



School of Computer Science & Engineering
COMP9242 Advanced Operating Systems

2026 T3 Week 02 Part 1

OS Execution Models:

Events, Co-routines, Continuations, Threads

Gernot Heiser (adopted from Kevin Elphinstone)

Copyright Notice

These slides are distributed under the Creative Commons Attribution 4.0 International (CC BY 4.0) License

- You are free:
 - to share—to copy, distribute and transmit the work
 - to remix—to adapt the work
- under the following conditions:
 - **Attribution:** You must attribute the work (but not in any way that suggests that the author endorses you or your use of the work) as follows:

“Courtesy of Gernot Heiser, UNSW Sydney”

The complete license text can be found at
<http://creativecommons.org/licenses/by/4.0/legalcode>

Today's Lecture

- Execution models and how they apply to the OS
 - Events
 - Coroutines
 - Threads
 - Continuations
- Trade-offs and relation to SOS

System Building

General purpose OS needs to deal with concurrency

- Many user activities
 - potentially overlapping
 - may be interdependent
 - need to resume after something else happens
- Activities that depend on external events
 - may requiring waiting for completion (e.g. storage read)
 - reacting to external triggers (e.g. interrupts)

OS defines its execution model

- low-level language
- minimal runtime

Need a systematic approach to execution structure

Execution Models

- Events
- Coroutines
- Threads
- Continuations

Note: Focus is on uni-processor for now, multiprocessors later

Events

Events

- External entities generate (post) events.
 - keyboard presses, mouse clicks, system calls, IRQs
- *Event loop* waits for events and calls an appropriate *event handler*.
- *Event handler* is a function that runs until completion and returns to the *event loop*.

Some Definitions

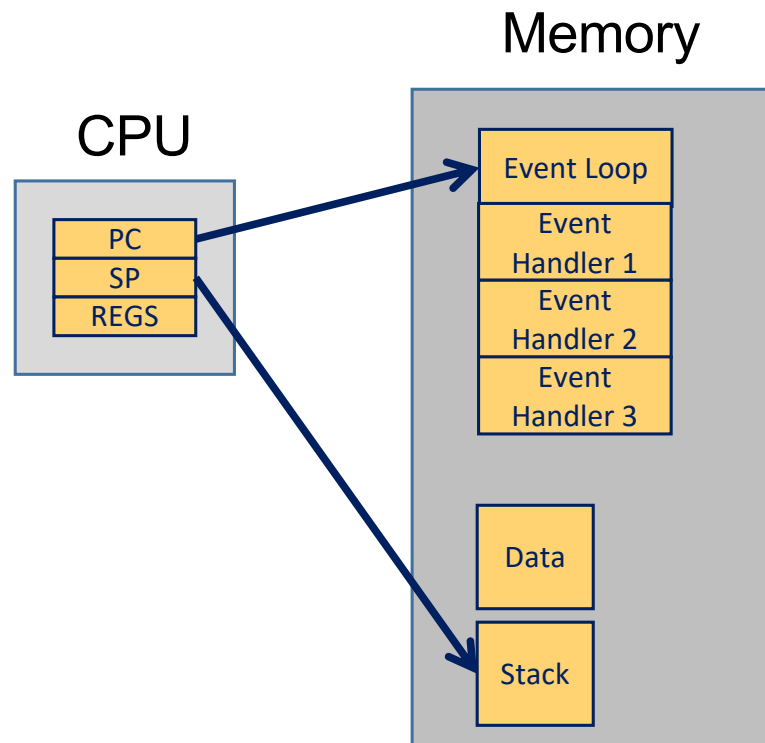
Block:

- Execution state is preserved
- *Marks current execution as blocked*
- *It is no longer considered Ready*
 - *Removed from a Ready Queue*
- *Requires an unblock to mark ready and rejoin the ready queue*
- Resumes from where it blocked

Yield:

- Execution state is preserved
- *The thread relinquishes execution*
- *Immediately placed in the ready queue*
- Resumes from where it yielded

Event Model



Only requires a single stack:

- Event handlers return to the event loop
 - No blocking
 - No yielding
- No preemption of handlers
 - Handler functions should be short!

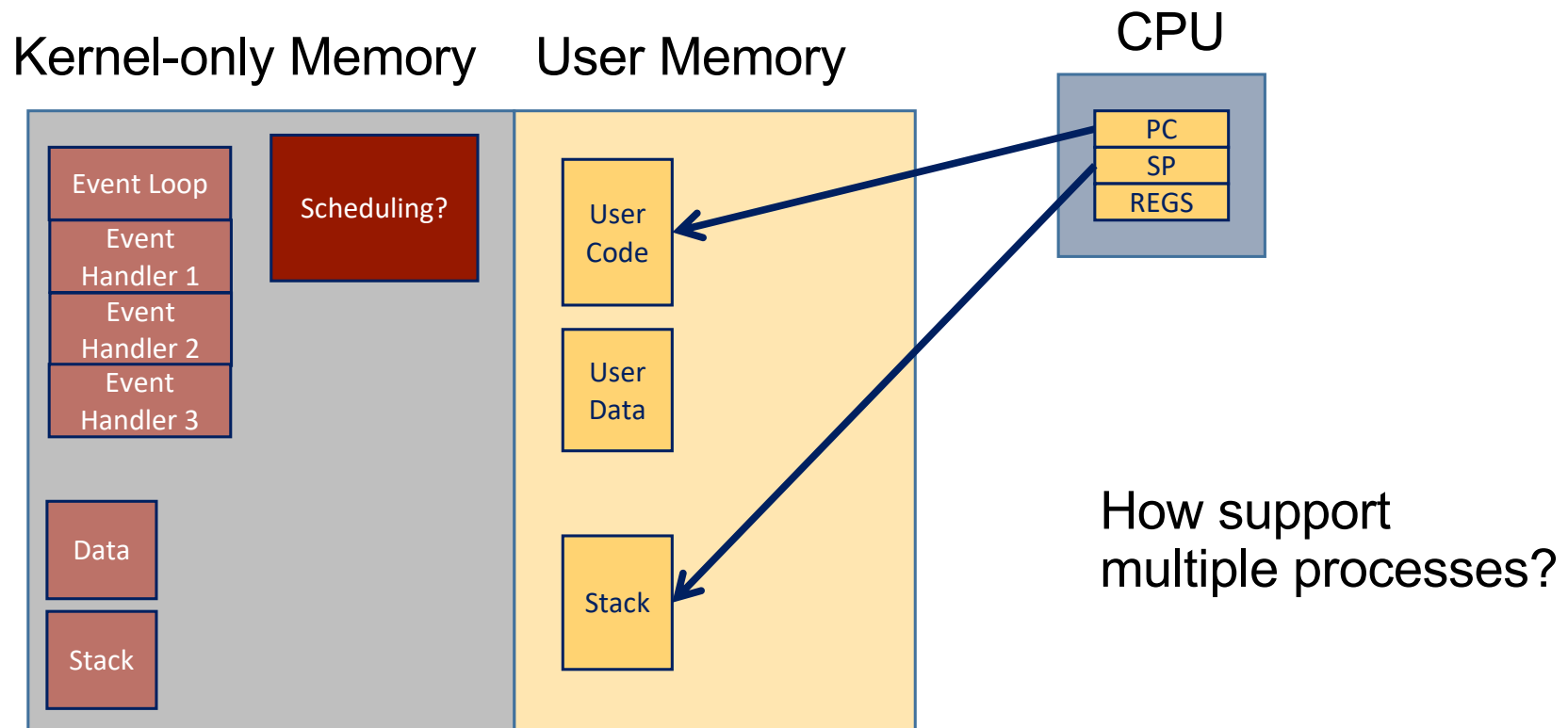
What is 'a'?

```
int a; /* global */
```

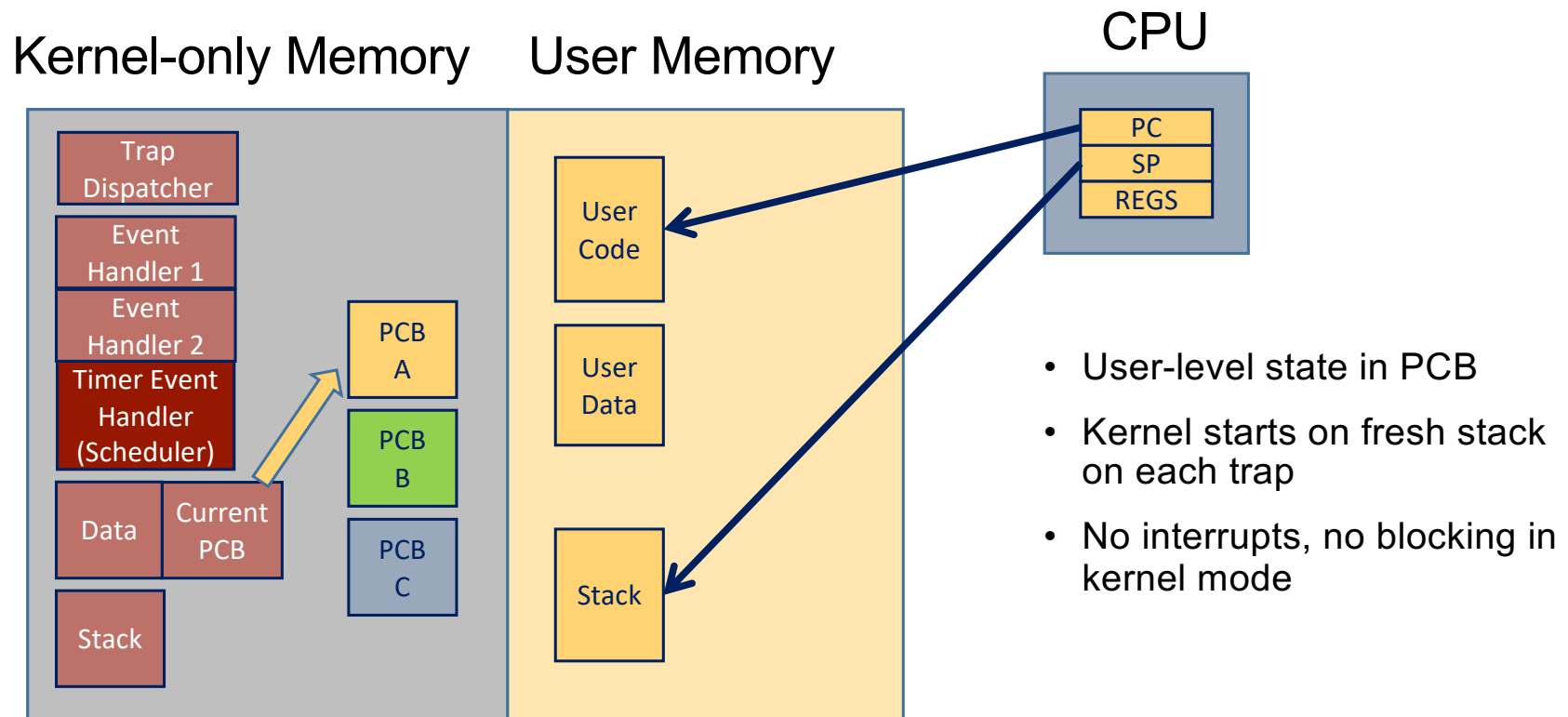
```
int func() {  
    a = 1;  
    if (a == 1) {  
        a = 2;  
    }  
    return a;  
}
```

