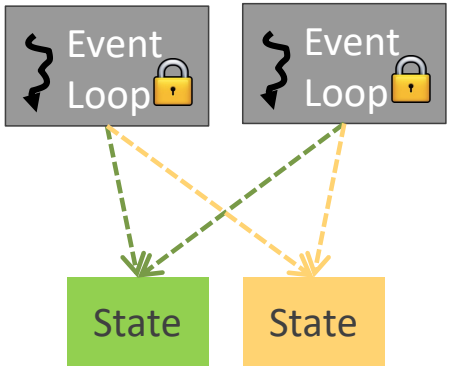
School of Computer Science & Engineering

COMP9242 Advanced Operating Systems

2025 T3 Week 02 Part 2

Threads *vs* or *and* Events?

Gernot Heiser



Copyright Notice

Today's Lecture

- Present classical pitches in favour of Events and Threads
- Present an alternative design
- Summarise the models

Why Threads Are A Bad Idea (for most purposes)

John Ousterhout

Sun Microsystems Laboratories

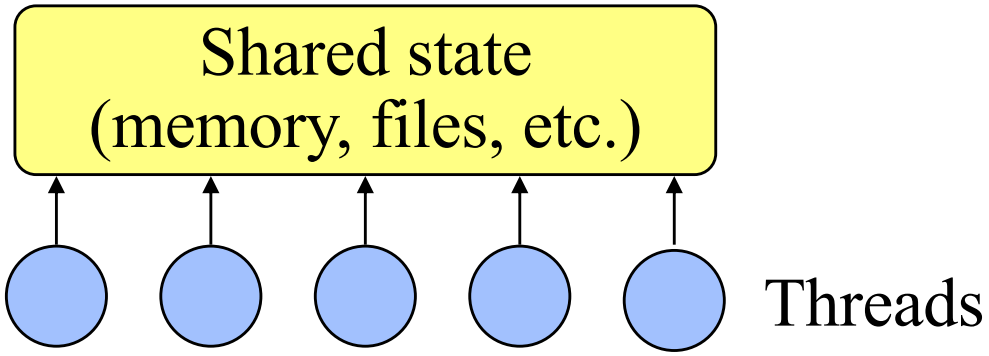
`john.ousterhout@eng.sun.com`
`http://www.sunlabs.com/~ouster`

Slide set courtesy of John Ousterhout, used with permission

Introduction

- **Threads:**
  - Grew up in OS world (processes).
  - Evolved into user-level tool.
  - Proposed as solution for a variety of problems.
  - Every programmer should be a threads programmer?
- **Problem: threads are very hard to program.**
- **Alternative: events.**
- **Claims:**
  - For most purposes proposed for threads, events are better.
  - Threads should be used only when true CPU concurrency is needed.

What Are Threads?

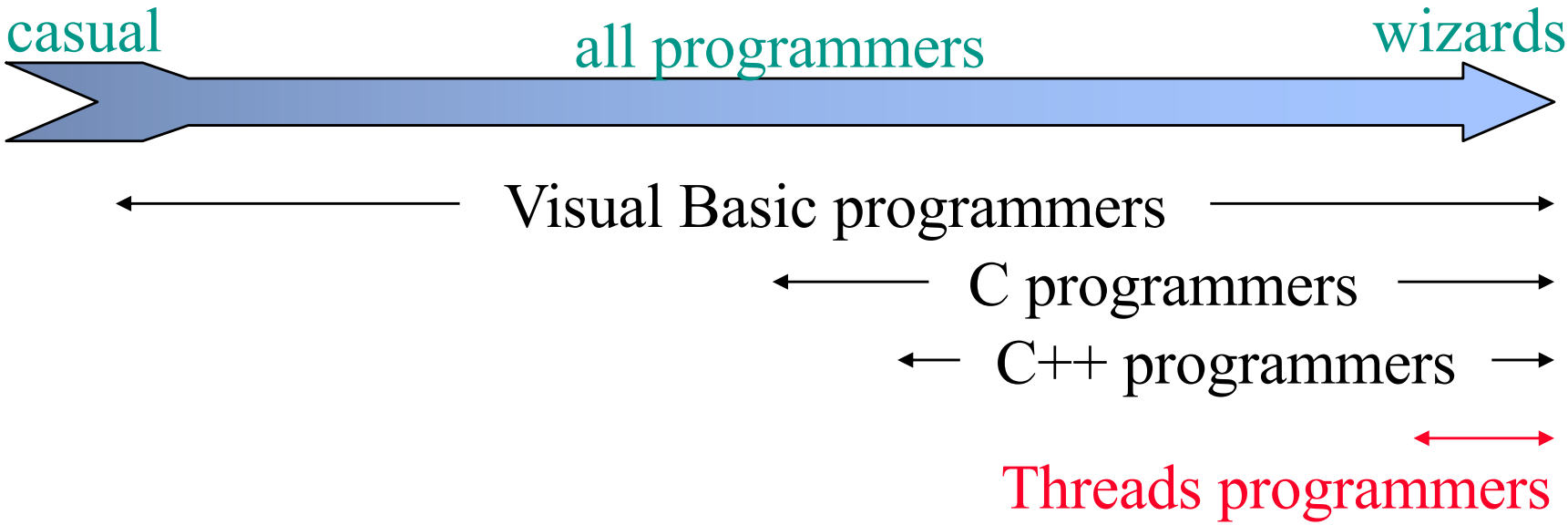


- **General-purpose solution for managing concurrency.**
- **Multiple independent execution streams.**
- **Shared state.**
- **Pre-emptive scheduling.**
- **Synchronization (e.g. locks, conditions).**

What Are Threads Used For?

- **Operating systems:** **one kernel thread for each user process.**
- **Scientific applications:** **one thread per CPU (solve problems more quickly).**
- **Distributed systems:** **process requests concurrently (overlap I/Os).**
- **GUIs:**
  - Threads correspond to user actions; can service display during long-running computations.
  - Multimedia, animations.

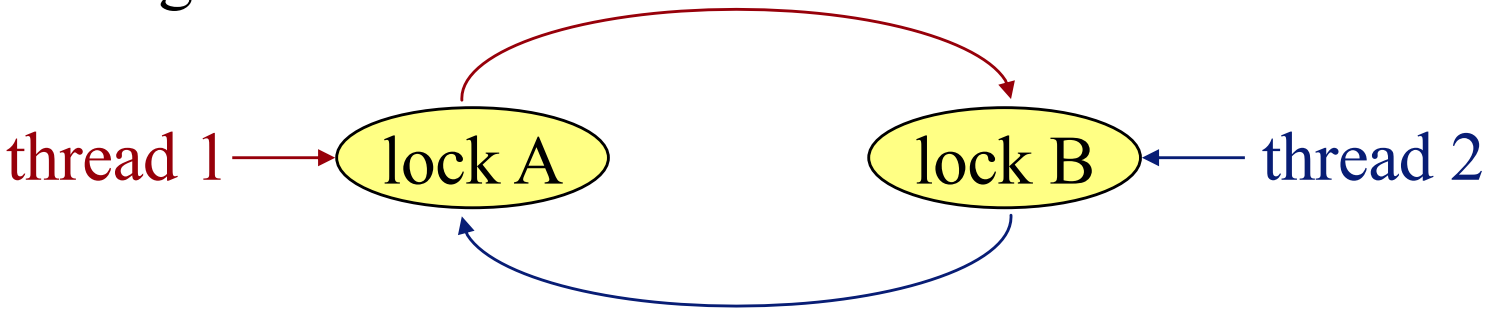
What's Wrong With Threads?



- **Too hard for most programmers to use.**
- **Even for experts, development is painful.**

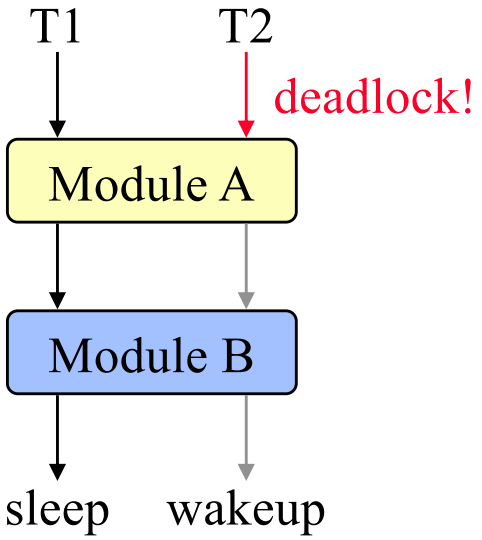
Why Threads Are Hard

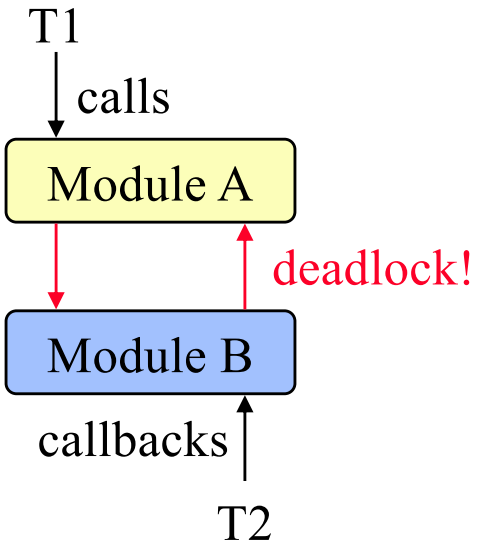
- **Synchronization:**
  - Must coordinate access to shared data with locks.
  - Forget a lock? Corrupted data.
- **Deadlock:**
  - Circular dependencies among locks.
  - Each process waits for some other process: system hangs.



