CSCI-1680
Network Programming II

Rodrigo Fonseca
Today

- **Network programming**
  - Programming Paradigms
  - Programming libraries
- **Final project**
Low-level Sockets

• **Address Family AF_PACKET**
  – Socket type: SOCK_RAW
    • See link-layer (Ethernet) headers. Can send broadcast on a LAN. Can get/create non-IP packets
  – Socket type: SOCK_DGRAM
    • See IP headers. Can get protocols other than TCP/UDP: ICMP, SCTP, DCCP, your own…
    • Can cook your own IP packets
  – Must have root privileges to play with these
Building High Performance Servers
The need for concurrency

• **How to improve throughput?**
  – Decrease latency (throughput $\alpha \frac{1}{\text{latency}}$)
  – Hard to do!
    • Optimize code (this you should try!)
    • Faster processor (no luck here, recently)
    • Speed of light isn’t changing anytime soon…
    • Disks have to deal with things like inertia!
  – Do multiple things at once

• **Concurrency**
  – Allows overlapping of computation and I/O
  – Allows use of multiple cores, machines
High-performance Servers
Common Patterns

Multiple processes

Single Process Event Driven

Multiple Threads

Single Process Event Driven with Helpers

Figures from Pai, et al., 1999 “Flash: An efficient and portable Web server”
Threads

- Usual model for achieving concurrency
- Uniform abstraction for single and multiple cores
- Concurrency with locks/mutexes
  - Threads may block, hold locks for long time
- Easy to reason about
  - Each thread has own stack
- Strong support from OS, libraries, debuggers
- Traditionally, problems with more than a few 100 threads
  - Memory overhead, O(n) operations
Performance, Thread-based server

The figure shows the flow of events from the measurements, and tasks are generated internally to negate network effects. The server is implemented in C and is running on a 4-way 500 MHz Pentium III with 2 GB of memory under Linux 2.2.14. As the number of concurrent clients increases, the number of threads grows exponentially, processing events of different types from a queue. Events may be driven by network notifications, timers, or other application-specific events. The event-driven approach implements the processing of each task as a finite state machine, where transitions between states in the FSM are triggered by interrupts, page faults, or garbage collection.

Threads are pre-allocated in the server to eliminate thread startup overhead but allow cache hits to proceed unabated. The throughput remains constant across a huge range in load, with the response time becoming substantial. As the number of tasks increases, the server is unable to inspect the internal load shedding. However, the server is unable to inspect the internal request stream to implement such a policy; all it knows is that the thread pool is saturated, and must arbitrarily reject work without knowledge of the source of the bottleneck.

For comparison, we have shown the linear (ideal) latency. The throughput of the server degrades substantially after which throughput degrades substantially. Response time becomes unbounded as task queue lengths increase; for comparison, we have shown the linear (ideal) latency. The tradeoffs between threaded and event-driven concurrency models are illustrated in Figure 3. As the number of tasks increases, the server throughput increases until the number of threads grows exponentially, processing events of different types from a queue. Events may be driven by notifications, timers, or other application-specific events. The event-driven approach implements the processing of each task as a finite state machine, where transitions between states in the FSM are triggered by interrupts, page faults, or garbage collection.

Throughput, tasks/sec

Latency, msec

Number of threads

Throughput
Latency
Linear (ideal) latency

Figure 2: Performance, Thread-based server

Figure 3: Throughput, tasks/sec versus Latency, msec

From Welsh, et al., SOSP 2001 “SEDA: An Architecture for Well-Conditioned, Scalable Internet Services”
Events

- **Small number of threads, one per CPU**
- **Threads do one thing:**
  
  ```
  while(1) {
      get event from queue
      Handle event to completion
  }
  ```
- **Events are network, I/O readiness and completion, timers, signals**
  - Remember select()?
- **Assume event handlers never block**
  - Helper threads handle blocking calls, like disk I/O
Events

• Many works in the early 2000’s claimed that events are needed for high performance servers
  – E.g., Flash, thttpd, Zeus, JAWS web servers

• Indeed, many of today’s fastest servers are event-driven
  – E.g., OKCupid, lighttpd, nginx, tornado

- Lighttpd: “Its event-driven architecture is optimized for a large number of parallel connections”
- Tornado: “Because it is non-blocking and uses epoll, it can handle thousands of simultaneous standing connections”
From Welsh, et al., SOSP 2001 “SEDA: An Architecture for Well-Conditioned, Scalable Internet Services
Flash Web Server

- Pai, Drushel, Zwaenepoel, 1999
- Influential work
- Compared four architectures
  - Multi-process servers
  - Multi-threaded servers
  - Single-process event-driven
  - Asymmetric Multi-process event driven
- AMPED was the fastest
Events (cont)

