6.828: Operating Systems Research Seminar

Adam Belay <abelay@mit.edu>

Welcome to 6.828!

- Theme for this semester: **Datacenter Operating Systems**
- This is a seminar-style class
- Goals:
  1. Read, discuss, and present research in operating systems
  2. Advance your research through an open-ended, final project
- We'll learn about cutting-edge systems built by researchers and industry
- And research problems that must be solved to build the next generation of datacenters

Background

- Undergraduate-level understanding of operating systems
- Low-level programming experience (e.g., C, C++, Rust, etc.)
- Ability to understand and think critically about research papers in computer systems

Which class to take

6.S081

- Basic operating systems concepts, intended for undergrads
- Projects focus on OS hacking

6.828

- Operating systems research, intended for graduate students and students who have already taken 6.S081 (or an equivalent class)
- Projects focus on OS research

Both classes meet on Monday and Wednesday at 1:00-2:30PM EST

Action items

- Fill out the class survey by the end of the week (09/4/20). Details will be posted on Piazza
  - Responses will inform the final class schedule.
- If you don't have the right background, consider registering for 6.S081 instead
  - Email us if you have questions

Logistics: Grading

- 20% Class Participation
- 20% Paper Summaries and Presentations
- 10% Warm Up Lab Assignment
- 50% Final Project

Logistics: Participation

- Read every paper and be prepared to discuss them in class
- Interactive lectures are encouraged! It's okay to unmute your microphone and ask a question anytime
- The night before each class, submit a question you have about the paper on Piazza

Logistics: Paper Presentations

- Each student will present a paper (one most likely) from the reading list and lead a discussion in the class
- We will meet with you ~1 week before to provide feedback on a draft of the presentation
- After the presentation, your slides (or notes) will be posted on the course website
- The course staff will present the first few papers

Logistics: Warm Up Lab

- Goal: Gain experience in low-level systems hacking and performance optimization
- Preliminary plan: DPDK hands-on exercise
- CloudLab is a helpful resource
  - Will provide a tutorial next week
- Stay tuned for more details

Logistics: Final Project

- Groups-based projects (current plan is two)
- Three checkpoints: Proposal, Draft Design, and Results
- Schedule and project ideas will be posted soon
- Presentations at end of class
- Good projects will make a research contribution, not just OS hacking
- It's okay if it's related to your current graduate research, but must align with the theme of the course

6.828 Themes

1. Host Networking
2. Compute + Accelerators
3. Memory + Storage
4. New Datacenter Applications

- Shifts occurring in each of these spaces
- Challenge: **Balance**: A system is balanced when each resource is sized appropriately for the demands placed on it
- Amdahl: 1Mbit/s of IO for every 1Mhz of compute