Why Threads Are Hard, cont'd

- **Hard to debug:** data dependencies, timing dependencies.
- **Threads break abstraction:** can't design modules independently.
- **Callbacks don't work with locks.**



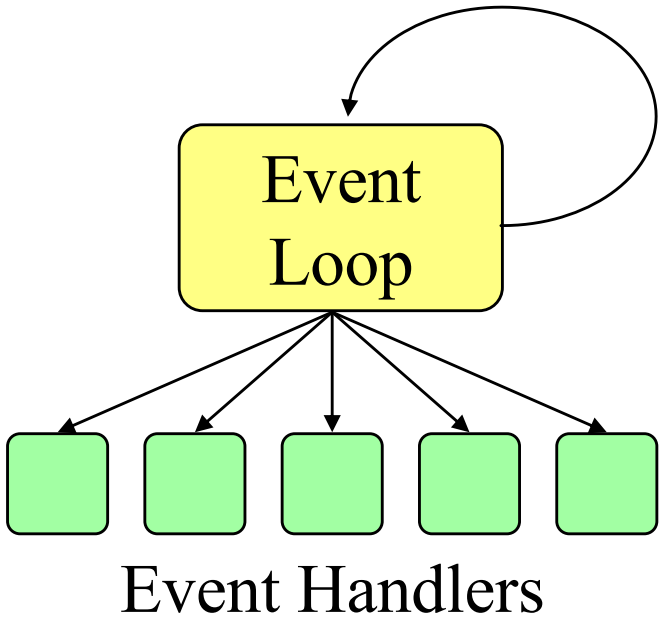


Why Threads Are Hard, cont'd

- **Achieving good performance is hard:**
  - Simple locking (e.g. monitors) yields low concurrency.
  - Fine-grain locking increases complexity, reduces performance in normal case.
  - OSes limit performance (scheduling, context switches).
- **Threads not well supported:**
  - Hard to port threaded code (PCs? Macs?).
  - Standard libraries not thread-safe.
  - Kernel calls, window systems not multi-threaded.
  - Few debugging tools (LockLint, debuggers?).
- **Often don't want concurrency anyway (e.g. window events).**

Event-Driven Programming

- **One execution stream: no CPU concurrency.**
- **Register interest in events (callbacks).**
- **Event loop waits for events, invokes handlers.**
- **No preemption of event handlers.**
- **Handlers generally short-lived.**



What Are Events Used For?

- **Mostly GUIs:**
  - One handler for each event (press button, invoke menu entry, etc.).
  - Handler implements behavior (undo, delete file, etc.).
- **Distributed systems:**
  - One handler for each source of input (socket, etc.).
  - Handler processes incoming request, sends response.
  - Event-driven I/O for I/O overlap.

Problems With Events

- **Long-running handlers make application non-responsive.**
  - Fork off subprocesses for long-running things (e.g. multimedia), use events to find out when done.
  - Break up handlers (e.g. event-driven I/O).
  - Periodically call event loop in handler (reentrancy adds complexity).
- **Can't maintain local state across events (handler must return).**
- **No CPU concurrency (not suitable for scientific apps).**
- **Event-driven I/O not always well supported (e.g. poor write buffering).**

Events vs. Threads

- **Events avoid concurrency as much as possible, threads embrace:**
  - Easy to get started with events: no concurrency, no preemption, no synchronization, no deadlock.
  - Use complicated techniques only for unusual cases.
  - With threads, even the simplest application faces the full complexity.
- **Debugging easier with events:**
  - Timing dependencies only related to events, not to internal scheduling.
  - Problems easier to track down: slow response to button vs. corrupted memory.

Events vs. Threads, cont'd

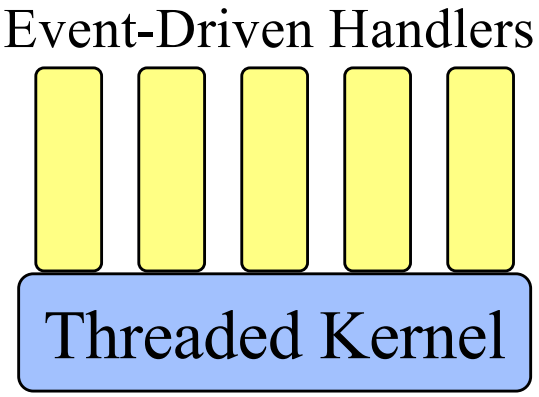
- **Events faster than threads on single CPU:**
  - No locking overheads.
  - No context switching.

- ~~**Events more portable than threads.**~~

- **Threads provide true concurrency:**
  - Can have long-running stateful handlers without freezes.
  - Scalable performance on multiple CPUs.

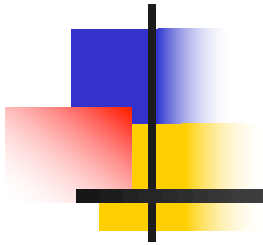
Should You Abandon Threads?

- **No: important for high-end servers (e.g. databases).**
- **But, avoid threads wherever possible:**
  - Use events, not threads, for GUIs, distributed systems, low-end servers.
  - Only use threads where true CPU concurrency is needed.
  - Where threads needed, isolate usage in threaded application kernel: keep most of code single-threaded.



Conclusions

- **Concurrency is fundamentally hard; avoid whenever possible.**
- **Threads more powerful than events, but power is rarely needed.**
- **Threads much harder to program than events; for experts only.**
- **Use events as primary development tool (both GUIs and distributed systems).**
- **Use threads only for performance-critical kernels.**



Why Events Are A Bad Idea
(for high-concurrency servers)

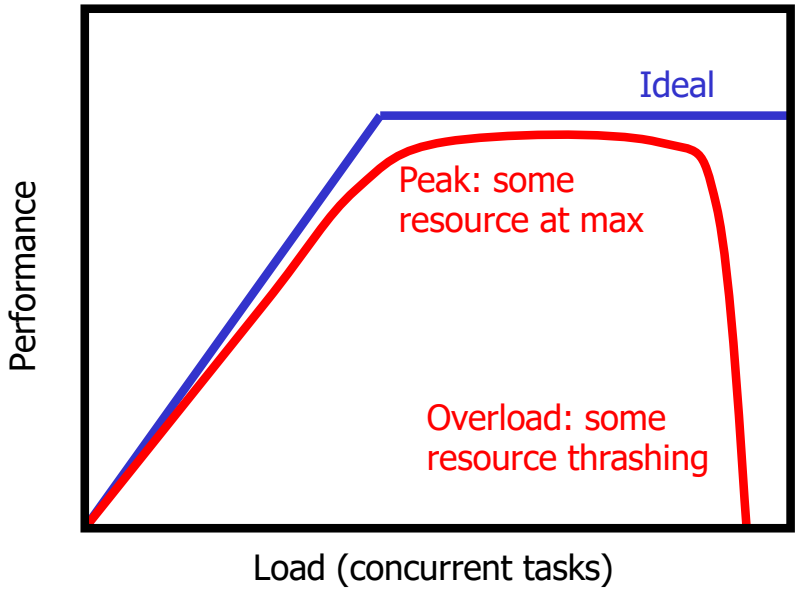
Rob von Behren, Jeremy Condit and Eric Brewer
University of California at Berkeley
{jrvb,jcondit,brewer}@cs.berkeley.edu
http://capriccio.cs.berkeley.edu

