User-level threads…

… with threads.
Paul Turner <pjt@google.com>
Threading Models

- **1:1 (Kernel Threading)**
  User contexts correspond directly with their own kernel schedulable entity. Everything does this (e.g. Linux, Windows, Solaris, NetBSD, FreeBSD).

- **N:1 (User Threading)**
  User-level threading multiplexed onto a single kernel context. No kernel awareness of user-level threading structure.

- **M:N (Hybrid)**
  Kernel assisted N:1 threading, using M kernel contexts. Classic example is *Scheduler Activations*
Parallel programming models

- Synchronous (Thread/Request)
- Delegate Event (Asynchronous callbacks)
- Message passing / Event Loops
Callback Types

- **Asynchronous callbacks** do not block their caller. They are typically run either within a separate thread, or after their invoker’s completion. e.g.:
  
  ```
  Executor() -> Add(Callback(...))
  ```

- **Synchronous callbacks** are always completed (often within the same thread) before control is returned to the caller. e.g.:
  
  ```
  foo->Lookup(&context, arg, &result);
  ```
**Complexity: “Own” vs “View”**

```c
int x;

(1) Foo(&x);

vs

(2) Executor() -> Add(Callback(Foo, &x));
```

In (2), the reader must immediately be concerned with:
- Synchronization of access to `x`.
- Co-ordination of `x`’s lifetime.
- What happens after `Foo` completes?
Callbacks are not a Programming Model

- Threads are base unit of concurrency... **but**
- **Requests** are the typical “currency” servers must build parallelism around.
Programming Models: Thread per request

Advantages
● Simple programming model
● Good data-locality

Challenges
● Harder to realize parallelism within a request
● Latency predictability varies inversely with load
  ○ 1000 outstanding requests means 1000 threads. Do you know where your threads are?
Programming Models: Asynchronous Worker Objects

Advantages
- Greater control of work partitioning, improved latency predictability.
- Lower overheads achievable.

Challenges
- Complex programming model; control and data-flow now require encapsulation. No longer strictly linear. Additional resource boundaries introduced. Code written under this model depends more heavily on primitives such as *Conditions*.
- Loss of data locality.
Crux

crux, n: something that torments by its puzzling nature; a perplexing difficulty.

We ‘fixed’ thread-per-request by introducing concurrency objects that are smaller than a request.

... yet many of thread-per-requests issues caused by concurrency!

... communication still cumbersome.
CSP: Go’s take

Go provides constructs allowing for a more synchronous model; allowing control flow to be represented in a linear fashion, while realizing available concurrency.

Key features:
● Goroutines
● Channels
● Select statement

What makes this type of model hard to achieve in C++?

Where does this hybrid face challenges? Are they barriers to adoption within C++?
How much does a context switch cost?

```
~# for ((i=0;i<10;i++)) do time ./pipe_test 500000; done
real    0m2.911s
real    0m3.052s
real    0m5.282s
real    0m4.724s
real    0m6.780s
real    0m1.250s
...
```

Why the inconsistency?
How about a raw futex?

- `sys_futex()` allows a program to wait for an address to change, or signal anyone waiting on a given address.

- Benchmark	Time (ns)	CPU (ns)	Iterations

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>Time (ns)</th>
<th>CPU (ns)</th>
<th>Iterations</th>
</tr>
</thead>
<tbody>
<tr>
<td>BM_Futex</td>
<td>4705</td>
<td>3555</td>
<td>1000000</td>
</tr>
<tr>
<td>BM_Futex</td>
<td>2757</td>
<td>1917</td>
<td>1000000</td>
</tr>
<tr>
<td>BM_Futex</td>
<td>2931</td>
<td>1983</td>
<td>1000000</td>
</tr>
<tr>
<td>BM_Futex</td>
<td>2791</td>
<td>1935</td>
<td>1000000</td>
</tr>
<tr>
<td>BM_Futex</td>
<td>2932</td>
<td>1933</td>
<td>1000000</td>
</tr>
</tbody>
</table>

- A little faster, ~2.7 usec/switch typical.
Wake-up CPU interactions

<table>
<thead>
<tr>
<th>CPU 0</th>
<th>CPU 1</th>
<th>CPU 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>a.  &lt;t1 wakes t2&gt;</td>
<td>&lt;idle&gt;</td>
<td></td>
</tr>
<tr>
<td>b.  &lt;t1 sleeps&gt;</td>
<td></td>
<td></td>
</tr>
<tr>
<td>c.  &lt;idle&gt;</td>
<td></td>
<td></td>
</tr>
<tr>
<td>d.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>e.  &lt;t1 wakes t2&gt;</td>
<td>&lt;t3 resumes&gt;</td>
<td>&lt;enqueue t2&gt;</td>
</tr>
</tbody>
</table>

- IPI
So what's the true cost?

