Why Threads Are A Bad Idea

(for most purposes)

John Ousterhout
Sun Microsystems Laboratories
john.ousterhout@eng.sun.com
http://www.sunlabs.com/~ouster
1996 USENIX Technical Conference
(January 25, 1996)

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.

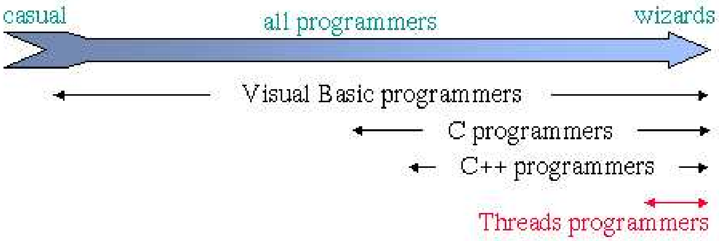
What Are Threads?

- 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.

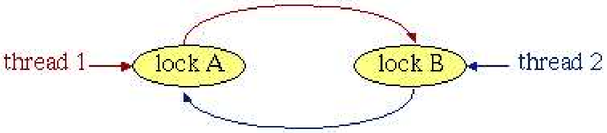
What's Wrong With Threads?



- Too hard for most programmers to use.
- Even for experts, development is painful.

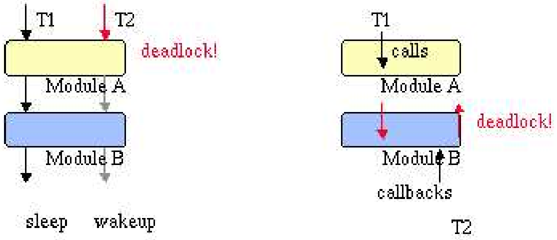
Why Threads Are Hard

- 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.



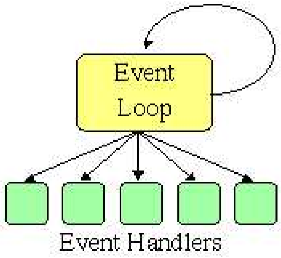
Why Threads Are Hard, cont’d

- 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 (reentrancy 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.

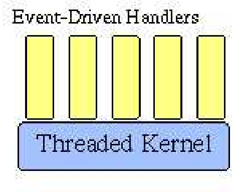
Events vs. Threads, cont’d

- 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.
