Concurrency, Thread and Event

CS6410 Sept 6, 2011

Ji-Yong Shin

What is a Thread?

- Same as multiple processes sharing an address space.
- 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

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

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”

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, …

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!

Common Problems

- Priority inversion, deadlock, etc. much the same with threads
- Stack ripping
  - No local variables are preserved
  - Callbacks should explicitly get more and more variables

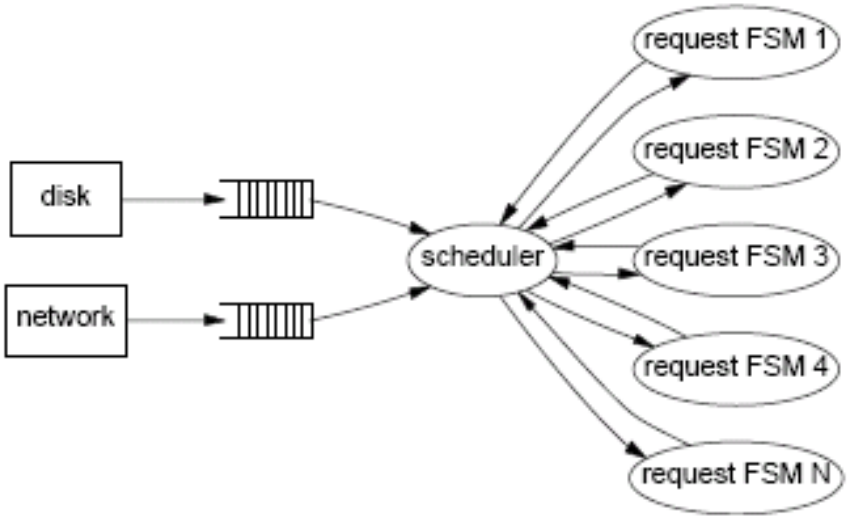
On the Duality of Operating System Structure

- Hugh C. Lauer
  - Adjunct Prof., Worcester Polytechnic Institute
  - Xerox, Apollo Computer, Mitsubishi Electronic Research Lab, etc.

- Roger M. Needham
  - Prof., Cambridge University
  - Microsoft Research, Cambridge Lab

Message-Oriented System (Event)

- Small, static # of process
- Explicit messaging
- Limited data sharing in memory
- Identification of address space or context with processes



Message Oriented System

- Characteristics
  - Queuing for congested resource
  - Data structure passed by reference (no concurrent access)
  - Peripheral devices treated as processes
  - Priority of process statically determined
  - No global naming scheme is useful

Message Oriented System

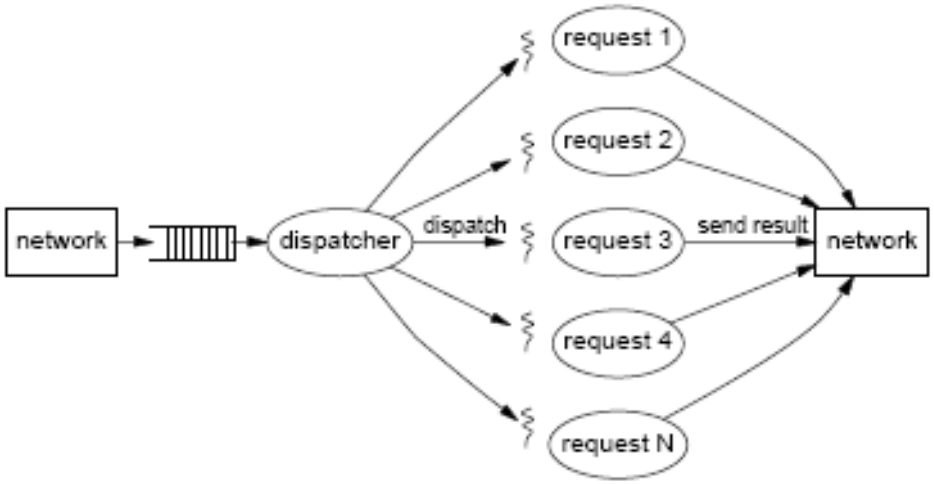
- Facilities
  - Message with identifiers: points to larger data
  - Message channel and ports:
    - Channel: destination of message
    - Port: type specific queue leading to a process
- Operations
  - SendMessage
  - AwaitReply
  - WaitForMessage
  - SendReply
- Process declaration
- CreateProcess

