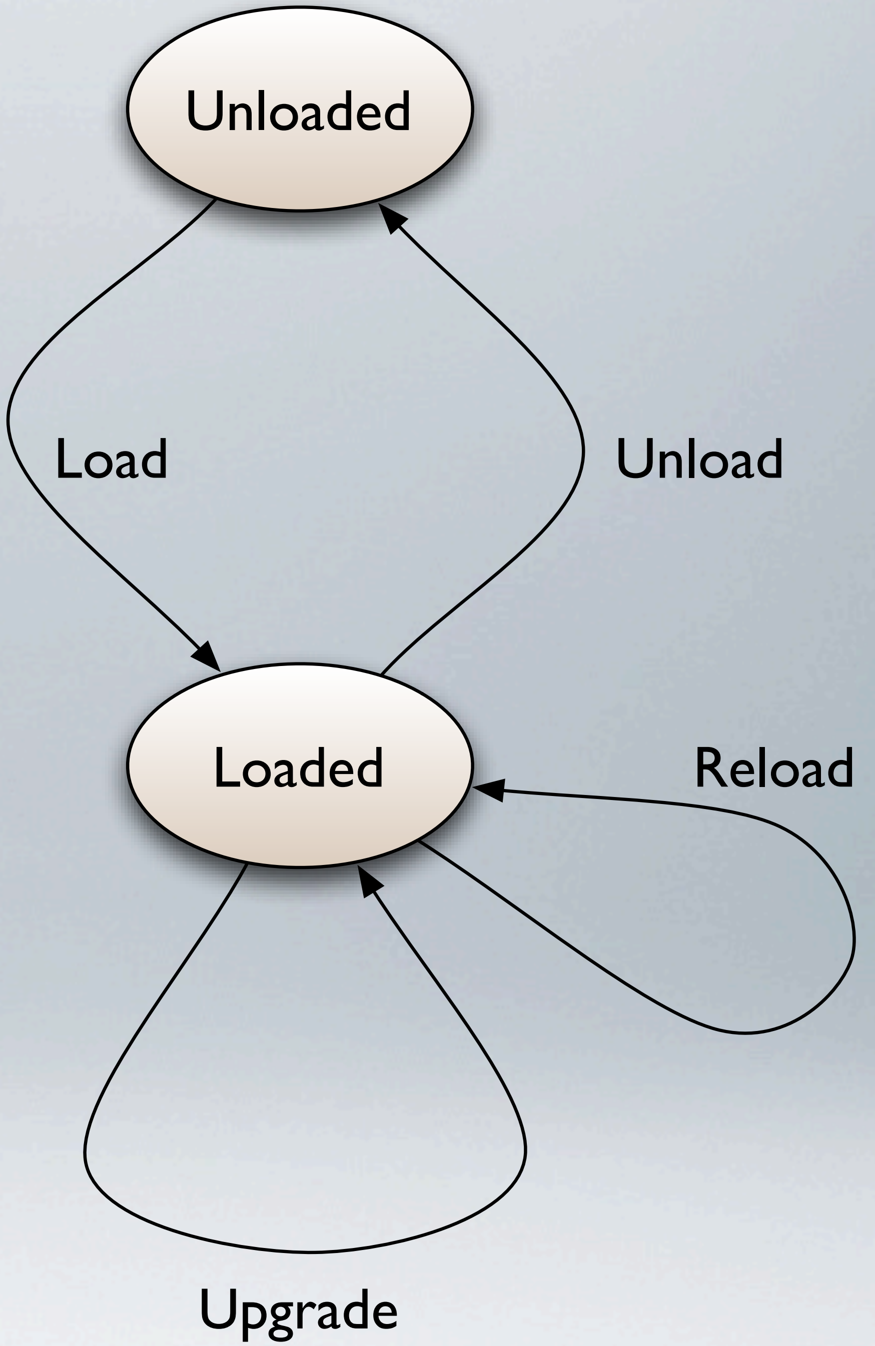
Lifecycle Of A NIF Library

```
load(ErlNifEnv* env,
    void** priv_data,
    ERL_NIF_TERM load_info);

reload(ErlNifEnv* env,
    void** priv_data,
    ERL_NIF_TERM load_info);

upgrade(ErlNifEnv* env,
    void** priv_data,
    void** old_priv_data,
    ERL_NIF_TERM load_info);

unload(ErlNifEnv* env,
    void* priv_data);
```