Significant advances in I/O

2010

2020

Networks

Latency: ~10 us
Throughput: 10 Gbps

5x

10x

Latency: ~2 us
Throughput: 100 Gbps

Storage

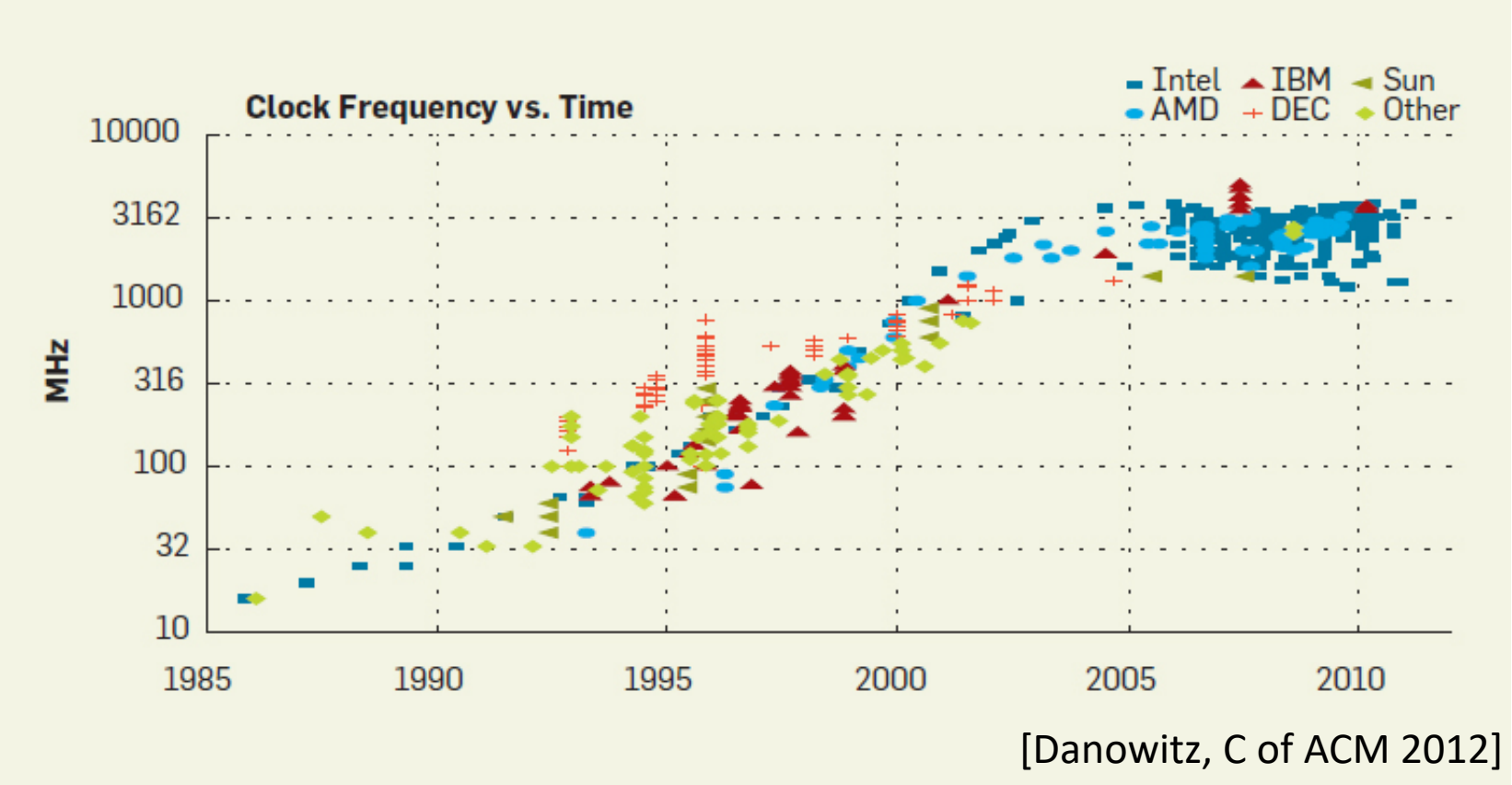
Latency: 75 us
Throughput: 35 krps

7.5x

21x

Latency: 10 us
Throughput: 750 krps

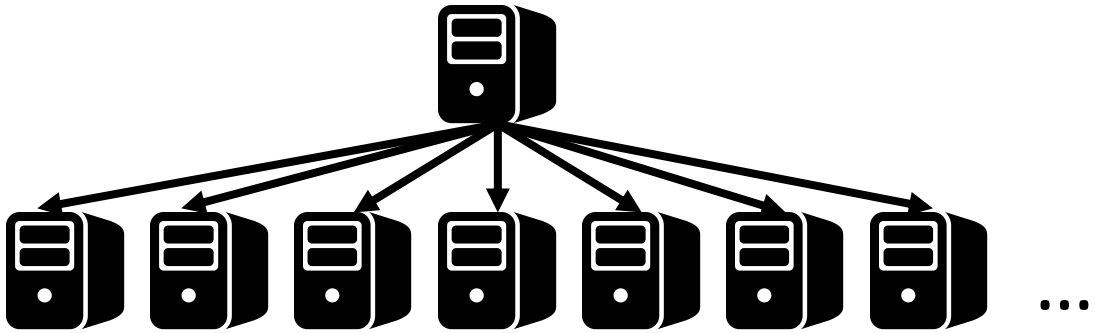
But end of Dennard Scaling



[Danowitz, C of ACM 2012]

Opportunity: Scaling out

- Today, CPUs are 20x slower than they would be if they followed Moore's law
- Use more machines instead!
- Complex control flow, resource disaggregation
- Requires OS support for low **tail latency** and high **networking throughput**