Message Oriented System

- Canonical model
  - begin
    Do forever
      WaitForMessages
      case port
            port 1: …;
            port 2: …; SendReply; …;
      end case
    end loop
  end

Procedure-Oriented System (Thread)

- Large # of small processes
- Rapidly changing # of processes
- Communication using direct sharing and interlocking of data
- Identification of context of execution with function being executed



Procedure Oriented System

- Characteristics
  - Synchronization and congestion control associates with waiting for locks
  - Data is shared directly and lock lasts for short period of time
  - Control of peripheral devices are in form of manipulating locks
  - Priority is dynamically determined by the execution context
  - Global naming and context is important

Procedure Oriented System

- Facilities
  - Procedure: implements algorithm and accesses global data
  - Procedure call facility
    - Synchronous call: regular call
    - Asynchronous call: FORK (creation) and JOIN (syncs termination)
  - Modules: collection of procedure and data
  - Monitor: modules with locking facility
  - NEW and START: instantiate and run modules (monitors)
  - WAIT condVar and SIGNAL condVar

Procedure Oriented System

- Canonical model
  - Monitor

```
-- global data and state info for the process
proc1: entry procedure
proc2: entry procedure returns
        begin
                If resourceExhausted then WAIT; …;
                RETURN result; …;
        end
proc L: Entry procedure
        begin
                …; SIGNAL; …
        end;
endloop;
initialize;
end
```

Dual Mapping

Event	Thread
Processes: CreateProcess	Monitors: NEW/START
Message channel	External procedure id
Message port	Entry procedure id
Send msg (immediate); AwaitReply	Simple procedure call
Send msg (delayed); AwaitReply	FORK; … JOIN
Send reply	Return from procedure
Main loop of std resource manager, wait for message stmt, case stmt	Monitor lock, ENTRY attribute
Arms of case statement	ENTRY proc declaration
Selective waiting	Condition vars, WAIT, SIGNAL

Summary

- One canonical implementation can be switched to the other implementation

- Performance can be the same as well (?)
  - But depending on the machine architecture and programming constraints, one model might be preferred over the other

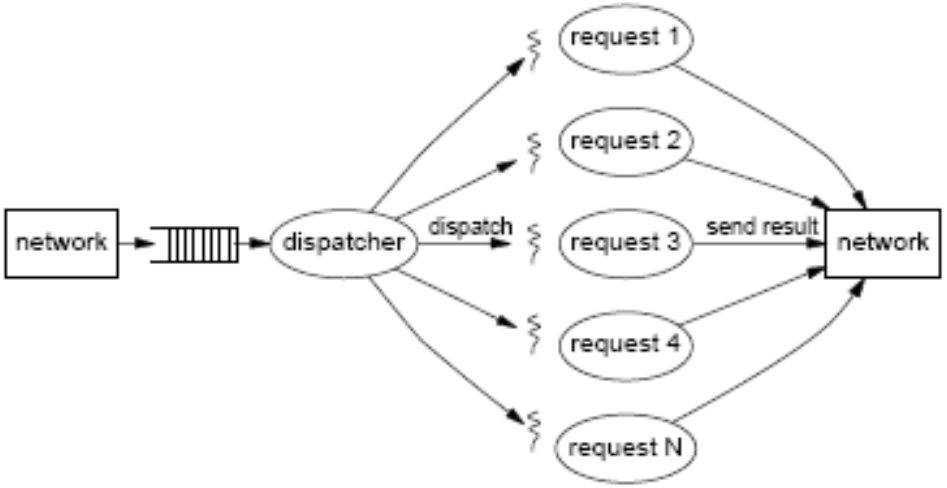
SEDA: An Architecture for Well-Conditioned Scalable Internet Services