No concurrency issues
within a handler

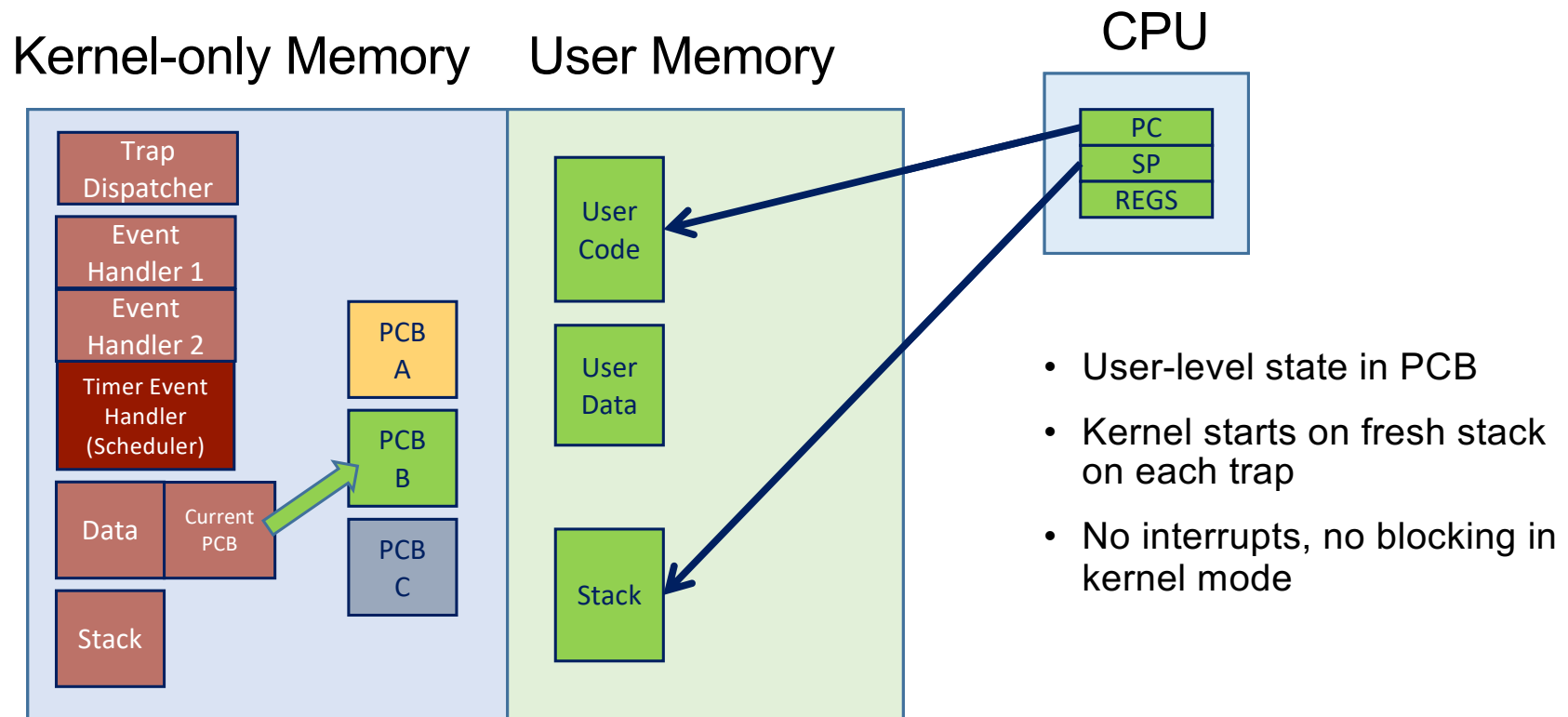
Event-based kernel on CPU with protection



Event-based kernel on CPU with protection



Event-based kernel on CPU with protection



Coroutines

Coroutines

- Old idea:

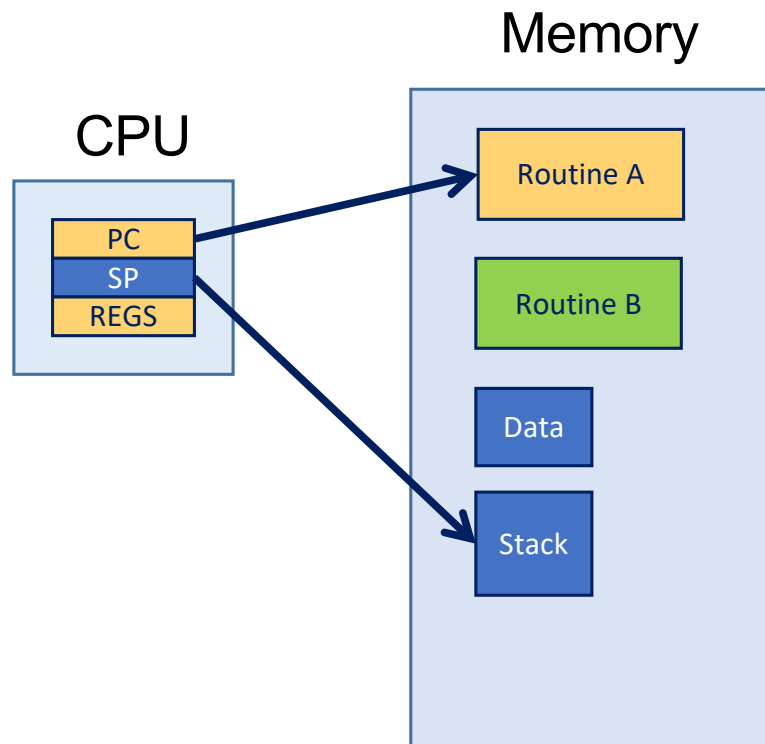
Melvin E. Conway. 1963. *Design of a separable transition-diagram compiler*. Commun. ACM 6, 7 (July 1963), 396-408.

DOI=<http://dx.doi.org/10.1145/366663.366704>

- Analogous to a “subroutine” with extra entry and exit points

- Exit/enter via `yield()`
- Supports long running subroutines
- Can implement sync primitives that wait for a condition to be true
 - `while (condition != true) yield();`

Coroutines



- `yield()` saves state of routine A and starts routine B
 - or resumes B's state from its previous `yield()` point.
- No pre-emption, any switching is explicit via `yield()` in code

What is 'a'?

```
int a; /* global */
```

```
int func() {  
    a = 1;  
    if (a == 1) {  
        yield();  
        a = 2;  
    }  
    return a;  
}
```

Bending over
backwards to shoot
yourself in the foot...

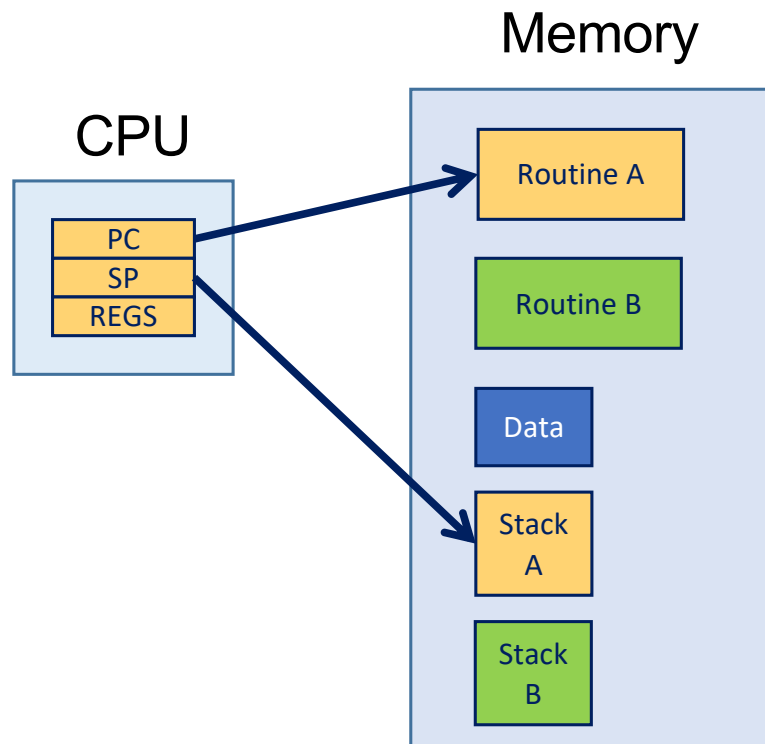
What is 'a'?

```
int a; /* global */
```

```
int func() {  
    a = 1;  
    yield();  
    if (a == 1) {  
        a = 2;  
    }  
    return a;  
}
```

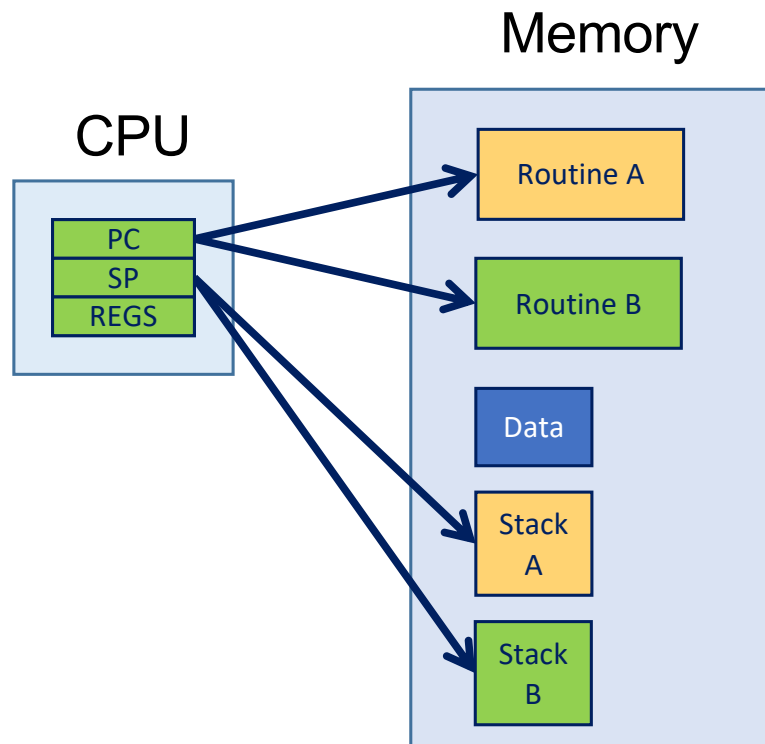
Limited concurrency issues/races, as
globals are exclusive between yields()

Coroutines Implementation strategy?



