Networking TS & Threadpools

Exploring interactions between networking TS and system threadpools.
Networking TS – io_context

A brief overview of what Networking TS offers

EXECUTION POLICY

thread 1
io_context::run()

thread 2
io_context::run()

... (more async sources)

thread N
io_context::run()

io_context::stop()
io_context::stopped()
io_context::restart()

ASYNC SOURCES

executor

socket

async_accept
async_connect
async_read/write

timer

async_wait

post
defer

dispatch

... (more async sources)
Simple io_context example

```cpp
int main() {
    io_context io;

    system_timer slow_timer(io, hours(15));
    slow_timer.async_wait([](auto) {
        puts("Timer fired");
    });

    system_timer fast_timer(io, seconds(1));
    fast_timer.async_wait([&io](auto) {
        io.stop();
    });

    io.run();
}
```
Windows TP & GCD & Linux GCD*

Thread Factories:
- Observe I/O completions
- Threads getting blocked
- Broker resources across processes
- NUMA aware

Platforms have highly efficient system threadpools. We would like to take advantage of them with Networking TS

* Linux GCD does not yet have a require kernel component and thus not as efficient as native OS X implementation
io_context vs threadpool

- *io_context* offers services similar to GCD/libdispatch or Windows Threadpool, but without thread creation policy (i.e. bring your own thread)
- **idea**: introduce *tp_context* as a representation of a system threadpool usable with all async sources: (sockets, timers, executors, etc) as io_context, but, with different execution policies (no run(), poll(), etc)
  - possibly also, *tp_private_context*(min-threads, max-threads) which uses a private threadpool that does not share threads with others.
Networking TS + tp_context

Idea: Same sources, different execution model

EXECUTION POLICY

thread 1
thread 2
... thread N

OS/Runtime manager

tp_context::async_join(cb)
tp_context::cancel()
tp_context::cancelling()

tp_context::join

ASYNC SOURCES

post
defer
dispatch
executor

socket
async_accept
async_connect
async_read/write

timer
async_wait

... (more async sources)
Simple tp_context example

```c
int main() {
    tp_context tp;

    system_timer slow_timer(tp, hours(15));
    slow_timer.async_wait([](auto) {
        puts("Timer fired");
    });

    system_timer fast_timer(tp, seconds(1));
    fast_timer.async_wait([&tp](auto) {
        tp.cancel();
    });

    tp.join();
}
```

Keeps the usage very similar to how Networking TS work today.
### io_context VS tp_context

<table>
<thead>
<tr>
<th>io_context</th>
<th>tp_context</th>
</tr>
</thead>
<tbody>
<tr>
<td>get_executor()</td>
<td>get_executor()</td>
</tr>
<tr>
<td>stop()</td>
<td>cancel()</td>
</tr>
<tr>
<td>stopped()</td>
<td>cancelling()</td>
</tr>
<tr>
<td>restart()</td>
<td>restart()</td>
</tr>
<tr>
<td>run()</td>
<td>async_join(cb)</td>
</tr>
<tr>
<td>run_for(rel_time)</td>
<td>join()</td>
</tr>
<tr>
<td>run_until(abs_time)</td>
<td></td>
</tr>
<tr>
<td>run_one()</td>
<td></td>
</tr>
<tr>
<td>run_one_for(rel_time)</td>
<td></td>
</tr>
<tr>
<td>run_one_until(abs_time)</td>
<td></td>
</tr>
<tr>
<td>poll()</td>
<td></td>
</tr>
<tr>
<td>poll_one()</td>
<td></td>
</tr>
</tbody>
</table>

Possible interface of tp_context
tp_contexts are purely work trackers

join/cancel only affect work issued through a particular tp_context.
No control over actual threadpool threads.
Could be hierarchical. Cancellation/Task Lifetime domains

Can act as cancellation/task lifetime domains.

Can cancel or join leaves, or subtrees without affecting others.

OS Threadpool / Process Wide TP
More `tp_contexts`?

```
<table>
<thead>
<tr>
<th>tp_raw_context</th>
</tr>
</thead>
<tbody>
<tr>
<td>get_executor()</td>
</tr>
</tbody>
</table>
```

```
<table>
<thead>
<tr>
<th>tp_context</th>
</tr>
</thead>
<tbody>
<tr>
<td>get_executor()</td>
</tr>
<tr>
<td>cancel()</td>
</tr>
<tr>
<td>cancelling()</td>
</tr>
<tr>
<td>restart()</td>
</tr>
<tr>
<td>async_join(cb)</td>
</tr>
<tr>
<td>join()</td>
</tr>
</tbody>
</table>
```

```
<table>
<thead>
<tr>
<th>tp_suspendable_context</th>
</tr>
</thead>
<tbody>
<tr>
<td>get_executor()</td>
</tr>
<tr>
<td>cancel()</td>
</tr>
<tr>
<td>cancelling()</td>
</tr>
<tr>
<td>restart()</td>
</tr>
<tr>
<td>async_join(cb)</td>
</tr>
<tr>
<td>join()</td>
</tr>
<tr>
<td>suspend()</td>
</tr>
<tr>
<td>resume()</td>
</tr>
</tbody>
</table>
```

Maybe. If you never cancel, join and exit your program with `exit(0)`.

Only if having `suspend/resume` adds overhead. Otherwise those could be part of `tp_context`.

Threadpools & Networking TS

11
How to integrate tp_context into Networking TS

• Make IoContext template parameter:
  • most flexible
  • most disruptive to existing users (deduction guides helps only with trivial examples)

• Make io_context a base class with two concrete implementations:
  • tp_context (join/cancel)
  • io_context_runner (which has run(), poll(), etc)

• Make io_context switch the behavior based on ctor
  • io_context(system_threadpool_t)
  • io_context(private_threadpool_t, min, max)
  • io_context(io_context&) –hierarchical
  • - run/poll/etc become less meaningful if run by the threadpool
Conclusion

• A longer paper to come if this general direction deemed promising.