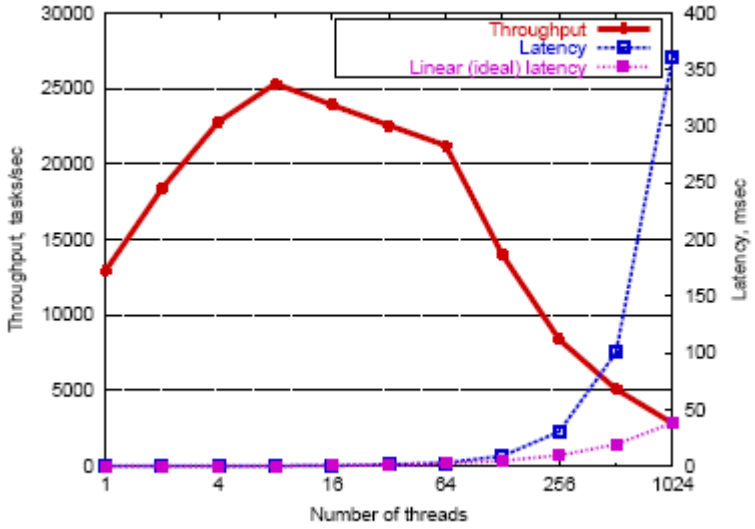
- Matt Welsh
  - Cornell Alumnus
  - Google
- David Culler
  - Faculty at UC Berkeley
- Eric Brewer
  - Faculty at UC Berkeley

Motivation

- Web services becoming larger
  - Billions of access per day
  - Complex services and dynamic contents
- Traditional concurrency model
  - Performance does not scale
- Aim to build well-conditioned service
  - Graceful degradation as load increases