- Highly efficient code
  - Little or no switching overhead
  - Easy concurrency control
- Common complaint: hard to program and reason about
  - For people and tools
- Main reason: stack ripping
Events criticism: control flow

- Events obscure control flow
  - For programmers and tools

### Threads

```c
thread_main(int sock) {
    struct session s;
    accept_conn(sock, &s);
    read_request(&s);
    pin_cache(&s);
    write_response(&s);
    unpin(&s);
}

pin_cache(struct session *s) {
    pin(&s);
    if( !in_cache(s) ) ReadFileHandler.enqueue(s);
    else ResponseHandler.enqueue(s);
}
```

### Events

```c
CacheHandler(struct session *s) {
    pin(s);
    if( !in_cache(s) ) ReadFileHandler.enqueue(s);
    else ResponseHandler.enqueue(s);
}
RequestHandler(struct session *s) {
    ...; CacheHandler.enqueue(s);
}
ExitHandler(struct session *s) {
    ...; unpin(&s); free_session(s);
}
AcceptHandler(event e) {
    struct session *s = new_session(e);
    RequestHandler.enqueue(s); }
```
Events criticism: Exceptions

- Exceptions complicate control flow
  - Harder to understand program flow
  - Cause bugs in cleanup code

```c
thread_main(int sock) {
    struct session s;
    accept_conn(sock, &s);
    if(!read_request(&s))
        return;
    pin_cache(&s);
    write_response(&s);
    unpinn(&s);
}

pin_cache(struct session *s) {
    pin(&s);
    if(!in_cache(&s))
        ReadFileHandler.enqueue(s);
    else
        ResponseHandler.enqueue(s);
}

CacheHandler(struct session *s) {
    pin(s);
    if(!in_cache(s))
        ReadFileHandler.enqueue(s);
    else
        ResponseHandler.enqueue(s);
}

RequestHandler(struct session *s) {
    ...; if(error) return; CacheHandler.enqueue(s);
}

ExitHandler(event e) {
    struct session *s = new_session(e);
    RequestHandler.enqueue(s);
}
```
Events criticism: State Management

- Events require manual state management
- Hard to know when to free
  - Use GC or risk bugs

**Threads**

```
thread_main(int sock) {
    struct session s;
    accept_conn(sock, &s);
    if( !read_request(&s) )
        return;
    pin_cache(&s);
    write_response(&s);
    unpin(&s);
}
```

```
pin_cache(struct session *s) {
    pin(&s);
    if( !in_cache(&s) )  ReadFileHandler.enqueue(s);
    else                    ResponseHandler.enqueue(s);
}
```

```
RequestHandler(struct session *s) {
    …; if( error ) return;  CacheHandler.enqueue(s);
}
```

```
ExitHandler(struct session *s) {
    …; unpin(&s); free_session(s);
}
```

**Events**

```
CacheHandler(struct session *s) {
    pin(s);
    if( !in_cache(s) ) ReadFileHandler.enqueue(s);
    else                    ResponseHandler.enqueue(s);
}
```

```
AcceptHandler(event e) {
    struct session *s = new_session(e);
    RequestHandler.enqueue(s); }
```

Web Server

- Accept Conn.
- Read Request
- Pin Cache
- Read File
- Write Response
- Exit
Usual Arguments

• Events:
  – Hard to program (stack ripping)
  – Easy to deal with concurrency (cooperative task management)
    • Shared state is more explicit
  – High performance (low overhead, no switching, no blocking)

• Threads
  – Easy to reason about flow, state (automatic stack management)
  – Hard to deal with concurrency (preemptive task management)
    • Everything is shared
  – Lower performance (thread switching cost, memory overhead)
Capriccio (2003)

- Showed threads can perform as well as events
  - Avoid $O(n)$ operations
  - Cooperative lightweight user-level threads
    - (still one kernel thread per core)
  - Asynchronous I/O
    - Handled by the library
  - Variable-length stacks
  - The thread library runs an event-based system underneath!

![Graph comparing requests per second for Threaded and Event-Based Servers](chart.png)
Artificial Dichotomy!

