Events vs. Threads

COS 316: Principles of Computer System Design

Amit Levy & Wyatt Lloyd

Two Abstractions for Supporting Concurrency

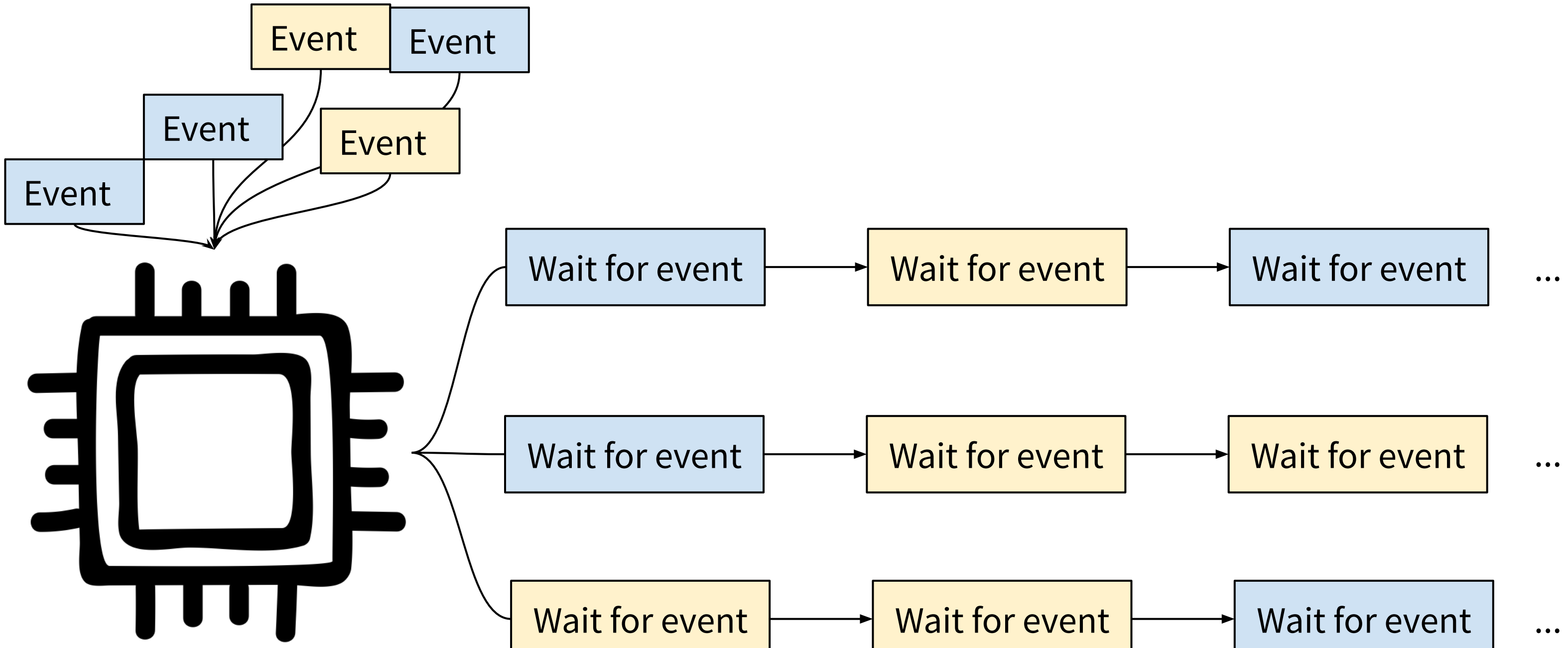
Problem: computationally light applications with large state-space

- How to support graphical interfaces with many possible interactions in each state?
- How to scale servers to handle many simultaneous requests?

Corollary: how do we best utilize the CPU in I/O-bound workloads?