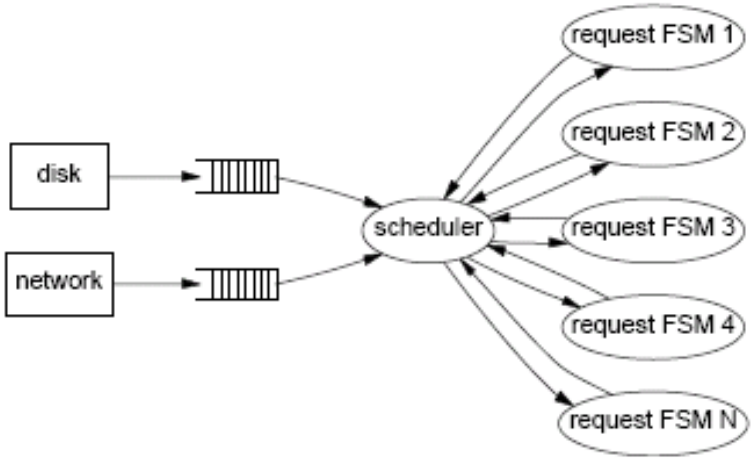
Thread

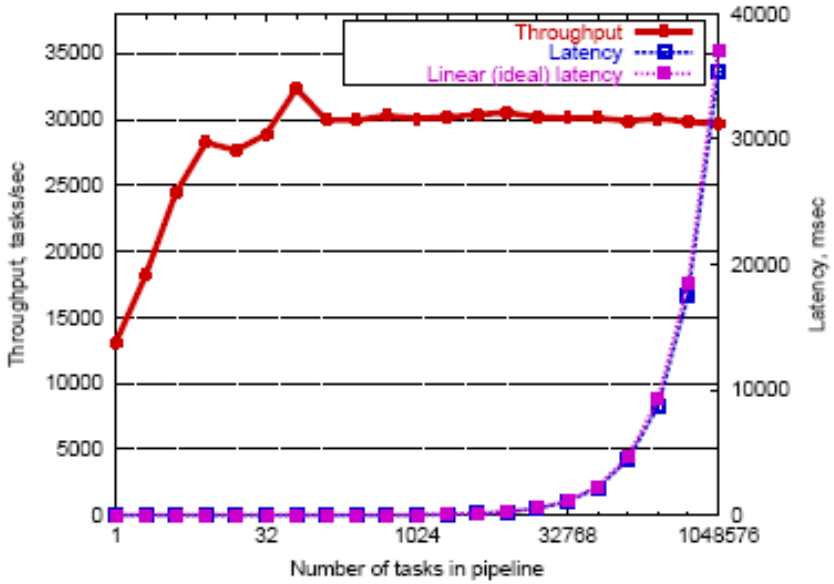




- Easy to program
- TLB miss, scheduling overhead, lock contention
- Bounded thread pool leads to unfairness

Event





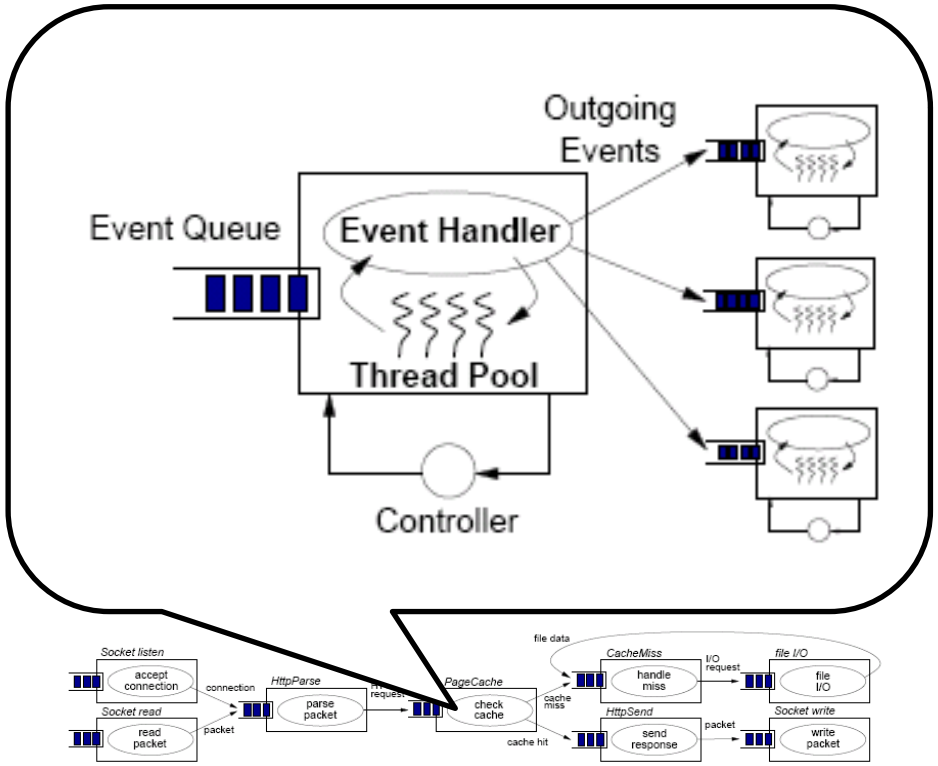
- Non-blocking event handling process assumed
- Scheduling issue for different FSMs
- Structured event queue: SEDA extends this approach

SEDA

- Goal
  - Support massive concurrency
  - Simplify construction of well conditioned service
  - Enable introspection by application
  - Support self-tuning resource management

SEDA Design = Event + Thread (Staged Event Driven Architecture)

- Stage
  - Event queue
  - Event handler
  - Thread pool
  - Controller
    - Scheduling
    - Thread pool size
  - Explicit control boundary