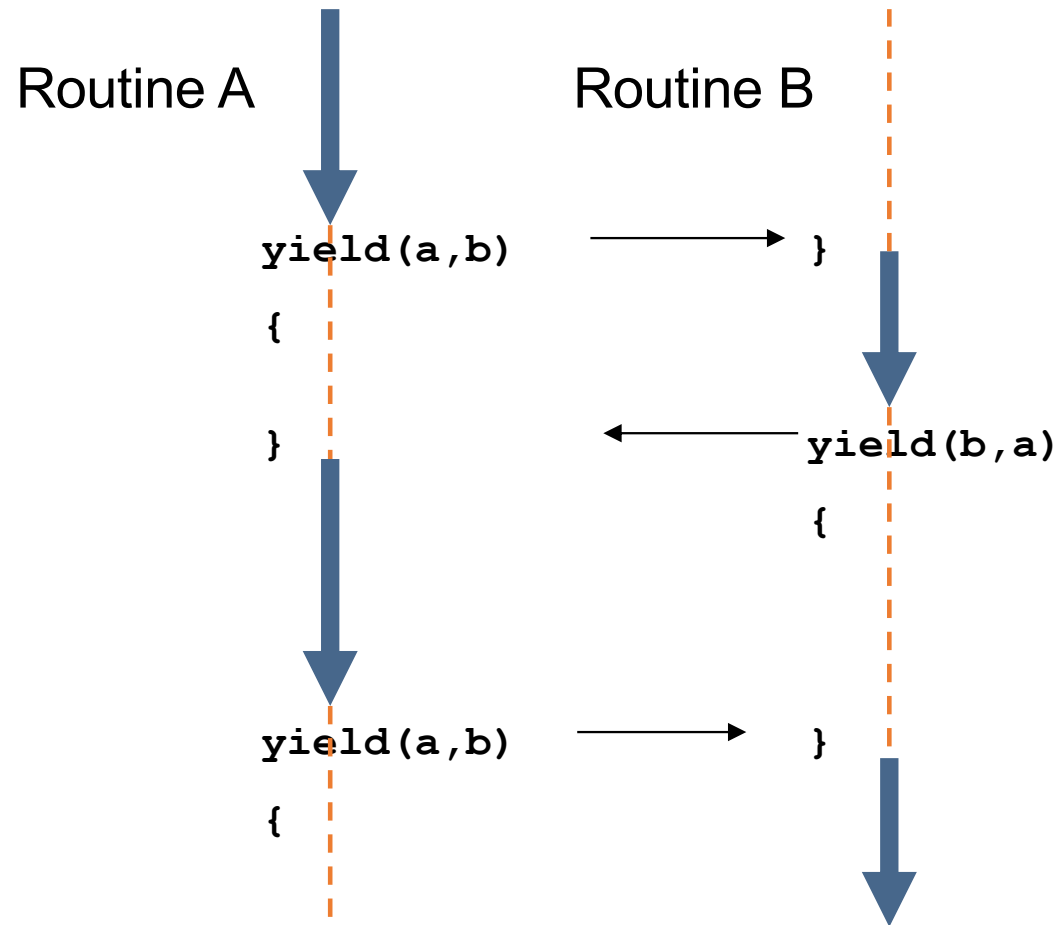
- Usually implemented with a stack per routine
- Preserves current state of execution of the routine

Coroutines Implementation strategy?



- Routine A state currently loaded
- Routine B state stored on stack
- Routine switch from A $\rightarrow$ B
 - saving state of A
 - registers, SP, PC
 - restoring the state of B
 - registers, SP, PC

Yield



What is 'a'?

```
int a; /* global */
```

```
int func() {  
    a = 1;  
    func2();  
    if (a == 1) {  
        a = 2;  
    }  
    return a;  
}
```

Does `func2()` `yield()`?

Use at your own risk!

- Build has `libco` (used by gdb thread):
 - <https://github.com/higan-emu/libco>
- Tony Finch's `picoro`: <https://dotat.at/git/picoro.git>

Threads

Cooperative Multithreading

- Also called *green threads*
- Conservatively assumes a multithreading model
 - i.e. uses synchronisation (locks) to avoid races,
 - and makes no assumption about subroutine behaviour
 - Everything thing can potentially yield()

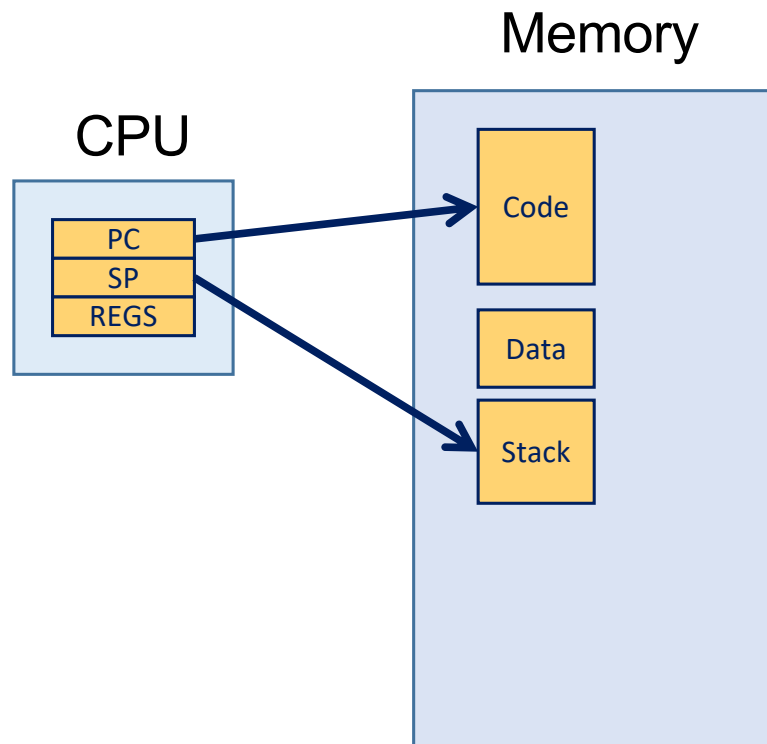
Green Threads

```
int a; /* global */
lock_t a_lock;
int func() {
    int t;
    lock_acquire(a_lock)
    a = 1;
    func2();
    if (a == 1) {
        a = 2;
    }
    t = a;
    lock_release(a_lock);
    return t;
}
```

Pessimistic locking

Deadlocks?

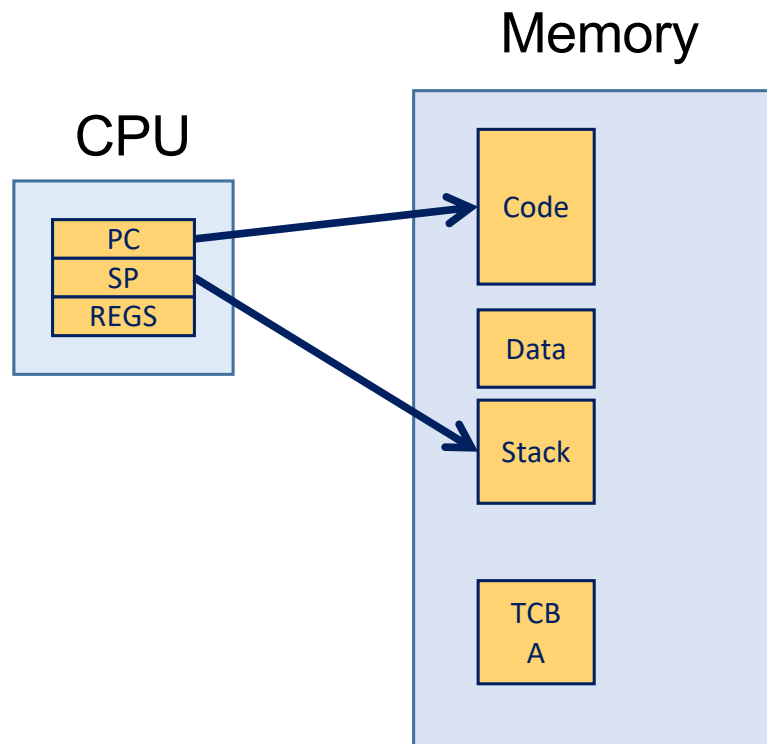
A Thread



Thread attributes

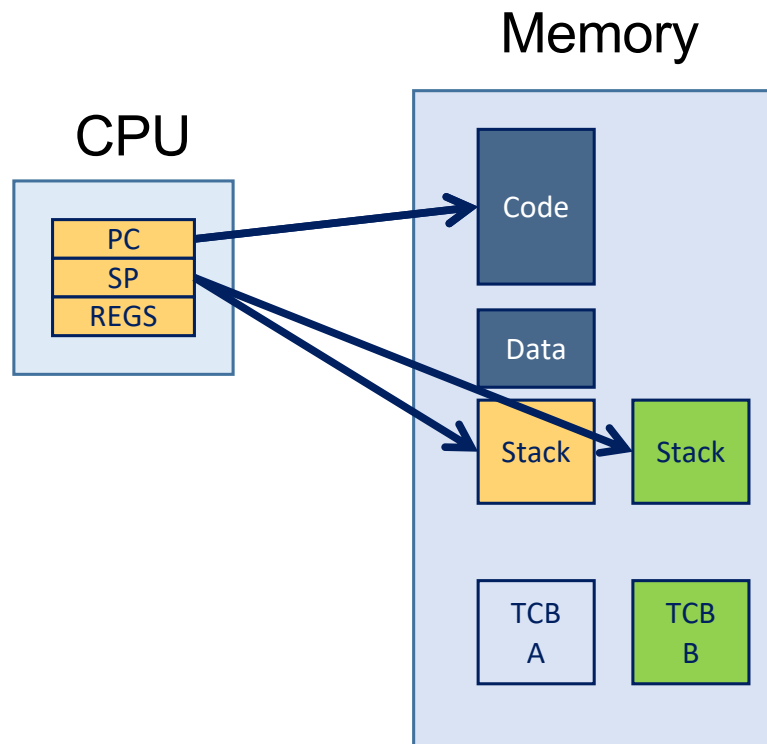
- processor related:
 - memory
 - program counter
 - stack pointer
 - registers (and status)
- OS/package related:
 - state (running/blocked)
 - identity
 - scheduler (queues, priority)
 - etc...

Thread Control Block (TCB)



- To support more than a single thread we need to store thread state and attributes
- Stored in per-thread thread control block
 - also indirectly in stack

Thread A and Thread B



- Thread A state currently loaded
- Thread B state stored in TCB B
- Thread switch from A $\rightarrow$ B
 - saving state of thread A
 - regs, sp, pc
 - restoring the state of thread B
 - regs, sp, pc
- Note: registers and PC can be stored on the stack, and only SP stored in TCB

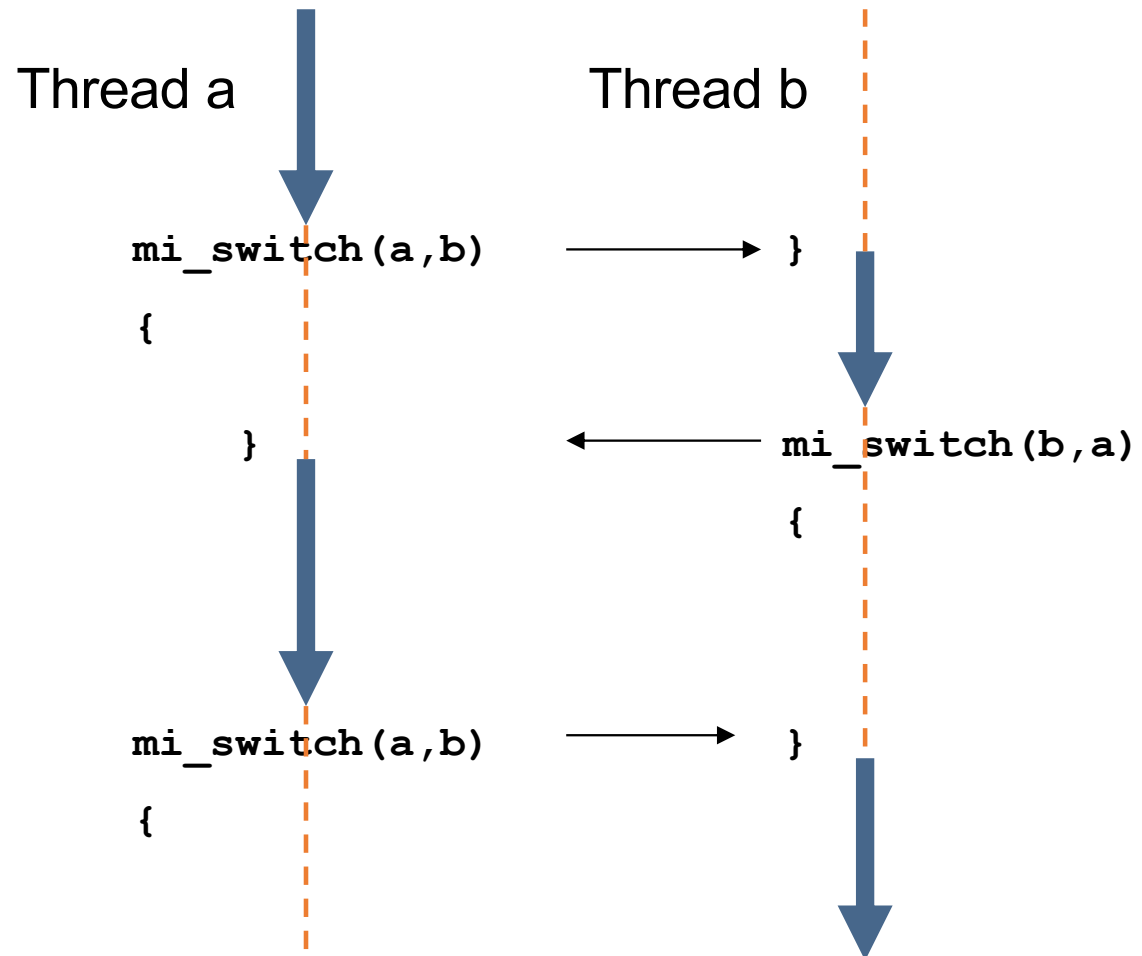
OS Pseudo-Code

```
mi_switch()
{
    struct thread *cur, *next;
    next = scheduler();

    /* update curthread */
    cur = curthread;
    curthread = next;
    /*
     * Call the machine-dependent code that actually does the
     * context switch.
     */
    md_switch(&cur->t_sp, &next->t_sp);
    /* back running in same thread */
}
```