- Ideally, CPU should not be a bottleneck for saturating I/O

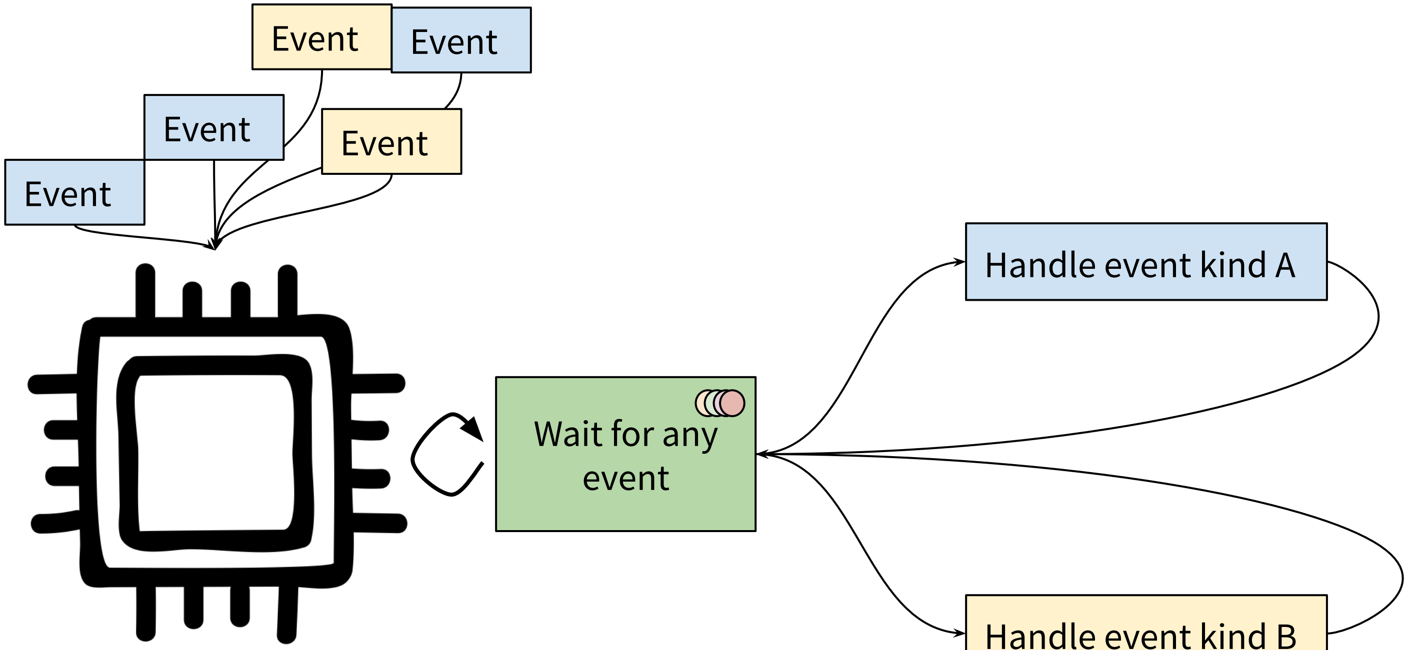
Threads


Event
Event
Event
Event
Event
Event
Wait for event
Wait for event
Wait for event
...
Wait for event
Wait for event
Wait for event
...
Wait for event
Wait for event
Wait for event
...

Threads

- One thread per sequence of related I/O operations
  - E.g. one thread per client TCP connection
- When next I/O isn't ready, block thread until it is
  - i.e. return to event handler instead of blocking for next event
- State machine is implicit in the sequential program
  - E.g., parse HTTP request, then send DB query, then read file from disk, then send HTTP response
- Only keeping track of relevant next I/O operations
- Any shared state must be synchronized
  - E.g. with locks, channels, etc
- In theory, scale by having a thread per connection

Events


Event
Event
Event
Event
Event
Event
Wait for any event
Handle event kind A
Handle event kind B

Events

- Single "event-loop" waits for the next event (of any kind)
- When event arrives, handle it "to completion"
  - i.e. return to event handler instead of blocking for next event
- Event loop keeps track of state machine explicitly
  - How to handle an event depends on which state I'm in
  - E.g. an HTTP parser will parse next incoming packet differently depending on what we've parsed so far
- In theory, one thread can handle infinite concurrent potential events
  - E.g. infinite number of client TCP connections
- In theory, no synchronization necessary
  - Each event handler has non-concurrent access to shared state
- In theory, scale by having an event-loop per CPU core

Which style should systems support?

