Lecture 22: Events vs. Threads

Mothy Roscoe and Joe Hellerstein

November 10, 2005

Background

We start with the two most common models for dealing with concurrency that we see today.

Threads:

- The standard PARC/SRC model (see Birrell paper)
- Concurrent threads with shared-memory synchronization (locks/mutex/condition variables etc.)
- Blocking system calls
- May hold locks for long time
- Problems with more than 100 or so threads due to OS overhead (why?)
  - Context switch overhead
  - Stack size
  - Kernel resources
  - Bad implementation . . .
- Much support from OS, libraries, debuggers, languages (Java), . . .
- Much used . . .

Events:

- Lots of small handlers that run to completion
- Basic model is event arrives and runs a handler

  - state is global or part of handler (not much in between)

- Handlers are dispatched from a loop which waits for events to arrive
- No context switch, just procedure call
- Threads exist, but just run event loop → handler → event loop
  - The stack trace is not useful for debugging - since the stack is "ripped" into tiny pieces
  - Typically one thread per CPU (any more doesn't add anything since threads never block)
  - Sometimes have extra threads for things that may block; largely a workaround for lack of asynchronous I/O support in the kernel, or to "wrap" 3rd-party libraries that contain blocking calls.
- Natural fit with finite-state machines (FSMs)
  - Handlers implement state transitions in response to events
  - Blocking invocations on the kernel, or other "split-phase" operations are split into two states (before and after the call).
- Allows very high concurrency
  - multiplex 10,000 FSMs over a small number of threads
- Also much used: e.g. Click, P2, SFS, Chord, ...

Lauer & Needham

- Proposed two canonical models not of events or threads, but of how to structure a concurrent system.
  1. Message-oriented: processes only communicate by sending messages. State is encapsulated in processes – servers, if you like.
  2. Procedure-oriented: processes synchronize using locks, and state is encapsuled in monitors.
- Note that the latter corresponds somewhat to a well-structured thread model.
- The former does not really correspond to events (actually, it looks more like CSP or Erlang).

- Subtle point: threads and events are both “logical” and “physical” constructs - they are programming abstractions which also affect how the processor is scheduled. Since fancy languages weren’t around much (and weren’t used for system programming at any rate), Lauer abnd Needham are focussing on the physical structure of the system.
- This explains why their performance equivalence result is not borne out by the events vs. threads comparison - it could be if the implementations were different.
- Also explains why it didn’t end the debate . . .

Split-phase actions

Split-phase action: something which is initiated in one phase, and then completes much later (e.g. disk I/O, RPC, etc.). Essential part of dealing with concurrency - this is the time when the processor shoud have better things to do.

- Threads: not too bad – just block until the action completes (synchronous). Threads can be viewed as a programming abstraction for split-phase actions.
  - Assumes other threads run in the meantime
  - Ties up considerable memory (full stack)
  - Easy memory management: stack allocation/deallocation matches natural lifetime
- Events: hard
  - Must store live state in a continuation (on the heap usually). Handler lifetime is too short, so need to explicitly allocate and deallocate later
  - Scoping is bad too: need a multi-handler scope, which usually implies global scope
  - Rips the function into two functions: before and after
  - Debugging is hard
  - Evolution is hard: adding a yielding call implies more ripping to do; converting a non-yielding call into a yielding call is worse – every call site needs to be ripped and those sites may become yielding which cascades the problem

Ousterhout's case for Events

Ousterhout advocated using threads only:

- in performance-critical situations
- to exploit multiple processors
- for coarse-grained isolation of event-driven components

Because:

- Most programmers find threads too difficult
- Locking disciplines are fragile with threads (can you call a function under a lock or not?)
- Break module boundaries

Adya et.al.

Written by the team building FarSite, a distributed P2P filing system at Microsoft Research. Most of these folks have been round the block several times on this kind of thing. They distingish between:

- Task management (preemptive, serial, cooperative)
- Stack management (automatic vs. manual)
- I/O management (does I/O block?)
- Conflict management (atomic operations and synchronization)
- Data partitioning (between tasks)

Automatic stack management: stack preserved across task calls
Manual stack management: stack ripped by programmer into discrete handlers, each of which constructs a continuation for the next.

Task Management

Preemptive: tasks may be interrupted at any time

- must use locks/mutex to get atomicity

- may get pre-empted while holding a lock – others must wait until you are rescheduled
- might want to differentiate short and long atomic sections (short should finish up work)

Serial: tasks run to completion

- basic event handlers, which are atomic
- not allowed to block
- what if they run too long? (not much to do about that, could kill them; implies might be better for friendly systems)
- hard to support multiprocessors

Cooperative: tasks are not pre-empted, but do yield the processor

- sometimes known as coroutines to old folks like me
- can use stacks and make calls, but still interleaved
- better with compiler help: is a call a yield point or not?
- yield points are not atomic: limits what you can do in an atomic section
- hard to support multiprocessors

Note: pre-emption is OK if it can't affect the current atomic section. Easy way to achieve this is data partitioning! Only threads that access the shared state are aproblem!

- Can pre-empt for system routines
- Can pre-empt to switch to a different process (with its own set of threads), but assumes processes don't share state

The FarSite observation

- Fibers (coroutines) give you automatic stack management (no ripping), but cooperative task management
- Handle blocking I/O with dedicated fibers which yield to the main coroutine
- Design patterns for calling into both event-driven (manual stack management) and blocking code without losing control of scheduler.

- Q. Is this really the best of both worlds?
- Q. When would you want to mix the two worlds?

Capriccio

Idea: instead of switching to events, just fix threads

- leverage async I/O
- scale to 100,000 threads (qualitative difference: one per connection!)
- enable compiler support and invariants

Why user-level threads? (see scheduler activations next week)

- easy, low-cost synchronization (but not for I/O which is slow anyway)
- *control* over thread semantics, invariants
- enable application-specific behavoir (e.g. scheduling) and optimizations
- enable compiler assistance (e.g. safe stacks)

But, still has problems:

- still have two schedulers
- async I/O mitigates this by eliminating largest cause of blocking (in kernel)
- still can block unexpectedly – page faults or close() example
- current version actually stops running in such cases (so must be rare)
- can’t schedule multple processes at user-level, only threads within a process (not a problem for dedicated machines like servers)

Specific:

- POSIX interface for legacy apps, but now at user level with a runtime library
- make all thread ops O(1)

- deal with stack space for 100,000 threads (can't just give each 2MB)
- this uses less stack space
- and is faster!
- and is safer! (any fixed amount might not be enough)

Async I/O

- allows lots of user threads to map to small number of kernel threads
- allows better disk throughput
- different mechanisms for network (epoll) and disk (AIO), but this is hidden from users

Resource-aware scheduling:

- goal: transparent but application specific (!)
- note: lots of earlier work on OS extensibility that was hard to use and not well justified
- here we are not requiring work on the part of the programmer, and we are focused on servers, which actually do need different support

Other approaches

- Sophisticated lock replacement / refactoring for C (e.g. Intel LockBend).
- Execution of lock-based programs without locks.
- CSP-like rendezvous programming constructs (e.g. Ada, Limbo, Concurrent ML, etc.)
- Combination of the above with pure functional languages (Erlang, Concurrent Haskell) can be extremely powerful.
- Join- and π-calculus approaches (CΩ, JoCaml, etc.)

- Recent hacks to make events debuggable (e.g. Eddie Kohler's `libeel`).

Main conclusion (from Eric Brewer et. al.: compilers are the key to concurrency in the future.