Fast Servers

Or: Religious Wars, part I, Events vs. Threads

Robert Grimm
New York University
Overview

- **Challenge**
  - Make server go fast

- **Approach**
  - Cache content in memory
  - Overlap I/O and processing

- **Some issues**
  - Performance characteristics
  - Costs/benefits of optimizations & features
  - Programmability, portability, evolution
Server Architectures

- **Multi-Process (MP)**
  - One request per process
    - Easily overlaps I/O and processing
    - No synchronization necessary

- **Multi-Threaded (MT)**
  - One request per thread
    - Kernel/user threads?
    - Enables optimizations based on shared state
    - May introduce synchronization overhead
Server Architectures (cont.)

- **Single Process Event Driven (SPED)**
  - Request processing broken into separate steps
  - Step processing initiated by application scheduler
    - In response to completed I/O
  - OS needs to provide support
    - Asynchronous `read()`, `write()`, `select()` for sockets
    - But typically not for disk I/O
Asymmetric Multi-Process Event Driven (AMPED)

- Like SPED, but helpers handle disk I/O
  - Helpers invoked through pipes (IPC channel)
  - Helpers rely on \texttt{mmap()}, \texttt{mincore()}
- Why?
Staged Event Driven (SEDA)

- Targeted at higher-level runtimes (e.g., Java VM)
  - No explicit control over memory (e.g., GC)
- Each stage is event driven, but uses its own threads to process events
Flash Implementation

- Map pathname to file
  - Use pathname translation cache
- Create response header
  - Use response header cache
    - Aligned to 32 byte boundaries → Why? `writev()`
- Write response header (asynchronously)
- Memory map file
  - Use cache of file chunks (with LRU replacement)
- Write file contents (asynchronously)
Flash Helpers

- Main process sends request over pipe
- Helper accesses necessary pages
  - `mincore()`
  - Feedback-based heuristic
    - Second-guess OS
- Helper notifies main process over pipe
- Why pipes?
  - `select()`-able
- How many helpers?
  - Enough to saturate disks
Costs and Benefits

- **Information gathering**
  - MP: requires IPC
  - MT: requires consolidation or fine-grained synchronization
  - SPED, AMPED: no IPC, no synchronization

- **Application-level caching**
  - MP: many caches
  - MT, SPED, AMPED: unified cache

- **Long-lived connections**
  - MP: process
  - MT: thread
  - SPED, AMPED: connection information
Performance Expectations

- In general
  - Cached
    - SPED, AMPED, Zeus > MT > MP, Apache
  - Disk-bound
    - AMPED > MT > MP, Apache >> SPED, Zeus

- What if Zeus had as many processes as Flash helpers?
  - Cached: Worse than regular Zeus b/c of cache partitioning
  - Disk-bound: Same as MP
Experimental Methodology

- 6 servers
  - Flash, Flash-MT, Flash-MP, Flash-SPED
  - Zeus 1.30 (SPED), Apache 1.3.1 (MP)
- 2 operating systems
  - Solaris 2.6, FreeBSD 2.2.6
- 2 types of workloads
  - Synthetic
  - Trace-based
- 1 type of hardware
  - 333 MHz PII, 128 MB RAM, 100 MBit/s Ethernet
Experiments

- Single file tests
  - Repeatedly request the same file
  - Vary file size
  - Provides baseline
    - Servers can perform at their highest capacity
- Real workloads
  - Measure throughput by replaying traces
  - Vary data set size
  - Evaluate impact of caching
Single File Tests

Solaris

FreeBSD
Real Workloads

Solaris

FreeBSD
Some Questions

- How does Flash-* compare to Zeus?
- Why is Apache slower?
- Why does Flash-SPED outperform Flash for cache-bound workloads?
- Why does Flash outperform Apache disk-bound?
- Why do Flash-MT, Flash-MP lag?
- Why does Zeus lag for files between 10 and 100KB on FreeBSD?
- Why no Flash-MT on FreeBSD?
- Which OS would you choose?
Flash Optimizations

- Test effect of different optimizations

- What do we learn?
WAN Conditions

- Test effect of WAN conditions
  - Less bandwidth
  - Higher packet loss

- What do we learn?
In Summary, Flash-MT or Flash?

