Threads vs. Events
Forms of task management

serial

preemptive

cooperative

(interrupt)

(yield)
Programming Models

thread model

Process address space

event model

event handler

blocking operations

events

run-time/kernel
Stack (really state) Management

Run-time stack

automatic

stack frame

A

B

C

state

Run-time stack

manual

event manager
Threads vs. Events

Threads are Bad

- Difficult to program
  - Synchronizing access to shared state
  - Deadlock
  - Hard to debug (race conditions, repeatability)
- Break abstractions
  - Modules must be designed “thread safe”
- Difficult to achieve good performance
  - simple locking lowers concurrency
  - context switching costs
- OS support inconsistent
  - semantics and tools vary across platforms/systems
- May not be right model
  - Window events do not map to threads but to events
Events are Bad - Threads are Good

- **Thread advantages**
  - Avoids “stack ripping” to maintain application context
  - Exception handling simpler due to history recorded in stack
  - Exploits available hardware concurrency

- **Events and Threads are duals**
  - Performance of well designed thread system equivalent to well designed event system (for high concurrency servers)
  - Each can cater to the common control flow patterns (a call/return pattern is needed for the acknowledgement required to build robust systems)
  - Each can accommodate cooperative multitasking
  - Stack maintenance problems avoided in event systems and can be mitigated in thread systems
Stack Ripping

```
CAInfo GetCAInfoBlocking(CAID caId) {
    CAInfo caInfo = LookupHashTable(caId);
    if (caInfo != NULL) {
        // Found node in the hash table
        return caInfo;
    }
    caInfo = new CAInfo();
    // DiskRead blocks waiting for
    // the disk I/O to complete.
    DiskRead(caId, caInfo);
    InsertHashTable(caId, CaInfo);
    return caInfo;
}
```
Ripped Code

```cpp
void GetCAInfoHandler1(CAID caId, Continuation *callerCont) {
    // Return the result immediately if in cache
    CAInfo *caInfo = LookupHashTable(caId);
    if (caInfo != NULL) {
        // Call caller's continuation with result
        (*callerCont->function)(caInfo);
        return;
    }

    // Make buffer space for disk read
    caInfo = new CAInfo();
    // Save return address & live variables
    Continuation *cont = new
    Continuation(&GetCAInfoHandler2, caId, caInfo, callerCont);
    // Send request
    EventHandle eh =
    InitAsyncDiskRead(caId, caInfo);
    // Schedule event handler to run on reply
    // by registering continuation
    RegisterContinuation(eh, cont);
}

void GetCAInfoHandler2(Continuation *cont) {
    // Recover live variables
    CAID caId = (CAID) cont->arg1;
    CAInfo *caInfo = (CAInfo*) cont->arg2;
    Continuation *callerCont =
    (Continuation*) cont->arg3;
    // Stash CAInfo object in hash
    InsertHashTable(caId, caInfo);
    // Now "return" results to original caller
    (*callerCont->function)(callerCont);
}
```
Ousterhout’s conclusions

Why Threads Are A Bad Idea
(for most purposes)

John Ousterhout
Sun Microsystems Laboratories

john.ousterhout@eng.sun.com
http://www.sunlabs.com/~onster

Conclusions

- Concurrency is fundamentally hard; avoid whenever possible.
- Threads more powerful than events, but power is rarely needed.
- **Threads much harder to program than events; for experts only.**
- Use events as primary development tool (both GUIs and distributed systems).
- Use threads only for performance-critical kernels.

Why Threads Are A Bad Idea

September 28, 1995, slide 15
Two approaches

- **Capriccio**
  - Each service request bound to an independent thread
  - Each thread executes all stages of the computation

- **Seda**
  - Each thread bound to one stage of the computation
  - Each service request proceeds through successive stages
Cappricio

- Philosophy
  - Thread model is useful
  - Improve implementation to remove barriers to scalability

- Techniques
  - User-level threads
  - Linked stack management
  - Resource aware scheduling

- Tools
  - Compiler-analysis
  - Run-time monitoring
Capriccio – user level threads

- User-level threading with fast context switch
- Cooperative scheduling (via yielding)
- Thread management costs independent of number of threads (except for sleep queue)

- Intercepts and converts blocking I/O into asynchronous I/O
- Does polling to determine I/O completion
Compiler Analysis - Checkpoints