• Old debate! Lauer and Needham, 78
  – Duality between process-based and message-passing
  – Updated by the Capriccio folks, 2003

<table>
<thead>
<tr>
<th>Threads</th>
<th>Events</th>
</tr>
</thead>
<tbody>
<tr>
<td>Monitors</td>
<td>Event handler &amp; queue</td>
</tr>
<tr>
<td>Exported functions</td>
<td>Events accepted</td>
</tr>
<tr>
<td>Call/return and fork/join</td>
<td>Send message / await reply</td>
</tr>
<tr>
<td>Wait on condition variable</td>
<td>Wait for new messages</td>
</tr>
</tbody>
</table>

• Performance should be similar
  – No inherent reason for threads to be worse
  – Implementation is key
Artificial Dichotomy

• **Threads**
  – Preemptive multitasking
  – Automatic stack management

• **Events**
  – Cooperative multitasking
  – Manual stack management (stack ripping)

• **Adya, 2002: you can choose your features!**
  – They show that you can have cooperative multitasking with automatic stack management

Adya, A. et al., 2002. “Cooperative Task Management without Manual Stack Management or Event-driven Programming is Not the Opposite of Threaded Programming
Threads vs. Events

• Today you still have to mostly choose either style (complete packages)
  – Thread-based servers very dependent on OS, threading libraries

• **Some promising directions!**
  – TAME allows you to write sequential C++ code (with some annotations), converts it into event-based
  – Scala (oo/functional language that runs on the JVM) makes threaded and event-based code look almost identical
Popular Event-Based Frameworks

• libevent
• libasync (SFS, SFS-light)
• Javascript
  – All browser code
  – Node.js at the server side
• GUI programming
Some available libraries

With material from Igor Ganichev
Python

• **Rich standard library**
  – url/http/ftp/pop/imap/smtp/telnet
  – SocketServer, HTTPServer, DocXMLRPCServer, etc

• **Twisted**
  – Very popular
  – Has *a lot* of stuff, but quite modular
  – Event-driven, many design patterns. Steep learning curve…
  – Well maintained and documented
Java

• Mature RPC library: RMI
• River: RMI + service discovery, mobile code
• Java.NIO
  – High-level wrapping of OS primitives
    • Select -> Selector . Socket -> Channel
  – Good, efficient buffer abstraction
• Jetty
  – Extensible, event-driven framework
  – High-performance
  – Avoid unnecessary copies
  – Other side doesn’t have to be in Java
• **Sockets!**
• **Direct access to what the OS provides**
• **Libevent**
  – Simple, somewhat portable abstraction of select() with uniform access to events: I/O, timers, signals
  – Supports /dev/poll, kqueue(2), event ports, select(2), poll(2) and epoll(4).
  – Well maintained, actively developed
  – Behind many very high-performance servers
    • Memcached
C++

• **Boost.ASIO**
  – Clean, lightweight, portable abstraction of sockets and other features
  – Not a lot of higher-level protocol support
  – Has support for both synchronous and asynchronous operations, threads (from other parts of Boost)

• **Others: ACE, POCO**
ICE

- Cross-language middleware + framework
  - Think twisted + protocol buffers
- Open source but owned by a company
- SSL, sync/async, threads, resource allocation, firewall traversal, event distribution, fault tolerance
- Supports many languages
  - C++, Java, .NET-languages (such as C# or Visual Basic), Objective-C, Python, PHP, and Ruby
Other “cool” approaches

• **Erlang, Scala, Objective C**
  – Support the Actor model: program is a bunch of actors sending messages to each other
  – Naturally extends to multi-core and multiple machines, as sending messages is the same

• **Go**
  – Built for concurrency, uses ‘Goroutines’, no shared state
  – “Don’t share memory to communicate, communicate to share memory”
Node.js

- Javascript server framework
- Leverages highly efficient Chrome V8 Javascript JIT runtime
- Completely event-based
- Many high-level libraries

```javascript
var http = require('http');
http.createServer(function (req, res) {
    res.writeHead(200, {'Content-Type': 'text/plain'});
    res.end('Hello World
');
}).listen(8124, "127.0.0.1");
console.log('Server running at http://127.0.0.1:8124/');
```
Final Assignment
Final Project

• Tethering IP over 3G
• Problem: Laptop in need of internet, no Wi-Fi available.
• On hand: Smartphone with 3G connection.
• Native applications don’t always allow custom network programming.
  — iOS App Store guidelines.
Custom Tethering Solution

- **Websockets to the rescue!**
  - Implemented in browsers.
  - Bi-directional, full-duplex connection over a single TCP socket.
  - Modern smartphone browsers have implemented websockets.
Implementation
Some questions

• How to connect phone to laptop?
• How to encode data?
• Virtual interfaces: TUN or TAP?
• Client: setting up routes
• Server: what to do with the packets you receive?
Some Resources

- **TUN/TAP Interfaces**
  - TunTap package for Mac OSX

- **Websocket Server**
  - Twisted

- **NAT**
  - Scapy

- **Base64 Encoding**