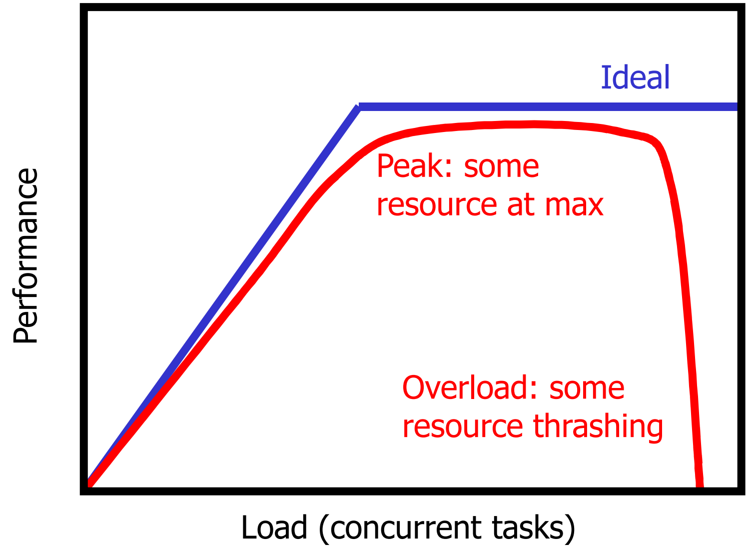
Why Events Are A Bad Idea
(for high-concurrency servers)

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 HotOS 2003

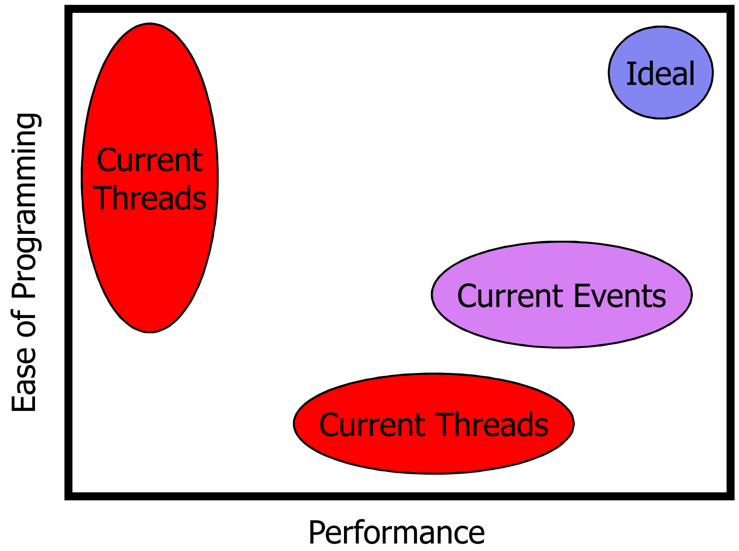
The Stage

- 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

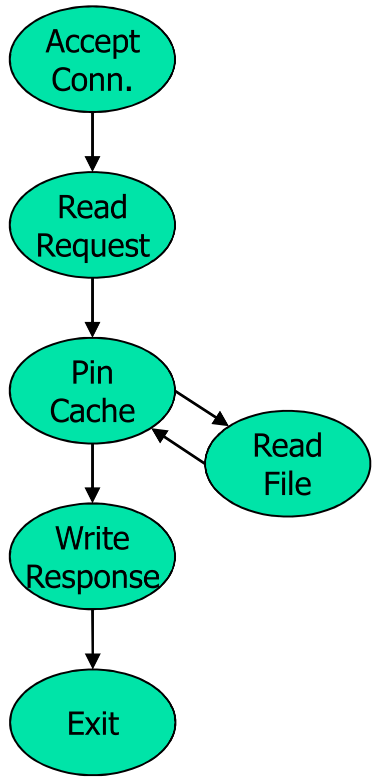
- Thread-event duality still holds
- But threads are better anyway
  - More natural to program
  - Better fit with tools and hardware
- Compiler-runtime integration is key

