Concurrency, Threads, and Events

Robbert van Renesse

Using Threads in Interactive Systems: A Case Study (Hauser et al 1993)

- Analyzes two interactive computing systems
- Classifies thread usage
- Finds that programmers are still struggling
  - (pre-Java)
- Limited scheduling support
  - Priority-inversion

SEDA: An Architecture for Well-Conditioned, Scalable Internet Services (Welsh, 2001)

- Analyzes threads vs event-based systems, finds problems with both
- Suggests trade-off: stage-driven architecture
- Evaluated for two applications
  - Easy to program and performs well

What is a thread?

- A traditional “process” is an address space and a thread of control.
- Now add multiple thread of controls
  - Share address space
  - Individual program counters and stacks
- Same as multiple processes sharing an address space.

Thread Switching

- To switch from thread T1 to T2:
  - Thread T1 saves its registers (including pc) on its stack
  - Scheduler remembers T1's stack pointer
  - Scheduler restores T2' stack pointer
  - T2 restores its registers
  - T2 resumes

Thread Scheduler

- Maintains the stack pointer of each thread
- Decides what thread to run next
  – E.g., based on priority or resource usage
- Decides when to pre-empt a running thread
  – E.g., based on a timer
- Needs to deal with multiple cores
  – Didn’t use to be the case
- “fork” creates a new thread

Synchronization Primitives

- Semaphores
  - P(S): block if semaphore is “taken”
  - V(S): release semaphore
- Monitors:
  - Only one thread active in a module at a time
  - Threads can block waiting for some condition using the WAIT primitive
  - Threads need to signal using NOTIFY or BROADCAST

Uses of threads

- To exploit CPU parallelism
  - Run two CPUs at once in the same program
- To exploit I/O parallelism
  - Run I/O while computing, or do multiple I/O
  - I/O may be "remote procedure call"
- For program structuring
  - E.g., timers

Hauser's categorization

- Defer Work: asynchronous activity
  - Print, e-mail, create new window, etc.
- Pumps: pipeline components
  - Wait on input queue; send to output queue
  - E.g., *slack process*: add latency for buffering
- Sleepers & one-shots
  - Periodic activity & timers

Categorization, cont'd

- Deadlock Avoiders
  - Avoid deadlock through ordered acquisition of locks
  - When needing more locks, roll-back and re-acquire
- Task Rejuvenation: recovery
  - Start new thread when old one dies, say because of uncaught exception

Categorization, cont’d

- Serializers: event loop
  - for (;;) { get_next_event(); handle_event(); }
- Concurrency Exploiters
  - Use multiple CPUs
- Encapsulated Forks
  - Hidden threads used in library packages
  - E.g., menu-button queue

Common Problems

- Priority Inversion
  - High priority thread waits for low priority thread
  - Solution: temporarily push priority up (rejected??)
- Deadlock
  - X waits for Y, Y waits for X
- Incorrect Synchronization
  - Forgetting to release a lock
- Failed “fork”
- Tuning
  - E.g. timer values in different environment

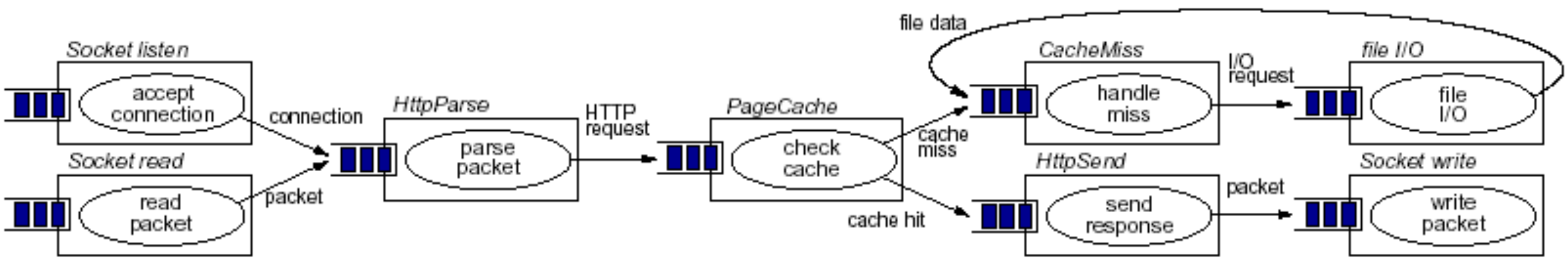
Criticism of Hauser

- Systems old but/and not representative
- Pre-Java

What is an Event?