A Talk at HotOS 2003

Slide set courtesy of Rob von Behren, used with permission

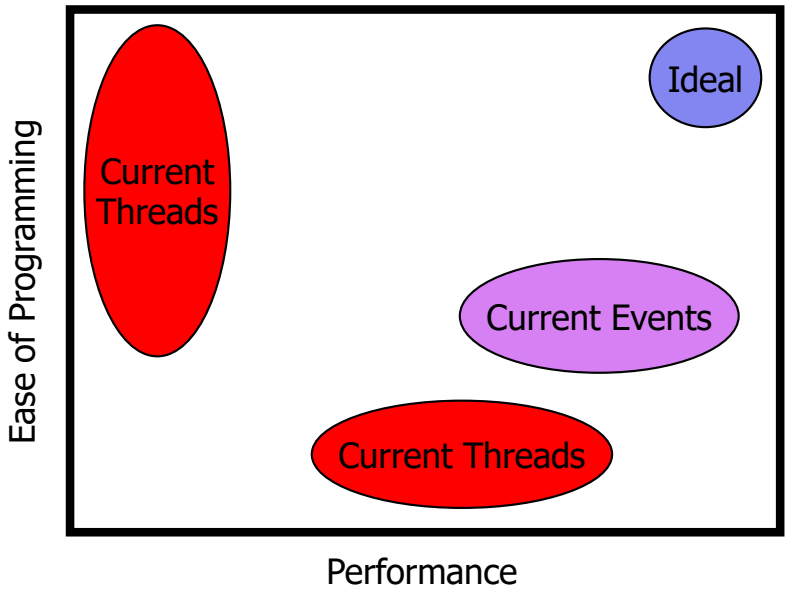
The Stage

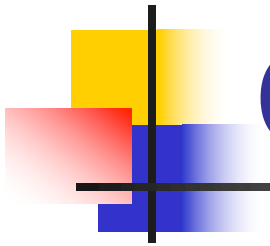
- Highly concurrent applications
  - Internet servers (Flash, Ninja, SEDA)
  - Transaction processing databases
- Workload
  - Operate "near the knee"
  - Avoid thrashing!
- What makes concurrency hard?
  - Race conditions
  - Scalability (no *O(n)* operations)
  - Scheduling & resource sensitivity
  - Inevitable overload
  - Code complexity



The Debate

- Performance vs. Programmability
  - Current threads pick one
  - Events somewhat better
- Questions
  - Threads vs. Events?
  - How do we get performance and programmability?





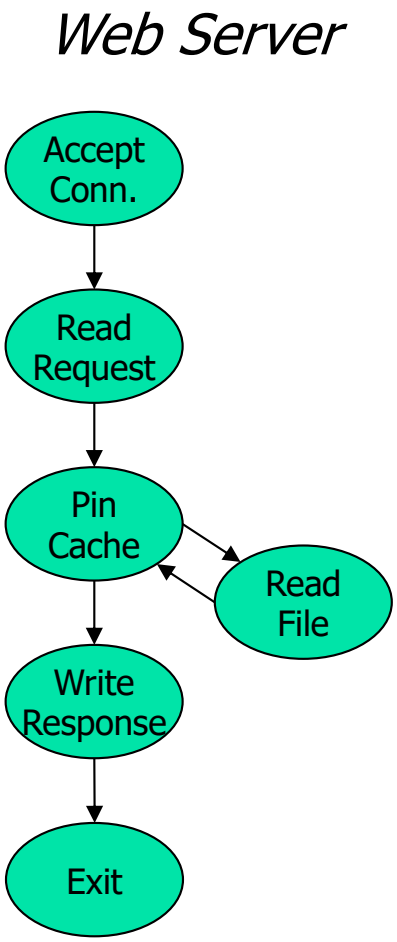
Our Position

- Thread-event duality still holds
- But threads are better anyway
  - More natural to program
  - Better fit with tools and hardware
- Compiler-runtime integration is key

The Duality Argument

- General assumption: follow "good practices"
- Observations
  - Major concepts are analogous
  - Program structure is similar
  - Performance should be similar
    - Given good implementations!

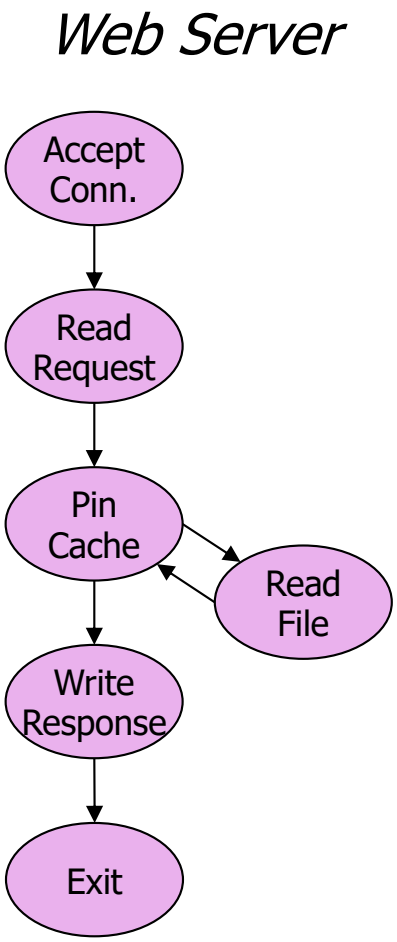
Threads	*Events*
▪ Monitors	▪ Event handler & queue
▪ Exported functions	▪ Events accepted
▪ Call/return and fork/join	▪ Send message / await reply
▪ Wait on condition variable	▪ Wait for new messages



The Duality Argument

- General assumption: follow "good practices"
- Observations
  - Major concepts are analogous
  - Program structure is similar
  - Performance should be similar
    - Given good implementations!

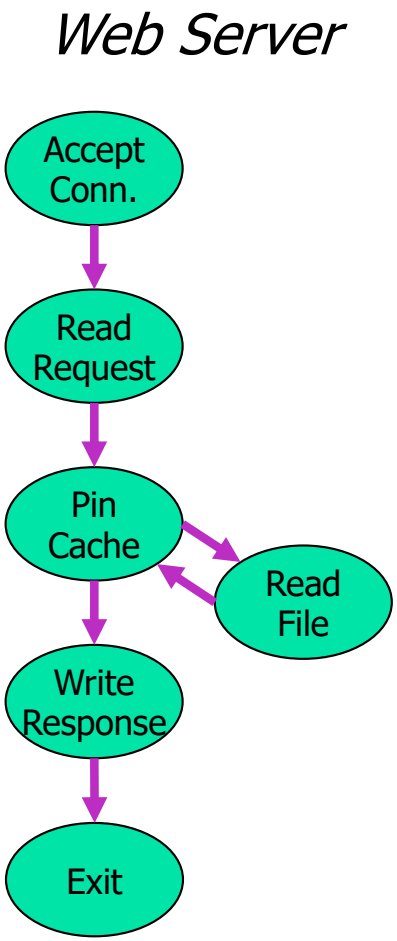
Threads	*Events*
▪ Monitors	▪ Event handler & queue
▪ Exported functions	▪ Events accepted
▪ Call/return and fork/join	▪ Send message / await reply
▪ Wait on condition variable	▪ Wait for new messages

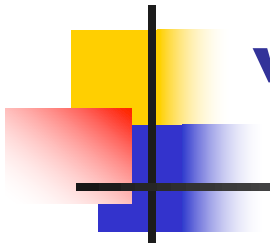


The Duality Argument

- General assumption: follow "good practices"
- Observations
  - Major concepts are analogous
  - Program structure is similar
  - Performance should be similar
    - Given good implementations!

Threads	*Events*
▪ Monitors	▪ Event handler & queue
▪ Exported functions	▪ Events accepted
▪ Call/return and fork/join	▪ Send message / await reply
▪ Wait on condition variable	▪ Wait for new messages



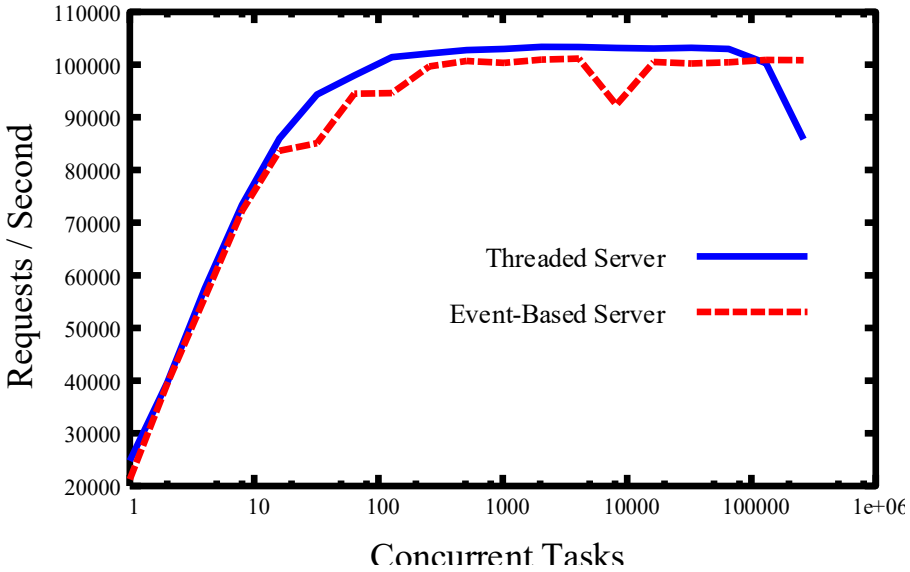


"But Events *Are* Better!"