Explanation Of Arguments

- ErlNifEnv* env
  - Opaque context for the NIF
- void** priv_data
  - Stash for state between NIF calls
- ERL_NIF_TERM load_info
  - Second argument of erlang:load_nif/2

Erl_nif.c

```
for (i=0; i < entry->num_of_funcs; i++)
{
    Uint* code_ptr;
    erts_atom_get(entry->funcs[i].name, sys_strlen(entry->funcs[i].name), &f_atom);
    code_ptr = *get_func_pp(mod->code, f_atom, entry->funcs[i].arity);

    if (code_ptr[1] == 0) {
  code_ptr[5+0] = (Uint) BeamOp(op_call_nif);
    }
    else { /* Function traced, patch the original instruction word */
  BpData* bp = (BpData*) code_ptr[1];
        bp->orig_instr = (Uint) BeamOp(op_call_nif);
    }
    code_ptr[5+1] = (Uint) entry->funcs[i].fptr;
    code_ptr[5+2] = (Uint) lib;
}
```

Erl_nif.c

```
for (i=0; i < entry->num_of_funcs; i++)
{
    Uint* code_ptr;
    erts_atom_get(entry->funcs[i].name, sys_strlen(entry->funcs[i].name), &f_atom);
    code_ptr = *get_func_pp(mod->code, f_atom, entry->funcs[i].arity);

    if (code_ptr[1] == 0) {
  code_ptr[5+0] = (Uint) BeamOp(op_call_nif);
    }
    else { /* Function traced, patch the original instruction word */
  BpData* bp = (BpData*) code_ptr[1];
        bp->orig_instr = (Uint) BeamOp(op_call_nif);
    }
    code_ptr[5+1] = (Uint) entry->funcs[i].fptr;
    code_ptr[5+2] = (Uint) lib;
}
```

FNV Hash

Case Study

Terribly Slow

```erlang
-module(fnv_offset).

-export([hash/1]).

-define(OFFSET_BASIS, 2166136261).
-define(FNV_PRIME, 16777619).

hash(Term) when is_binary(Term) ->
  fnv_int(?OFFSET_BASIS, 0, Term);

hash(Term) ->
  fnv_int(?OFFSET_BASIS, 0, term_to_binary(Term)).

fnv_int(Hash, ByteOffset, Bin) when erlang:byte_size(Bin) == ByteOffset ->
  Hash;

fnv_int(Hash, ByteOffset, Bin) ->
  <<_:ByteOffset/binary, Octet:8, _/binary>> = Bin,
  Xord = Hash bxor Octet,
  fnv_int((Xord * ?FNV_PRIME) rem (2 bsl 31), ByteOffset+1, Bin).
```

Slow

```erlang
-module(fnv_slow).

-export([hash/1]).

-define(OFFSET_BASIS, 2166136261).
-define(FNV_PRIME, 16777619).

hash(Term) when is_binary(Term) ->
  fnv_int(?OFFSET_BASIS, Term);

hash(Term) ->
  fnv_int(?OFFSET_BASIS, term_to_binary(Term)).

fnv_int(Hash, <<>>) ->
  Hash;
fnv_int(Hash, <<Byte:8, Tail/binary>>) ->
  Xord = ((Hash bxor Byte) * ?FNV_PRIME) bor (2 bsl 31),
  fnv_int(Xord rem (2 bsl 31), Tail).
```