- Cynical
  - Don’t bother with Flash

- Practical
  - Flash easier than kernel-level threads
  - Flash scales better than Flash-MT with many, long-lived connections

- However:
  - What about read/write workloads?
  - What about SMP machines?
Do We Really Have to Choose Between Threads and Events?
Remember SEDA...?

- **Staged Event Driven (SEDA)**
  - Targeted at higher-level runtimes (e.g., Java VM)
  - Each stage is event driven, but uses its own threads to process events

- Why would we want this?
- What’s the problem?
Let’s Try Something Different...
Checking Our Vocabulary

- Task management
  - Serial, preemptive, cooperative
- Stack management
  - Automatic, manual
- I/O management
  - Synchronous, asynchronous
- Conflict management
  - With concurrency, need locks, semaphores, monitors
- Data partitioning
  - Shared, task-specific
Separate Stack and Task Management!

- Religious war conflates two orthogonal axes
  - Stack management
  - Task management
Automatic vs. Manual Stack Management In More Detail

- **Automatic**
  - Each complete task a procedure/method/…
  - Task state stored on stack

- **Manual**
  - Each step an event handler
  - Event handlers invoked by scheduler
  - Control flow expressed through *continuations*
    - Necessary state + next event handler
    - Scheme: `call/cc` reifies stack and control flow
call/cc in Action

- \((+ 1 (\text{call/cc})\)
  \((\lambda (k)\)
  \((+ 2 (k 3)))\))
  - Continuation reified by call/cc represents \((+ 1 [])\)
  - When applying continuation on 3, do we get
    - 4
    - 6

- Thanks to Dorai Sitaram,
  Teach Yourself Scheme in Fixnum Days
(define r #f)
(+ 1 (call/cc
     (lambda (k)
       (set! r k)
       (+ 2 (k 3))))))

- Results in?

(r 5)
- Results in?
Manual Stack Management: Stack Ripping

- As we add blocking calls to event-based code
  - Need to break procedures into event handlers

- Issues
  - Procedure scoping
    - From one to many procedures
  - Automatic variables
    - From stack to heap
  - Control structures
    - Loops can get nasty (really?)
  - Debugging
    - Need to recover call stack
So, Why Bother with Manual Stacks?

- Hidden assumptions become explicit
  - Concurrency
    - Static check: `yielding, atomic`
    - Dynamic check: `startAtomic(), endAtomic(), yield()`
  - Remote communications (RPC)
    - Take much longer, have different failure modes

- Better performance, scalability
- Easier to implement
Hybrid Approach

- Cooperative task management
  - Avoid synchronization issues
- Automatic stack management
  - For the software engineering wonks amongst us
- Manual stack management
  - For “real men”
Implementation

- Based on Win32 fibers
  - User-level, cooperative threads
- Main fiber
  - Event scheduler
  - Event handlers
- Auxiliary fibers
  - Blocking code
- Macros to
  - Adapt between manual and automatic
  - Wrap I/O operations
Manual Calling

Automatic

- Set up continuation
  - Copy result
  - Invoke original continuation
- Set up fiber
- Switch to fiber
- Issue: I/O
  - Are we really blocking?
  - No, we use asynchronous I/O and yield back to main fiber
Automatic Calling
Manual

- Set up special continuation
  - Test whether we actually switched fibers
    - If not, simply return
- Invoke event handler
- Return to main fiber
- When done with task
  - Resume fiber
What Do We Learn?

- Adaptors induce headaches…
- Even the authors can’t get the examples right…
  - Sometimes `caInfo`, sometimes `*caInfo`, etc.
- More seriously, implicit trade-off
  - Manual
    - Optimized continuations vs. stack ripping
  - Automatic
    - Larger continuations (stack-based) vs. more familiar programming model
- Performance implications???
Who has written event-based code?
What about user interfaces?
  - MacOS, Windows, Java Swing
Who has written multi-threaded code?
Who has used Scheme’s continuations?
What do you think?