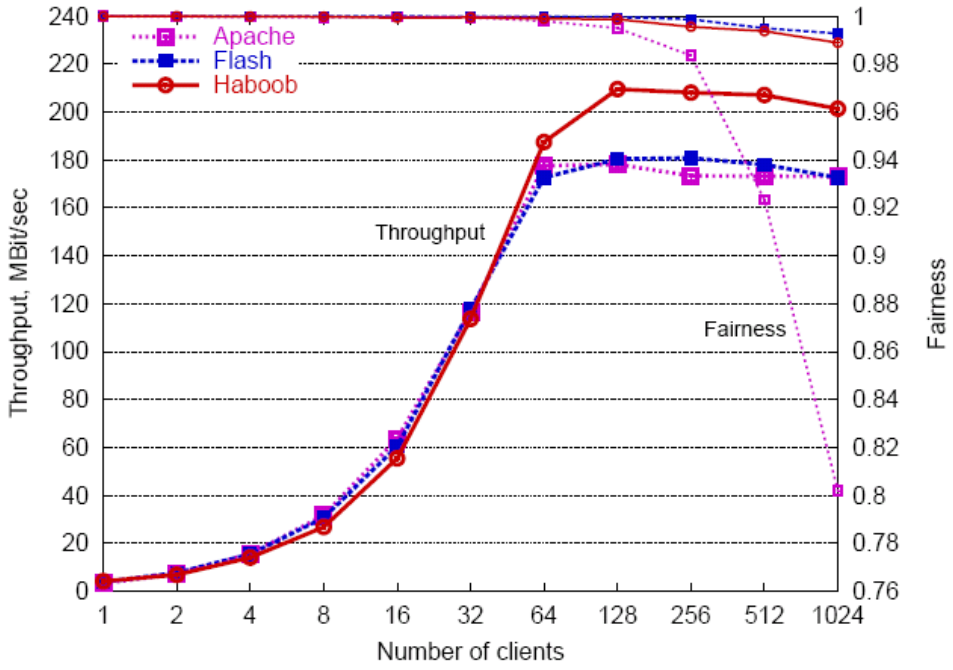


Dynamic Resource Control

- Thread pool controller
  - Based on queue length and idle period
    - Queue length ↑ → # Thread ↑
    - Thread Idle → # Thread ↓
- Batching controller
  - Based on output rate of a stage
    - Output rate ↓ → Batching factor ↑

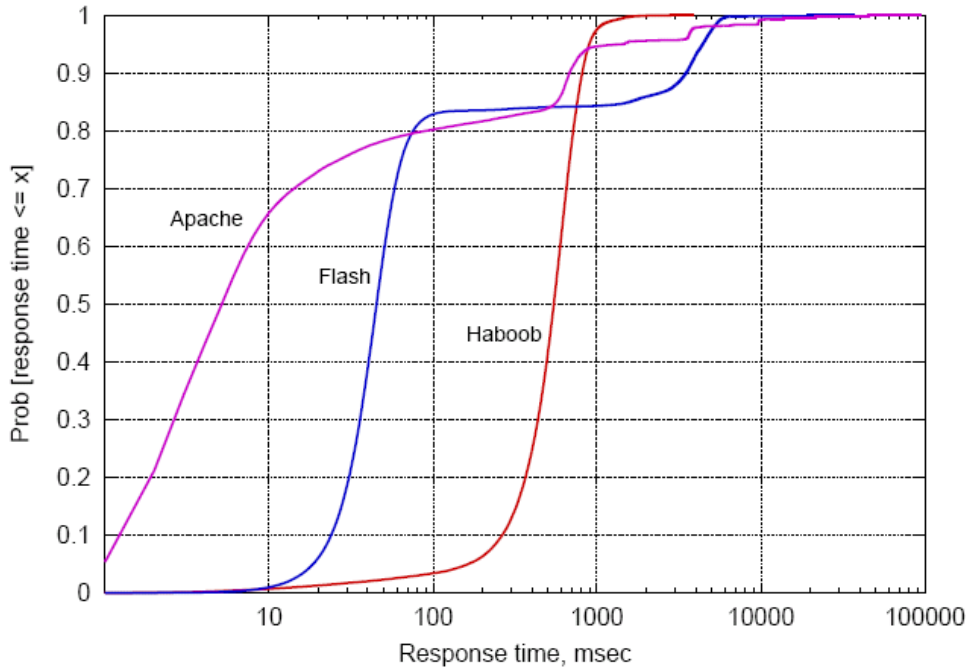
Evaluation: Throughput & Fairness

- Haboob: SEDA based
- Apache: Thread based (max 150 threads)
- Flash: Event based (max 500 connections)



