

Document Number: P0399R0

Audience: SG1 & LEWG

Date: 2017-10-15

Revises: None

Reply to: Gor Nishanov (gorn@microsoft.com)

# Networking TS & Threadpools

Exploring interactions between networking TS and  
system threadpools.

# Networking TS – io\_context

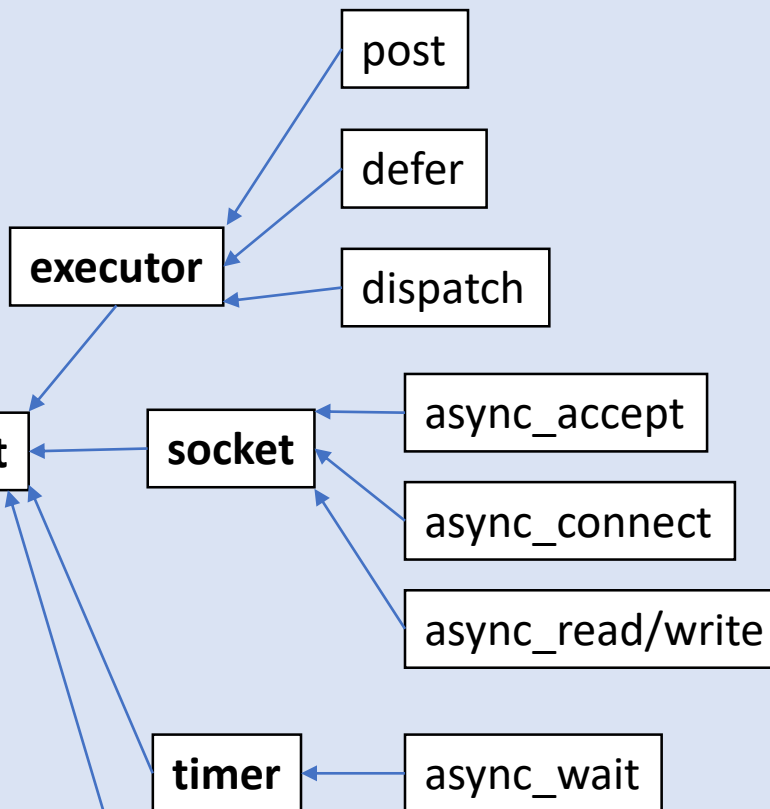
A brief overview of what Networking TS offers

## EXECUTION POLICY

thread 1  
io\_context::run()  
thread 2  
io\_context::run()  
...  
thread N  
io\_context::run()

io\_context::stop()  
io\_context::stopped()  
io\_context::restart()  
...

## ASYNC SOURCES



... (more async sources)