Hash NIF C

```c
/* niftest.c */
#include <stdio.h>
#include "erl_nif.h"
#include "fnv.h"

static ERL_NIF_TERM hash(ErlNifEnv* env, int argc, const ERL_NIF_TERM argv[]) {
    ErlNifBinary bin;
    long seed;
    int hash;

    if (enif_inspect_binary(env, argv[0], &bin)) {
      int res = enif_get_long(env, argv[1], &seed);
      hash = fnv_hash(bin.data, bin.size, seed);
      return enif_make_int(env, hash);
    }
    return enif_make_atom(env, "badarg");
}
static ErlNifFunc nif_funcs[] = {
    {"hash", 2, hash}
};
ERL_NIF_INIT(fnv_nif,nif_funcs,NULL,NULL,NULL,NULL)
```

Hash NIF C

```c
/* niftest.c */
#include <stdio.h>
#include "erl_nif.h"
#include "fnv.h"

static ERL_NIF_TERM hash(ErlNifEnv* env, int argc, const ERL_NIF_TERM argv[]) {
    ErlNifBinary bin;
    long seed;
    int hash;

    if (enif_inspect_binary(env, argv[0], &bin)) {
      int res = enif_get_long(env, argv[1], &seed);
      hash = fnv_hash(bin.data, bin.size, seed);
      return enif_make_int(env, hash);
    }
    return enif_make_atom(env, "badarg");
}
static ErlNifFunc nif_funcs[] = {
    {"hash", 2, hash}
};
ERL_NIF_INIT(fnv_nif,nif_funcs,NULL,NULL,NULL,NULL)
```

Hash NIF C

```c
/* niftest.c */
#include <stdio.h>
#include "erl_nif.h"
#include "fnv.h"

static ERL_NIF_TERM hash(ErlNifEnv* env, int argc, const ERL_NIF_TERM argv[]) {
    ErlNifBinary bin;
    long seed;
    int hash;

    if (enif_inspect_binary(env, argv[0], &bin)) {
      int res = enif_get_long(env, argv[1], &seed);
      hash = fnv_hash(bin.data, bin.size, seed);
      return enif_make_int(env, hash);
    }
    return enif_make_atom(env, "badarg");
}
static ErlNifFunc nif_funcs[] = {
    {"hash", 2, hash}
};
ERL_NIF_INIT(fnv_nif,nif_funcs,NULL,NULL,NULL,NULL)
```

Hash NIF C

```c
/* niftest.c */
#include <stdio.h>
#include "erl_nif.h"
#include "fnv.h"

static ERL_NIF_TERM hash(ErlNifEnv* env, int argc, const ERL_NIF_TERM argv[]) {
    ErlNifBinary bin;
    long seed;
    int hash;

    if (enif_inspect_binary(env, argv[0], &bin)) {
      int res = enif_get_long(env, argv[1], &seed);
      hash = fnv_hash(bin.data, bin.size, seed);
      return enif_make_int(env, hash);
    }
    return enif_make_atom(env, "badarg");
}
static ErlNifFunc nif_funcs[] = {
    {"hash", 2, hash}
};
ERL_NIF_INIT(fnv_nif,nif_funcs,NULL,NULL,NULL,NULL)
```

Hash NIF C

```c
/* niftest.c */
#include <stdio.h>
#include "erl_nif.h"
#include "fnv.h"

static ERL_NIF_TERM hash(ErlNifEnv* env, int argc, const ERL_NIF_TERM argv[]) {
    ErlNifBinary bin;
    long seed;
    int hash;

    if (enif_inspect_binary(env, argv[0], &bin)) {
      int res = enif_get_long(env, argv[1], &seed);
      hash = fnv_hash(bin.data, bin.size, seed);
      return enif_make_int(env, hash);
    }
    return enif_make_atom(env, "badarg");
}
static ErlNifFunc nif_funcs[] = {
    {"hash", 2, hash}
};
ERL_NIF_INIT(fnv_nif,nif_funcs,NULL,NULL,NULL,NULL)
```