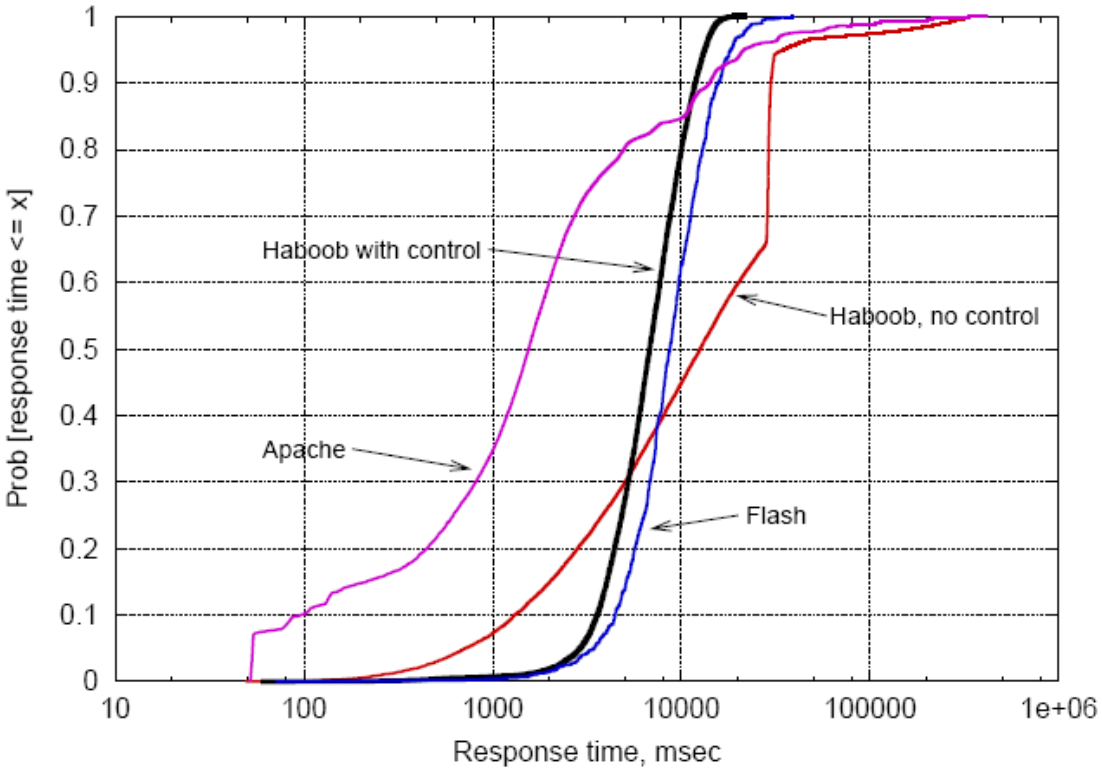
Evaluation: Distribution of Response Time

- Haboob: SEDA based
- Apache: thread based (max 150 threads)
- Flash: Event based (max 500 connections)



Evaluation: Adaptive Load Shedding

- Dynamic page request: IO intensive load
- Flash: CGI bug (?)
- Haboob: Customized controller improves fairness



Summary

- Staged event driven architecture
  - Merits of thread and event
  - Easy to program and debug
    - Modularity
    - Sequential stages
  - Controller dynamically handles heavy loads
- Global arbitration of controllers (?)
  - Detecting load
  - Handling load

Discussion

- Thread vs Event? Are they the same?
  - Why threads are a bad idea [Ousterhout, 1996]
  - Why events are a bad idea, [Von Behren, 2003]
  - Events can make sense, [Krohn, 2007]
  - …
- Compromise solution?

Reference

- http://www.cs.cornell.edu/courses/CS6410/2010fa/lectures/03-concurrency.pptx