Threads are an illusion

A guide to thinking about Program Design with Boost.Asio

When we hear “concurrency”...

When we hear “concurrency”...

... we usually think threads

When we hear “concurrency”...

... we usually think threads

but threads are an illusion

When we hear “concurrency”...

... we usually think threads

but threads are an ~~illusion~~ abstraction

Threads are an abstraction

- A state machine, implemented using
  - stack frame +
  - thread locals +
  - registers +
  - sequence of instructions +
  - a program counter
- Threads are a *general purpose* abstraction

If we know our problem domain...

If we know our problem domain...

... we can do better

If we know our problem domain...

... we can do better

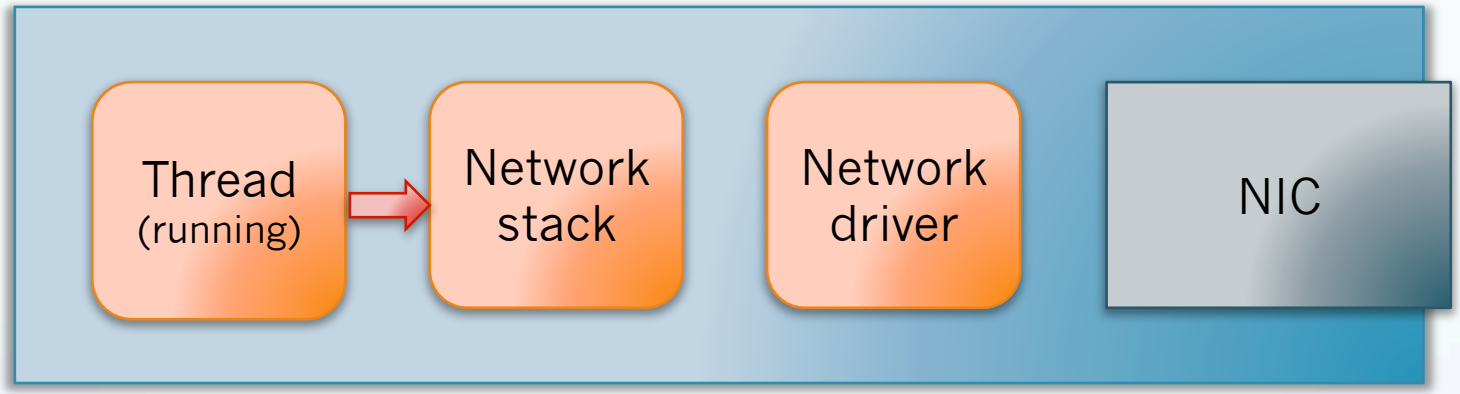
Networking is a domain where it is easy to do better

Thread per connection

```
array<unsigned char, 1024> data;
while (!ec)
{
  size_t n = my_socket.read_some(buffer(data), ec);
  // Do something with data ...
}
```