- An object queued for some module
- Operations:
  - create_event_queue(handler) → EQ
  - enqueue_event(EQ, event-object)
    - Invokes, eventually, handler(event-object)
- Handler is *not* allowed to block
  - Blocking could cause entire system to block
  - But page faults, garbage collection, …

Example Event System



(Also common in telecommunications industry, where it's called "workflow programming")

Event Scheduler

- Decides which event queue to handle next.
  - Based on priority, CPU usage, etc.
- Never pre-empts event handlers!
  - No need for stack / event handler
- May need to deal with multiple CPUs

Synchronization?

- Handlers cannot block → no synchronization
- Handlers should not share memory
  - At least not in parallel
- All communication through events

Uses of Events

- CPU parallelism
  - Different handlers on different CPUs
- I/O concurrency
  - Completion of I/O signaled by event
  - Other activities can happen in parallel
- Program structuring
  - Not so great…
  - But can use multiple programming languages!

Hauser's categorization ?!

- Defer Work: asynchronous activity
  - Send event to printer, etc
- Pumps: pipeline components
  - Natural use of events!
- Sleepers & one-shots
  - Periodic events & timer events

Categorization, cont'd

- Deadlock Avoiders
  - Ordered lock acquisition still works
- Task Rejuvenation: recovery
  - Watchdog events?