- Recent arguments for events
  - Lower runtime overhead
  - Better live state management
  - Inexpensive synchronization
  - More flexible control flow
  - Better scheduling and locality
- All true but…
  - No *inherent* problem with threads!
  - Thread implementations can be improved

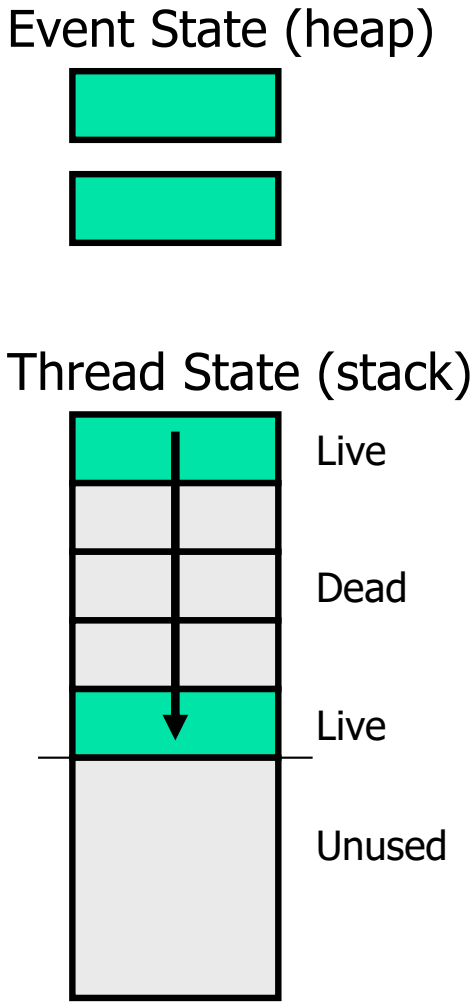
Runtime Overhead

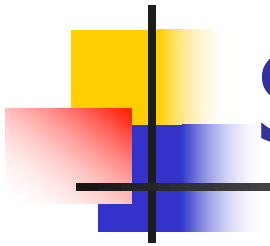
- *Criticism: Threads don't perform well for high concurrency*
- Response
  - Avoid *O(n)* operations
  - Minimize context switch overhead
- Simple scalability test
  - Slightly modified GNU Pth
  - Thread-per-task vs. single thread
  - Same performance!



Live State Management

- *Criticism: Stacks are bad for live state*
- Response
  - Fix with compiler help
  - Stack overflow vs. wasted space
    - Dynamically link stack frames
  - Retain dead state
    - Static lifetime analysis
    - Plan arrangement of stack
    - Put some data on heap
    - Pop stack before tail calls
  - Encourage inefficiency
    - Warn about inefficiency



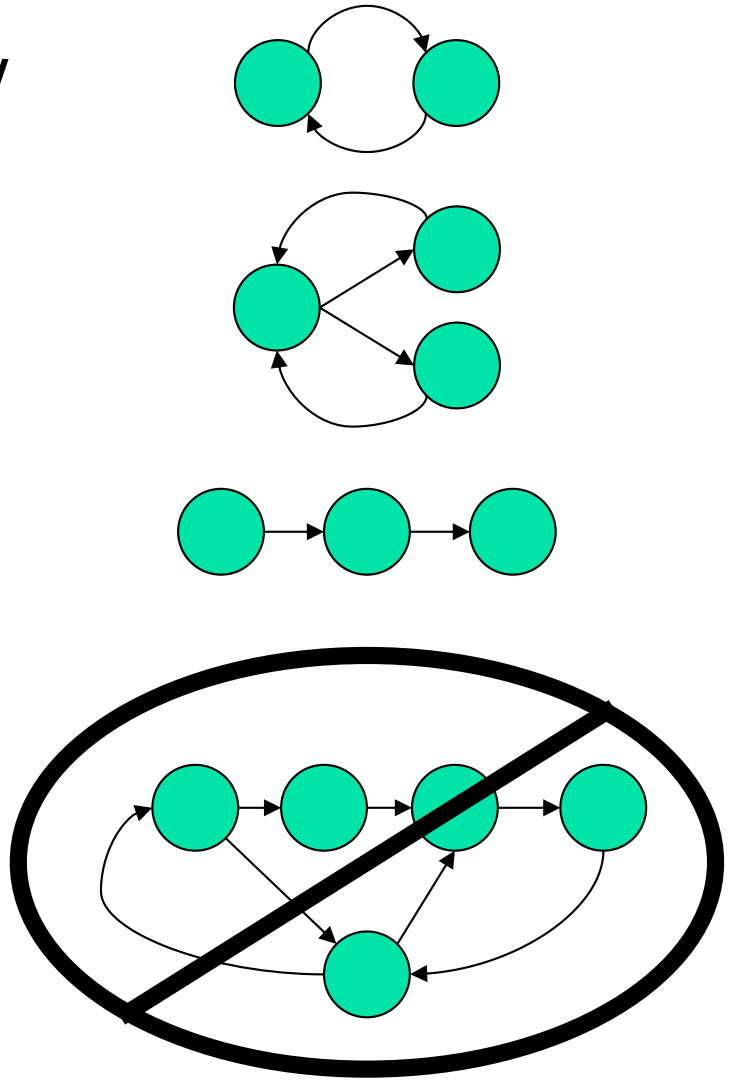


Synchronization

- *Criticism: Thread synchronization is heavyweight*
- Response
  - Cooperative multitasking works for threads, too!
  - Also presents same problems
    - Starvation & fairness
    - Multiprocessors
    - Unexpected blocking (page faults, etc.)
  - Compiler support helps

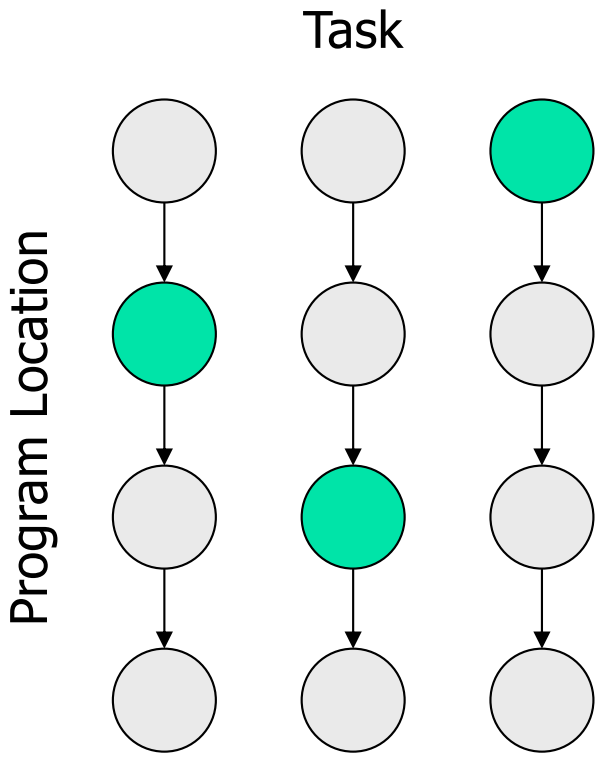
Control Flow

- *Criticism: Threads have restricted control flow*
- Response
  - Programmers use simple patterns
    - Call / return
    - Parallel calls
    - Pipelines
  - Complicated patterns are unnatural
    - Hard to understand
    - Likely to cause bugs



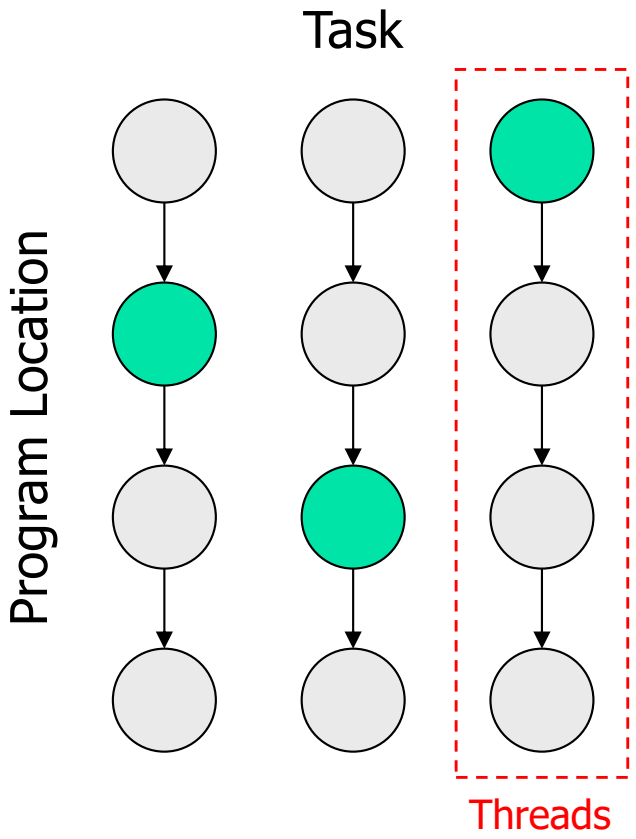
Scheduling

- *Criticism: Thread schedulers are too generic*
  - Can't use application-specific information
- Response
  - 2D scheduling: task & program location
    - Threads schedule based on task only
    - Events schedule by location (e.g. SEDA)
      - Allows batching
      - Allows prediction for SRCT
  - Threads can use 2D, too!
    - Runtime system tracks current location
    - Call graph allows prediction