Resource Disaggregation

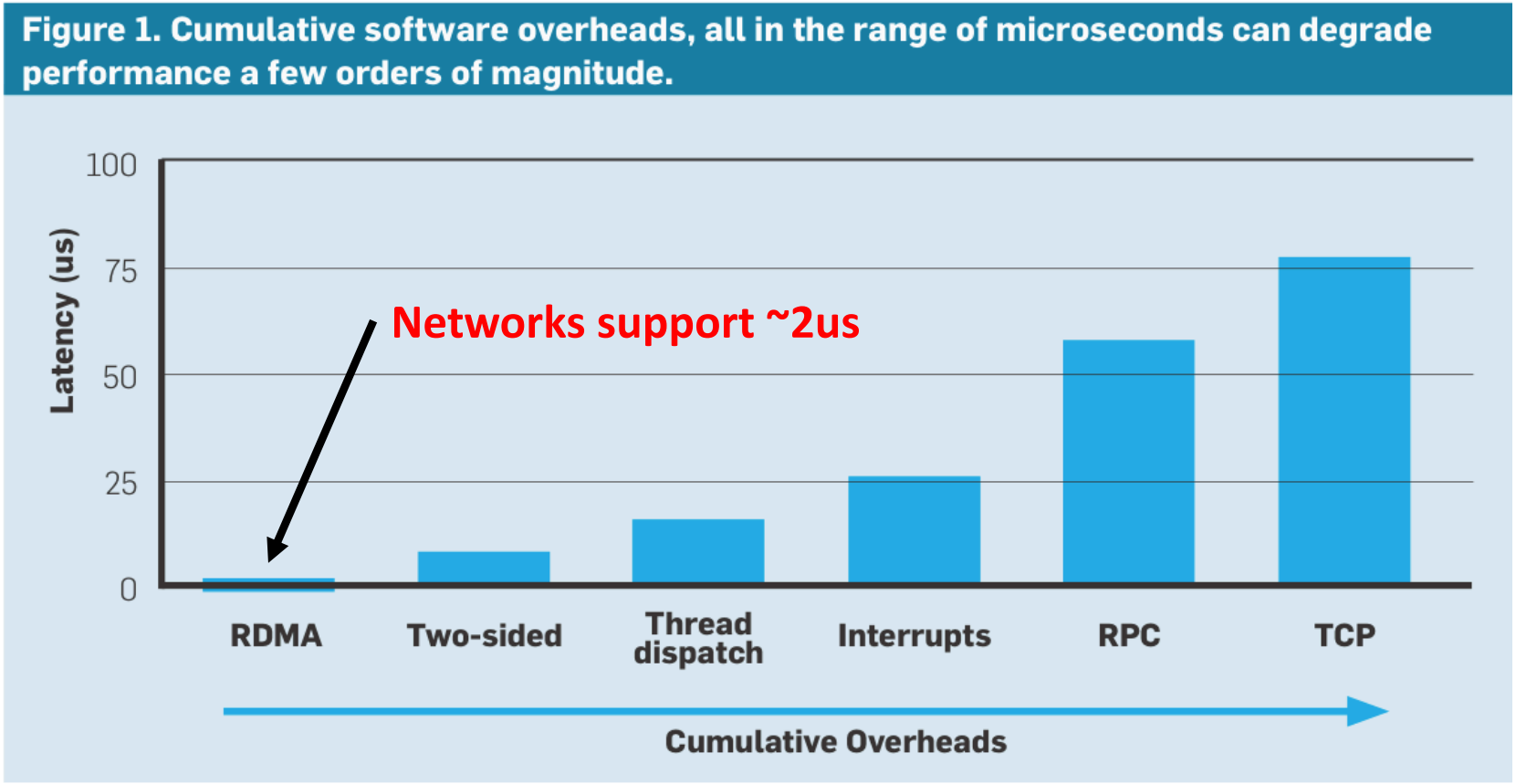
- Assume a typical server has 64 cores @ 3 Ghz
- Back of the envelope (Amdahl):

  64 * 3000 * 1 Mbit / s = 192 Gbit / s
- 100 GbE available now, 200 GbE soon
- Implication: Remote resources (e.g. flash, memory, accelerators, etc.) are becoming as fast as local resources!
- Easier to achieve balance between resources!