Note: global
variable curthread

Thread Switch

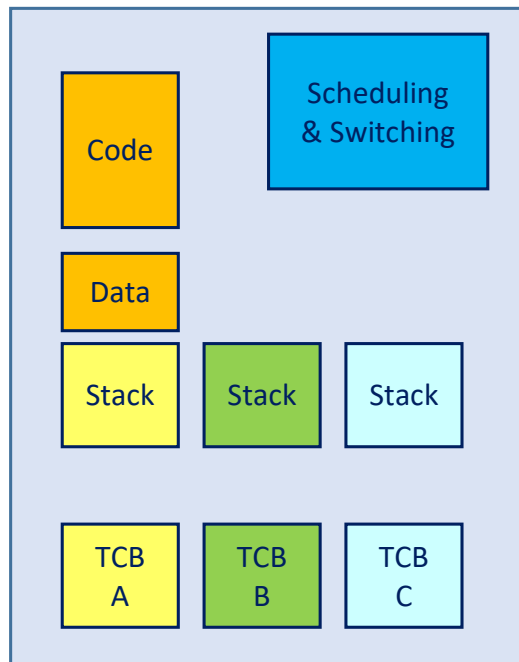


Preemptive Multithreading

- Switch can be triggered by asynchronous external event
 - eg. timer interrupt
- Asynchronous interrupt triggers saving current state
 - on current stack, if in kernel (nesting)
 - on kernel stack or in TCB if coming from user-level
- Call `thread_switch()`

Threads on simple CPU

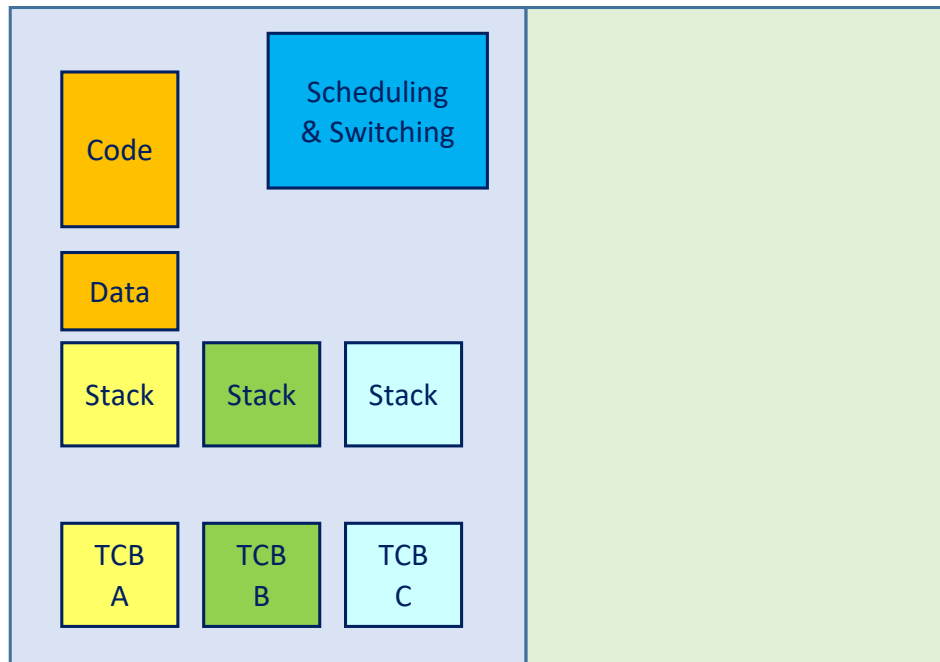
Memory



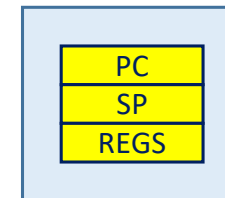
Threads on CPU with protection

Kernel-only Memory

User Memory



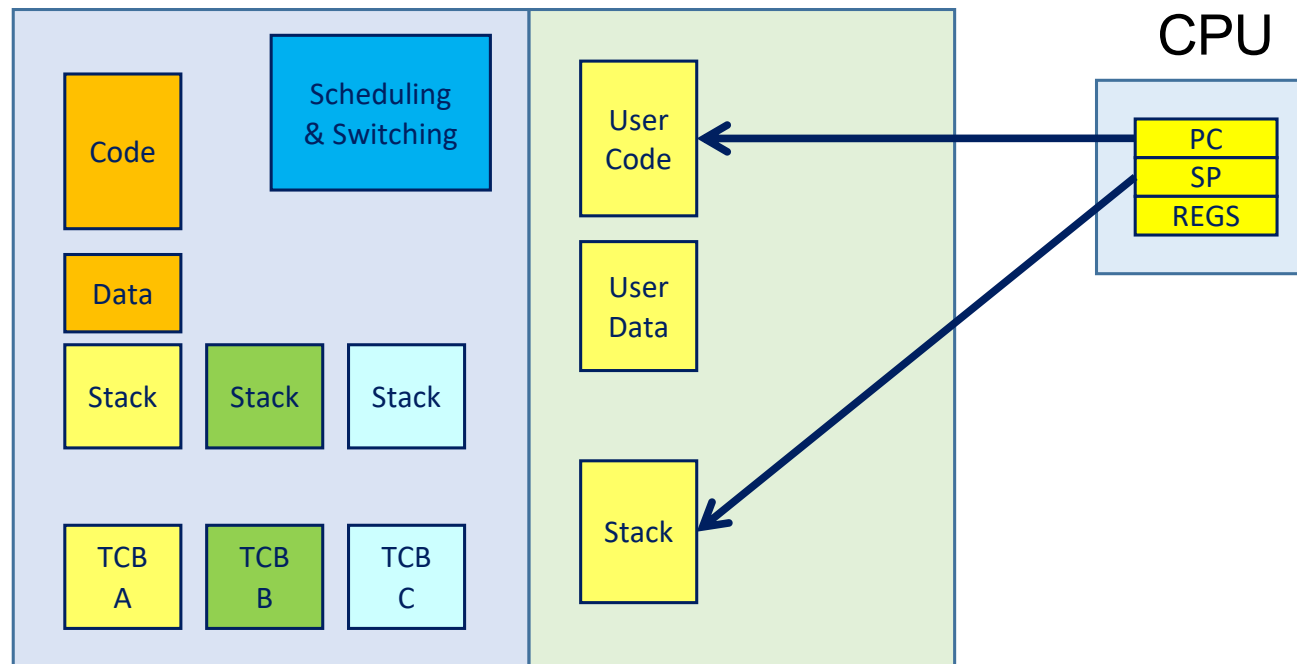
CPU



What is missing?

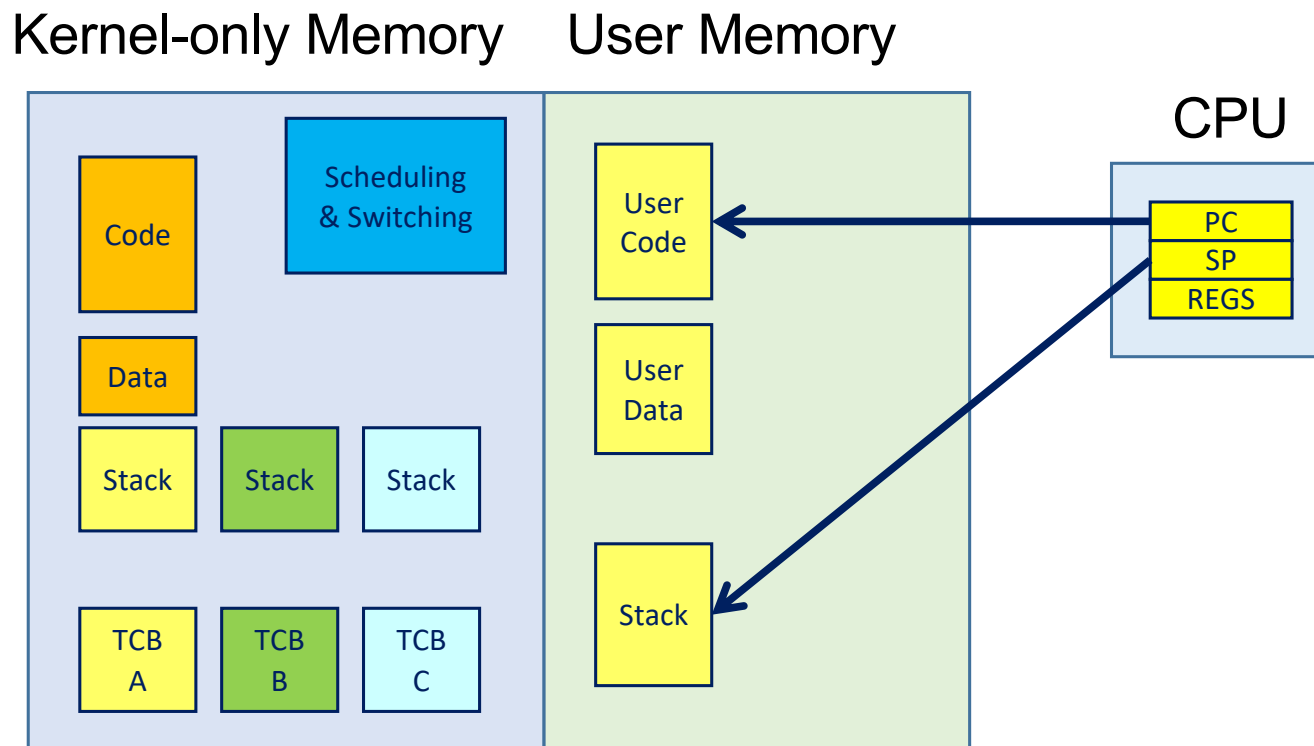
Threads on CPU with protection

Kernel-only Memory User Memory



- What happens on kernel entry and exit?