# Simple io\_context example

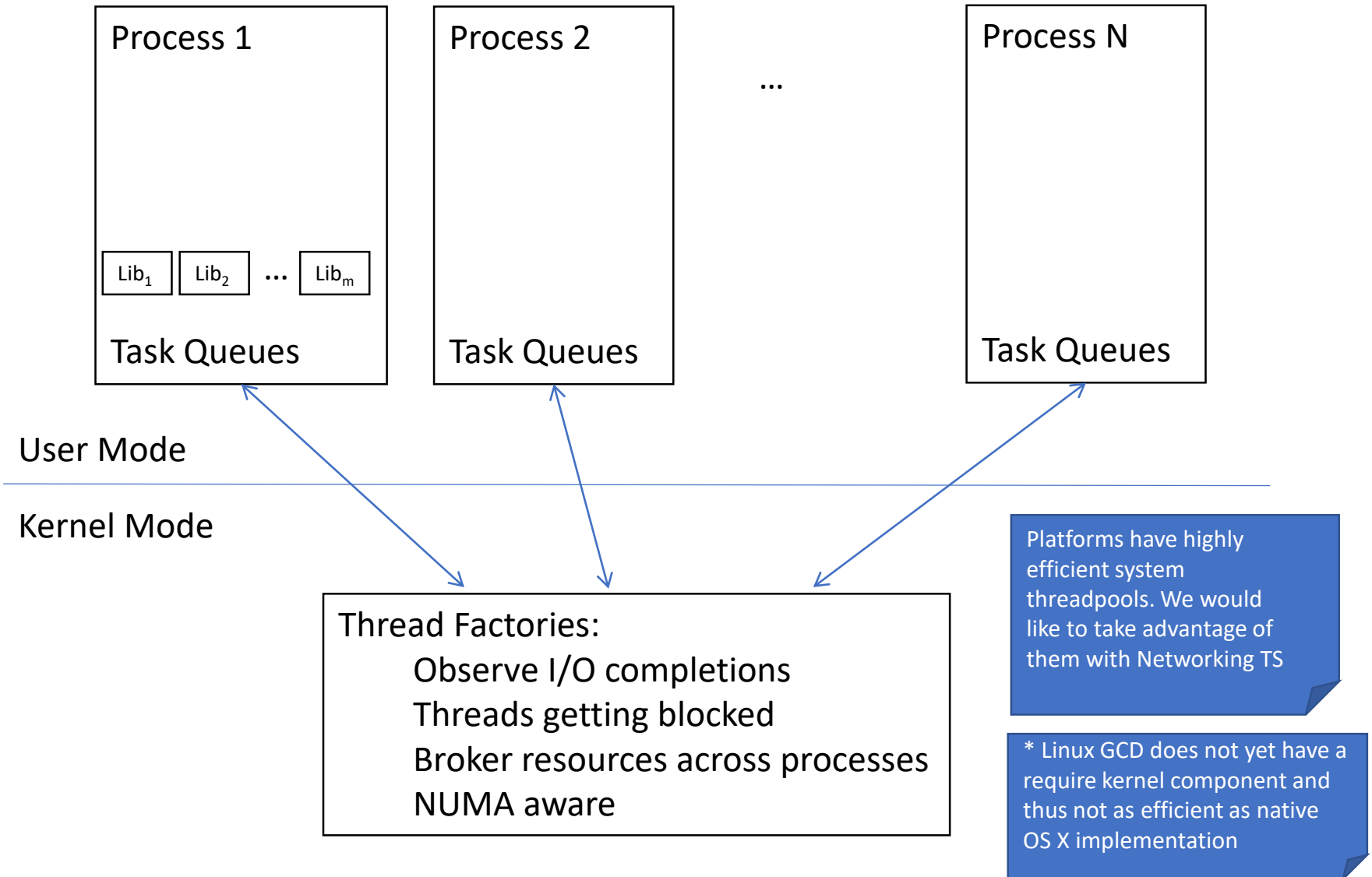
```
int main() {
    io_context io;

    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\*