Paper Discussion: Attack of the Killer Microseconds

BY LUIZ BARROSO, MIKE MARTY, DAVID PATTERSON, AND PARTHASARATHY RANGANATHAN

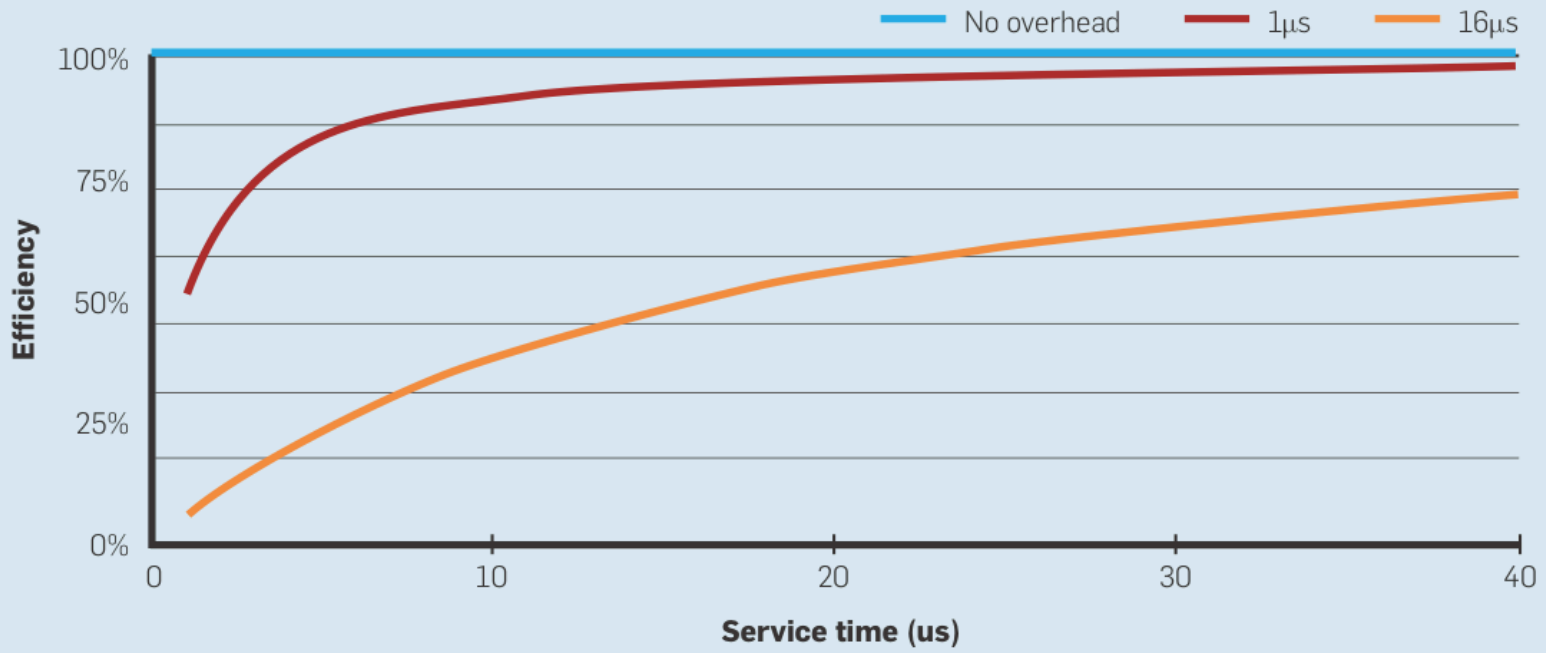
How to waste a fast network



Attack of the Killer Microseconds [CACM'17]