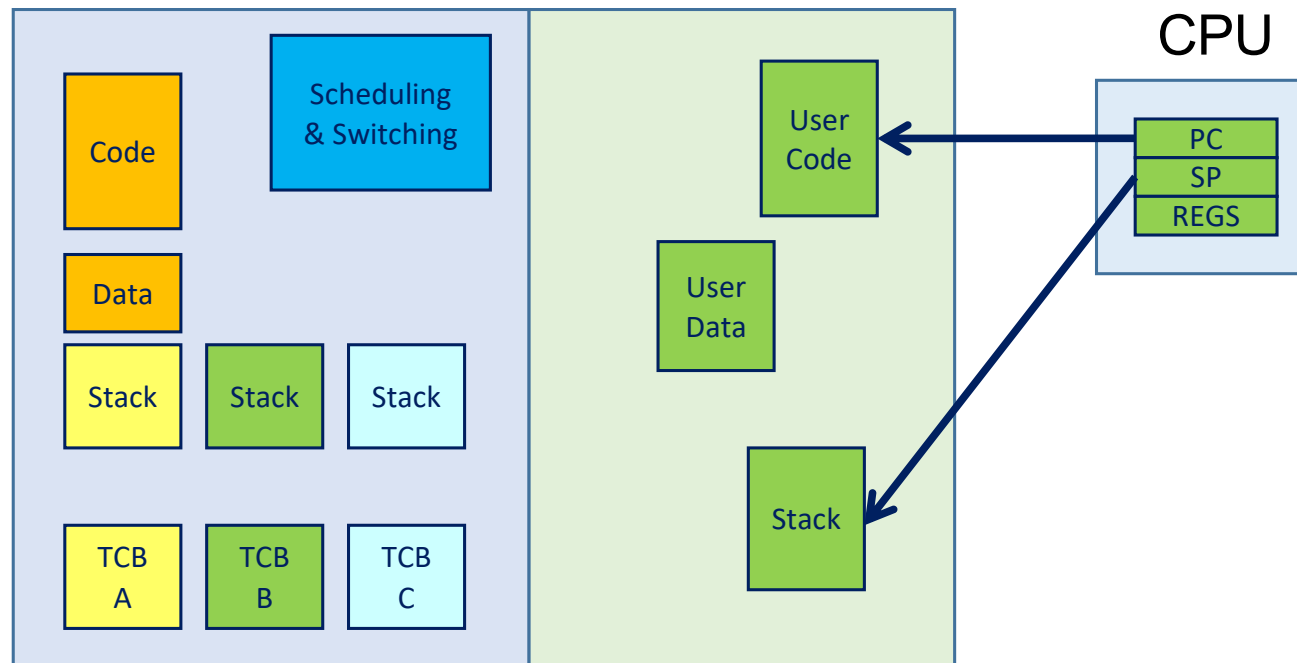
Thread Switch Switching Address Space: Process



Thread Switch Switching Address Space: Process

Kernel-only Memory

User Memory



Continuations

Continuations

Continuation:

- representation of an instance of a computation at a point in time
- the state and code where to *continue* from

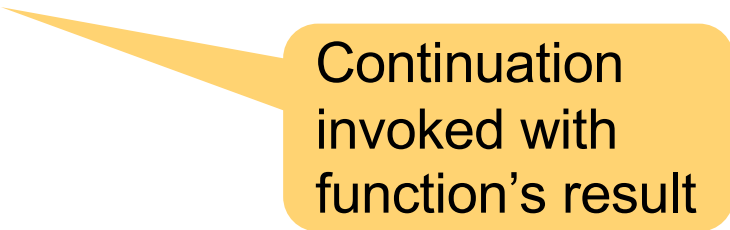
Continuations in PLs: Python

- Traditional function that returns:

```
def func(x):  
    return x+1
```

- Function with a continuation indicating where to continue

```
def func_cps(x, c):  
    c(x+1)
```



Continuation
invoked with
function's result

Continuations

The concept of capturing current (stack) state to continue the computation in the future

- In the general case can restore same state repeatedly
- C has one-shot continuations: `setjmp()`/`longjmp()`

OS Execution Models

OS Execution Model Alternatives

Single Kernel Stack

- One kernel stack supports all user threads
- “Event model” / “interrupt model”

Per-Thread Kernel Stack

- Each user thread has a separate kernel stack
- “Process model”

Per-Thread Kernel Stack

A thread's kernel state is implicitly encoded in the kernel activation stack

- If the thread must block in-kernel, we can simply switch from the current stack, to another threads stack until thread is resumed
- Resuming is simply switching back to the original stack
- Preemption is easy

```
example(arg1, arg2) {  
    P1(arg1, arg2);  
    if (need_to_block) {  
        thread_block();  
        P2(arg2);  
    } else {  
        P3();  
    }  
    /* return control to user */  
    return SUCCESS;  
}
```

- Dump registers on stack
- Switch stack
- Restore registers

Single Kernel Stack

How do we use a single kernel stack to support many threads?

- Issue: How are system calls that block handled?

⇒ Use *continuations*

- Used in Mach: *Using Continuations to Implement Thread Management and Communication in Operating Systems*. [Draves et al., 1991]

⇒ Use *stateless kernel* (event model)

- Used in Fluke: *Interface and Execution Models in the Fluke Kernel*. [Ford et al., 1999]
- Also used in seL4

Continuations

State required to resume a blocked thread is explicitly saved in a TCB

- A function pointer
- Variables

Stack can be discarded and reused to support new thread

Resuming involves discarding current stack, restoring the continuation, and continuing

```
example(arg1, arg2) {  
    P1(arg1, arg2);  
    if (need_to_block) {  
        save_arg_in_TCB;  
        thread_block(example_continue);  
        /* NOT REACHED */  
    } else {  
        P3();  
    }  
    thread_syscall_return(SUCCESS);  
}  
  
example_continue() {  
    recover_arg2_from_TCB;  
    P2(recovered arg2);  
    thread_syscall_return(SUCCESS);  
}
```

Logically, **p2 (arg2)**
exceuted here

Stateless Kernel

System calls cannot block within the kernel

- If syscall must block (resource unavailable)
 - Modify user-state such that syscall is restarted when resources become available
 - Stack content is discarded (functions all return)

Preemption within kernel difficult to achieve.

⇒ Must (partially) roll syscall back to a restart point

Avoid page faults within kernel code

⇒ Syscall arguments in registers

- Page fault during roll-back to restart (due to a page fault) is fatal.

Summary

Single Kernel Stack

- Either *continuations*
 - complex to program
 - must be conservative in state saved (any state that *might* be needed)
 - Mach (Draves), L4Ka::Strawberry, NICTA Pistachio, OKL4
- or *stateless kernel*
 - no kernel threads, kernel not interruptible, difficult to program
 - request all potentially required resources prior to execution
 - blocking syscalls must always be re-startable
 - Processor-provided stack management can get in the way
 - system calls need to be kept simple “atomic”.
 - e.g. the fluke kernel from Utah, seL4
- low cache footprint
 - always use the same stack!
 - reduced memory footprint

Per-Thread Kernel Stack

- simple, flexible
 - kernel can always use threads, no special techniques required for keeping state while interrupted / blocked
 - no conceptual difference between kernel mode and user mode
 - e.g. traditional L4, Linux, Windows, OS/161
- but larger cache footprint
- and higher memory consumption
- ... and more concurrency issues!

Why Threads Are A Bad Idea (for most purposes)

John Ousterhout

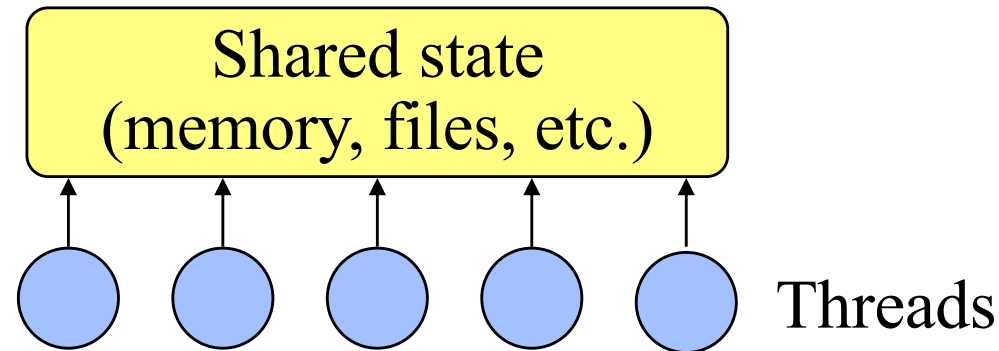
Sun Microsystems Laboratories

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

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?

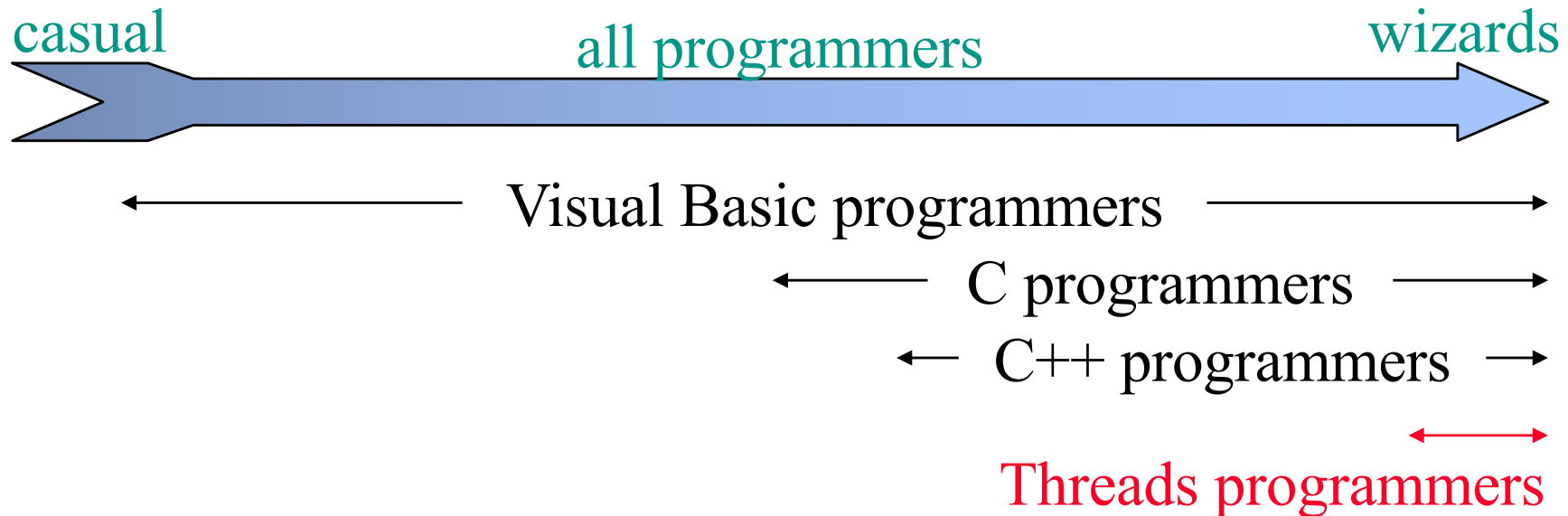


- ◆ **General-purpose solution for managing concurrency.**
- ◆ **Multiple independent execution streams.**
- ◆ **Shared state.**
- ◆ **Pre-emptive scheduling.**
- ◆ **Synchronization (e.g. locks, conditions).**

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.