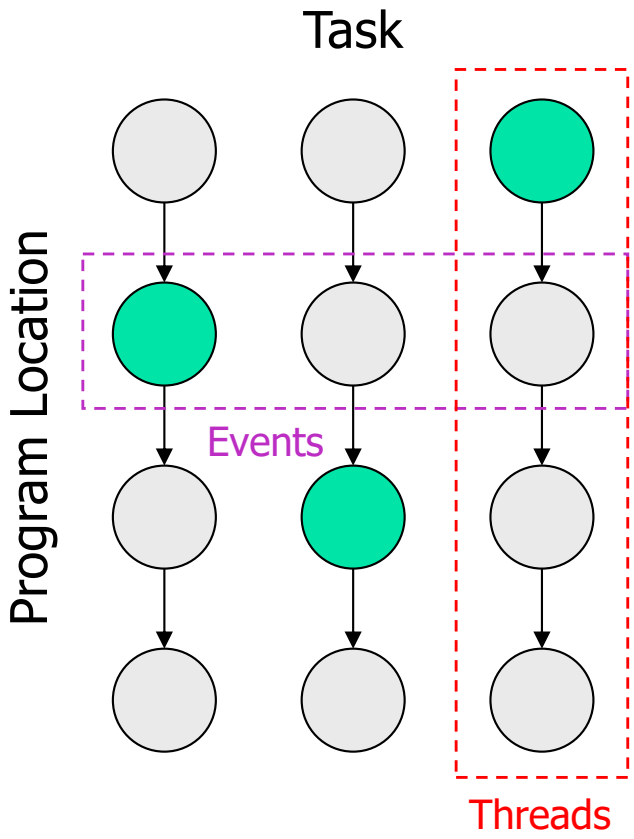
Scheduling

- *Criticism: Thread schedulers are too generic*
  - Can't use application-specific information
- Response
  - 2D scheduling: task & program location
    - Threads schedule based on task only
    - Events schedule by location (e.g. SEDA)
      - Allows batching
      - Allows prediction for SRCT
  - Threads can use 2D, too!
    - Runtime system tracks current location
    - Call graph allows prediction



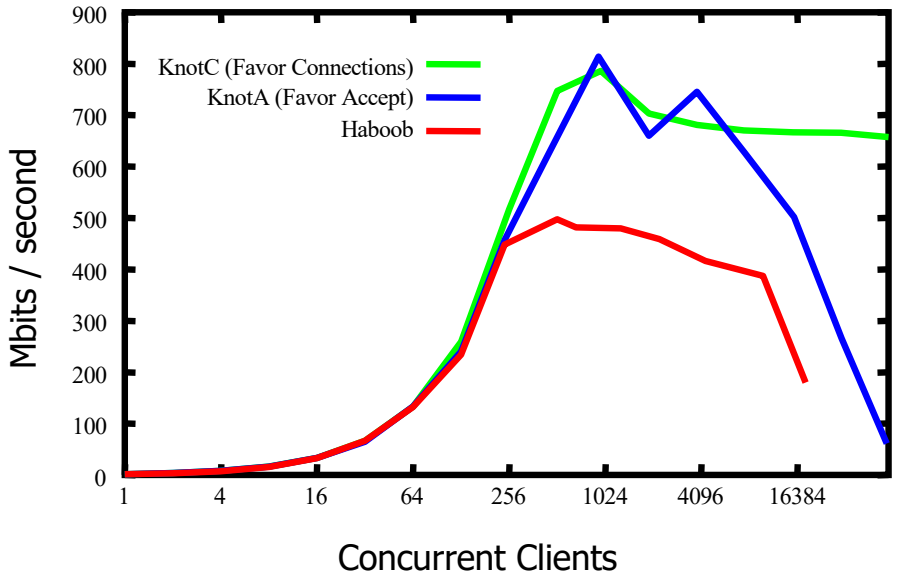
Scheduling

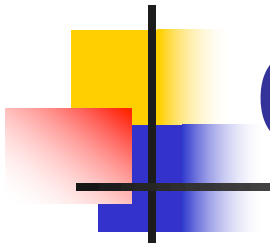
- *Criticism: Thread schedulers are too generic*
  - Can't use application-specific information
- Response
  - 2D scheduling: task & program location
    - Threads schedule based on task only
    - Events schedule by location (e.g. SEDA)
      - Allows batching
      - Allows prediction for SRCT
  - Threads can use 2D, too!
    - Runtime system tracks current location
    - Call graph allows prediction



The Proof's in the Pudding

- User-level threads package
  - Subset of pthreads
  - Intercept blocking system calls
  - No *O(n)* operations
  - Support > 100K threads
  - 5000 lines of C code
- Simple web server: Knot
  - 700 lines of C code
- Similar performance
  - Linear increase, then steady
  - Drop-off due to `poll()` overhead





Our Big But…

- More natural programming model
  - Control flow is more apparent
  - Exception handling is easier
  - State management is automatic
- Better fit with current tools & hardware
  - Better existing infrastructure
  - Allows better performance?

Control Flow

- Events obscure control flow
  - For programmers *and* tools

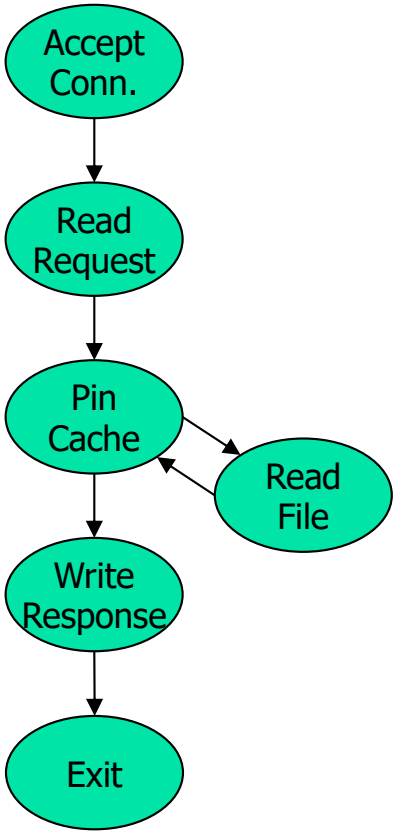
Threads

```
thread_main(int sock) {
    struct session s;
    accept_conn(sock, &s);
    read_request(&s);
    pin_cache(&s);
    write_response(&s);
    unpin(&s);
}

pin_cache(struct session *s) {
    pin(&s);
    if( !in_cache(&s) )
        read_file(&s);
}
```

Events

```
AcceptHandler(event e) {
    struct session *s = new_session(e);
    RequestHandler.enqueue(s);
}
RequestHandler(struct session *s) {
    …; CacheHandler.enqueue(s);
}
CacheHandler(struct session *s) {
    pin(s);
    if( !in_cache(s) )  ReadFileHandler.enqueue(s);
    else                ResponseHandler.enqueue(s);
}
. . .
ExitHandlerr(struct session *s) {
    …;  unpin(&s);  free_session(s);  }
```

Web Server



Control Flow

- Events obscure control flow
  - For programmers *and* tools

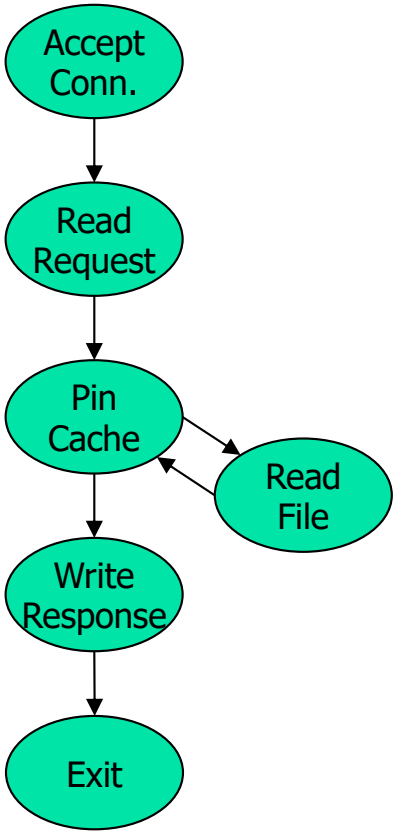
Threads

```
thread_main(int sock) {
    struct session s;
    accept_conn(sock, &s);
    read_request(&s);
    pin_cache(&s);
    write_response(&s);
    unpin(&s);
}

pin_cache(struct session *s) {
    pin(&s);
    if( !in_cache(&s) )
        read_file(&s);
}
```