1 million context switches
~1.326 usec per switch

Can we do better?
Futex (pinned)

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>Time(ns)</th>
<th>CPU(ns)</th>
<th>Iterations</th>
</tr>
</thead>
<tbody>
<tr>
<td>BM_Futex</td>
<td>1028</td>
<td>1022</td>
<td>1000000</td>
</tr>
<tr>
<td>BM_Futex</td>
<td>1030</td>
<td>1024</td>
<td>1000000</td>
</tr>
<tr>
<td>BM_Futex</td>
<td>1021</td>
<td>1016</td>
<td>1000000</td>
</tr>
<tr>
<td>BM_Futex</td>
<td>1022</td>
<td>1016</td>
<td>1000000</td>
</tr>
<tr>
<td>BM_Futex</td>
<td>1012</td>
<td>1006</td>
<td>1000000</td>
</tr>
</tbody>
</table>

Down to ~1 usec, getting better.. but little else we can do.
Context-switch cost: key observations

- The switch into kernel mode \((\text{ring0})\) is surprisingly inexpensive
  - \(<50\text{ns round trip}.\)

- Majority of the context-switching cost attributable to the complexity of the scheduling decision by a modern SMP cpu scheduler.
Syscall API

`pid_t switchto_wait(timespec *timeout)`
- Enter an 'unscheduled state', until our control is re-initiated by another thread or external event (signal).

`void switchto_resume(pid_t tid)`
- Resume regular execution of tid

`pid_t switchto_switch(pid_t tid)`
- Synchronously transfer control to target sibling thread, leaving the current thread unscheduled.
- Analogous to:
  - \textit{Atomically} \{ Resume(t1); Wait(NULL); \}
Kernel View

CPU i:

Thread A

Thread B

Minimal scheduling operation.
- B inherits A’s virtual runtime.
- B was not runnable, so we don’t need to remove it from runqueues.
- B holds references on same objects as A.

(Unscheduled state is \texttt{TASK\_INTERRUPTIBLE} with a special return stack.)
API choices/Considerations

- Operations must be commutative (reversible). 
  \( \{T1:Wait, \; T2:\text{Switch}(T1)\} \) should behave the same as \( \{T2:\text{Switch}(T1), \; T1:Wait\} \)

- Requiring a re-entrant (asynchronous) user-scheduler entry classically hard; prefer a synchronous programming model.

- User scheduling id compatible with kernel scheduling; the kernel scheduler grants us quanta, we schedule \textit{within} that quanta.

- Load-balancing is best left to the load-balancer.
# Context-switch performance

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>Time(ns)</th>
<th>CPU(ns)</th>
<th>Iterations</th>
</tr>
</thead>
<tbody>
<tr>
<td>BM_Futex</td>
<td>2905</td>
<td>1958</td>
<td>10000000</td>
</tr>
<tr>
<td>BM_GoogleMutex</td>
<td>3102</td>
<td>2326</td>
<td>10000000</td>
</tr>
<tr>
<td><strong>BM_SwitchTo</strong></td>
<td><strong>179</strong></td>
<td><strong>178</strong></td>
<td><strong>3917412</strong></td>
</tr>
<tr>
<td>BM_SwitchResume</td>
<td>2734</td>
<td>1554</td>
<td>10000000</td>
</tr>
</tbody>
</table>
Advantages of maintaining a 1:1 threading model

- Semantics dependent on thread identity (e.g. TLS, tid, etc) are preserved.
- Existing debugging and profiling tools work naturally.
- Existing thread management APIs (e.g. nice(2), tkill) continue to work.
- **Compatible with existing code.**
Related: Socket locality

- Thread A makes request, sends on socket, waits on response
- Response comes to Thread B, a networking thread
- B needs to wake A
  - B would like A to run on the same CPU (locality)
Context-switching lacks context

static void ContextSwitcher(Mutex* m, ...) {
    for (; n > 0; n--) {
        a) m->LockWhen(Condition(own_mutex(), val));
        b) <mutex_owner = next thread>; m->Unlock();
    }
}

- When releasing resource, no way of advertising that our execution is about to stop.
Backup
Managed Concurrency: SwitchToGroups

T1, T2, T3, ..., Tn

Blocking Delegate 1

Context
Managed Concurrency

1. $t_1:u$ read(2) $\rightarrow$ $t_1:k$ blocks $\rightarrow$ SwitchTo $\rightarrow$ $t_D:k$

   *IF IDLE:*
   a. $t_D:u$ No other threads $\rightarrow$ WaitForUnblockingOrNew()
   b. $t_1:k$ read returns, $t_1:k$ allowed to unblock instead of fast-wait
   c. $t_1:u$ read(2) returns

   *ELSE (suppose runnable $t_2$ exists)*
   a. $t_D:u$ $\rightarrow$ SwitchTo + BecomeDesignate $\rightarrow$ $t_2$
   b. $t_2:u$ resumes working
   c. ($t_1:k$ read returns, enters a fast-wait state)

Since $t_2$ is running (and we chose to have 1 active thread) we've explicitly chosen to defer the processing of $t_1$'s wake-up; unlike the 1:1 case, $t_2$'s execution proceeds undisturbed, skipping work of the re-enqueue and preemption.