History of the Debate

- 1978 - Lauer & Needham
  - "procedure-oriented" and "message-oriented" systems are duals
- 1995 - Ousterhout: "Why Threads Are A Bad Idea"
- 2001/2002 - Matt Welch builds SEDA
- 2002 - Adya et al. "Cooperative Task Management without Manual Stack Management"
- 2003 - Behrer, Condit, Brewer: "Why Events Are A Bad Idea"
- 2009/10 - NodeJS & ***Golang***
- 2012 - Linux addresses non-blocking I/O performance issues with `epoll`
- 2020 - is the debate still relevant?
  - Spoiler alert: I think it is!

How do Threads and Events Compare?

Some of the debate has been around usability---which is easier to program.

Much of the debate has focused on “object” measures:

1. Performance
2. Control Flow
3. Synchronization
4. State Management
5. Scheduling

Performance

Threads

- Heavier weight *abstraction* of hardware
  - Blocking I/O naturally translates to polling, but that's not what we want
- Context switching can be expensive
- System ops for threads often O(n)
  - But this is not fundamental!

Events

- Abstraction maps neatly to most hardware
  - With interrupts turned off, a CPU-core is basically an event system
- Result: easier to implement efficiently
- Why: complexity is deferred to applications
  - This can be good for apps!
- Operations can be O(n) in event types
  - Fixed in the last decade

Control Flow

Threads

- Linear control flow implied
  - Do this I/O, then this I/O, then this I/O
  - This is the *most* common control flow
- Simple to support
  - Call/return
  - Parallel calls
  - Pipelines
- More complex flows don’t fit well
  - Dynamic fan-in or fan-out

Events

- No particular control-flow implied
  - Call/return
  - Parallel
  - Pipelines
  - Fan-in/fan-out
- In practice, complex control-flows are uncommon
- Simple control-flows are “harder” to program
  - Ousterhout disagrees!
  - Notably hard: exceptions & error handling

Synchronization

Threads

- Synchronization across threads is hard
  - Locks, condition variables, channels
  - Difficult to reason about
  - ***Very*** difficult to test
- Synchronization necessary even when not fundamental
  - E.g. on a uniprocessor

Events

- Synchronization for “free”
  - Single thread of execution means no need to synchronize on shared state
- Well… only true for uniprocessors
  - Lots of applications are single-core
  - Scalable servers typically multi-core
  - Multi-core event-loops still need explicit synchronization

State Management

Threads

- State encapsulated on the stack
- Programmer doesn't need to think about what state to track between I/O
  - Implied in local variables, etc
- State-machine implied in linear flow of program
- Need a stack for each frame
  - How big should the stack be?
  - Too small? Stack overflow
  - Too big? Memory waste

Events

- Programmer tracks state explicitly
  - Called "Stack ripping"
- Global, shared variables
- Can make state difficult to track
  - Particularly if state-space is large
- No need for separate stacks
- Memory usage (can be) optimal

Scheduling

Threads

- Typically: threads scheduled *preemptively* by system
- Provides performance isolation
  - Threads with long-running computation don't starve other threads
- Programmer doesn't have to think about scheduling
- Requires worst-case context switching
- Scheduling decisions can be sub-optimal for application

Events

- Cooperatively scheduled
  - Event handlers yield by returning to event loop
  - Next event handler runs only when previous completes
- Scheduling decisions deferred to application
  - E.g. event loop can prioritize advantageous events
  - Only true if you write your own event loop
- Hard to achieve fairness
  - Hard to debug and test fairness regressions

Which style should systems support?