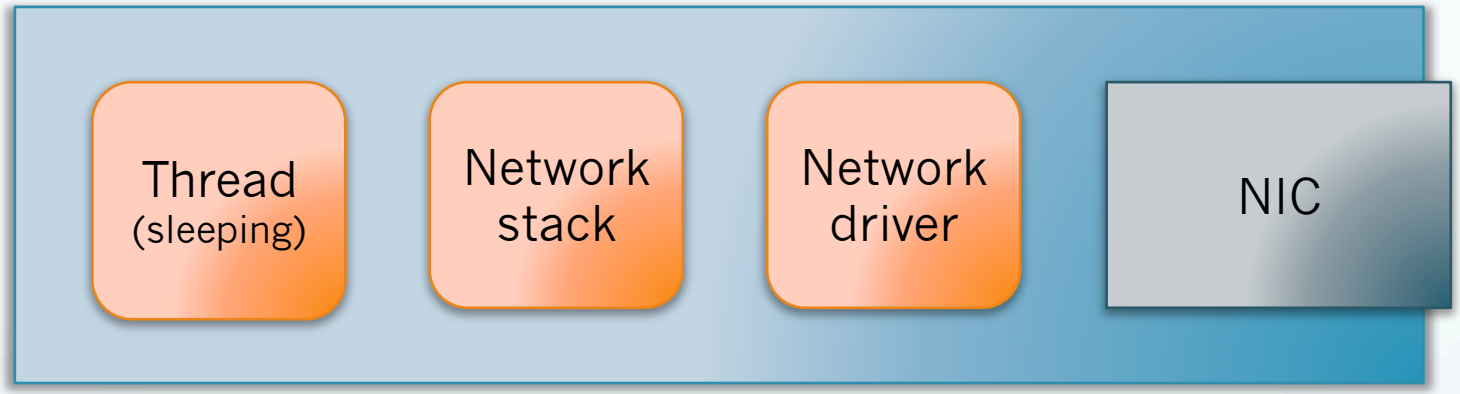
Thread per connection

Thread: `size_t n = my_socket.read_some(buffer(data), ec);`



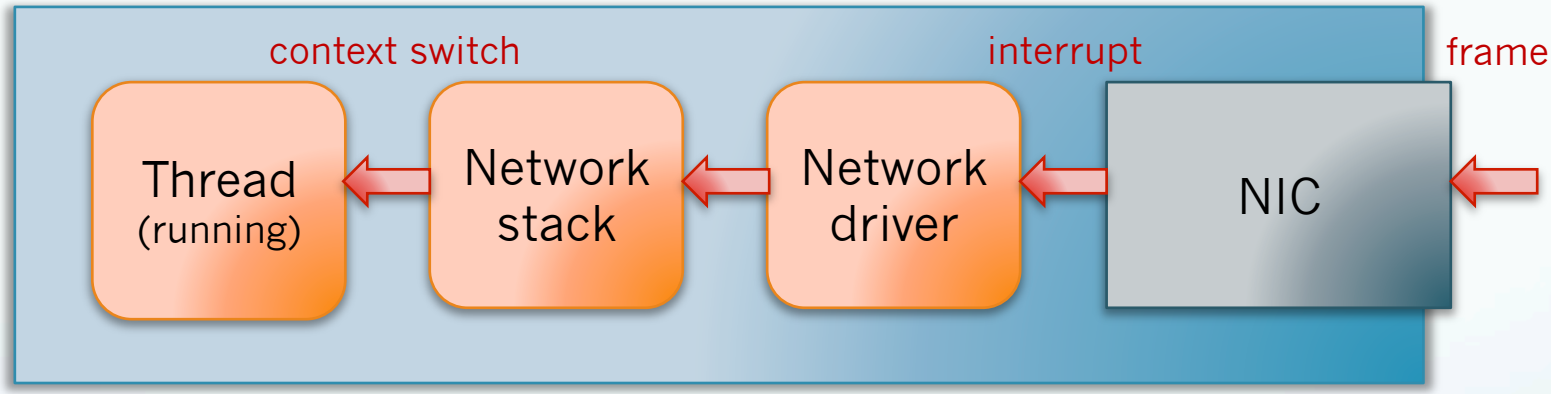
Thread per connection

Thread: `size_t n = my_socket.read_some(buffer(data), ec);`



Thread per connection

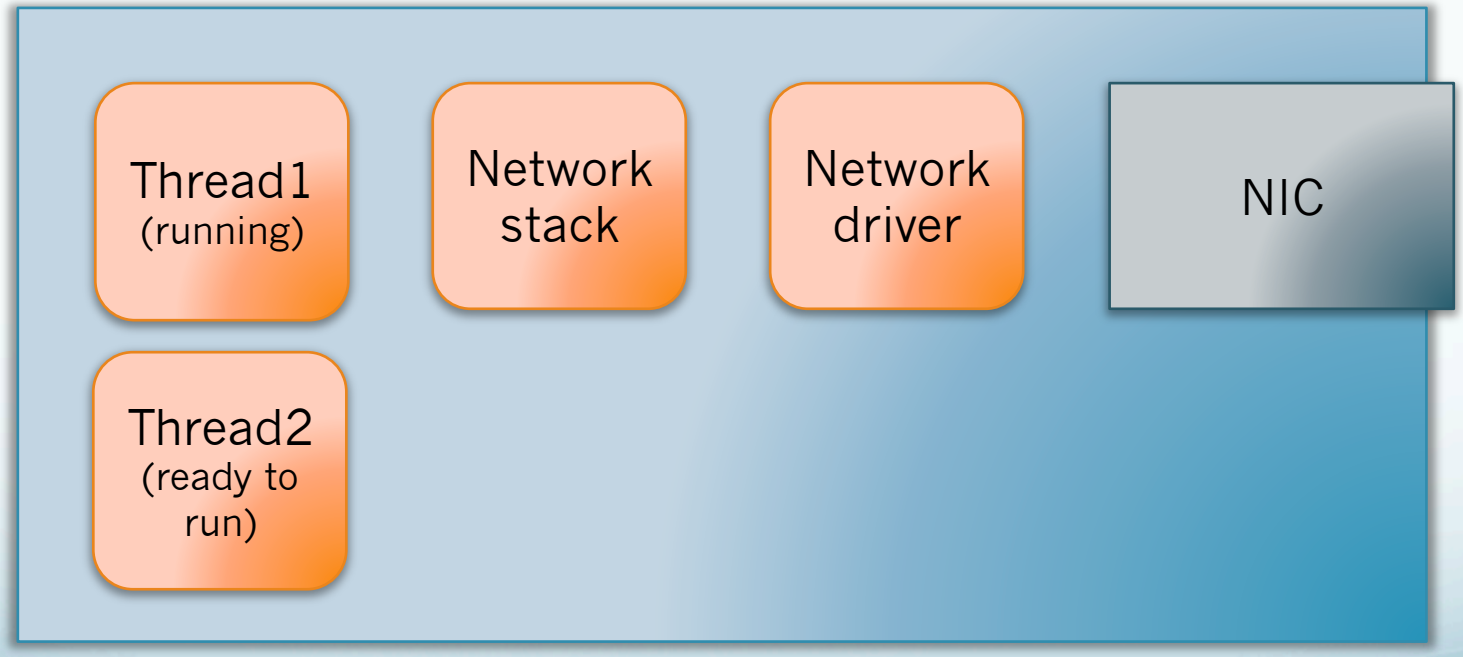
Thread: `size_t n = my_socket.read_some(buffer(data), ec);`



What if connections > number of CPUs?

Thread 1: `size_t n = my_socket.read_some(buffer(data), ec);`

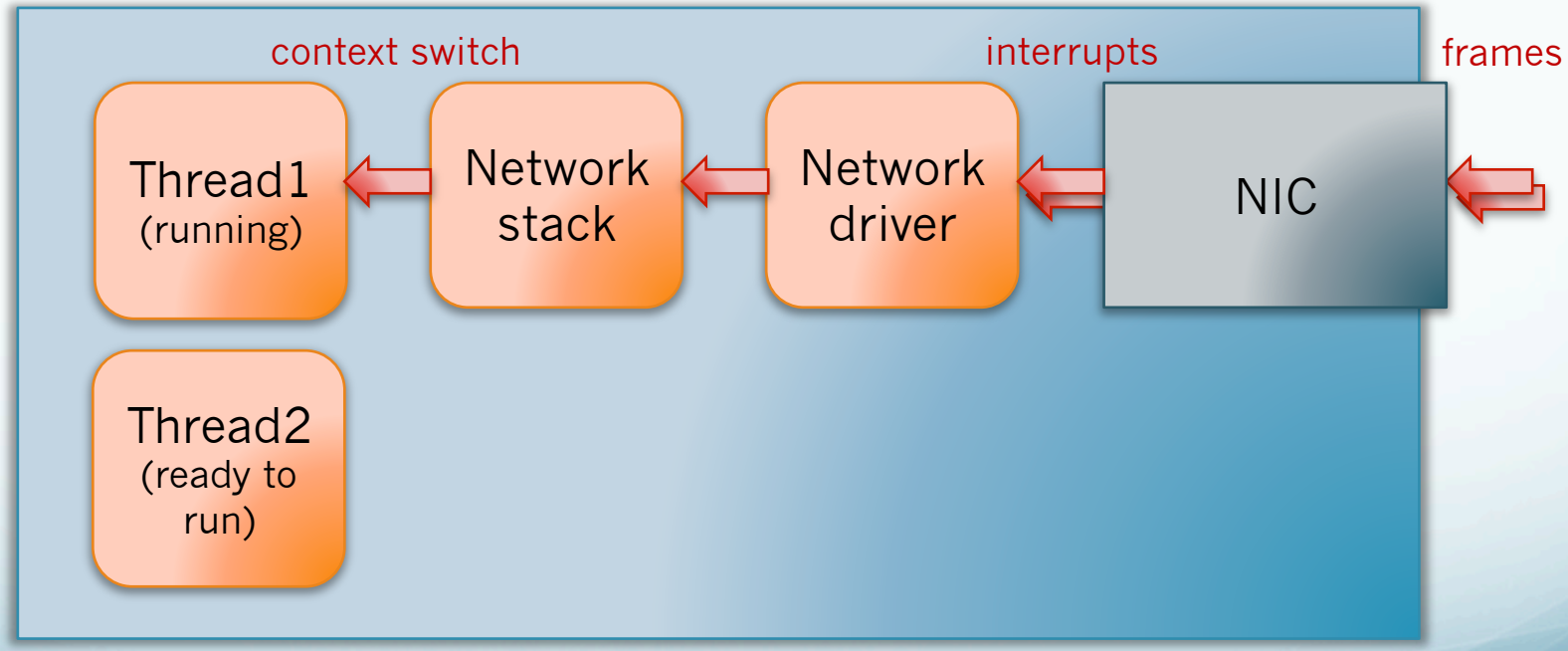
Thread 2: `size_t n = my_socket.read_some(buffer(data), ec);`



What if connections > number of CPUs?

Thread 1: `size_t n = my_socket.read_some(buffer(data), ec);`

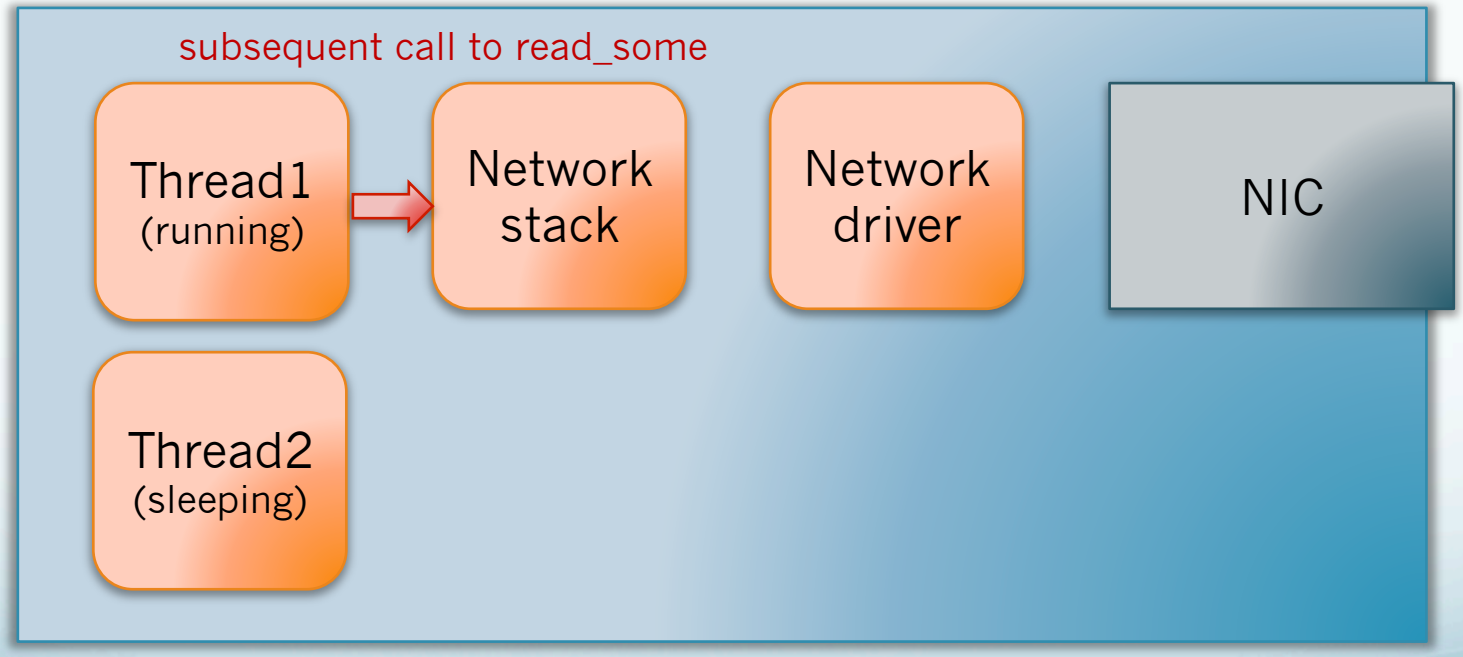
Thread 2: `size_t n = my_socket.read_some(buffer(data), ec);`



What if connections > number of CPUs?