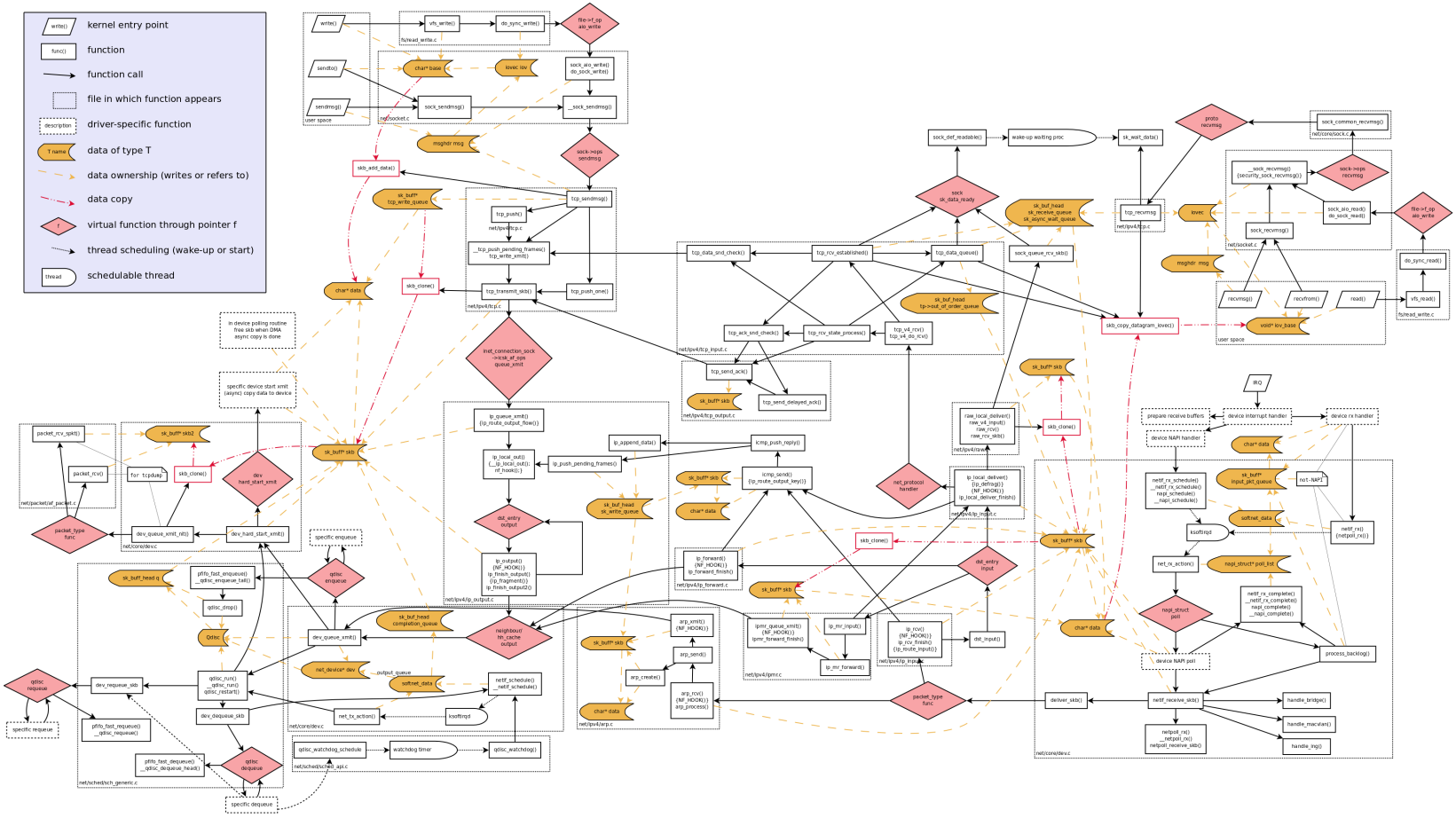
How to waste a fast CPU

Figure 2. Efficiency degrades significantly at low service times due to microsecond-scale overheads.



Attack of the Killer Microseconds [CACM'17]

The Linux Kernel networking stack



The OS is the bottleneck

- High software overheads
  - Too many layers, too general purpose
  - Programmers don’t care about TCP sockets!
- Not designed for microseconds
  - OS scheduler can add millisecond delays
  - Competition over shared CPU resources can too
  - Designed to hide disk seek latency (milliseconds)
- Strawman: Busy-poll for I/O completions

Different objectives

HPC:

- Okay to waste resources (e.g. MPI often spins waiting)
- Fixed set of data structures, time to optimize
- Emphasis: Performance

Data warehouses:

- Maximizing resource utilization critical for cost and energy
- Rapid churn, new code shipped every couple weeks
- Emphasis: Total cost of ownership (Utilization)

Latency hiding

- Can’t busy poll! Would waste the CPU.