- Call graph – each node is a procedure annotated with maximum stack size needed to execute that procedure; each edge represents a call
- Maximum stack size for thread executing call graph cannot be determined statically
  - Recursion (cycles in graph)
  - Sub-optimal allocation (different paths may require substantially different stack sizes)
- Insert checkpoints to allocate additional stack space (“chunk”) dynamically
  - On entry (e.g., C₀)
  - On each back-edge (e.g. C₁)
  - On each edge where the needed (maximum) stack space to reach a leaf node or the next checkpoints exceeds a given limit (MaxPath) (e.g., C₂ and C₃ if limit is 1KB)
- Checkpoint code added by source-source translation
Linked Stacks

- Thread stack is collection of non-contiguous blocks (‘chunks’)
- MinChunk: smallest stack block allocated
- Stack blocks “linked” by saving stack pointer for “old” block in field of “new” block; frame pointer remains unchanged
- Two kinds of wasted memory
  - Internal (within a block) (yellow)
  - External (in last block) (blue)
- Two controlling parameters
  - MaxPath: tradeoff between amount of instrumentation and run-time overhead vs. internal memory waste
  - MaxChunk: tradeoff between internal memory waste and external memory waste
- Memory advantages
  - Avoids pre-allocation of large stacks
  - Improves paging behavior by (1) leveraging LIFO stack usage pattern to share chunks among threads and (2) placing multiple chunks on the same page
Resource-aware scheduling

- **Blocking graph**
  - Nodes are points where the program blocks
  - Arcs connect successive blocking points
- Blocking graph formed dynamically
  - Appropriate for long-running program (e.g. web servers)
- Scheduling annotations
  - Edge – exponentially weighted average resource usage
  - Node – weighted average of its edge values (average resource usage of next edge)
  - Resources – CPU, memory, stack, sockets
- Resource-aware scheduling:
  - Dynamically prioritize nodes/threads based on whether the thread will increase or decrease its use of each resource
  - When a resource is scarce, schedule threads that release that resource
- Limitations
  - Difficult to determine the maximum capacity of a resource
  - Application-managed resources cannot be seen
  - Applications that do not yield
Performance comparison

- Apache – standard distribution
- Haboob – event-based web server
- Knot – simple, threaded specially developed web server
SEDA – Staged Event-Driven Architecture

- **Goals**
  - **Massive concurrency**
    - required for heavily used web servers
    - large spikes in load (100x increase in demand)
    - requires efficient, non-blocking I/O
  - **Simplify constructing well-conditioned services**
    - “well conditioned”: behaves like a simple pipeline
    - offers graceful degradation, maintaining high throughput as load exceeds capacity
    - provides modular architecture (defining and interconnecting “stages”)
    - hides resource management details
  - **Introspection**
    - ability to analyze and adapt to the request stream
  - **Self-tuning resource management**
    - thread pool sizing
    - dynamic event scheduling

- **Hybrid model**
  - combines threads (within stages) and events (between stages)
SEDAs point of view

Thread model and performance

Event model and performance
SEDA - structure

- Event queue – holds incoming requests
- Thread pool
  - Takes requests from event queue and invokes event handler
  - Limited number of threads per stage
- Event handler
  - Application defined
  - Performs application processing and possibly generates events for other stages
  - Does not manage thread pool or event queue
- Controller – performs scheduling and thread management
Resource Controllers

Thread pool controller
- Thread added (up to a maximum) when event queue exceeds threshold
- Thread deleted when idle for a given period

Batching controller
- Adjusts batching factor: the number of events processed at a time
- High batching factor improves throughput
- Low batching factor improves response time
- Goal: find lowest batching factor that sustains high throughput
Asynchronous Socket layer

- Implemented as a set of SEDA stages
- Each asyncSocket stage has two event queues
- Thread in each stage serves each queue alternately based on time-out
- Similar use of stages for file I/O
Threads vs. Events

Performance

- Apache
  - process-per-request design
- Flash
  - event-driven design
  - one process handling most tasks
- Haboob
  - SEDA-based design

Fairness

- Measure of number of requests completed per client
- Value of 1 indicates equal treatment of clients
- Value of k/N indicates k clients received equal treatment and n-k clients received no service
TAME

- expressive abstractions for event-based programming
- implemented via source-source translation
- avoids stack ripping
- type safety and composability via templates

A typical thread programming problem

Problem: the thread becomes blocked in the called routine (f) and the caller (c) is unable to continue even if it logically is able to do so.
A partial solution

Issues

- Synchronization: how does the caller know when the signal has occurred without busy-waiting?
- Data: how does the caller know what data resulted from the operation?
A “Tame” solution

A non-blocking, asynchronous operation, e.g., I/O

\[
\text{e(a): event<T>}
\]

\[
e.\text{trigger}(\text{data})
\]