Events

```
CacheHandler(struct session *s) {
    pin(s);
    if( !in_cache(s) )  ReadFileHandler.enqueue(s);
    else                ResponseHandler.enqueue(s);
}
RequestHandler(struct session *s) {
    …; CacheHandler.enqueue(s);
}
. . .
ExitHandlerr(struct session *s) {
    …;  unpin(&s);  free_session(s);
}
AcceptHandler(event e) {
    struct session *s = new_session(e);
    RequestHandler.enqueue(s); }
```

Web Server



Exceptions

- Exceptions complicate control flow
  - Harder to understand program flow
  - Cause bugs in cleanup code

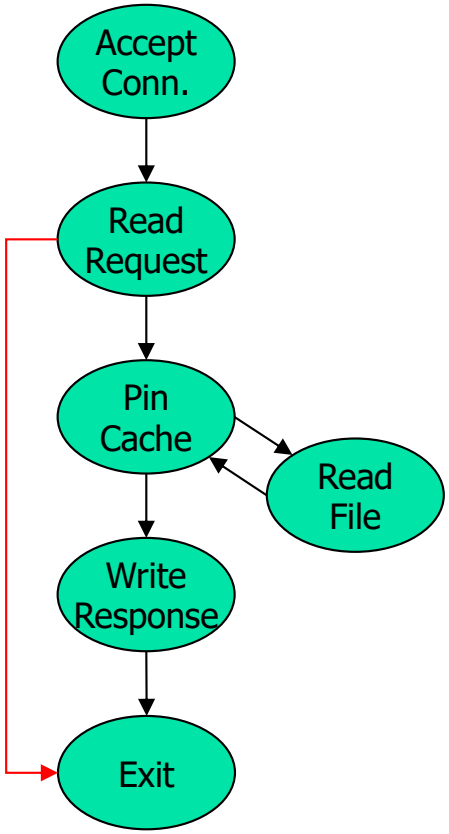
Threads

```
thread_main(int sock) {
    struct session s;
    accept_conn(sock, &s);
    if( !read_request(&s) )
        return;
    pin_cache(&s);
    write_response(&s);
    unpin(&s);
}

pin_cache(struct session *s) {
    pin(&s);
    if( !in_cache(&s) )
        read_file(&s);
}
```

Events

```
CacheHandler(struct session *s) {
    pin(s);
    if( !in_cache(s) )  ReadFileHandler.enqueue(s);
    else                ResponseHandler.enqueue(s);
}
RequestHandler(struct session *s) {
    …; if( error ) return;  CacheHandler.enqueue(s);
}
. . .
ExitHandlerr(struct session *s) {
    …;  unpin(&s);  free_session(s);
}
AcceptHandler(event e) {
    struct session *s = new_session(e);
    RequestHandler.enqueue(s); }
```

Web Server



State Management

- Events require manual state management
- Hard to know when to free
  - Use GC or risk bugs

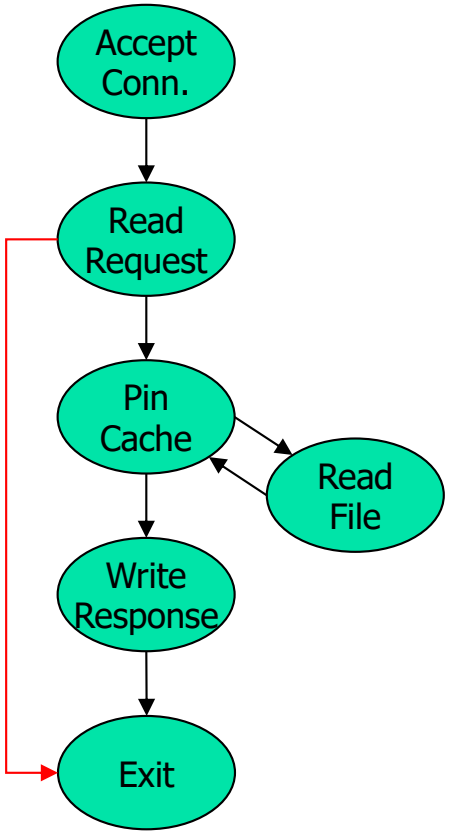
Threads

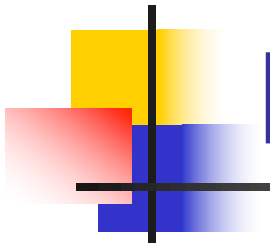
```
thread_main(int sock) {
    struct session s;
    accept_conn(sock, &s);
    if( !read_request(&s) )
        return;
    pin_cache(&s);
    write_response(&s);
    unpin(&s);
}

pin_cache(struct session *s) {
    pin(&s);
    if( !in_cache(&s) )
        read_file(&s);
}
```

Events

```
CacheHandler(struct session *s) {
    pin(s);
    if( !in_cache(s) )  ReadFileHandler.enqueue(s);
    else                ResponseHandler.enqueue(s);
}
RequestHandler(struct session *s) {
    …; if( error ) return;  CacheHandler.enqueue(s);
}
. . .
ExitHandlerr(struct session *s) {
    …;  unpin(&s);  free_session(s);
}
AcceptHandler(event e) {
    struct session *s = new_session(e);
    RequestHandler.enqueue(s); }
```

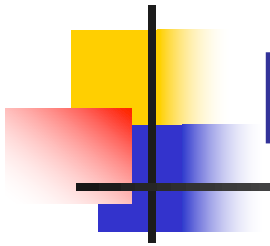
Web Server





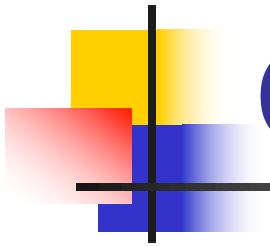
Existing Infrastructure

- Lots of infrastructure for threads
  - Debuggers
  - Languages & compilers
- Consequences
  - More amenable to analysis
  - Less effort to get working systems



Better Performance?

- Function pointers & dynamic dispatch
  - Limit compiler optimizations
  - Hurt branch prediction & I-cache locality
- More context switches with events?
  - Example: Haboob does 6x more than Knot
  - Natural result of queues
- More investigation needed!



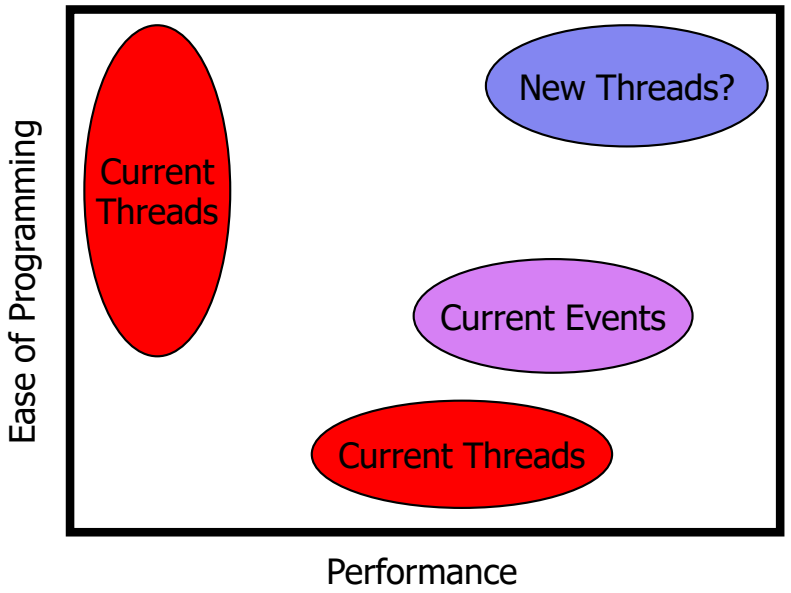
The Future: Compiler-Runtime Integration

- Insight
  - Automate things event programmers do by hand
  - Additional analysis for other things
- Specific targets
  - Dynamic stack growth*
  - Live state management
  - Synchronization
  - Scheduling*
- Improve performance *and* decrease complexity