Thread 1: `size_t n = my_socket.read_some(buffer(data), ec);`

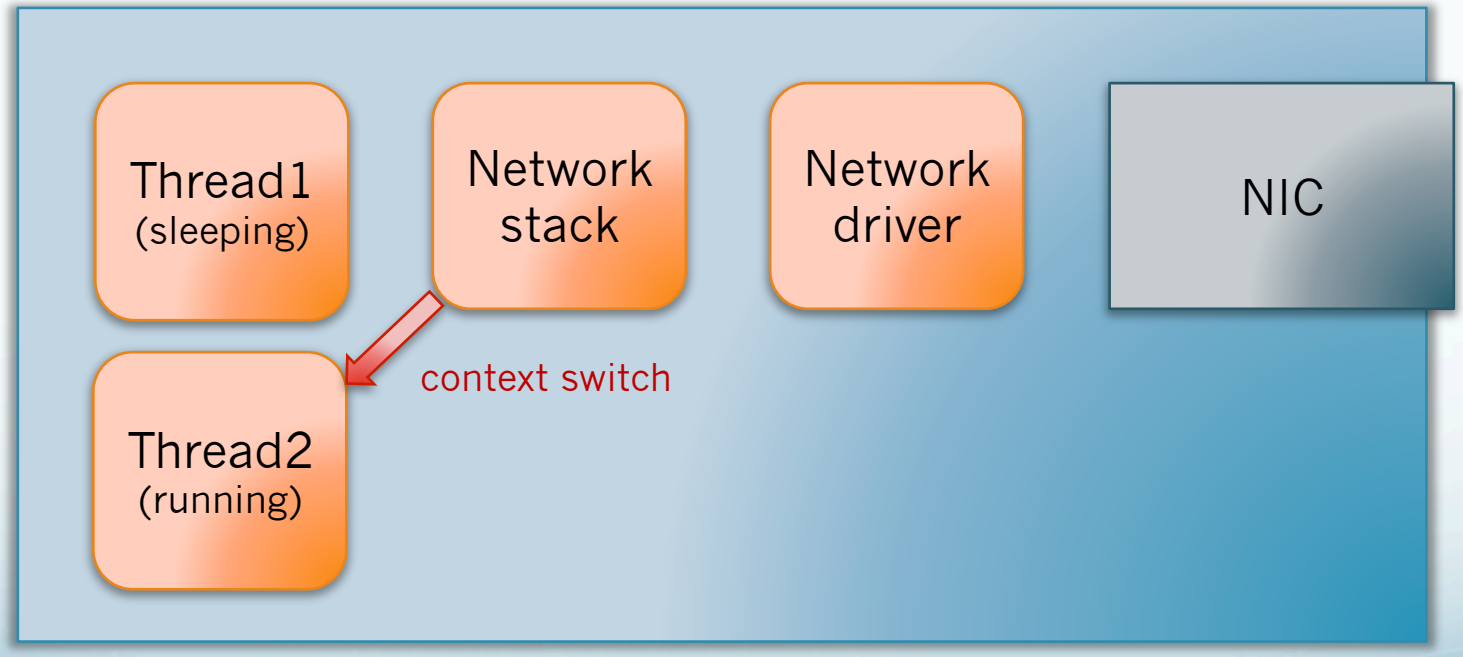
Thread 2: `size_t n = my_socket.read_some(buffer(data), ec);`



What if connections > number of CPUs?

Thread 1: `size_t n = my_socket.read_some(buffer(data), ec);`

Thread 2: `size_t n = my_socket.read_some(buffer(data), ec);`



What if connections > number of CPUs?

- Really a queuing problem
- Cost of running each task includes a context switch
- General problem is N ready connections > #CPUs
- Both throughput and latency are impacted

We could amortise costs

- Process network events in a batch
- One context switch per batch

Recall that threads are:

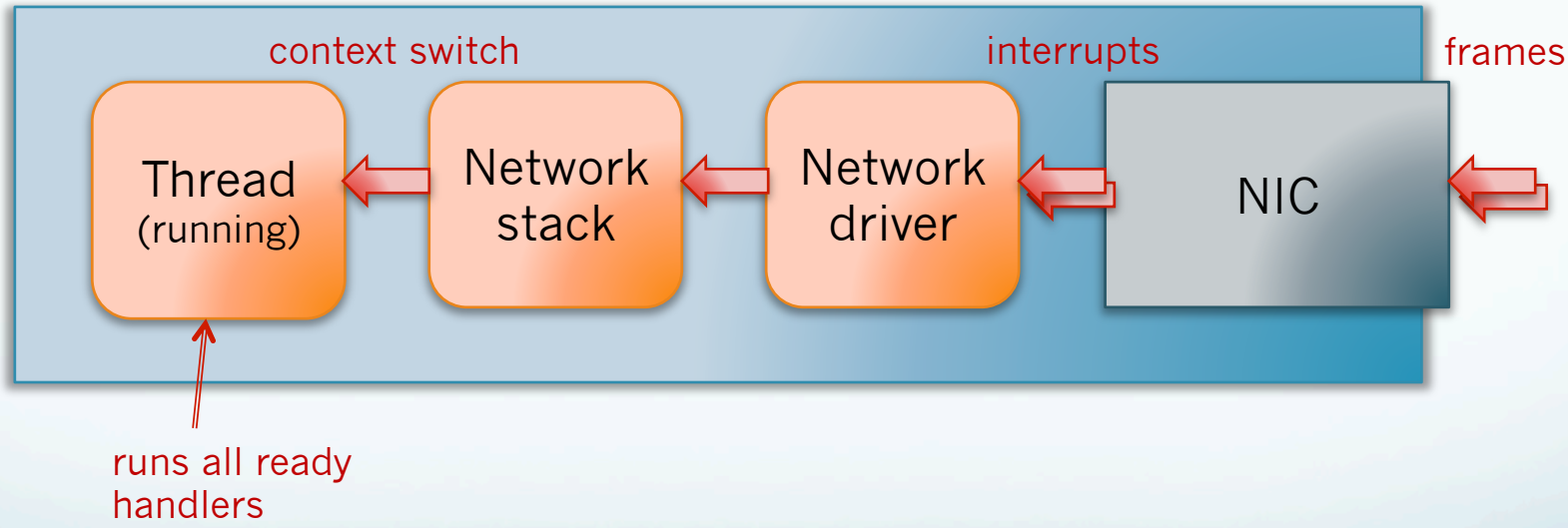
- A *state machine*, implemented using
  - stack frame +
  - thread locals +
  - registers +
  - sequence of instructions +
  - a program counter
- There are other ways of representing state machines...