Table 2. Service times measuring number of instructions between I/O events for a production-quality-tuned web search workload. The flash and DRAM data are measured for real production use; the bolded data for new memory/storage tiers is based on detailed trace analysis.

Task: data-intensive workload	Service time (task length between I/Os)
Flash	225K instructions = O(100μs)
Fast flash	***~20K instructions = O(10μs)***
New NVM memory	***~2K instructions = O(1μs)***
DRAM	500 instructions = O(100ns–1μs)

Attack of the Killer Microseconds [CACM’17]

Potential solutions

- Make context switching faster in hardware
  - Prior solutions trade single-threaded performance
  - But this kills latency, unacceptable
- Make context switching faster in software
  - E.g. Golang and Erlang provide “green threads” that bypass kernel
  - But I/O handling falls back on heavy OS mechanisms
- New hardware offloads needed?
  - E.g. MWAIT to track multiple memory locations
  - Cache thread registers?
  - Improved quality of service mechanisms?
  - Shift power to other cores when blocked?

Synchronous vs. Asynchronous

```
// Returns the number of words found in the document specified by 'doc_id'.
int CountWords(const string& doc_id) {
  Index index;
  bool status = ReadDocumentIndex(doc_id, &index);
  if (!status) return -1;
  string doc;
  status = ReadDocument(index.location, &doc);
  if (!status) return -1;
  return CountWordsInString(doc);
}
```

```
// Heap-allocated state tracked between asynchronous operations.
struct AsyncCountWordsState {
  bool status;
  std::function<void(int)> done_callback;
  Index index;
  string doc;
};

// Invokes the 'done' callback, passing the number of words found in the
// document specified by 'doc_id'.
void AsyncCountWords(const string& doc_id, std::function<void(int)> done) {
  // Kick off the first asynchronous operation, and invoke the
  // ReadDocumentIndexDone when it finishes. State between asynchronous
  // operations is tracked in a heap-allocated 'state' object.
  auto state = new AsyncCountWordsState();
  state->done_callback = done;
  AsyncReadDocumentIndex(doc_id, &state->status, &state->index,
                         std::bind(&ReadDocumentIndexDone, state));
}

// First callback function.
void ReadDocumentIndexDone(AsyncCountWordsState* state) {
  if (!state->status) {
    state->done_callback(-1);
    delete state;
  } else {
    // Kick off the second asynchronous operation, and invoke the
    // ReadDocumentDone function when it finishes. The 'state' object
    // is passed to the second callback for final cleanup.
    AsyncReadDocument(state->index.location, &state->status,
                      &state->doc, std::bind(&ReadDocumentDone, state));
  }
}

// Second callback function.
void ReadDocumentDone(AsyncCountWordsState* state) {
  if (!state->status) {
    state->done_callback(-1);
  } else {
    state->done_callback(CountWordsInString(state->doc));
  }
  delete state;
}
```

Attack of the Killer Microseconds [CACM'17]

History: Threads vs. events

- Debated over many decades!
- Why Threads Are a Bad Idea (for most purposes)
- Why Events Are A Bad Idea (for high-concurrency servers)
- C10k Problem in the 90's
  - Can a web server handle 10K clients simultaneously?
  - Balance constraint: Memory (Stack frames are 2MB)
  - Event-based programming won (e.g. Libevent)
  - Is that still the right answer?

How fast can threads be? (ns)

	Mutex Acquire Release	Yield PingPong	Condvar PingPong	Spawn Join
Linux	32	419	1,536	11,680
Golang	**24**	115	293	443
Arachne [OSDI 18]	59	90	214	642
Shenango [NSDI 19]	36	**27**	**84**	**130**

Events in 2020: Rust

```
async fn learn_and_sing() {
let song = learn_song().await;
   sing_song(song).await;
}

async fn async_main() {
   let f1 = learn_and_sing();
   let f2 = dance();
   // learn to sing and dance at the same time (concurrently)
   futures::join!(f1, f2);
}

fn main() {
   block_on(async_main());
}
```

But so far, overhead is high! (relatively speaking)

Conclusion

- New operating systems and systems software needed to keep up with hardware trends
- If solved, network advances enable disaggregation
- Soon: Remote resources = local resources
- Opportunities for 10x improvements in efficiency!
- Total cost of ownership is not the same as performance

Reminder: Fill out the class survey on Piazza.