Announcements

- Revised project proposals due next week
  - Meet w/me if necessary
- For survey papers:
  - Use proper citations (at least: author(s), title, published where/when)
  - Ideally, provide URL also
  - Finalize reading list
- Midterm upcoming: Nov 2

Recap

- Motivation for multi-threaded programming
- Implementation options
  - User vs. kernel level
  - Preemptive vs. nonpreemptive
  - (process-)blocking vs. nonblocking I/O

Lightweight Processes (M:N)

Outline for Today

- Threads vs. Events
  - As a programming model
  - In high-concurrency servers: SEDA vs. Capriccio

Programming with threads

- Ousterhout 1996: "Why Threads are a bad idea (for most purposes)"
- Threads are hard (only for experts)
  - Synchronization
  - Possibility of deadlock
  - Hard to debug
Ousterhout: Why … (cont’d)

- Good performance is hard to achieve
  - Simple locking ("BKL") yields low concurrency
  - Fine-grained locking has common-case overhead
  - Actual performance strongly implementation-dependent

- (1996) Threads not well-supported in legacy APIs: e.g., standard libraries

Ousterhout’s Conclusion

- Use threads only if true CPU concurrency is needed
  - Computations on multiprocessors
  - High-end servers:
    - Overlap I/O
    - Preempt long-running requests

- Otherwise, use events

Threads and Events

- Message req, res, in;
- fork(n, {
  for (;;) {
    req = getRequest(); // blocks
    res = doA(req); // blocks
    sendReply(req, res); // nonblk
  }
})
- for (;;) {
  item in = getMessage(); // blocks
  switch (in.type) {
    case REQUEST:
      doA(in); // nonblocking
      processing.put(in.key, in);
      break;
    case A_DONE:
      req = processing.get(in.key);
      sendReply(req, in); // nonblk
      break;
  }
}

Comparison of Programming Model

- Threads
  - One thread per request
  - Must use locks
  - State stored on stack
  - Calls may block

- Events
  - One thread overall
  - No locking necessary
  - State stored explicitly on heap
  - Calls must not block

Events vs. Threads: Performance

- Observation:
  - Threads don’t scale

- Why?
  - Intrinsic or not?


Alternative: SEDA

- Stages:
  - Explicit resource and concurrency control
Retort: Capriccio

- von Behren et al, 2003: *Why Events are a Bad Idea (for high-concurrency servers)*
- Problems with thread (implementations) can be overcome
- Problems with event (model) persist:
  - Call/return semantics hard to see
  - Saving state across events
  - Exception handling and recovery

Capriccio

- User-level threads
  - Nonpreemptive
  - All operations $O(1)$
- Efficient polling asynchronous I/O
- Linked Stacks

Capriccio Scheduling

- Resource needs are deduced from blocking graph
- Monitors and schedules resources associated with blocking points

Summary: Threads vs. Events

- Trade-offs in programming model, ease of implementation + use
  - New approaches: compiler support, resource-aware scheduling
- History:
  - Decade-long battle
  - First paper: Lauer/Needham, 1979: *On the duality of operating system structures*