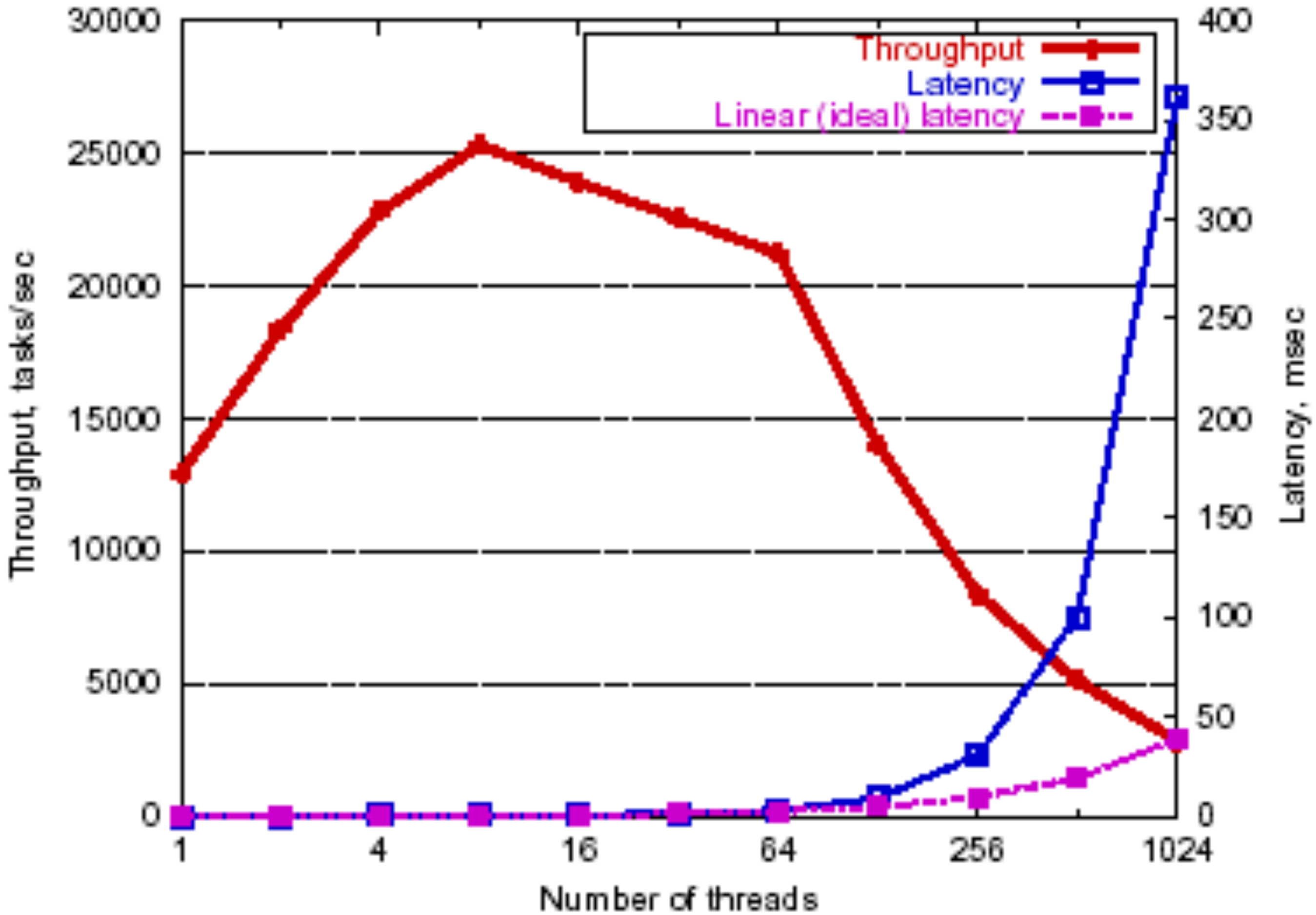
Categorization, cont'd

- Serializers: event loop
  - Natural use of events and handlers!
- Concurrency Exploiters
  - Use multiple CPUs
- Encapsulated Events
  - Hidden events used in library packages
  - E.g., menu-button queue