Unifying Events and Threads

Behrer, Condit & Brewer: "Why Events Are A Bad Idea (for high-concurrency servers)" HotOS 2003

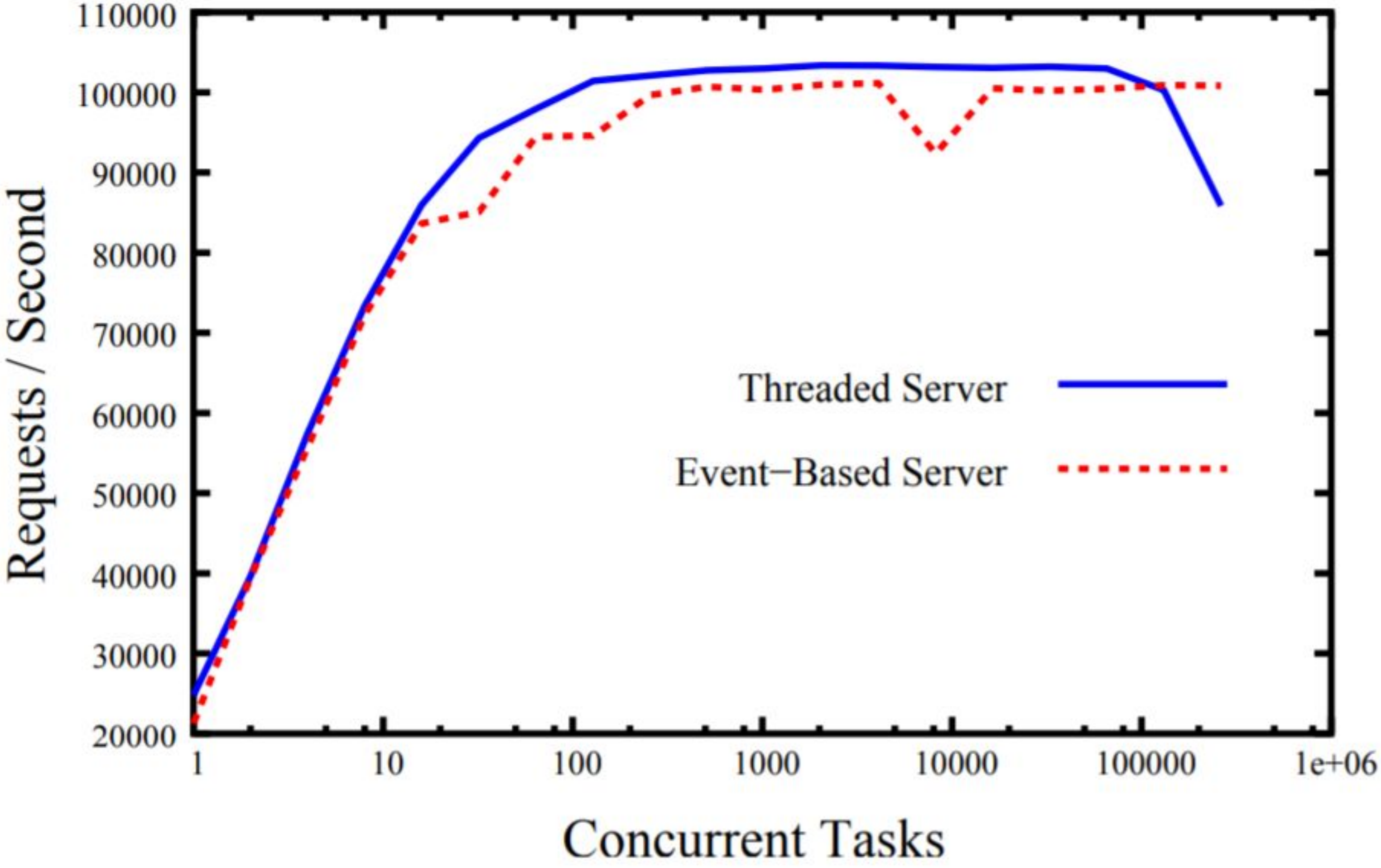
Premise: threads are easier to program than events

- This is probably not true for GUIs, games, etc
- It’s at least closest to what we learn in 126

Key idea: better implementations and the compiler can (and should!) address the problems with threads