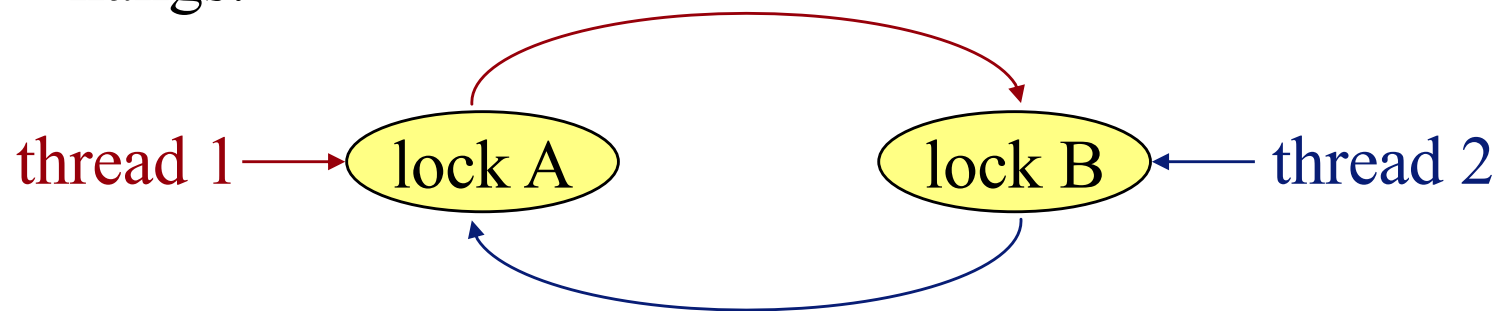
Why Threads Are Hard

◆ Synchronization:

- Must coordinate access to shared data with locks.
- Forget a lock? Corrupted data.

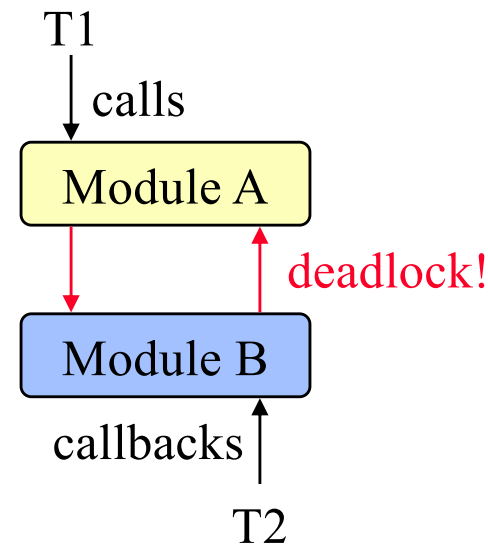
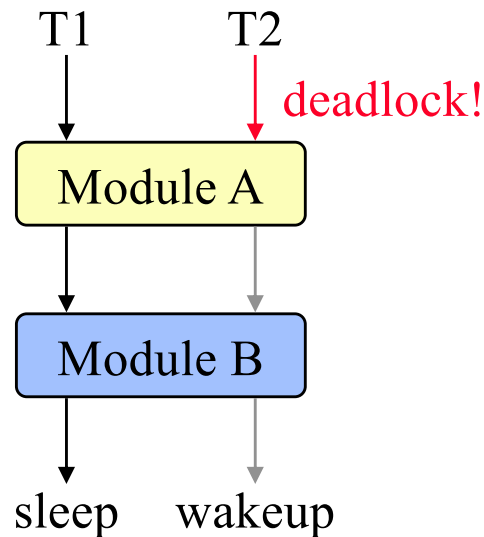
◆ Deadlock:

- 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.**

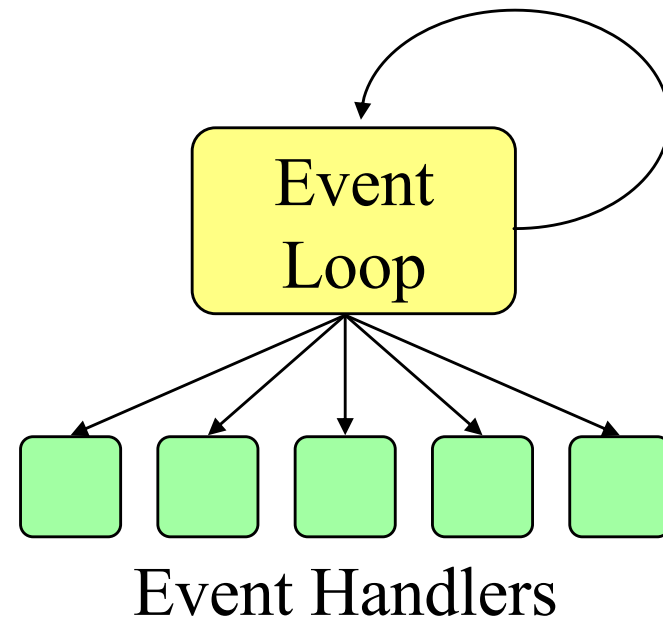


Why Threads Are Hard, cont'd

- ◆ **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 (reenetrancy 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

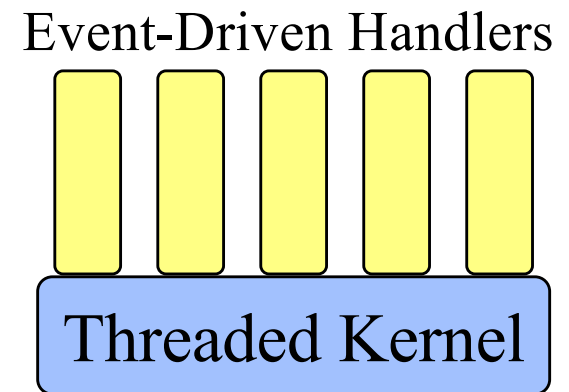
- ◆ **Events avoid concurrency as much as possible, threads embrace:**
 - Easy to get started with events: no concurrency, no preemption, no synchronization, no deadlock.
 - Use complicated techniques only for unusual cases.
 - With threads, even the simplest application faces the full complexity.
- ◆ **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?

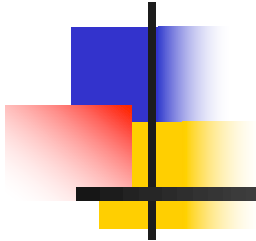
- ◆ **No:** important for high-end servers (e.g. databases).
- ◆ **But, avoid threads wherever possible:**
 - Use events, not threads, for GUIs, distributed systems, low-end servers.
 - Only use threads where true CPU concurrency is needed.
 - Where threads needed, isolate usage in threaded application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

... as in
“compute kernel”



Why Events Are A Bad Idea

(for high-concurrency servers)

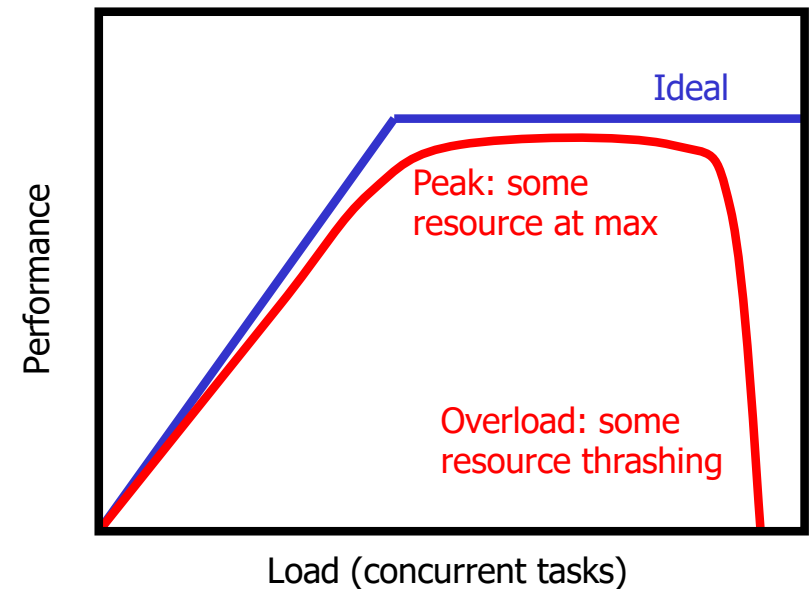
Rob von Behren, Jeremy Condit and Eric Brewer
University of California at Berkeley
{jrvb,jcondit,brewer}@cs.berkeley.edu
<http://capriccio.cs.berkeley.edu>

A Talk at HotOS 2003

Slide set courtesy of Rob von Behren, used with permission

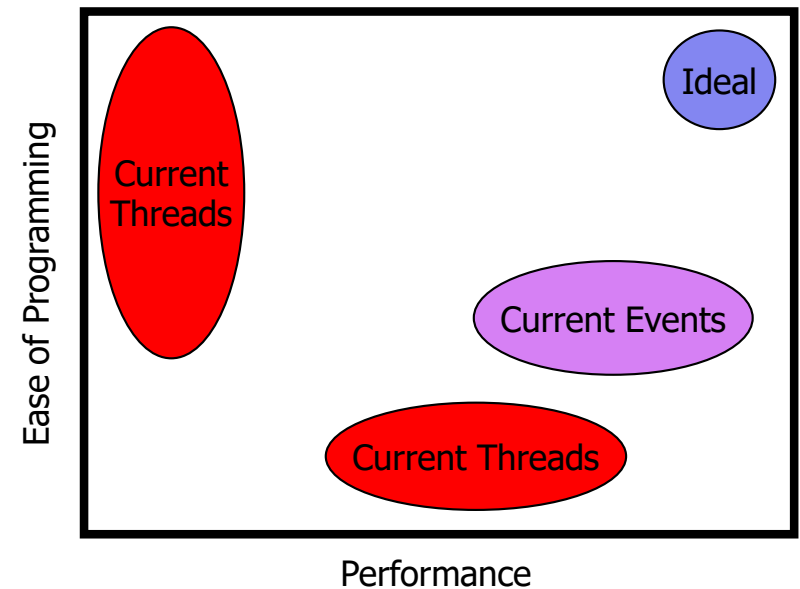
The Stage

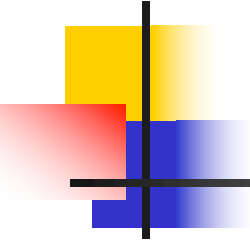
- Highly concurrent applications
 - Internet servers (Flash, Ninja, SEDA)
 - Transaction processing databases
- Workload
 - Operate “near the knee”
 - Avoid thrashing!
- What makes concurrency hard?
 - Race conditions
 - Scalability (no $O(n)$ operations)
 - Scheduling & resource sensitivity
 - Inevitable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

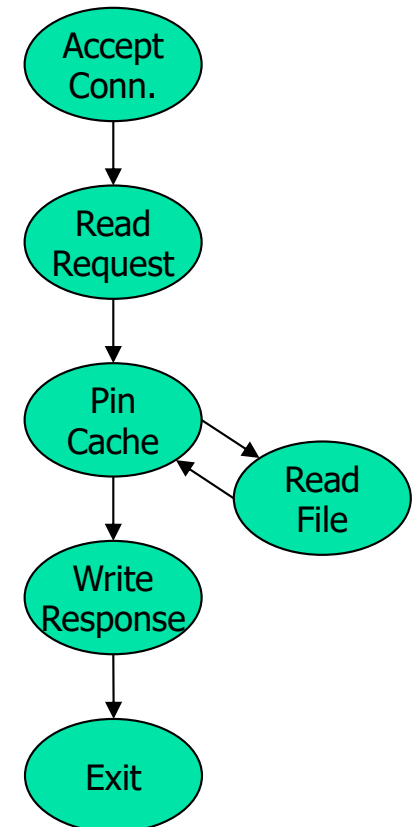
- Thread-event duality still holds
- But threads are better anyway
 - More natural to program
 - Better fit with tools and hardware
- Compiler-runtime integration is key