Hash NIF C

```c
/* niftest.c */
#include <stdio.h>
#include "erl_nif.h"
#include "fnv.h"

static ERL_NIF_TERM hash(ErlNifEnv* env, int argc, const ERL_NIF_TERM argv[]) {
    ErlNifBinary bin;
    long seed;
    int hash;

    if (enif_inspect_binary(env, argv[0], &bin)) {
      int res = enif_get_long(env, argv[1], &seed);
      hash = fnv_hash(bin.data, bin.size, seed);
      return enif_make_int(env, hash);
    }
    return enif_make_atom(env, "badarg");
}
static ErlNifFunc nif_funcs[] = {
    {"hash", 2, hash}
};
ERL_NIF_INIT(fnv_nif,nif_funcs,NULL,NULL,NULL,NULL)
```

Hash NIF C

```c
/* niftest.c */
#include <stdio.h>
#include "erl_nif.h"
#include "fnv.h"

static ERL_NIF_TERM hash(ErlNifEnv* env, int argc, const ERL_NIF_TERM argv[]) {
    ErlNifBinary bin;
    long seed;
    int hash;

    if (enif_inspect_binary(env, argv[0], &bin)) {
      int res = enif_get_long(env, argv[1], &seed);
      hash = fnv_hash(bin.data, bin.size, seed);
      return enif_make_int(env, hash);
    }
    return enif_make_atom(env, "badarg");
}
static ErlNifFunc nif_funcs[] = {
    {"hash", 2, hash}
};
ERL_NIF_INIT(fnv_nif,nif_funcs,NULL,NULL,NULL,NULL)
```

Hash NIF C

```c
/* niftest.c */
#include <stdio.h>
#include "erl_nif.h"
#include "fnv.h"

static ERL_NIF_TERM hash(ErlNifEnv* env, int argc, const ERL_NIF_TERM argv[]) {
    ErlNifBinary bin;
    long seed;
    int hash;

    if (enif_inspect_binary(env, argv[0], &bin)) {
      int res = enif_get_long(env, argv[1], &seed);
      hash = fnv_hash(bin.data, bin.size, seed);
      return enif_make_int(env, hash);
    }
    return enif_make_atom(env, "badarg");
}
static ErlNifFunc nif_funcs[] = {
    {"hash", 2, hash}
};
ERL_NIF_INIT(fnv_nif,nif_funcs,NULL,NULL,NULL,NULL)
```

Hash NIF C

```c
/* niftest.c */
#include <stdio.h>
#include "erl_nif.h"
#include "fnv.h"

static ERL_NIF_TERM hash(ErlNifEnv* env, int argc, const ERL_NIF_TERM argv[]) {
    ErlNifBinary bin;
    long seed;
    int hash;

    if (enif_inspect_binary(env, argv[0], &bin)) {
      int res = enif_get_long(env, argv[1], &seed);
      hash = fnv_hash(bin.data, bin.size, seed);
      return enif_make_int(env, hash);
    }
    return enif_make_atom(env, "badarg");
}
static ErlNifFunc nif_funcs[] = {
    {"hash", 2, hash}
};
ERL_NIF_INIT(fnv_nif,nif_funcs,NULL,NULL,NULL,NULL)
```

Hash NIF Erlang

```erlang
-module(fnv_nif).

-on_load(init/0).

-define(SEED, 2166136261).

-export([init/0, hash/1, hash/2]).

init() ->
  erlang:load_nif("priv/ef_examples_drv",0).

hash(Bin) ->
  hash(Bin, ?SEED).

hash(Bin, Seed) ->
  io:format("nif not loaded derp!~n").
```

Benchmark!