Unifying Events and Threads

- Scheduling: threads can be cooperatively scheduled too!
  - Key idea in Go's goroutines
- Performance: existence proof of a lightweight thread system
  - Many previous examples, especially from language runtimes (ML, Erlang)
- Control-flow: not really a problem
- Synchronization: compiler analysis can reduce/eliminate occurrence of concurrency bugs
  - Haskell, Rust languages take this to extreme
- State management
  - Dynamic stack growth: start with a small stack and dynamically allocate more as necessary
    - Go, Haskell
  - Compiler can purge unnecessary stack variables


110000
100000
90000
80000
70000
60000
50000
40000
30000
20000
Requests / Second
1
10
100
1000
10000
100000
1e+06
Concurrent Tasks
Threaded Server
Event−Based Server

Two Paths: Node.js & Go

- Around 2009, it seemed time for new runtimes for building servers
  - Java was showing its age
  - C++ perceived as unnecessarily hard
  - Popular languages not “fast enough” (PHP, Python, Ruby)
    - Note that most performance sensitive code in these languages actually in C
  - Cool kids don’t want to use good languages for some reason (ML, Lisp, Haskell etc)
- Node.js:
  - Chrome made JavaScript fast with V8’s just-in-time compiler
  - Everyone already knows JavaScript!
  - Events are faster than threads!
- Go:
  - Write a language and compiler from scratch targeted specifically at building scalable servers
  - I’m Ken “f&#*$ing” Thompson and I don’t have to care what language you already know!
  - Threads are easier to program, and we can make them just as fast!

Goroutines

- Threaded programming model
  - I/O operations are blocking
  - Spawn a goroutine per-connection, request, etc
- Lightweight:
  - Dynamically sized stacks (starts at 2KB, grows based on compiler analysis)
  - Context switching in language runtime, only save minimal CPU state
- Cooperatively scheduled
  - Uses implied save-points to yield:
    - Function calls
    - I/O operations
    - Recently, loop iterations
- M:N threading model: schedule M goroutines onto N CPU cores (kernel threads)

The State of Events vs. Threads in 2020

- General-purpose operating systems have converged on…. both
  - Posix Threads on Linux (LinuxThreads) continue to improve in performance
  - epoll solves bottlenecks in previous event system calls
  - Most highly-concurrent servers use multiple threads, each running an event-loop
- Switch to 64-bit address space & dramatic increase in RAM have circumvented the stack consumption problem on servers
  - 64-bit virtual addresses means it's easy to allocate many virtually unbounded stacks
  - Lots of memory means that overallocated unused memory isn't so bad (on the order of MBs)
- Most common language runtimes just use Posix Threads
  - Python, Ruby, Java, Rust
  - Turns out to be fine in the vast majority of applications

The State of Events vs. Threads in 2020

Some back of the envelope math:

- Facebook has millions of servers
- Facebook has ~1.5 billion users
- If every user established a TCP connection at the same time
  - 1,500,000,000 / 1,000,000 = 1,500 connections/server
  - 1,500 threads * 4MB/thread = ~6GB of RAM
- Most applications are not Facebook

Maybe it’s just not that important anymore. Whatever is most natural to program…

Not all Systems in 2020 are Network Servers

- Embedded "Internet of Things" systems have very low memory
  - 16KB to, at most, 1MB
  - 32-bit addresses and often no virtual memory
  - Even with Golang's small goroutine stacks, we could only have 4
- Real-time systems
  - Both threads and events can make it hard to estimate compute time
  - Threads: task switching modulates block I/O time
  - Events: can't reliably ensure event progress without preemption
- Performance isolation
  - When running untrusted code, we might *need* preemption to ensure performance isolation

Takeaways

- Often, there are no clear answers to system design questions
  - Are threads or events easier to program? How would you measure?
  - Are threads or events more performant? How to disentangle from implementation details?
- True not just for events vs. threads:
  - Garbage collection vs. manual memory management
  - Asynchronous messaging vs RPC
  - Hardware vs. software fault isolation
  - As we'll see, access control: mandatory vs. discretionary access control, capabilities vs. information flow control
- Measuring performance of an actual system is the best way to get a concrete answer to this question

You should always plan to build *at least* two systems from scratch. You will throw away the first one once you learn what are the hard problems.