Single-threaded asynchronous I/O

```
class session
{
  tcp::socket socket_;
  array<unsigned char, 1024> data_;
  ...
  void do_read()
  {
    socket_.async_read_some(buffer(data_),
        [this](std::error_code ec, std::size_t n)
        {
          // Do something with data ...
          do_read();
        });
  }
};

...

io_service.run();
```

Single-threaded asynchronous I/O

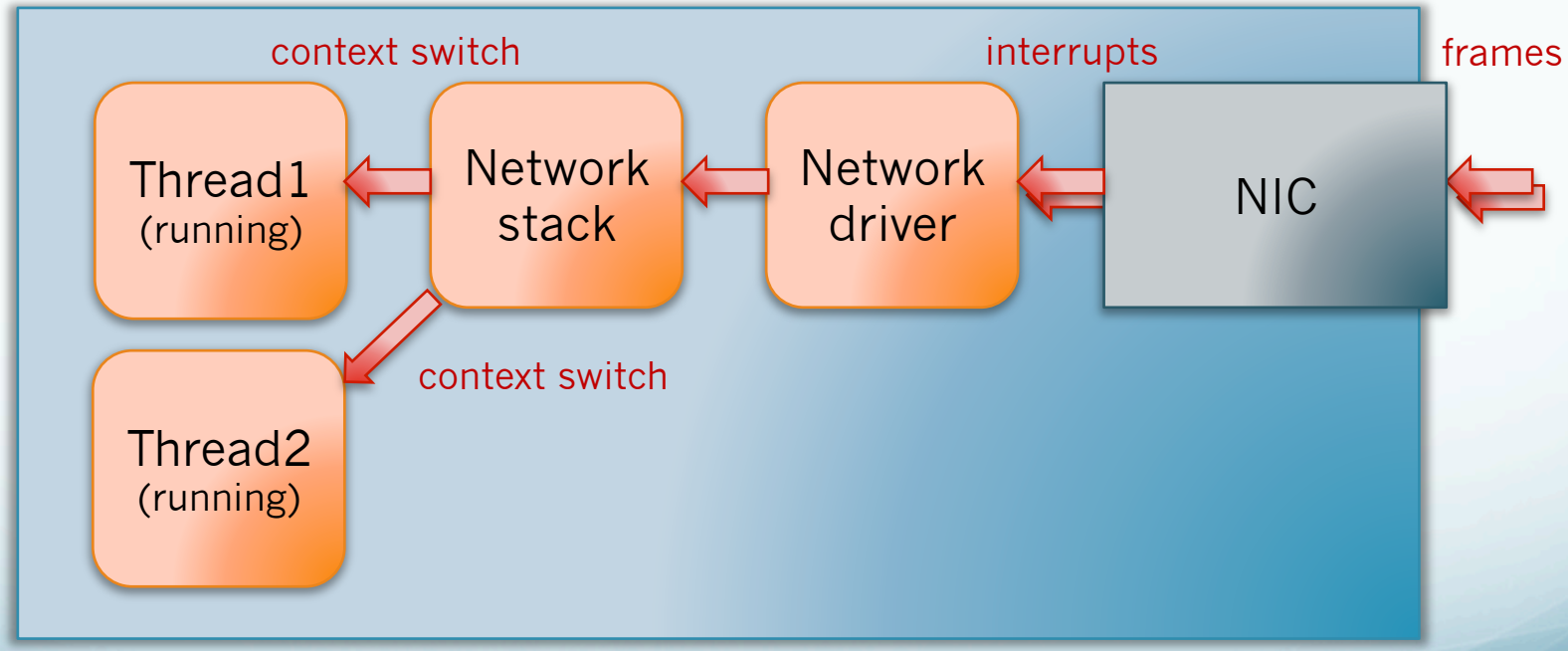
Thread: `io_service.run();`



What if connections ≤ number of CPUs?

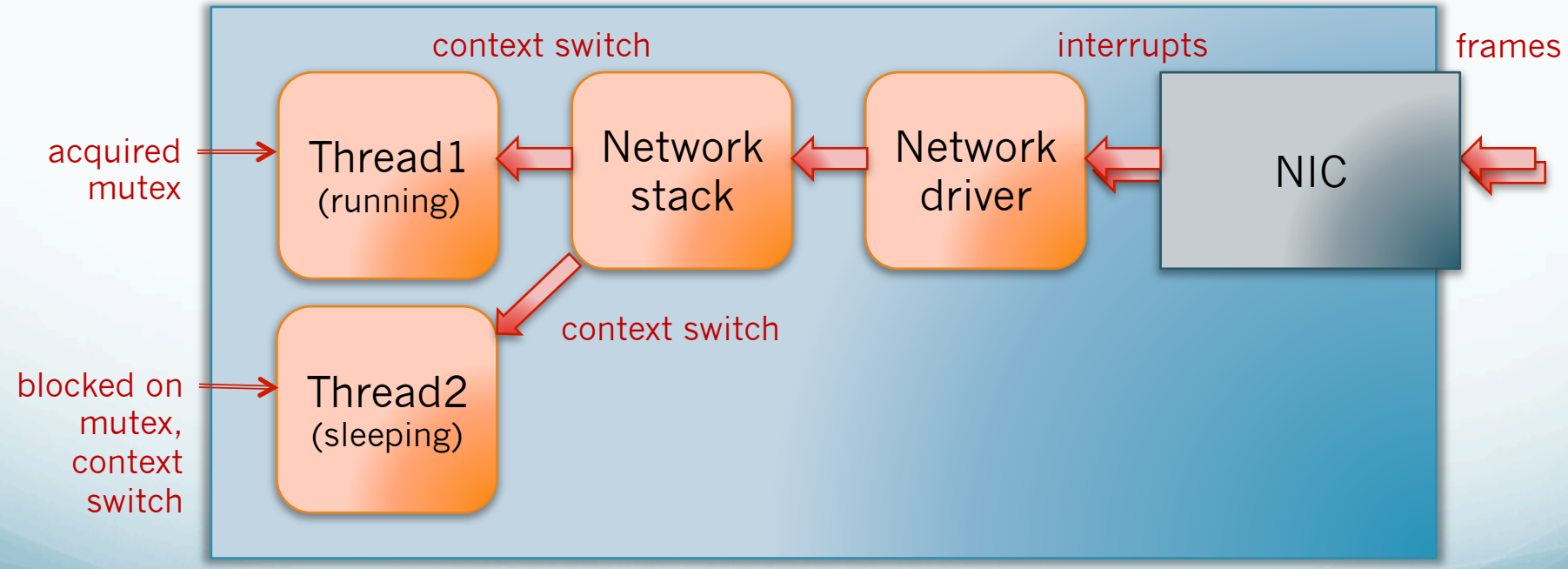
Thread 1: `size_t n = my_socket.read_some(buffer(data), ec);`

Thread 2: `size_t n = my_socket.read_some(buffer(data), ec);`



What if connections ≤ number of CPUs?

- Consider shared resources
  - Are connections truly independent or must they access or manipulate shared data



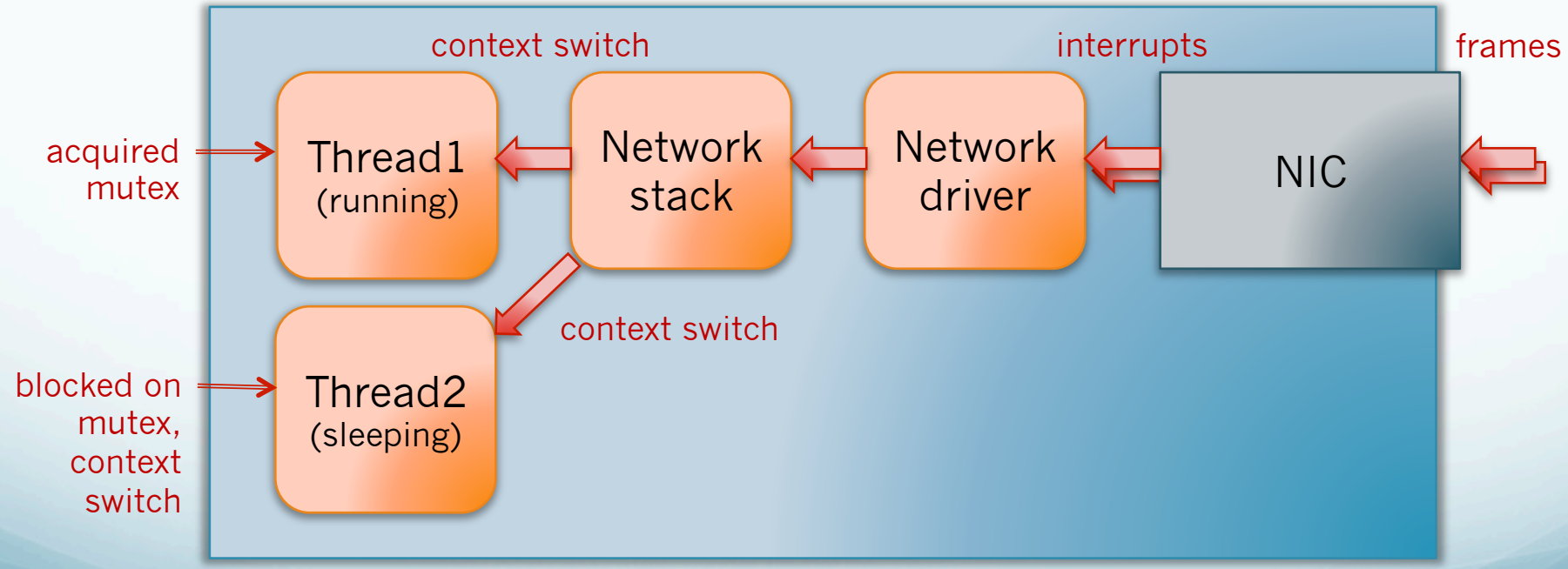
With single-threaded asynchronous I/O...

```
class session
{
  tcp::socket socket_;
  array<unsigned char, 1024> data_;
  ...
  void do_read()
  {
    socket_.async_read_some(buffer(data_),
        [this](std::error_code ec, std::size_t n)
        {
          // Do something with data ...
          do_read();
        });
  }
};

...

io_service.run();
```

Only one context switch per batch of events

What if connections ≤ number of CPUs?