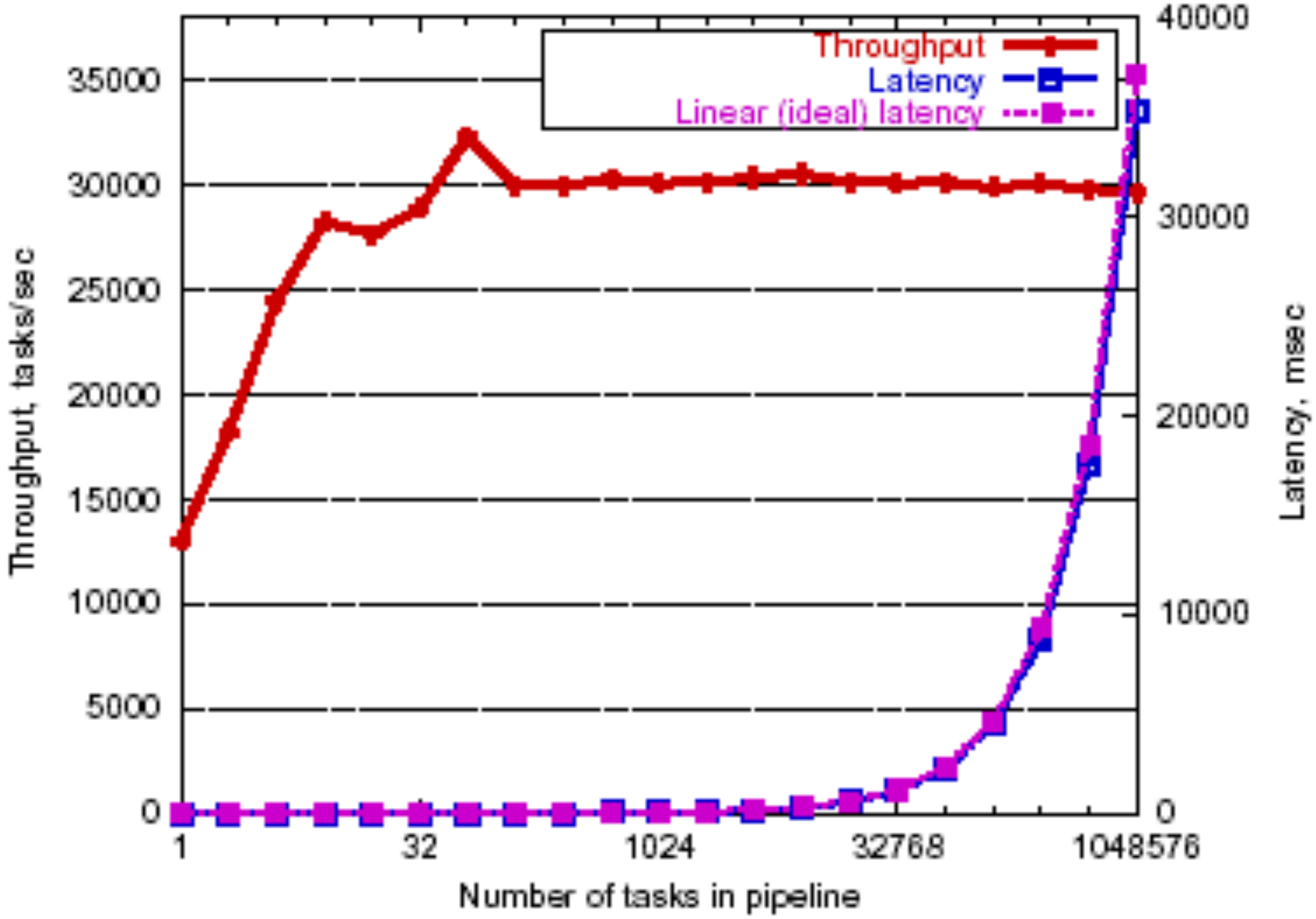
Common Problems

- Priority inversion, deadlock, etc. much the same with events

Threaded Server Throughput



Event-driven Server Throughput



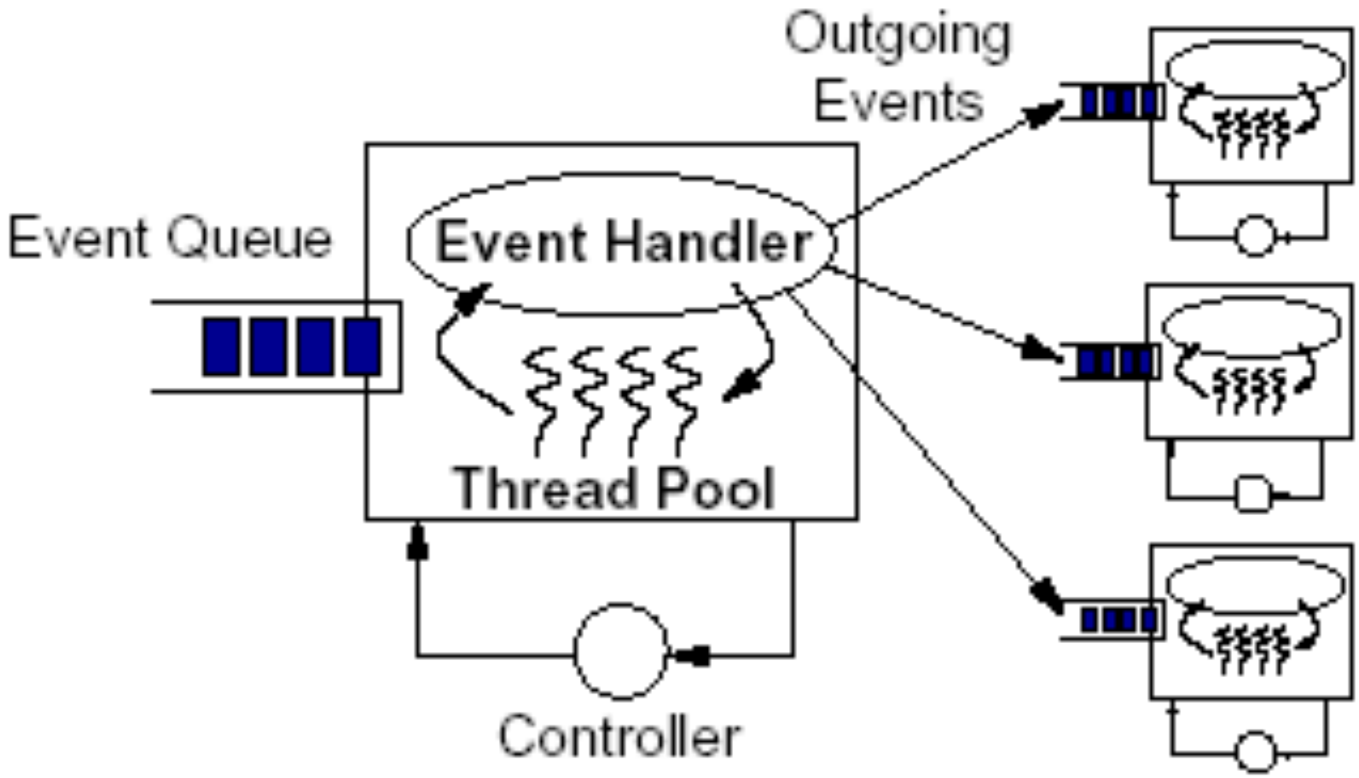
Threads vs. Events

- Events-based systems use fewer resources
  - Better performance (particularly scalability)
- Event-based systems harder to program
  - Have to avoid blocking at all cost
  - Block-structured programming doesn't work
  - How to do exception handling?
- In both cases, tuning is difficult

SEDA

- Mixture of models of threads and events
- Events, queues, and “pools of event handling threads”.
- Pools can be dynamically adjusted as need arises.

SEDA Stage


Outgoing
Events
Event Queue
Event Handler
Thread Pool
Controller

Best of both worlds

- Ease of programming of threads
  - Or even better
- Performance of events
  - Or even better