(1) e

(2) f

(3) (non-blocking, asynchronous operation, e.g., I/O)

(4) (signal + data)

(5) (6)

(7) (8) a <- data

(9) r: \langle I \rangle

(10) (11)

\[
n\text{handler}
\]

slot

\[
<T> \text{ a;}
\]

\[
\text{a \langle I \rangle rendezvous}<\langle I \rangle>
\]

\[
\text{wait point}
\]
## Tame Primitives

<table>
<thead>
<tr>
<th>Classes</th>
<th>Keywords &amp; Language Extensions</th>
<th>Functions &amp; Methods</th>
</tr>
</thead>
<tbody>
<tr>
<td>event&lt;&gt;</td>
<td>twait(r[i]); A wait point. Block on explicit rendezvous r, and optionally set the event ID i when control resumes. tamed</td>
<td>mkevent(r,i,s); Allocate a new event with event ID i. When triggered, it will awake rendezvous r and store trigger value in slot s.</td>
</tr>
<tr>
<td>event&lt;T&gt;</td>
<td>tvars { ... } Marks-safe local variables. twait { statements; } Wait-point syntactic sugar: block on an implicit rendezvous until all events created in statements have triggered.</td>
<td>mkevent(s); Allocate a new event for an implicit twait{} rendezvous. When triggered, store trigger value in slot s.</td>
</tr>
<tr>
<td>rendezvous&lt;&gt;</td>
<td>e.trigger(v); Trigger event e, with trigger value v. timer(t0,e); wait_on_fd(fd,rw,e);</td>
<td>Primitive event interface for timeouts and file descriptor events, respectively.</td>
</tr>
</tbody>
</table>

**Figure 2:** Tame primitives for event programming in C++.
An example

```c
1 void multidns(dnsname name[], ipaddr a[], int n) {
2     for (int i = 0; i < n; i++)
3         a[i] = gethostbyname(name[i]);
4 }
```

```c
1 tamed multidns_tame(dnsname name[], ipaddr a[],
                       int n, event<> done) {
2     tvars { int i; }
3     for (i = 0; i < n; i++)
4         twait { gethost_event(name[i], mkevent(a[i])); }
5     done.trigger();
6 }
```

tamed gethost_event(dsname name, event<ipaddr> e);
Variations on control flow

```
1  tamed multidns_par(dnsname name[], ipaddr a[],
                    int n, event<> done) {
2    twait {
3      for (int i = 0; i < sz; i++)
4        gethost_ev(name[i], mkevent(a[i]));
5    }
6    done.trigger();
7 }
```

parallel control flow

```
1  tamed multidns_win(dnsname name[], ipaddr a[],
                      int n, event<> done) {
2    tvars { int sent(0), recv(0); rendezvous<> r; }
3    while (recv < n)
4      if (sent < n && sent - recv < WINDOWSIZE) {
5        gethost_ev(name[sent], mkevent(r,a[sent]));
6        sent++;
7      } else {
8        twait(r);
9        recv++;
10     }
11    done.trigger();
12 }
```

window/pipeline control flow
Event IDs & Composability

```cpp
template <typename T> tamed
__add_timeout(event<T> &e_base, event<bool, T> e) {
  tvars { rendezvous<bool> r; T result; bool rok; }
  timer(TIMEOUT, mkevent(r, false));
  e_base = mkevent(r, true, result);
  twait(r, rok);
  e.trigger(rok, result);
  r.cancel();
}

template <typename T> event<T> add_timeout(event<bool, T> e) {
  event<T> e_base;
  __add_timeout(e_base, e);
  return e_base;
}
```

![Diagram showing the relationships between event IDs and composability, including a timer, event variable, and host event with triggers and wait operations.](image)
Closures

Smart pointers and reference counting insure correct deallocation of events, rendezvous, and closures.
### Performance (relative to Capriccio)

<table>
<thead>
<tr>
<th></th>
<th>Capriccio</th>
<th>Tame</th>
</tr>
</thead>
<tbody>
<tr>
<td>Throughput (connections/sec)</td>
<td>28,318</td>
<td>28,457</td>
</tr>
<tr>
<td>Number of threads</td>
<td>350</td>
<td>1</td>
</tr>
<tr>
<td>Physical memory (kB)</td>
<td>6,560</td>
<td>2,156</td>
</tr>
<tr>
<td>Virtual memory (kB)</td>
<td>49,517</td>
<td>10,740</td>
</tr>
</tbody>
</table>

**Figure 7**: Measurements of Knot at maximum throughput. Throughput is averaged over the whole one-minute run. Memory readings are taken after the warm-up period, as reported by ps.