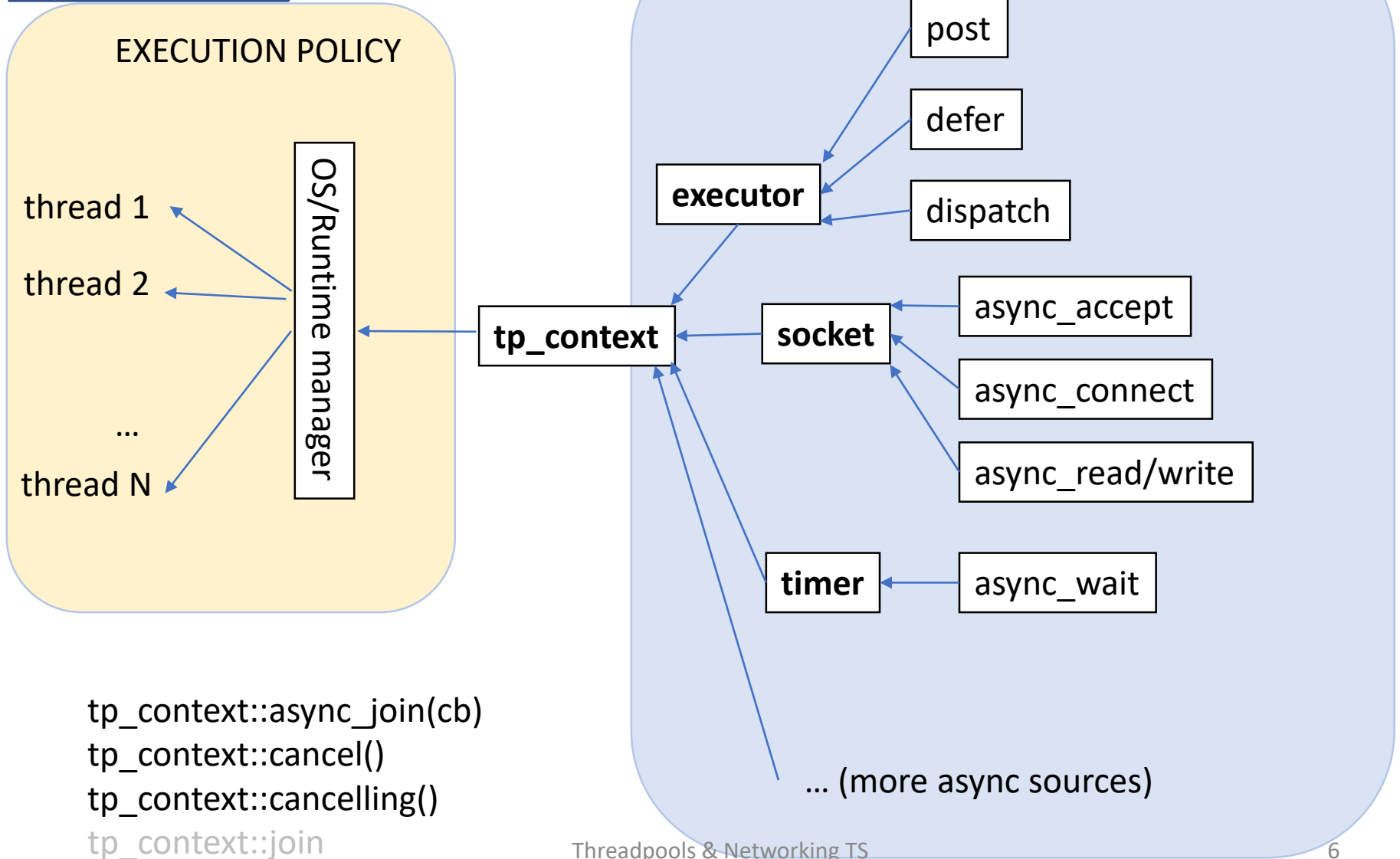


# io\_context vs threadpool

- *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



# Simple tp\_context example