The Duality Argument

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

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

Web Server

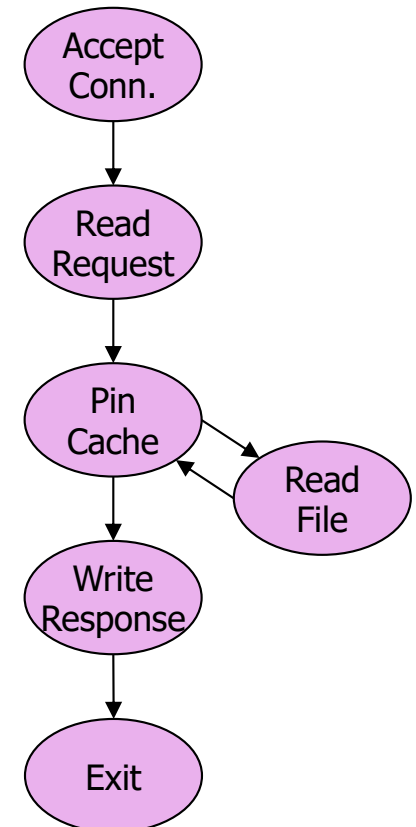


The Duality Argument

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

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

Web Server

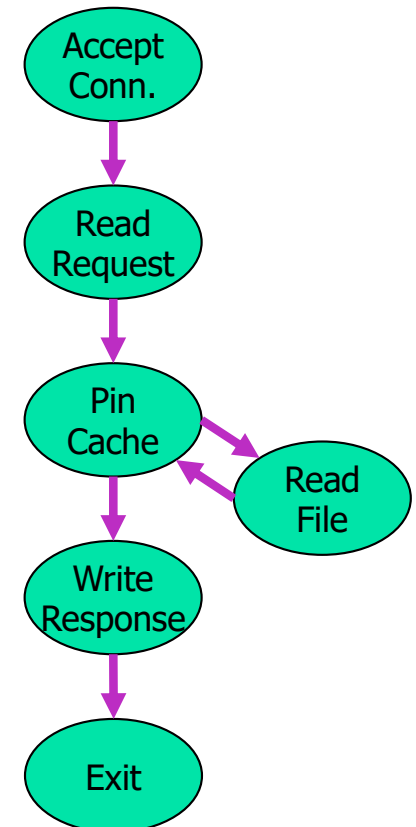


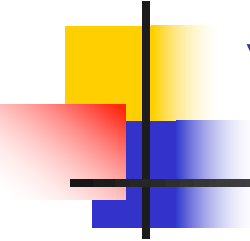
The Duality Argument

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

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

Web Server



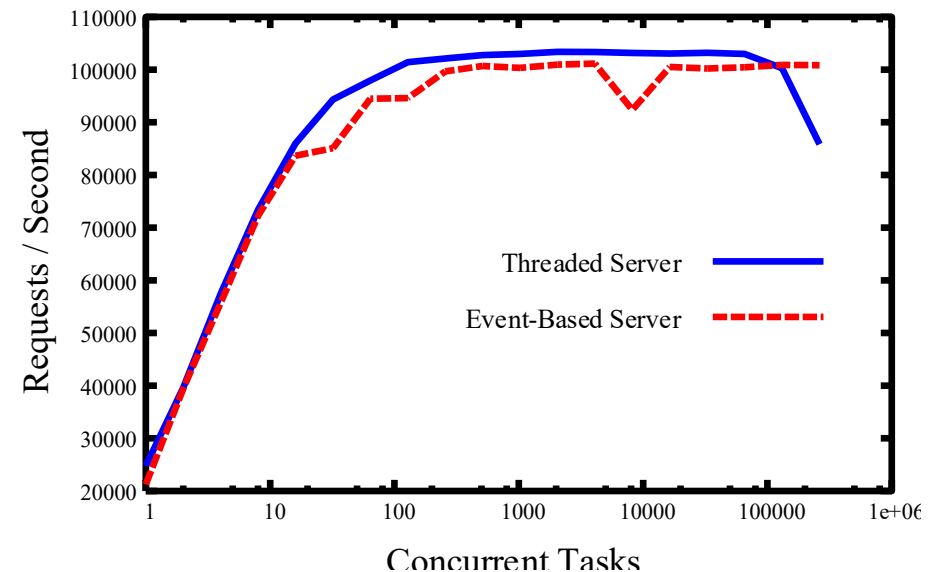


“But Events *Are* Better!”

- 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

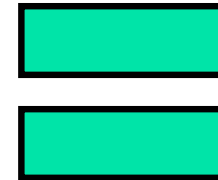
- *Criticism: Threads don't perform well for high concurrency*
- Response
 - Avoid $O(n)$ operations
 - Minimize context switch overhead
- Simple scalability test
 - Slightly modified GNU Pth
 - Thread-per-task vs. single thread
 - Same performance!



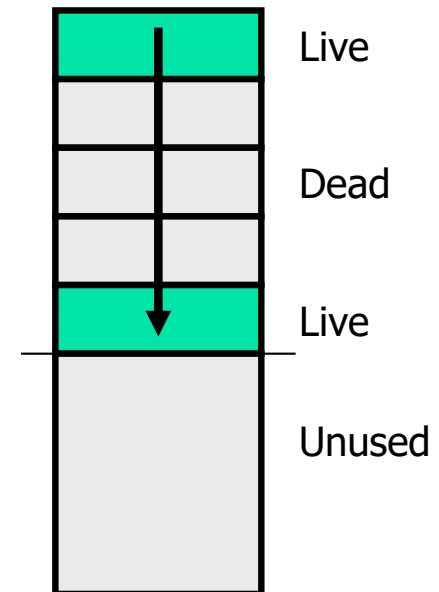
Live State Management

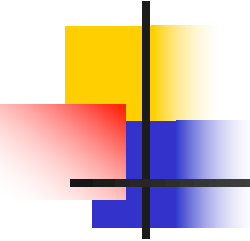
- *Criticism: Stacks are bad for live state*
- Response
 - Fix with compiler help
 - Stack overflow vs. wasted space
 - Dynamically link stack frames
 - Retain dead state
 - Static lifetime analysis
 - Plan arrangement of stack
 - Put some data on heap
 - Pop stack before tail calls
 - Encourage inefficiency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

Event State (heap)



Thread State (stack)



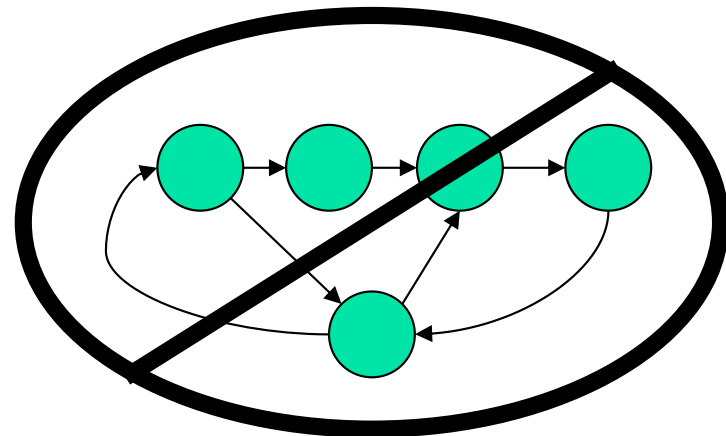
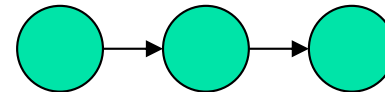
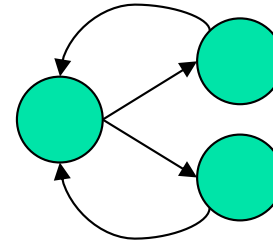
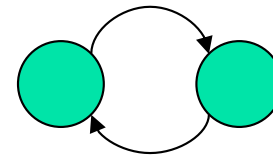


Synchronization

- *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

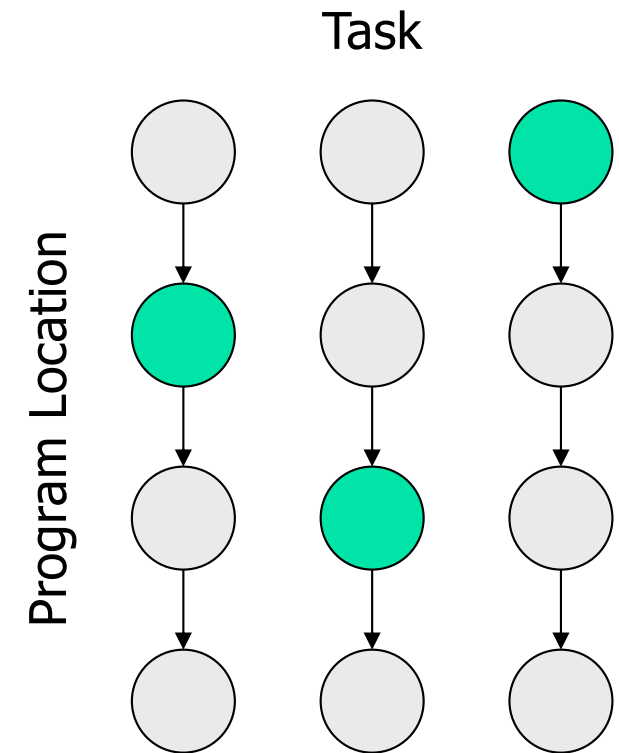
Control Flow

- *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



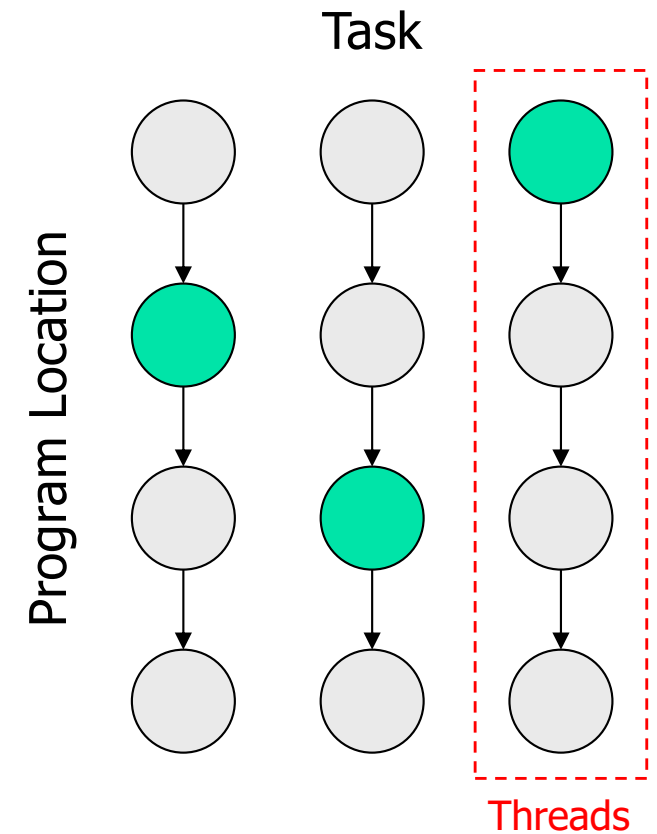
Scheduling

- *Criticism: Thread schedulers are too generic*
 - Can't use application-specific information



