Parallel & Concurrent Programming:
Server Architectures

Emery Berger
CMPSCI 691W - Spring 2006
Outline

Last time:
- Lock “improvements”
- Non-blocking operations
- java.nio library

Today:
- Server architectures
  - Focus: web servers
  - Performance & ease of programming
Web Servers

- Client (IE, Mozilla) requests http://foo.com/bar.html
- In response, web server
  - Accepts network connection
    - Persistent in http/1.1
  - Reads request (bar.html)
  - Reads requested file or execute CGI
  - Sends header and file / output
Example: Single-Thread

- **Single-threaded server:**
  - One process handles all web connections, step by step
- **Advantages:**
  - Easy! 1 thread = no race conditions, etc.
- **Disadvantages:**
  - Only one client at a time
  - Unacceptably simple

Figures from *Flash* [Pai et al., USENIX 99]
Web Server Goals

Performance goals:
- Support as many simultaneous clients as possible
  - High concurrency
  - Low memory consumption per client
- Provide high throughput, low response time (latency)

Software engineering goals:
- Simple to understand, extend, employ desired optimizations & features, and debug
Optimizations & Features

- **Optimizations: caching**
  - Pathname translations
  - Results of script executions
    - Turns dynamic pages into static pages
  - File reads
    - Avoids disk I/O, expensive systems calls: `stat()`
- **Features: logging, statistics gathering, access control...**

- *Lots of centralized data structures*
Server Architectures

- **MP/MT**
  - Multiprocess/multithreaded (Apache)

- **SPED**
  - Single-process event-driven
    (thttpd, Zeus)

- **AMPED**
  - asymmetric multiprocess event-driven
    (Flash)
**Multiprocess Architecture**

- **Advantages:**
  - Takes advantage of multiple processors
  - Debugging, etc.?

- **Disadvantages:**
  - IPC (maintain caches, logs)
  - Memory cost, limited # clients, context switches
Multithreaded

- **Advantages**
  - Takes advantage of multiple processors
  - Extensibility

- **Disadvantages**
  - Synch, races
  - Memory cost (kernel vs. user-level)
  - Startup cost? Context switches?
  - Blocking I/O
**Blocking I/O**

- Can specify “non-blocking” for some I/O calls, but:
  - Non-blocking supported for network I/O, but generally not disk operations
  - POSIX standard **AIO**: Asynchronous I/O
    - Supports only reads & writes, not `open()` or `stat()`

⇒ Must work around blocking I/O
- Single-process event-driven
  - Uses `select()` to check for ready file descriptors
  - Processes ready items, moves to next "stage"
    - One finite state machine per client
SPED Example (thttpd)

- Loop until shut down:
  - Accept new connections
  - For each ready file descriptor, switch (status):
    - READ_MODE - handle read
    - SEND_MODE - handle send
    - WRITE_MODE - handle write
SPED Pros & Cons

- Advantages:
  - No context switches, synchronization, IPC, etc.
  - Low memory overhead

- Disadvantages:
  - Multiple processors?
  - Blocking I/O?
  - Programming complexity...
Asymmetric MultiProcess Event-Driven

- Like SPED, but with helper processes for blocking I/O
  - e.g., one or two per disk, more for multi-arm disks
**AMPED Pros & Cons**

- **Advantages:**
  - Same as event-driven, but no blocking
  - No context switches, synchronization, IPC, etc.
  - Low memory overhead

- **Disadvantages:**
  - Multiple processors?
Throughput versus “Size”

- 96MB = available RAM for buffer cache
  - In RAM: SPED wins
  - On disk: blocking I/O dominates
**Throughput vs. # Clients**

- **WAN conditions**
  - Why does MP do so badly?
- **Note:** all experiments on uniprocessor
Problems with Events

- Do not take advantage of multiple processors
- Long-running handler = high latency
- Events obscure control flow
  - No state across request handlers
  - Break code into “call” event and “return” event
    - continuation-passing style
  - Hard to write, understand & debug
Problems with Threads

- Synchronization overhead & complexity, deadlock
- Race conditions difficult to debug
  - Timing dependencies result in Heisenbugs
- Priority inversion
- Hybrid approach: mixes thread pools with events
- Staged Event-Driven Architecture
  - Event-driven stages separated by queues
  - Thread pools per stage
  - Provides load conditioning: degrades service gracefully
    - Admission control
    - Load shedding
**Bursty Load**

- Web server logs for USGS site after 1999 earthquake
- 3 orders of magnitude increase
  - a.k.a. “Slashdotting”

Figures from SEDA [Welsh et al., SOSP 01]
Effect of Load

- Simulated on thread-pool server
- What happened?
Event-driven server (all in RAM)
- Events organized into **stages**
  - Connect output of one stage to input of next
**SEDA stages**

- Each stage: **thread pool** processes **batches** of events
  - Amortizes ops, locality
  - Can perform admissions control on own queue
    - Shed load, etc.
- **Controller:**
  - Adjusts resource allocations & scheduling
  - E.g., reduces # threads in pool when thruput degrades
- 64 clients
- Nearly identical response time curve
- 1024 clients
- Note the heavy tail (minutes!)
The End

- But isn’t it still painful to write event-driven code?
- Next time: alternatives
  - Capriccio, Flux