- Consider shared resources
  - Are connections implemented using multiple threads?



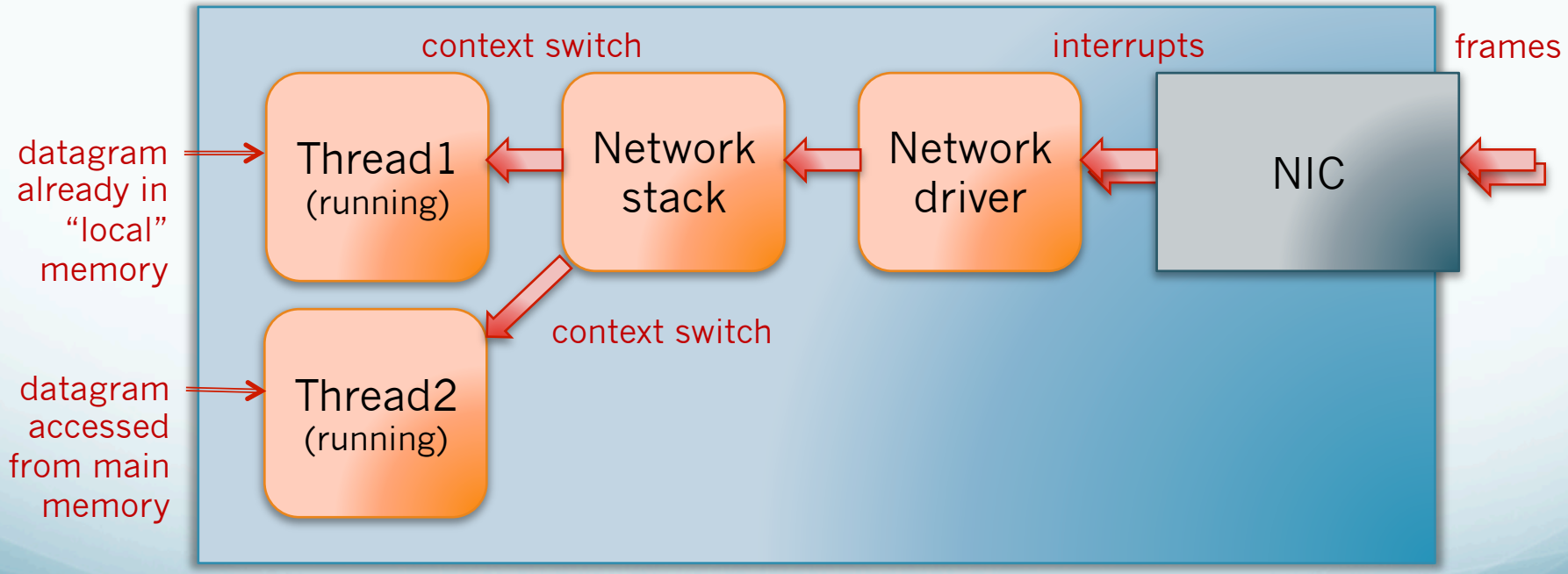
Single-threaded asynchronous I/O

```
class session
{
  tcp::socket socket_;
  array<unsigned char, 1024> data_;
  ...
  void do_read()
  {
    socket_.async_read_some(buffer(data_),
        [this](std::error_code ec, std::size_t n)
        {
          // Do something with data ...
          do_read();
        });
  }
};

...

io_service.run();
```


No mutex required because all state machines run on the one "real" thread

What if connections ≤ number of CPUs?

- Consider interrupts:
  - NIC may be closer to one physical CPU than another
  - Pay attention to where your interrupts go



With single-threaded asynchronous I/O...

```
class session
{
  tcp::socket socket_;
  array<unsigned char, 1024> data_;
  ...
  void do_read()
  {
    socket_.async_read_some(buffer(data_),
        [this](std::error_code ec, std::size_t n)
        {
          // Do something with data ...
          do_read();
        });
  }
};

...
pthread_np_setaffinity(...);
io_service.run();
```

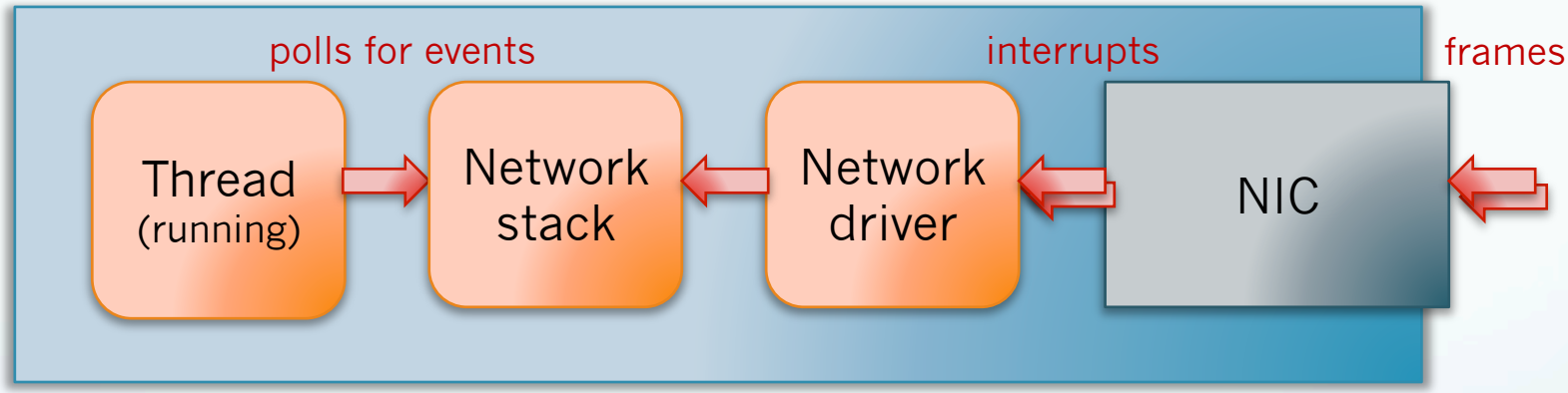
All state machines run on a specific CPU

Bonus: no context switches!