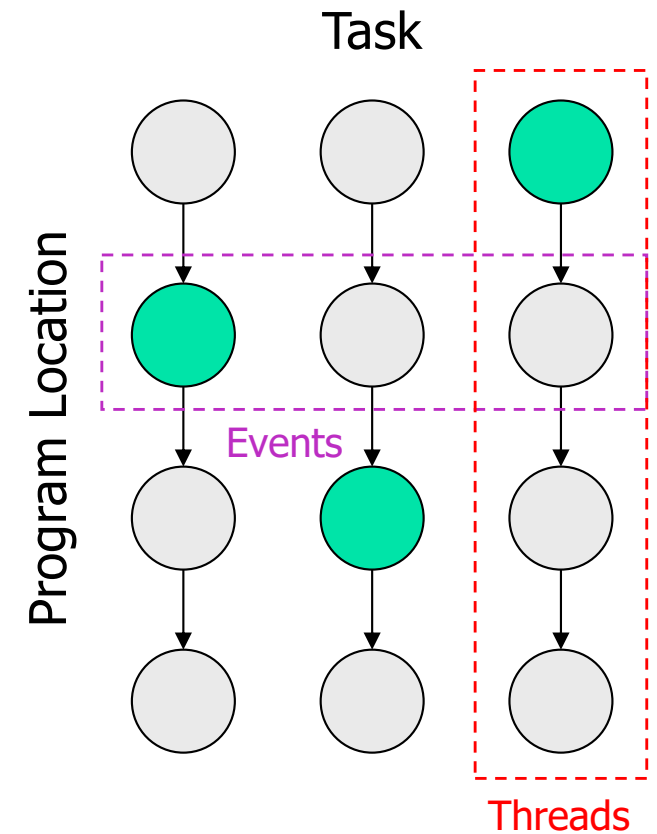
Scheduling

- *Criticism: Thread schedulers are too generic*
 - Can't use application-specific information



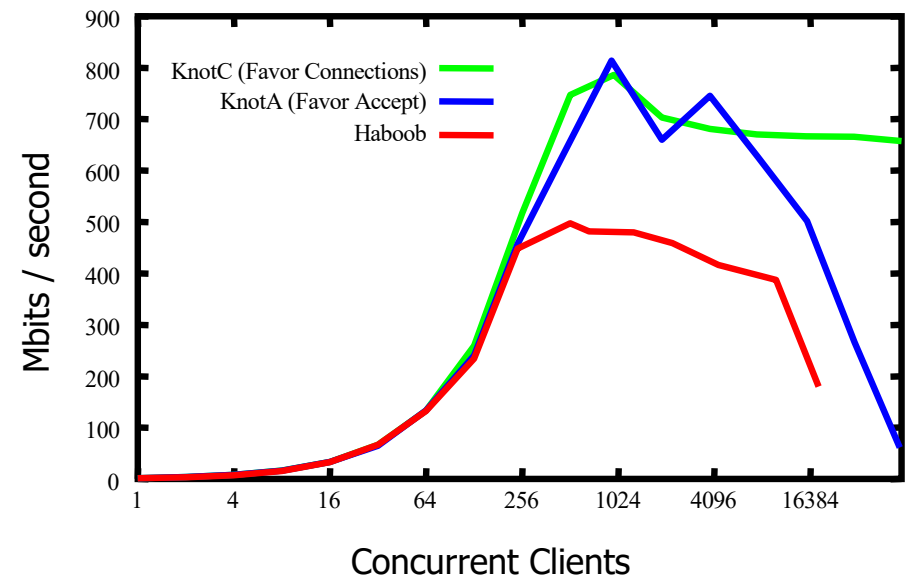
Scheduling

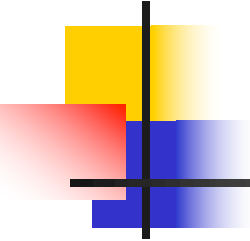
- *Criticism: Thread schedulers are too generic*
 - Can't use application-specific information
- Response
 - 2D scheduling: task & program location
 - Threads schedule based on task only
 - Events schedule by location (e.g. SEDA)
 - Allows batching
 - Allows prediction for SRCT
 - Threads can use 2D,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

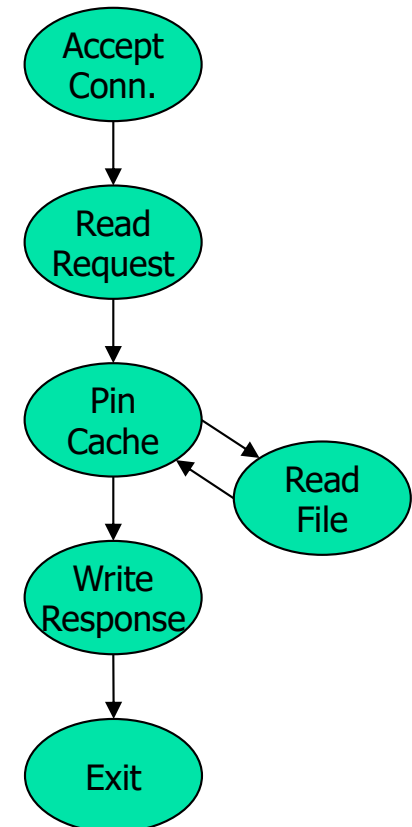
- More natural programming model
 - Control flow is more apparent
 - Exception handling is easier
 - State management is automatic
- Better fit with current tools & hardware
 - Better existing infrastructure
 - Allows better performance?

Control Flow

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

<i>Threads</i>	<i>Events</i>
<pre>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); }</pre>	<pre>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); } ... ExitHandler(struct session *s) { ...; unpin(&s); free_session(s); }</pre>

Web Server

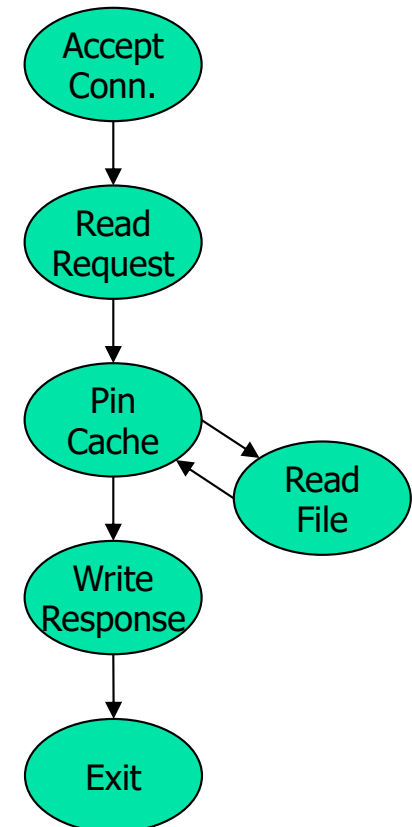


Control Flow

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

<i>Threads</i>	<i>Events</i>
<pre>thread_main(int sock) { struct session s; accept_conn(sock, &s); read_request(&s); pin_cache(&s); write_response(&s); unpin(&s); }</pre> <pre>pin_cache(struct session *s) { pin(&s); if(!in_cache(&s)) read_file(&s); }</pre>	<pre>CacheHandler(struct session *s) { pin(s); if(!in_cache(s)) ReadFileHandler.enqueue(s); else ResponseHandler.enqueue(s); } RequestHandler(struct session *s) { ...; CacheHandler.enqueue(s); } ... ExitHandler(struct session *s) { ...; unpin(&s); free_session(s); } AcceptHandler(event e) { struct session *s = new_session(e); RequestHandler.enqueue(s); }</pre>

Web Server

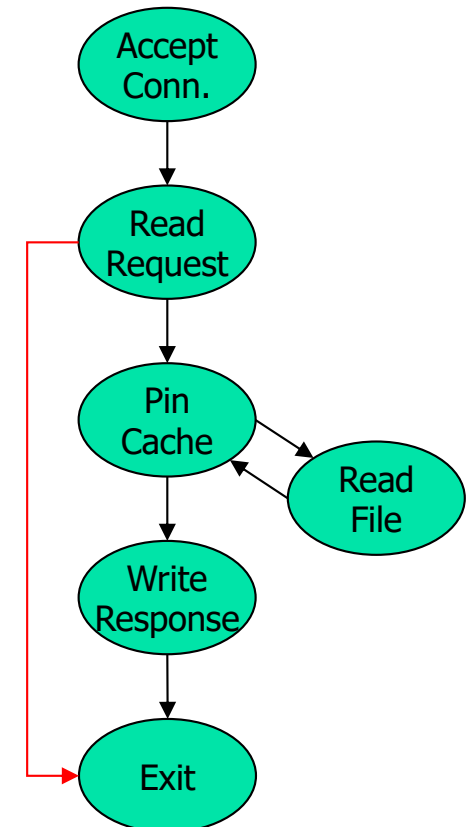


Exceptions

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

<i>Threads</i>	<i>Events</i>
<pre>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); }</pre>	<pre>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); } ... ExitHandler(struct session *s) { ...; unpin(&s); free_session(s); } AcceptHandler(event e) { struct session *s = new_session(e); RequestHandler.enqueue(s); }</pre>

Web Server



State Management

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

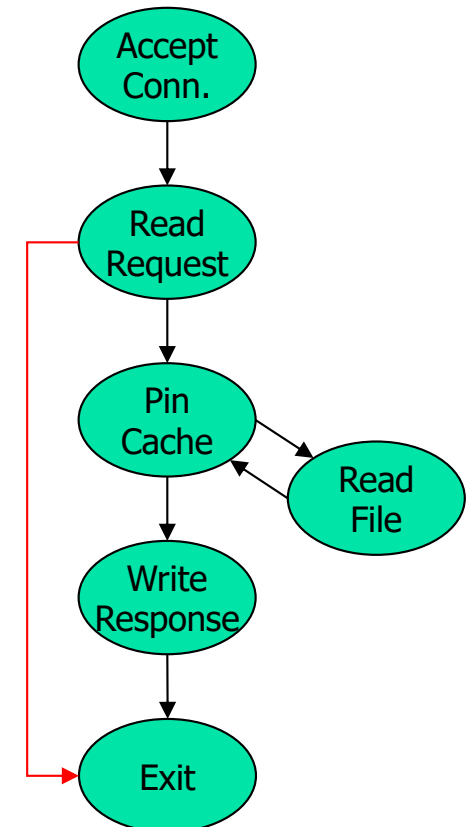
Threads

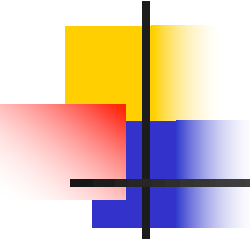
```
thread_main(int sock) {  
    struct session s;  
    accept_conn(sock, &s);  
    if( !read_request(&s) )  
        return;  
    pin_cache(&s);  
    write_response(&s);  
    unpin(&s);  
}  
  
pin_cache(struct session *s) {  
    pin(&s);  
    if( !in_cache(&s) )  
        read_file(&s);  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

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);  
}  
...  
  
ExitHandler(struct session *s) {  
    ...; unpin(&s); free_session(s);  
}  
  
AcceptHandler(event e) {  
    struct session *s = new_session(e);  
    RequestHandler.enqueue(s); }
```

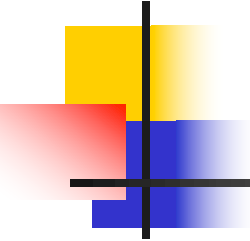
Web Server





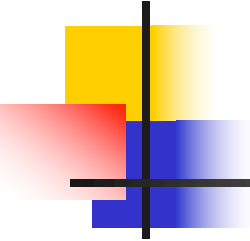
Existing Infrastructure

- 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