* Working prototype in threads package

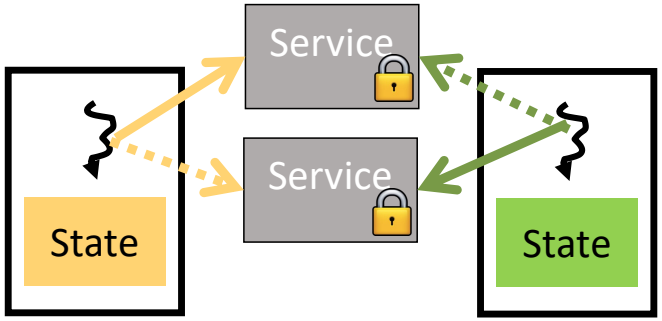
Conclusion

- Threads ≈ Events
  - Performance
  - Expressiveness
- Threads > Events
  - Complexity / Manageability
- Performance *and* Ease of use?
  - Compiler-runtime integration is key



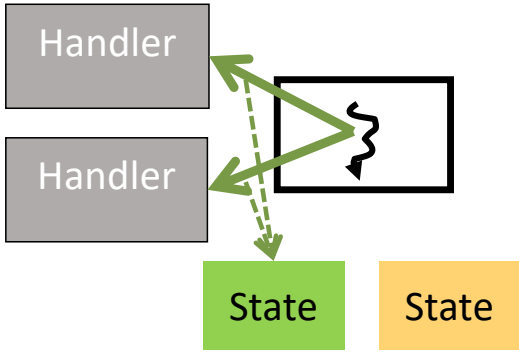
Threads vs Events

Threads



- OS thread per user process
  ⇒ extensive locking
- State with thread
- Suitable for multicore!

Events



- Stateless
- Single OS thread
  ⇒ no multicore!

LionsOS

Threads *and* Events?

LionsOS Motivation: seL4 Is A Microkernel

Microkernel:
- OS code that must execute in privileged mode
- Everything else belongs in user mode servers
- Servers are subject to the microkernel's security enforcement!

Assembly language of operating systems

Consequence:
- Small: 10 kLOC
- Only fundamental, policy-free mechanisms
- No application-oriented services/abstractions
- **BYO file system, memory manager, device drivers**

Need an actual OS!

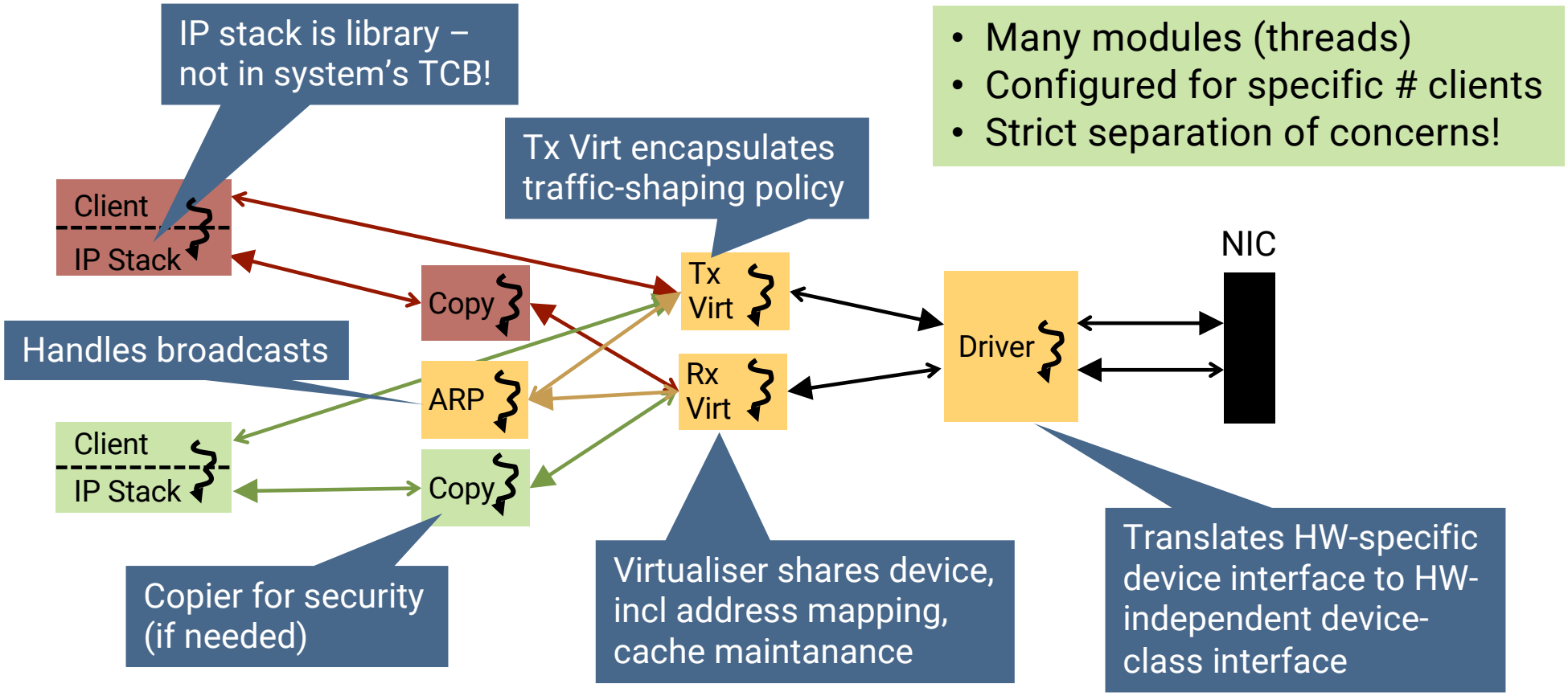
LionsOS Design Principle: KISS!

Radical simplicity:

- fine-grained modularity, strict separation of concerns
- event-driven programming model
- static system architecture
- use-case-specific policies

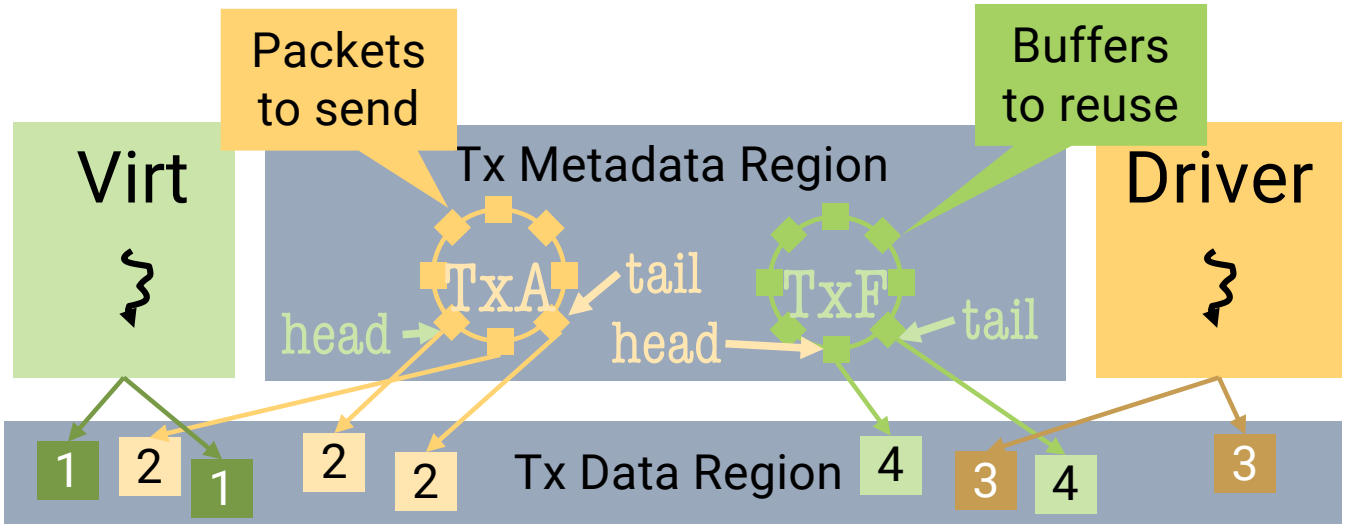
Use-case diversity by replacing components