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

Web Server

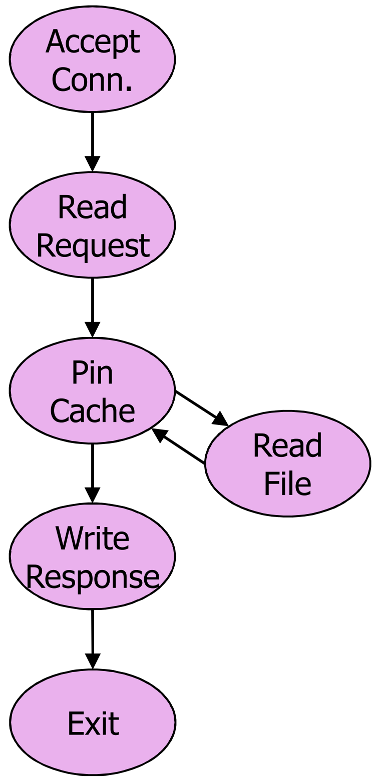


The Duality Argument

- 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

Web Server

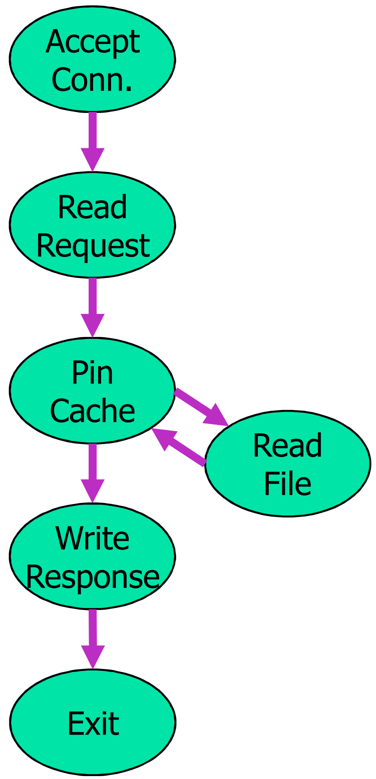


The Duality Argument

- 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

Web Server

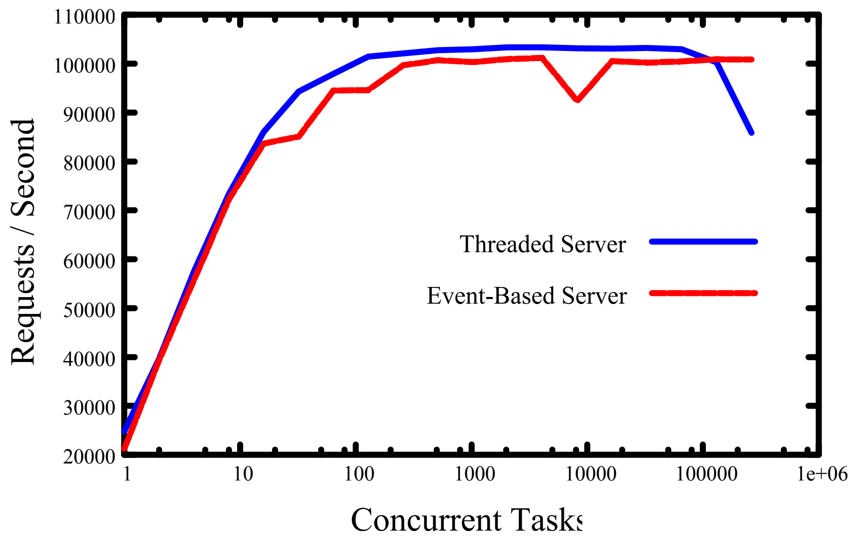


"But Events *Are* Better!"