Linked In Drivers

Nothing at all to do with the douche social network

Runtime Characteristics

- Input via command/control/call
- Output via messages
- Can use async thread pool
- Can manage it's own threads
- Timers / select for async scheduling

Driver Entry

```
static ErlDrvEntry queue_driver_entry = {
  NULL,                             /* init */
  init,
  stop,
  output,                           /* output */
  NULL,                             /* ready_input */
  NULL,                             /* ready_output */
  "queue_drv",                      /* the name of the driver */
  NULL,                             /* finish */
  NULL,                             /* handle */
  NULL,                             /* control */
  NULL,                             /* timeout */
  NULL,                             /* outputv */
  NULL,                             /* ready_async */
  NULL,                             /* flush */
  NULL,                             /* call */
  NULL,                             /* event */
  ERL_DRV_EXTENDED_MARKER,          /* ERL_DRV_EXTENDED_MARKER */
  ERL_DRV_EXTENDED_MAJOR_VERSION,   /* ERL_DRV_EXTENDED_MAJOR_VERSION */
  ERL_DRV_EXTENDED_MAJOR_VERSION,   /* ERL_DRV_EXTENDED_MINOR_VERSION */
  ERL_DRV_FLAG_USE_PORT_LOCKING     /* ERL_DRV_FLAGs */
};

DRIVER_INIT(queue_driver) {
  return &queue_driver_entry;
}
```

Control

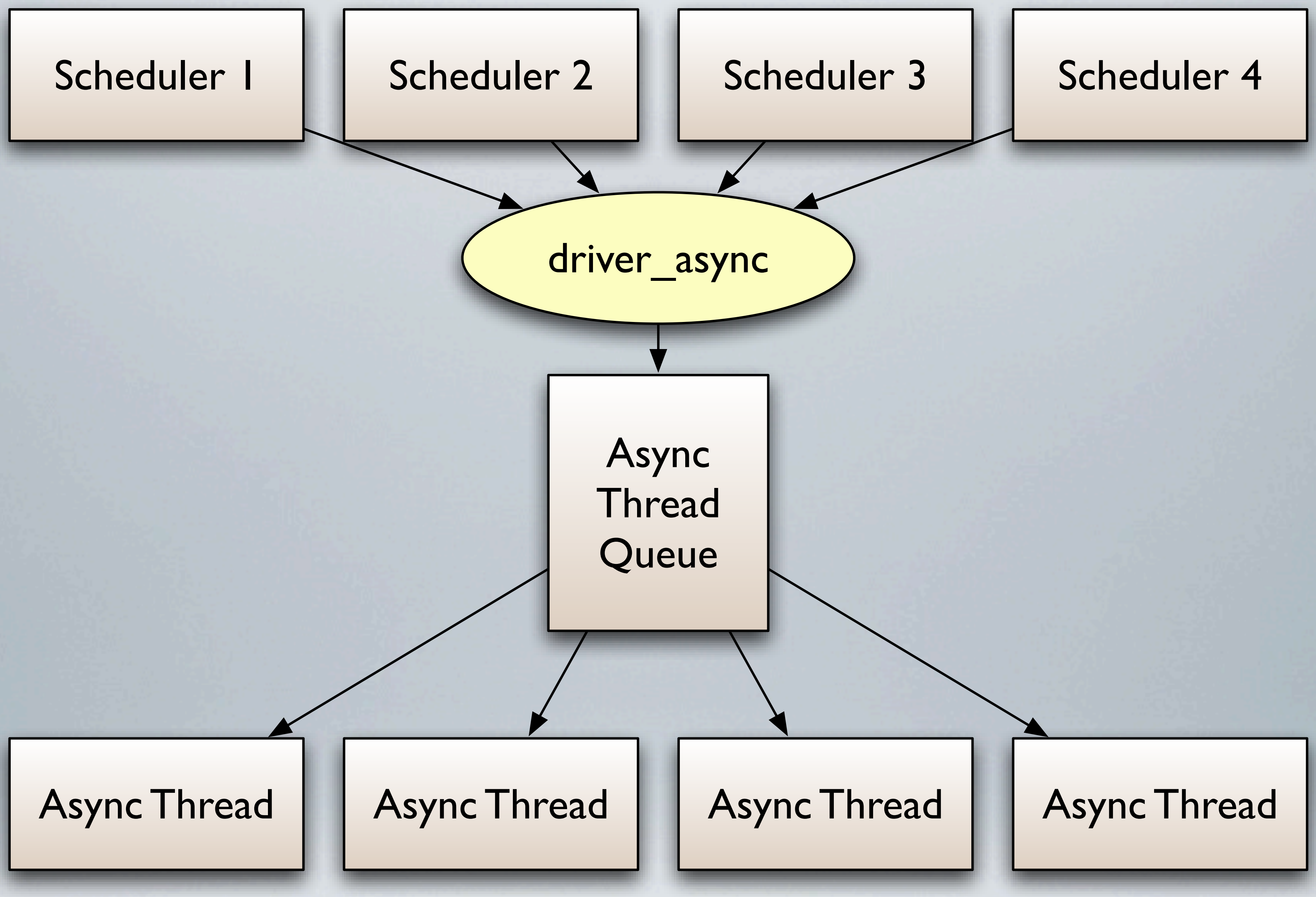
- Invokes the control callback in the driver entry
- Synchronous call
- Control is expected to return a value
- Can interface with any of the asynchronous facilities