Example: Networking Subsystem



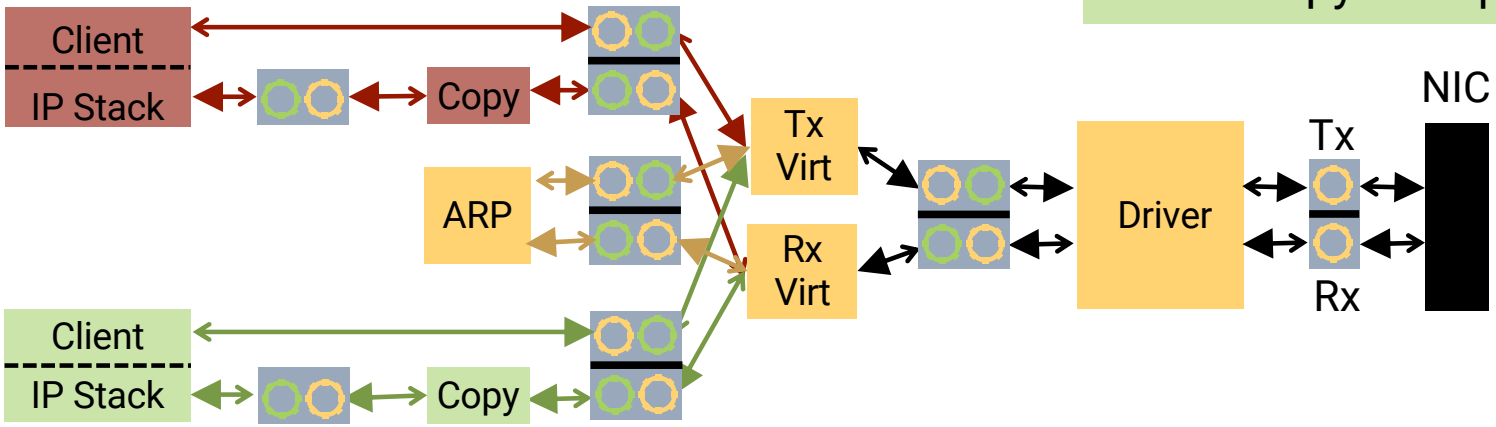
Zero-copy Data Transfer

- Lock-free bounded queues
- Single producer, single consumer
- Similar to ring buffers used by NICs
- Synchronised by semaphores



Networking Detail

Modules:
- simple event loops
- single-threaded
- zero-copy data passing



Location transparent modules
⇒ Distribute across cores!

Comparison to Linux on i.MX8M

Linux:
- NW driver: 3k lines
- NW system total: 1M lines

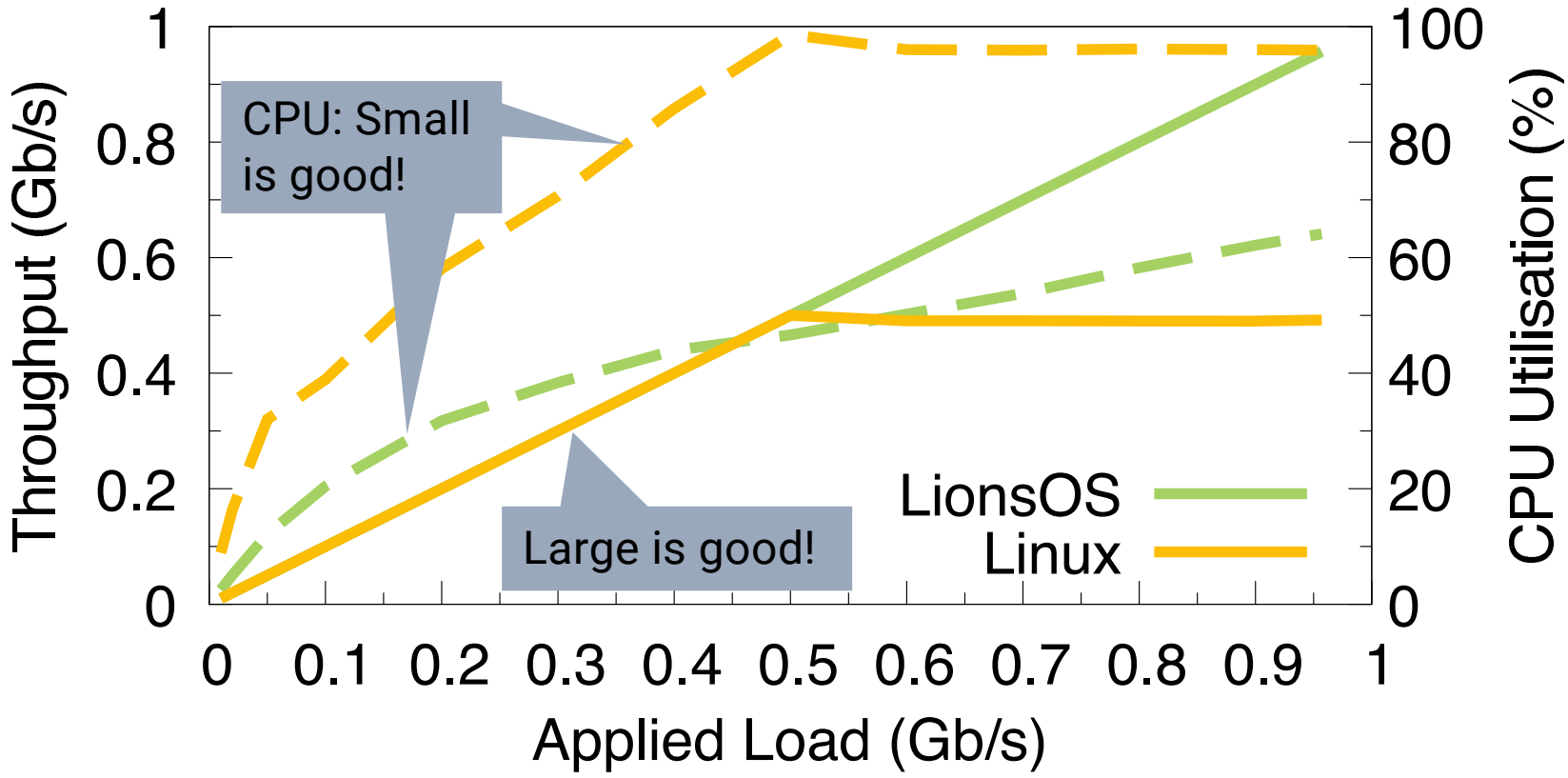
Performance?

Written by second-year student!

LionsOS:
- NW driver: 400 lines
- Virtualiser: 160 lines
- Copier: 80 lines
- IP stack: much simpler, client library
- shared NW system total: < 1,000 lines

Presently use lwip

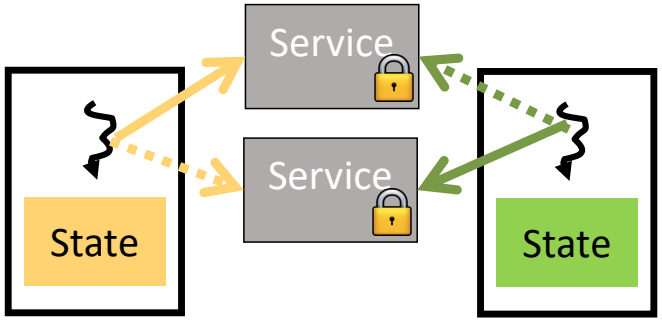
Performance: i.MX8M, 1Gb/s Eth, UDP



Single-core configuration

Threads vs Events

Threads

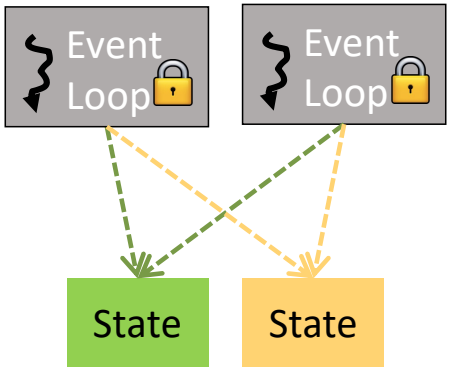


- OS thread per user process
  ⇒ extensive locking
- State with thread
- Suitable for multicore!

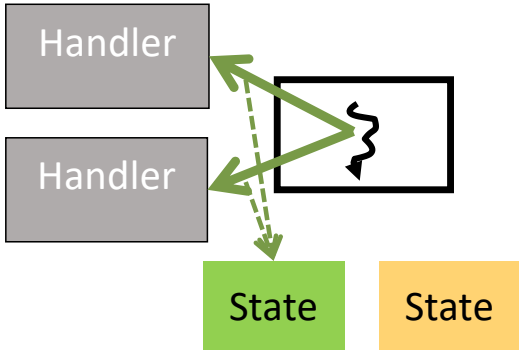
Many (micro-)services:
- 1 OS thread each
- Single-threaded
- Event-based
- Stateless

Suitable for multicore!

LionsOS



Events



- Stateless
- Single OS thread
  ⇒ no multicore!

John Lions Distinguished Lecture

Reminder: Taste of Research Internships

- Official site: https://www.unsw.edu.au/engineering/student-life/undergraduate-research-opportunities/
- TS topics: https://trustworthy.systems/students/internships
- Application deadline: 24 October
  - Talk to me before applying!