

# SEDA Capriccio

## I. Background

Threads:

- o The standard model (similar to Mesa)
- o Concurrent threads with locks/mutex/etc. for synchronization
- o Blocking calls
- o May hold locks for long time
- o Problems with more than 100 or so threads due to OS overhead (why?)
  - see SEDA graph of throughput vs. number of threads
- o Strong support from OS, libraries, debuggers, ....

Events:

- o Lots of small handlers that run to completion
- o Basic model is event arrives and runs a handler
  - state is global or part of handler (not much in between)
- o An event loops runs at the core waiting for arrivals, then calls handler
- o No context switch, just procedure call
- o Threads exist, but just run event loop -> handler -> event loop
  - stack trace is not useful for debugging!
  - typically one thread per CPU (any more doesn't add anything since threads don't block)
  - Sometimes have extra threads for things that may block; e.g. OSs that only support synchronous disk reads
- o Natural fit with finite-state machines (FSMs)
  - arrows are handlers that change states
  - blocking calls are split into two states (before and after the call)
- o Allows very high concurrency
  - multiplex 10,000 FSMs over a small number of threads

## II. Cooperative Task Scheduling

Tasks

## Week 4

- o **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)
- o **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
- o **cooperative**: tasks are not pre-empted, but do yield the processor
  - can use stacks and make calls, but still interleaved
  - yield points are not atomic: limits what you can do in an atomic section
  - better with compiler help: is a call a yield point or not?
  - hard to support multiprocessors
- o 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 a problem!
  - 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

Split-phase actions: how do you implement a split-phase action?

- o Threads: not too bad -- just block until the action completes (synchronous)
  - Assumes other threads run in the meantime
  - Ties up considerable memory (full stack)
  - Easy memory management: stack allocation/deallocation matches natural lifetime
- o 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

Atomic split-phase actions (really hard):

## Week 4

- o Threads -- pessimistic: acquire lock and then block
- o Threads -- optimistic: read state, block, try write, retry if fail (and re-block!)
- o Events -- pessimistic: acquire lock, store state in continuation ; later reply completes and releases lock. (seems hard to debug, what if event never comes? or comes more than once?)
- o Events -- optimistic: read state, store in continuation ; apply write, retry if fail
- o Basic problem: exclusive access can last a long time -- hard to make progress
- o General question: when can we move the lock to one side or the other?
- o One strategy:
  - structure as a sequence of actions that may or may not block (like cache reads)
  - acquire lock, walk through sequence, if block then release and start over
  - if get all the way through then action was short (and atomic)
  - This seems hard to automate! Compiler would need to know that some actions mostly won't block or won't block the second time... and then also know that something can be retried without multiple side effects...

Main conclusion (for me): compilers are the key to concurrency in the future

## III. SEDA

Problem to solve:

- o 1) need a better way to achieve concurrency than just threads
- o 2) need to provide graceful degradation
- o 3) need to enable feedback loops that adapt to changing conditions
- o Internet makes the concurrency higher, requires high availability and ensures that load will exceed the target range.

Graceful degradation:

- o want throughput to increase linearly and then stay flat as you exceed capacity
- o want response time to be low until saturation and then linearly increase
- o want fairness in the presence of overload
- o almost no systems provide this
- o key is to drop work early or to queue it for later (threads have implicit queues on locks, sockets, etc.)
- o virtualization makes it harder to know where you stand!

Problems with threads (claimed):

## Week 4

- o threads limits are too small in practice (about 100); some of this is due to linear searches in internal data structures, or limits on kernel memory allocation
- o claims about locks, overhead, TLB and cache misses are harder to understand -- don't seem to be fundamental over events
  - do events use less memory? probably some but not 50% less
  - do events have fewer misses? latencies are the same, so only if working set is smaller
  - is it bad to waste VM for stacks? only with tons of threads
  - is there a fragmentation issue with stacks (lots of partially full pages)? probably to some degree (each stack needs at least one page. If so, we can move to a model with non-contiguous stack frames (leads to a different kind of fragmentation. If so, we could allocate subpages (like FFS did for fragments), and thread a stack through subpages (but this needs compiler support and must recompile all libraries).
- o queues are implicit, which makes it hard to control or even identify the bottlenecks
- o key insight in SEDA: no user allocated threads: programmer defines what can be concurrent and SEDA manages the threads. Otherwise no way to control the overall number or distribution of threads

Event-based approach has problems too:

- o debugging
- o legacy code
- o stack ripping

Solution:

- o use threads within a stage, events between stages
- o stages have explicit queues and explicit concurrency
- o threads (in a stage) can block, just not too often
- o SEDA will add and remove threads from a stage as needed
- o simplifies modularity: queues decouple stages in terms of performance at some cost to latency
- o threads never cross stages, but events can be pass by value or pass by reference.
- o stage scheduling affects locality -- better to run one stage for a while than to follow an event through multiple stages. This should make up for the extra latency of crossing stages.

Feedback loops:

- o works for any measurable property that has smooth behavior (usually continuous as well). Property typically needs to be monotonic in the control area (else get lost in local minima/maxima)
- o within a stage: batching controller decides how many events to process at one time
  - balance high throughput of large batches with lower latency of small batches -- look for point where the throughput drops off
- o thread pool controller: find the minimum number of threads that keeps queue length low
- o global thread allocation based on priorities or queue lengths...

## Week 4

Performance is very good, degrades more gracefully, and is more fair!

- o but huge dropped requests to maintain response time goal
- o however, can't really do any better than this...

Events vs. threads revisited

- o how does SEDA do split-phase actions?
- o Intra-stage:
  - threads can just block
  - multiple threads within a stage, so shared state must be protected. Common case is that each event is mostly independent (think http requests)
- o Inter-stage:
  - rip action into two stages
  - usually one-way: no return (equivalent to tail recursion). This means that the continuation is just the contents of the event for the next stage.
  - loops in stages are harder: have to manually pass around the state
  - atomicity is tricky too: how do you hold locks across multiple stages? generally try to avoid, but otherwise need one stage to lock and a later one to unlock

## IV. Capriccio

Idea: instead of switching to events, let's just fix threads

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

Why user-level threads? (mostly same args from scheduler activations)

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

But, still has problems:

- o still have two schedulers
- o async I/O mitigates this by eliminating largest cause of blocking (in kernel)
- o still can block unexpectedly -- page faults or close() example

## Week 4

- o current version actually stops running in such cases (so must be rare)
- o can't schedule multiple processes at user-level, only threads within a process (not a problem for dedicated machines like servers)

### Specific:

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

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

### Resource-aware scheduling:

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