- 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

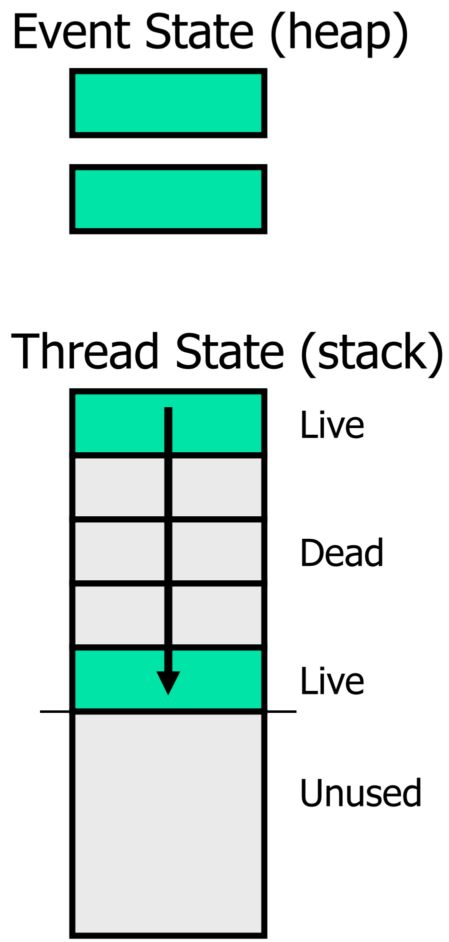
Runtime Overhead

- *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

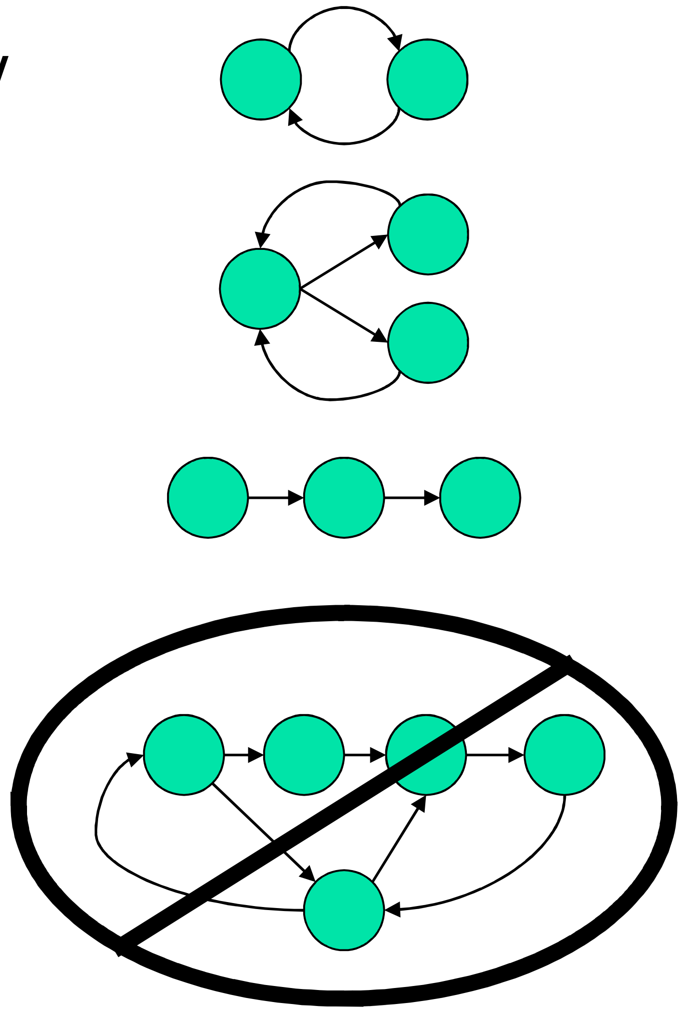


Synchronization

- *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

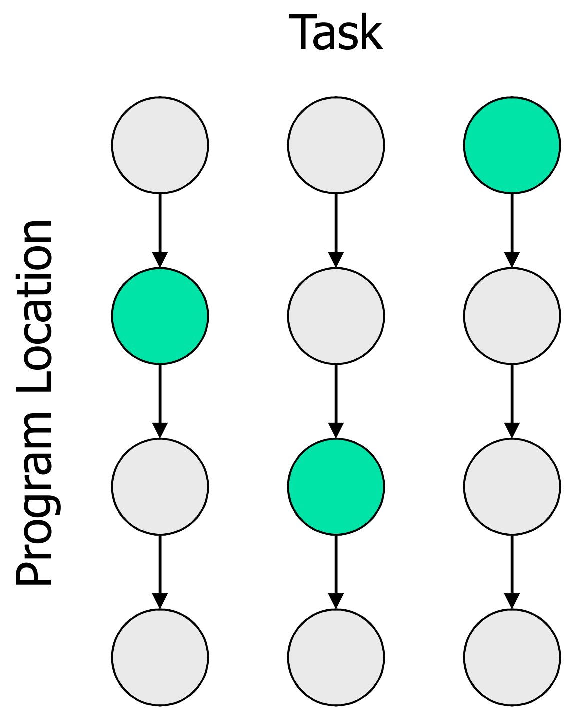
Control Flow