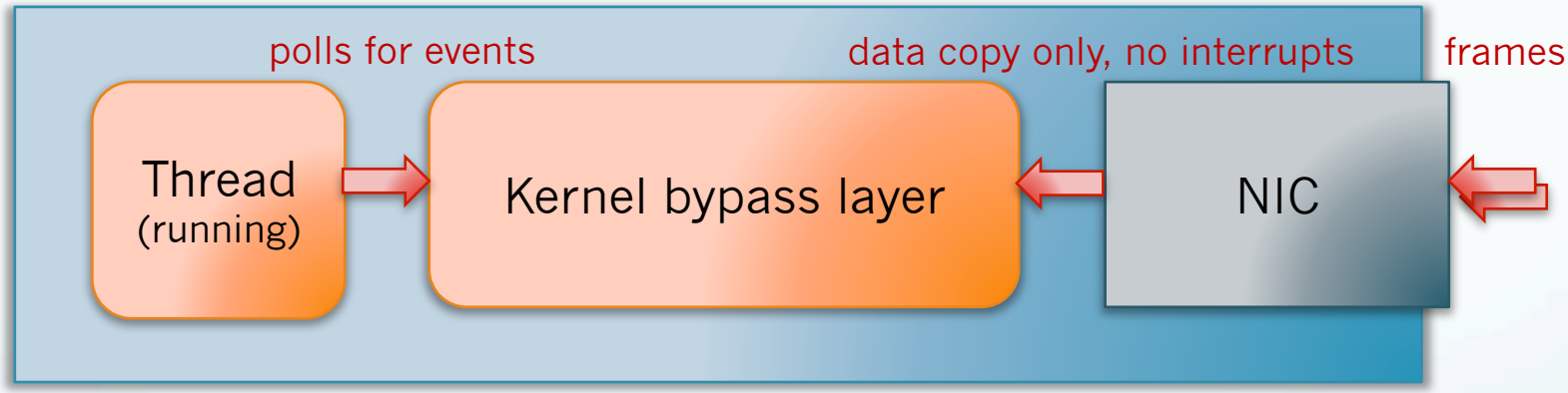
```
class session
{
  tcp::socket socket_;
  array<unsigned char, 1024> data_;
  ...
  void do_read()
  {
    socket_.async_read_some(buffer(data_),
        [this](std::error_code ec, std::size_t n)
        {
          // Do something with data ...
          do_read();
        });
  }
};

...
pthread_np_setaffinity(...);
for (;;) io_service.poll();
```

Thread polls for network events and never blocks

Bonus: no context switches!



Bonus: no context switches!



Design recommendations

Example Scenario

- TCP proxy:
  1. Client establishes connection to proxy
  2. Proxy establishes connection to server
  3. Proxy simultaneously:
     a. receives data from client and sends to server
     b. receives data from server and sends to client

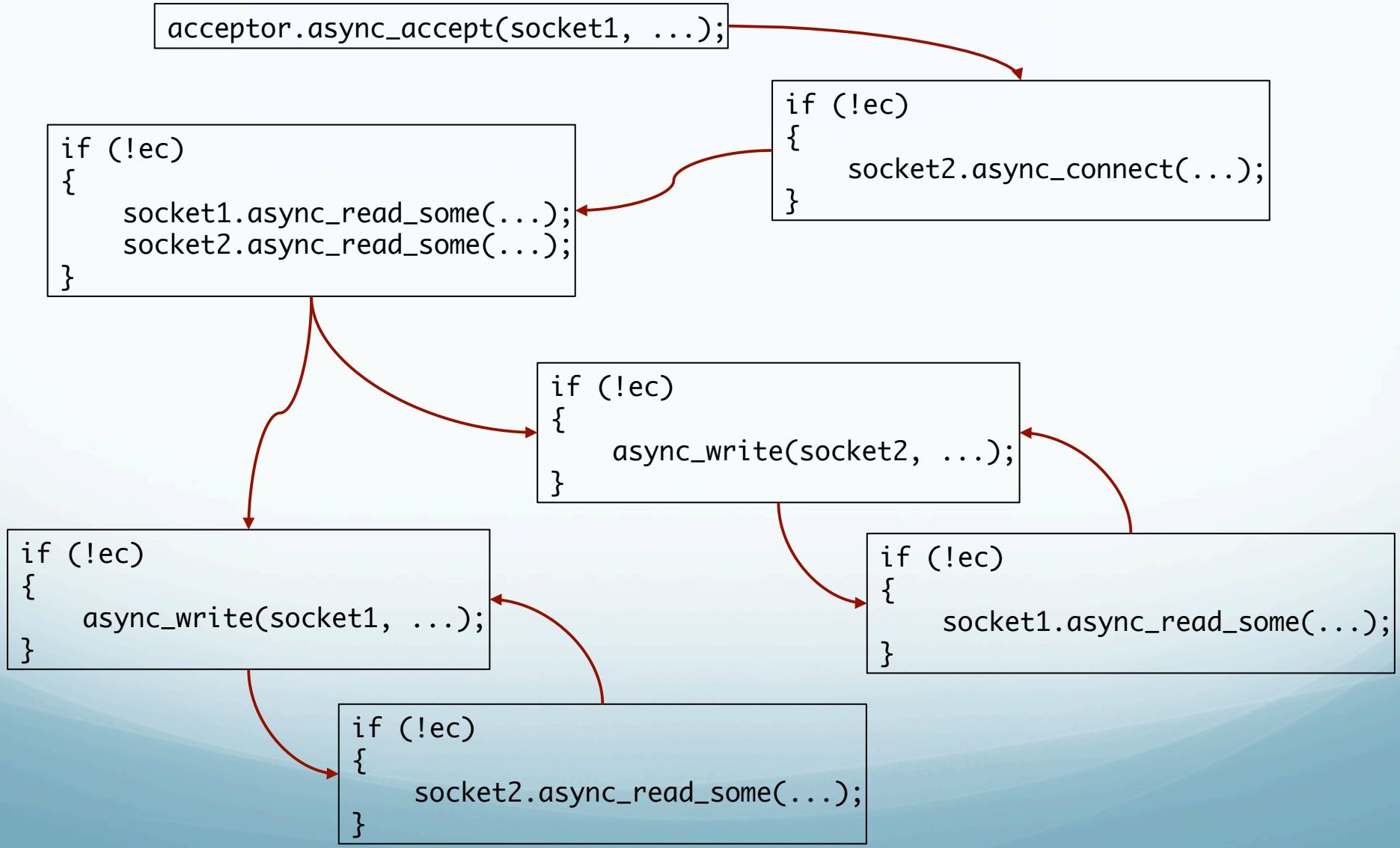
Recommendations

- Spectrum of approaches, in order of preference:
  1. Single-threaded
     1.1. Use threads for long-running tasks
  2. Multiple io_services, one thread each
  3. One io_service, many threads