Command

- Invokes the output callback in the driver entry
- Does not explicitly return a value
- Synchronous call
- Can interface with any of the asynchronous facilities

Async Thread Pool

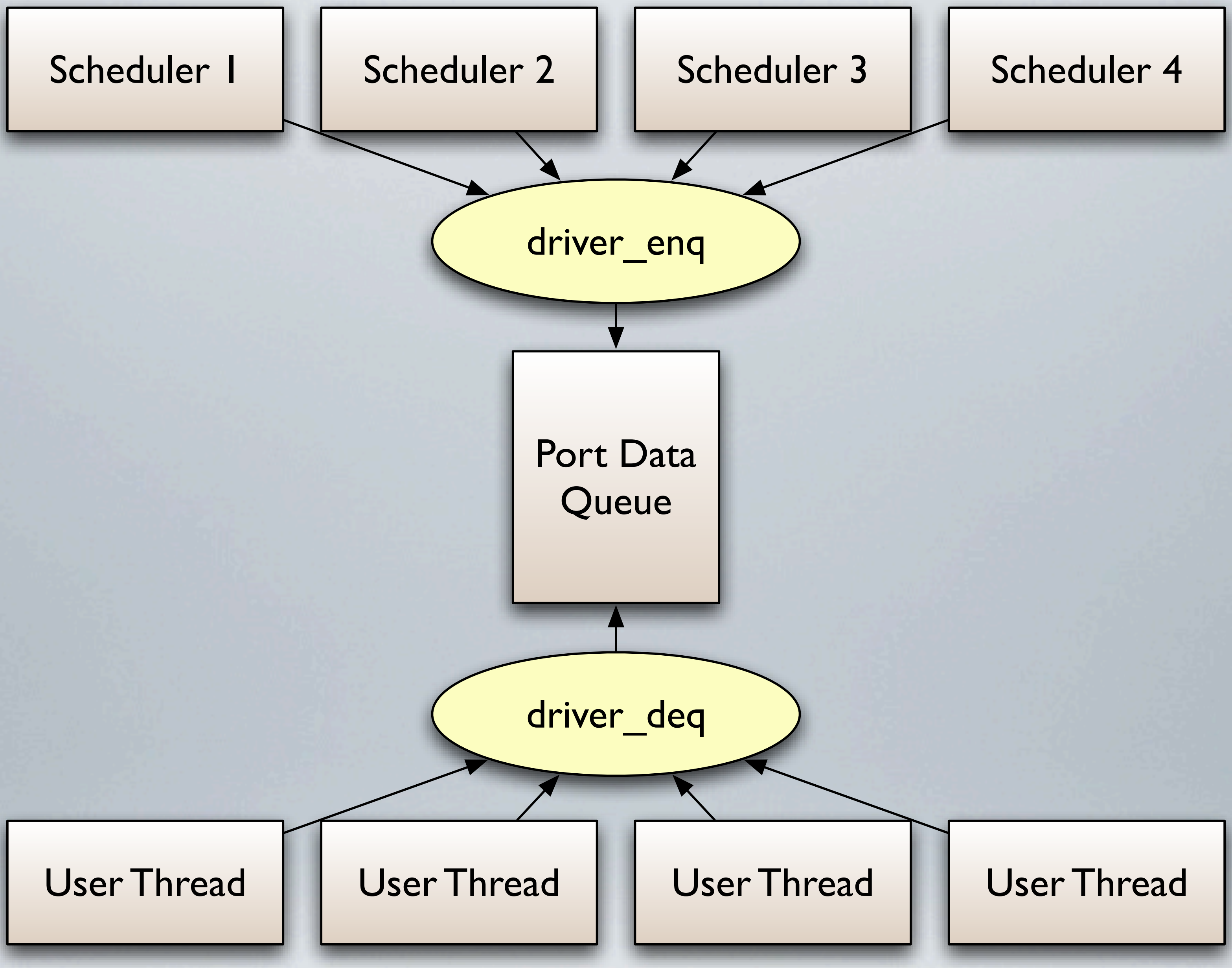
- driver_async submits work
- ready_async callback for when work is finished
- Can also signal via unix pipes
- thread affinity is available
- async callbacks must be threadsafe (duh)