- *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



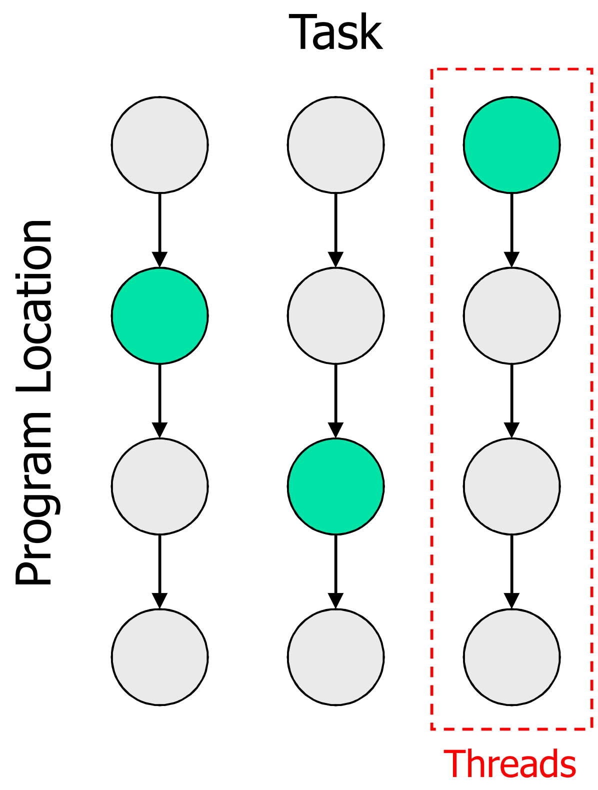
Scheduling

- *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



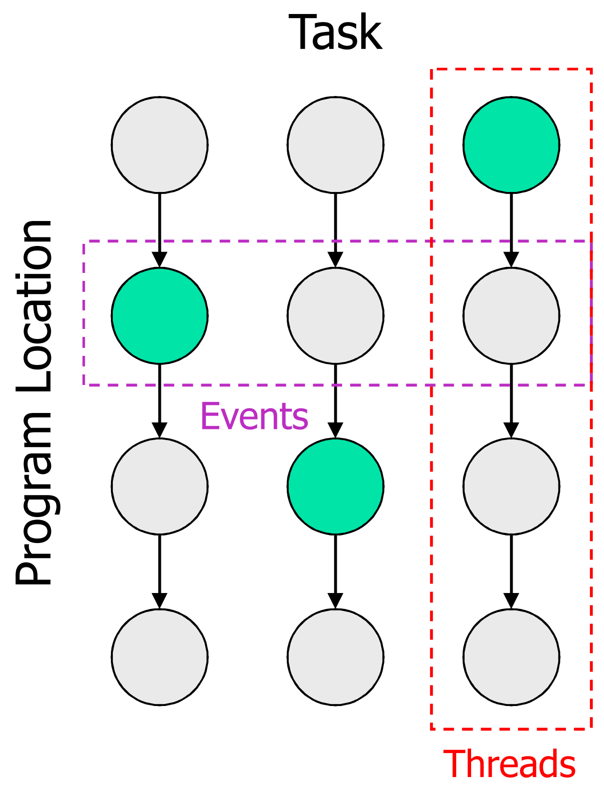
Scheduling

- *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



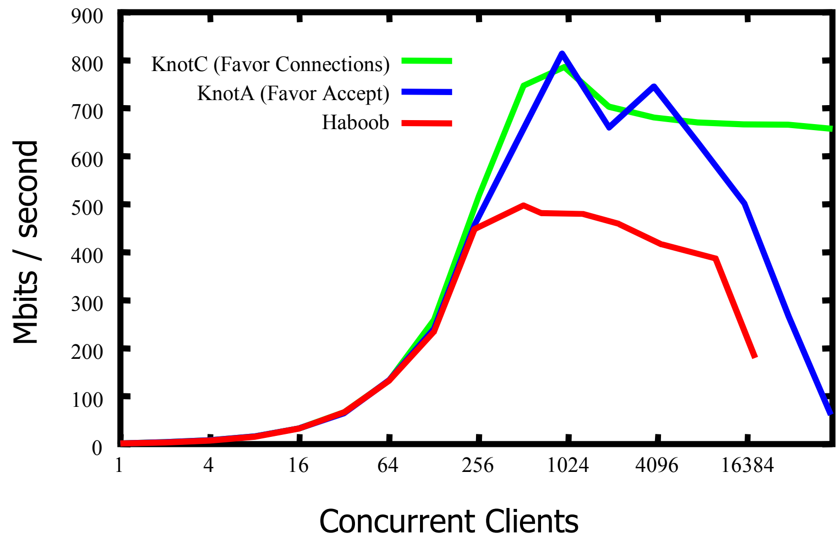
Scheduling

- *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?