1. Single-threaded

- Preferred starting point
- Keep handlers short and non-blocking

1. Single-threaded



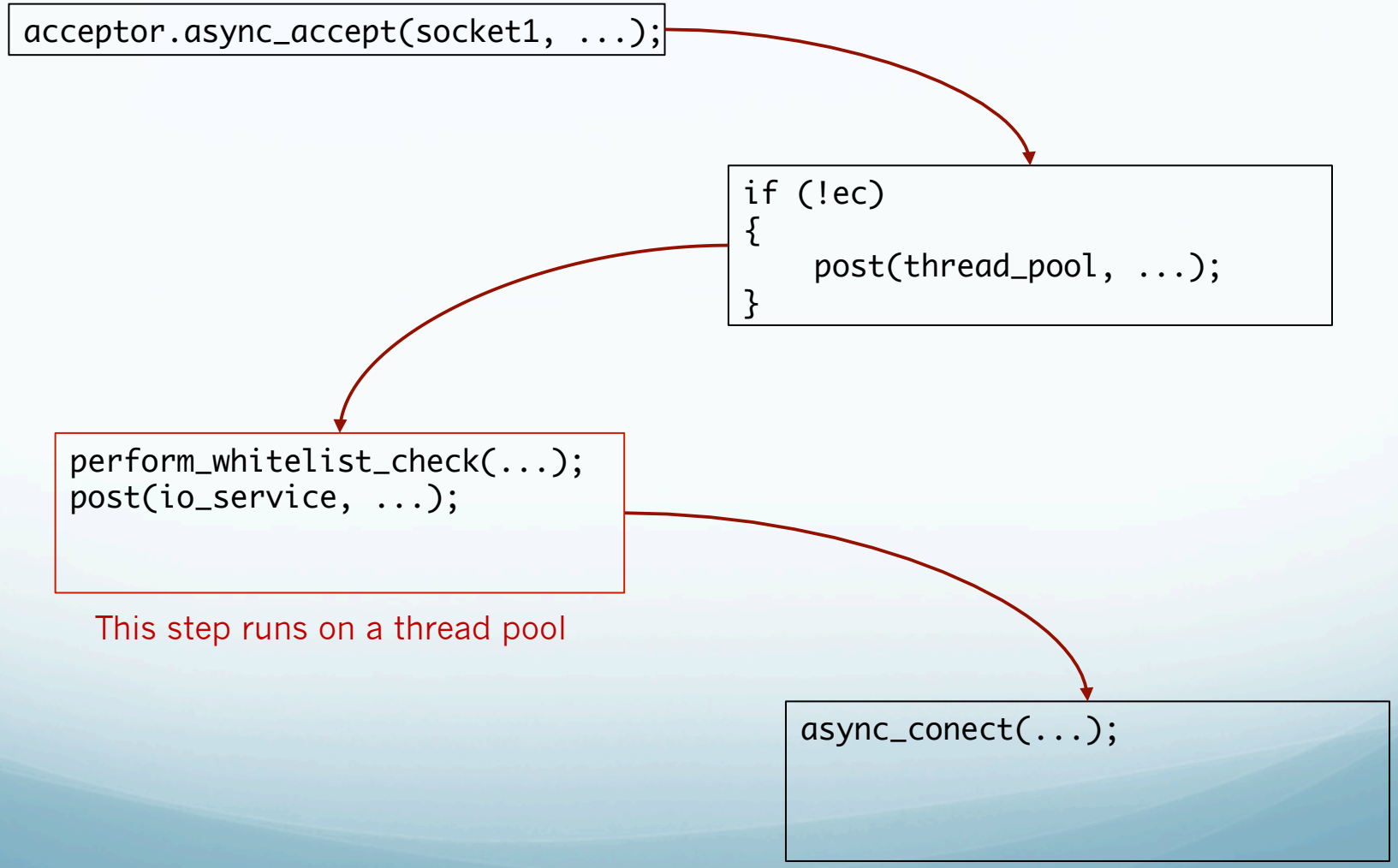
1.1. Use threads for long running tasks

- Networking state machine stays in network thread
  - i.e. networking is still single-threaded
- Pass work to background thread
- Pass result back to network thread

Modified example scenario

- TCP proxy:
  1. Client establishes connection to proxy
  2. Proxy checks client IP against whitelist database
  3. Proxy establishes connection to server
  4. Proxy simultaneously:
     a. receives data from client and sends to server
     b. receives data from server and sends to client

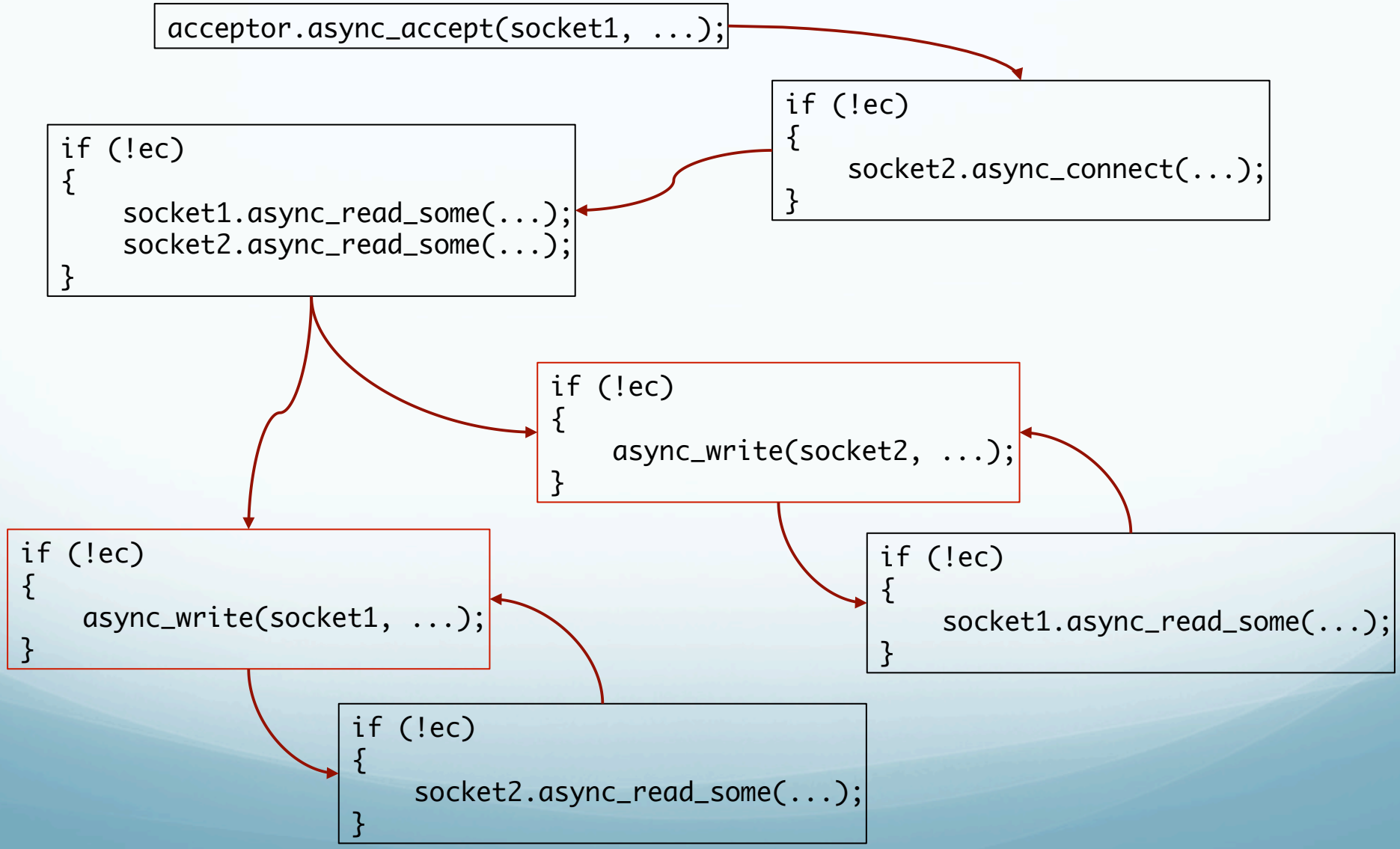
1.1. Use threads for long running tasks



2. Multiple io_services, one thread each

- Networking state machine stays in each object's “home” thread
- Keep handlers short and non-blocking
- Objects communicate via “message passing”

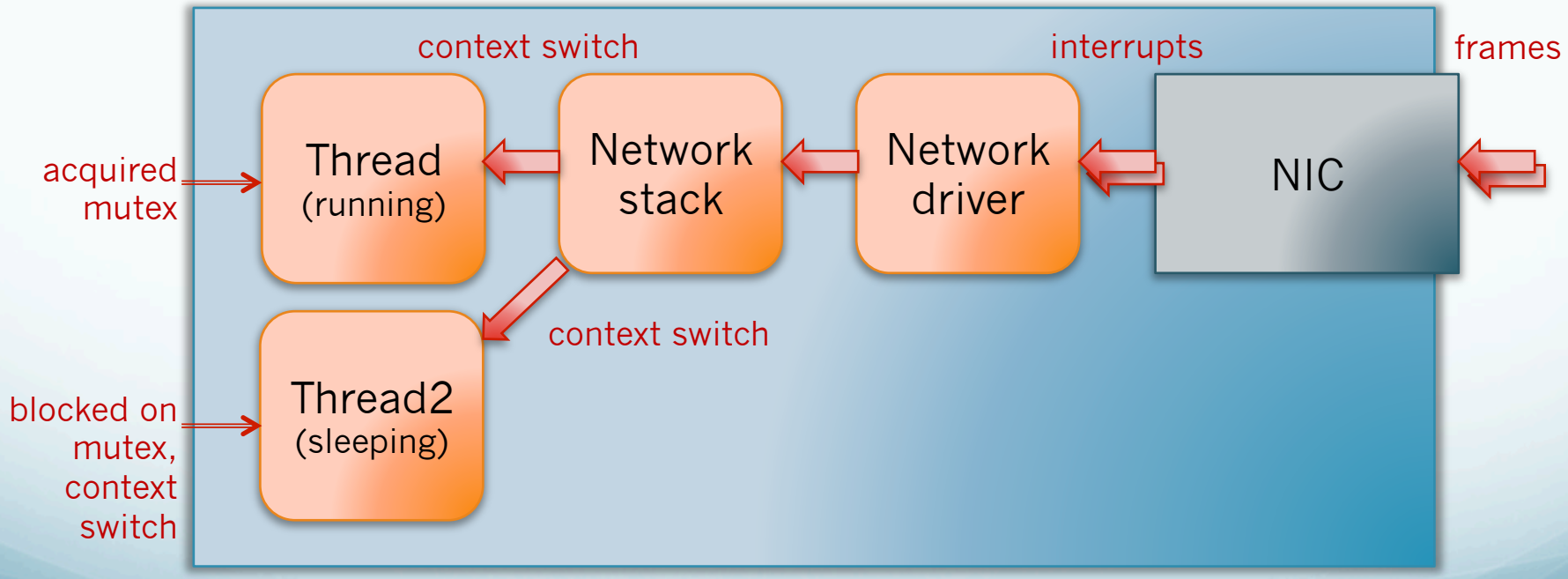
2. Multiple io_services, one thread each



