Threads vs. Events

TAME – Event Style Programming with Threads
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 handler is a convenient statement to express the non-blocking, asynchronous operation, e.g., I/O. For example, a handler can be expressed as follows:

```
handler:
e.trigger(data)
```

where `e` is an event and `data` is the data to be triggered.
# 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($r[i]$);</td>
<td>mkevent($r, i, s$);</td>
</tr>
<tr>
<td></td>
<td>• A basic event.</td>
<td>• Allocate a new event with event ID $i$.</td>
</tr>
<tr>
<td>event&lt;$T$&gt;</td>
<td>twait($r[i]$);</td>
<td>• When triggered, it will awake rendezvous $r$ and store trigger value in slot $s$.</td>
</tr>
<tr>
<td></td>
<td>• An event with a single trigger value of type $T$.</td>
<td>mkevent($s$);</td>
</tr>
<tr>
<td></td>
<td>This value is set when the event occurs; an example might be a character read from a file descriptor. Events may also have multiple trigger values of types $T_1...T_n$.</td>
<td>• 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>twait{};</td>
<td>e.trigger($v$);</td>
</tr>
<tr>
<td></td>
<td>• Represents a set of outstanding events with event IDs of type $I$. Callers name a rendezvous when they block, and unblock on the triggering of any associated event.</td>
<td>• Trigger event $e$, with trigger value $v$.</td>
</tr>
<tr>
<td></td>
<td>twait{};</td>
<td>timer($t, e$);</td>
</tr>
<tr>
<td></td>
<td>• Wait point syntactic sugar: block on an implicit rendezvous until all events created in statements have triggered.</td>
<td>• wait_on_fd($fd, rw, e$);</td>
</tr>
</tbody>
</table>

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

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

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

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

```c
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

```c
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

```
1 template <typename T> tamed
2    __add_timeout(event<T> &e_base, event<bool, T> e) {
3      tvars { rendezvous<bool> r; T result; bool rok; }
4      timer(TIMEOUT, mkevent(r, false));
5      e_base = mkevent(r, true, result);
6      twait(r, rok);
7      e.trigger(rok, result);
8      r.cancel();
9    }

9 template <typename T> event<T> add_timeout(event<bool, T> e) {
10     event<T> e_base;
11     __add_timeout(e_base, e);
12     return e_base;
13 }
```
Closures

```c
f(...params...)
{
    rendezvous<> r;
    tvars { ...locals...};
    twait(r);
    continue_here:
}
```

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`.