```
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

<b>io_context</b>
get_executor()
stop()
stopped()
restart()
run()
run_for(rel_time)
run_until(abs_time)
run_one()
run_one_for(rel_time)
run_one_until(abs_time)
poll()
poll_one()

<b>tp_context</b>
get_executor()
cancel()
cancelling()
restart()
async_join(cb)
join()

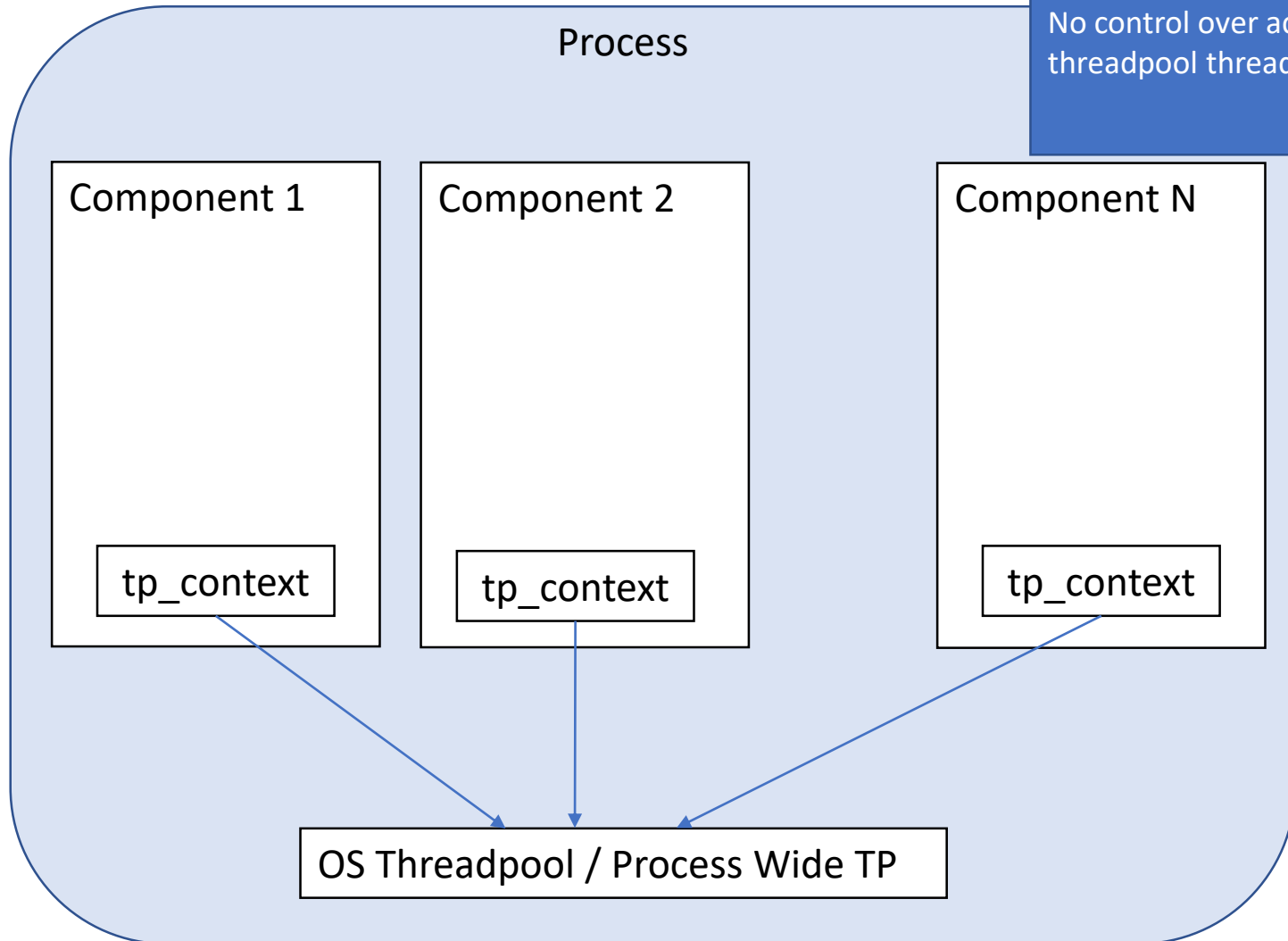
Possible interface  
of tp\_context



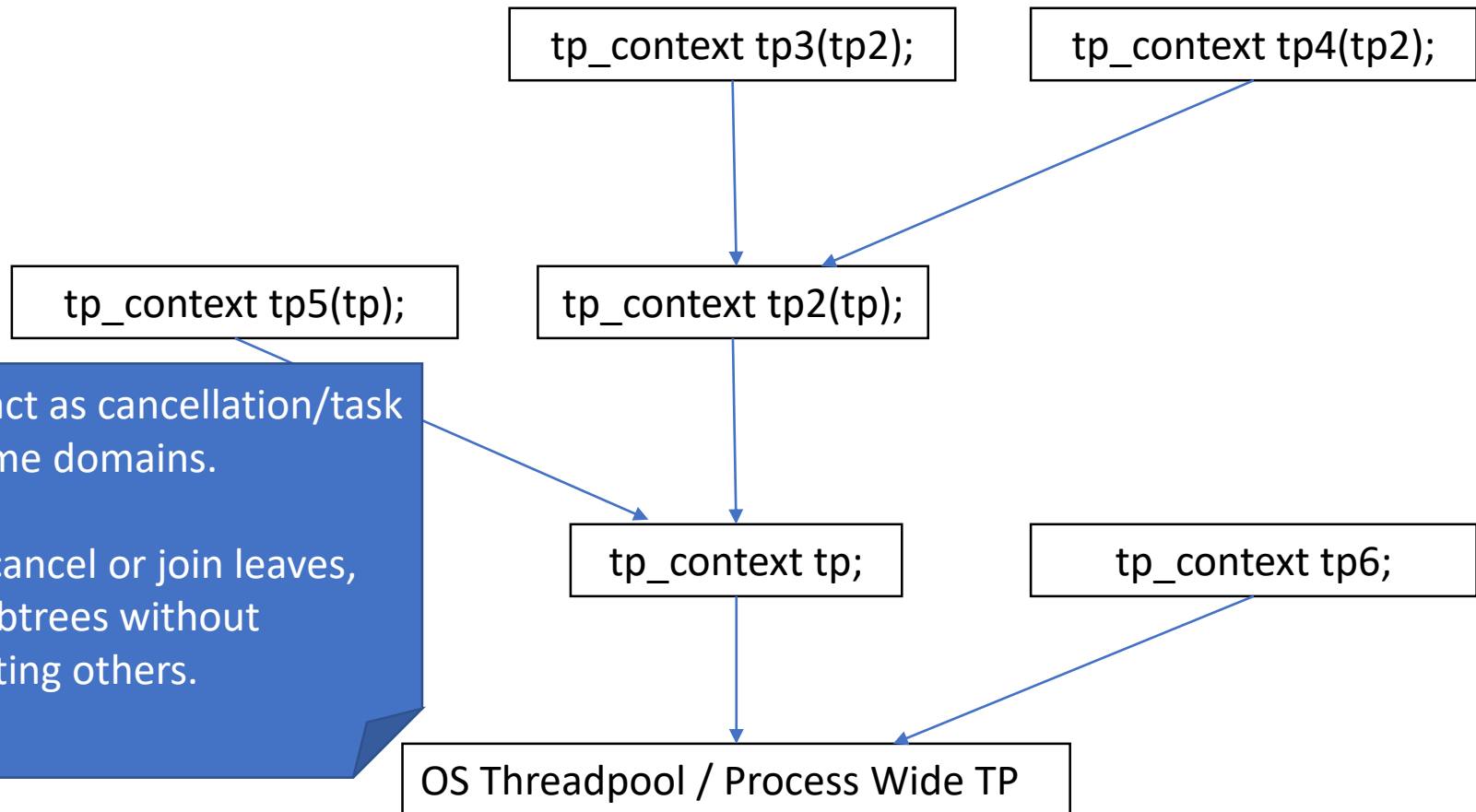
# tp\_contexts are purely work trackers

join/cancel only affect work issued through a particular tp\_context.

No control over actual threadpool threads.



# Could be hierarchical. Cancellation/Task Lifetime domains



# More tp\_contexts?

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

<b>tp_raw_context</b>
get_executor()

<b>tp_context</b>
get_executor()
cancel()
cancelling()
restart()
async_join(cb)
join()

<b>tp_suspendable_context</b>
get_executor()
cancel()
cancelling()
restart()
async_join(cb)
join()
suspend()
resume()

Only if having suspend/resume adds overhead. Otherwise those could be part of tp\_context

# 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.