Multithreaded asynchronous I/O

Thread1: `io_service.run();`

Thread2: `io_service.run();`



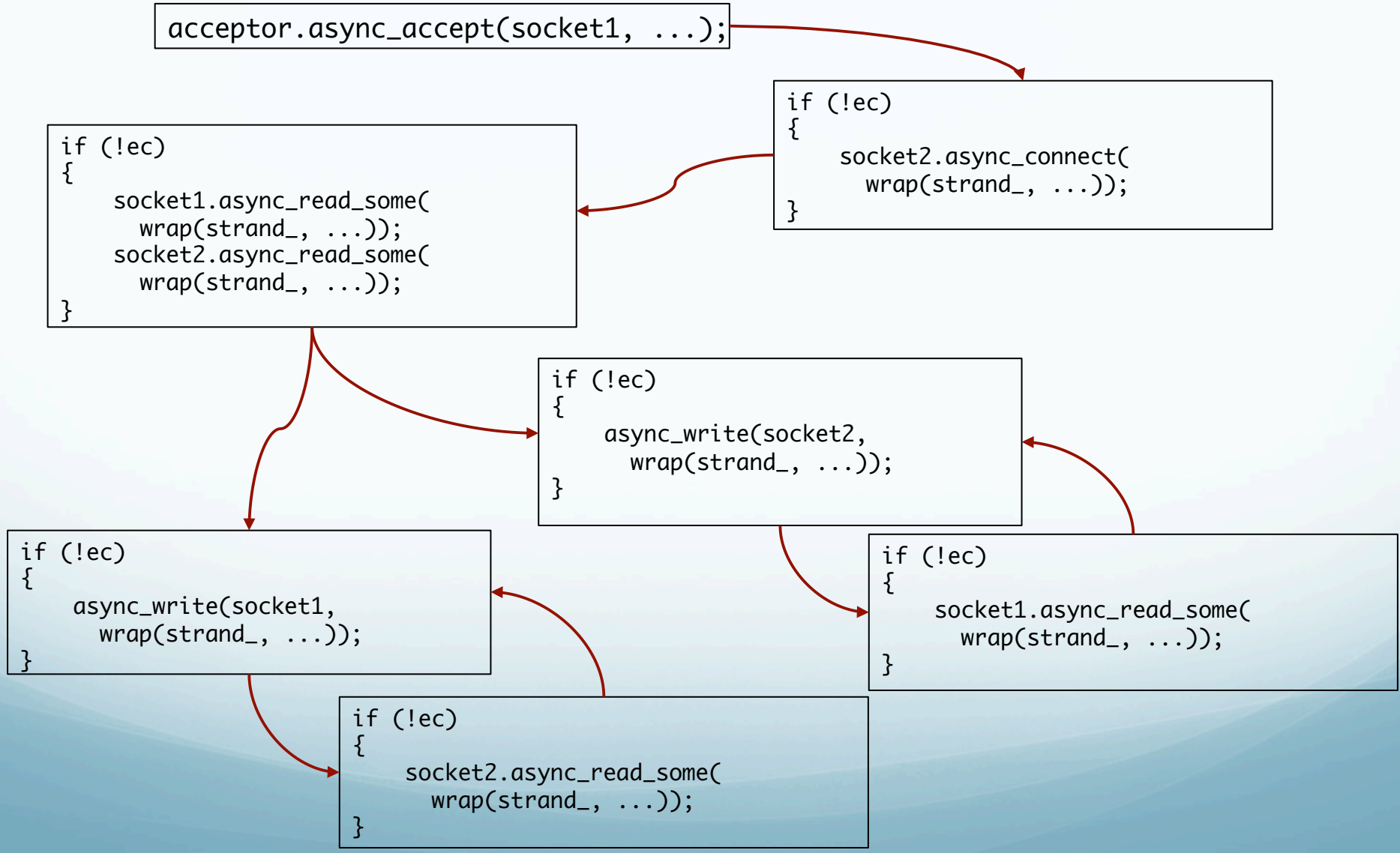
2. Multiple io_services, one thread each

- One io_service, multiple threads:
- Handlers can be called on any thread
- *Execute state machines in strands*
- Objects communicate via “message passing”

Strands

- Ensures:
  - non-concurrent execution of handlers
  - FIFO execution of handlers

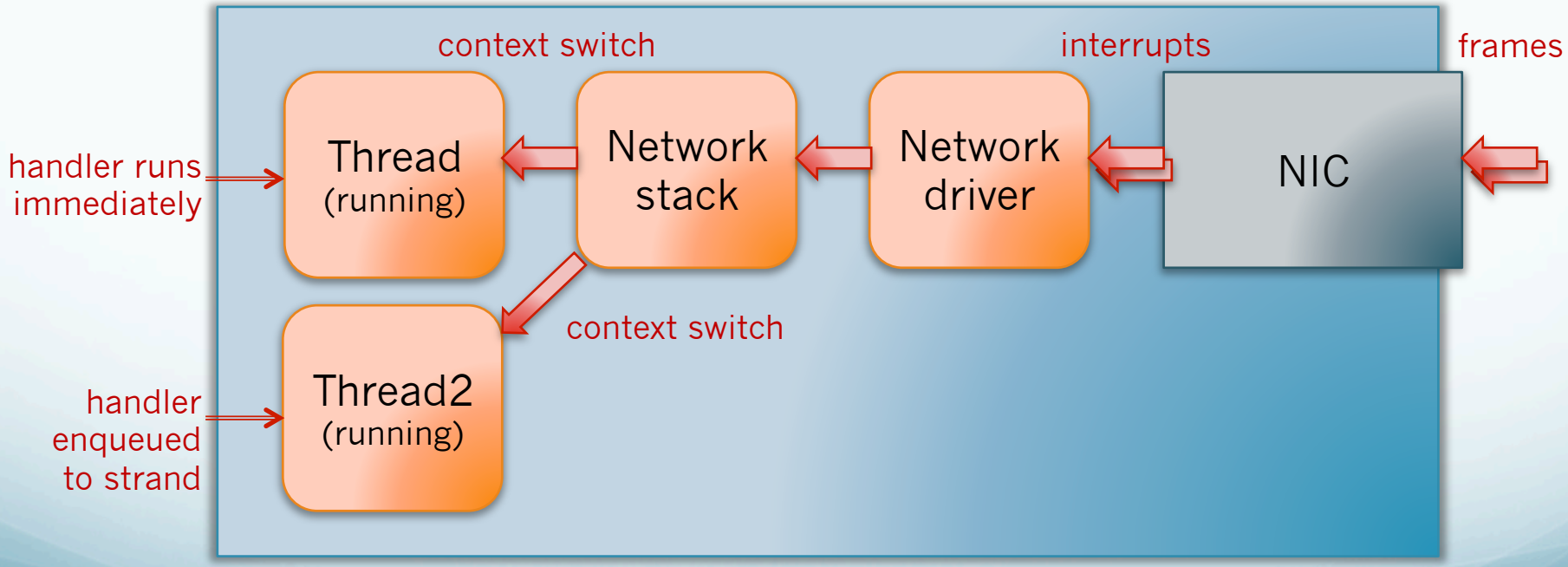
2. Multiple io_services, one thread each



Multithreaded asynchronous I/O

Thread1: `io_service.run();`

Thread2: `io_service.run();`



Recommendations

- Spectrum of approaches, in order of preference:
  1. Single-threaded
     1.1. Use threads for long-running tasks
  2. Multiple io_services, one thread each
  3. One io_service, many threads
- Whichever approach is used, pay attention to interrupts and CPU affinity
  - May provide an easy win, with no source changes

Questions?