Async Thread Pool

User Managed Threads

- erl_drv_thread_create - creates user managed threads
- driver_pdl_create - creates a lock for the port queue
- driver_enq - pushes data onto the port queue
- driver_deq - shifts data off the end of the port queue



Scheduler 1

Scheduler 2

Scheduler 3

Scheduler 4

driver_enq

Port Data Queue

driver_deq

User Thread

User Thread

User Thread

User Thread

User Threads

Event Based Async

- driver_select - wraps a kernel call to kqueue, epoll, or select
- ready_input - callback when the filehandle is ready to read
- ready_output - callback when ready to write
- ErlDrvEvent - type wrapper for the FD

Choosing An Async Model

- Erlang managed threads are easier
  - Thread pool needs to be configured
  - Stack size needs to be configured
- User managed threads can be customized
  - Higher complexity

Libgd

Case Study

Image Library

- Resize and crop images
- Long running operations
- Async linked-in drivers!

Async Dispatch

```c
static void output(ErlDrvData handle, char *buff, int len) {
  Gd *gd = (Gd*)handle;
  char cmd = buff[0];
  char *data = &buff[1];
  int size = len-1;
  asyncFun function;

  Cmd *command = driver_alloc(sizeof(Cmd) + size);
  command->gd = gd;
  command->size = size;
  memcpy(command->data, data, size);

  switch(cmd) {
  case SIZE:
    function = get_size;
    break;
...
  }

  driver_async(gd->port, NULL, function, command, driver_free);
}
```

Demo!

```erlang
-module(gd_test).

-export([test/0]).

test() ->
  {ok, Bin} = file:read_file("priv/riak_logo.jpg"),
  {ok, Gd} = gd:read(Bin, "image/jpg"),
  gd:resize(Gd, 700),
  {ok, Bin2} = gd:blob(Gd, 100),
  file:write_file("priv/riak_resized.jpg", Bin2).
```

TL;DR

- NIF for simple CPU bound operations
- Linked In Drivers for IO and fine grained concurrency

Linkz

- hash NIF - http://github.com/cliffmoon/ef_examples
- gd wrapper - http://github.com/cliffmoon/ef_gd
- Thanks, scros.