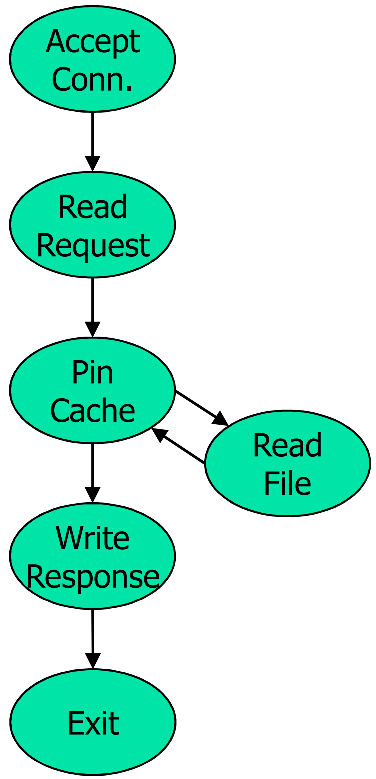
Control Flow

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

Threads	*Events*

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);
}
	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



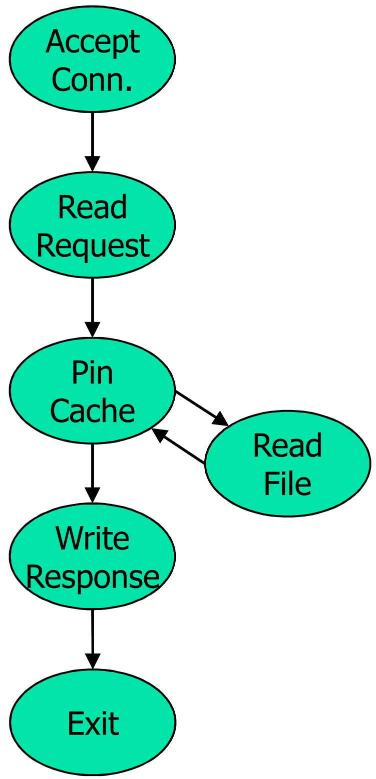
Control Flow

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

Threads	*Events*

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);
}
	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



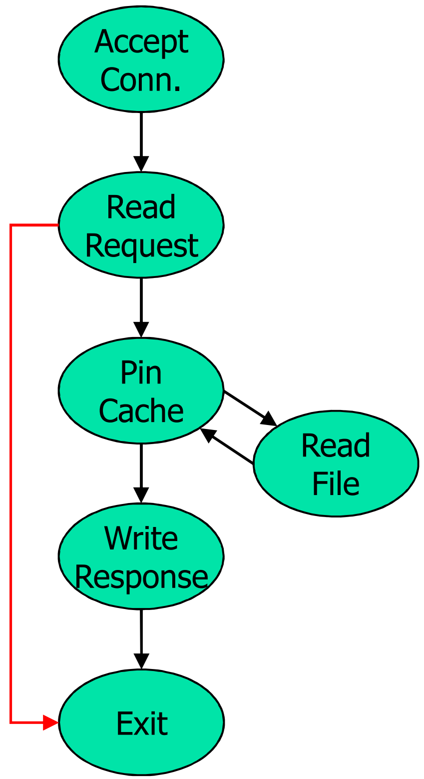
Exceptions

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

Threads	*Events*

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);
}
	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

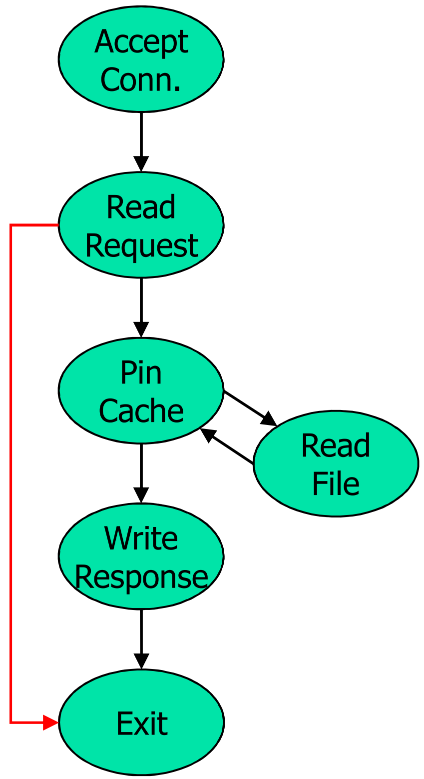
```
thread_main(int sock) {
   struct session s;
   accept_conn(sock, &s);
   if( !read_request(&s) )
      return;
   pin_cache(&s);
   write_response(&s);
   unpin(&s);
}

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