Pragmatic Multithreaded C++

High or Low level solution ?

- **First choice – high level structure and tools**
- Pros:
  - Quick results
  - Practical problem identification
  - (Typically) Better readability and portability
  - Quick pattern implementation
  - Better separation of responsibilities
- Cons:
  - Maybe not enought performance (mesure it)
  - The wrong tool/structure/pattern/containerf/… for the problem
  - Threads oversubscription
  - Destroy without calling get()

std::async – context of use

- For:
  - Simple background tasks
  - Return result at end (job like)
  - Easy, tracebility
  - Low thread execution control
- Not for:
  - Inter-thread synchronization
  - Shared data
  - Thread count control

Before we talk about std::async, what does it return?

std::future<...>

- Non-copyable – move semantics
- No thread safety! - one thread get()
- Moved data – MoveContructible
- .get() - move data
- .valid() return false after move (one-shoot move)
- Need MOVE to ONE thread

future_call.cpp

std::shared_future<...>

- Copyable
- Thread safety! - many get()
- Copied data – CopyConstructible
- .get() - return const ref
- .valid() return „permanent true” (if shared_future copy)
- Need COPY to MANY thread

shared_call.cpp

std::*future<...> state

- States:
  - future_status::deferred – lazy execution
  - future_status::ready – result ready
  - future_status::timeout – under calculation…
- Get by .wait*() call

state_call.cpp

Back to the std::async(...)

- Launch policy (BitMaskType):
  - std::launch::deferred – lazy (after .get()) run in current thread
  - std::launch::async – run on different thread
  - std::launch::default – *(not specified)*, implementation specified

    std::launch::default == std::launch::deferred | std::launch::async

- Arguments:
  - Callable object – function, lambda, functional object
  - Args to call – Copyable (if ref, std::ref(...)

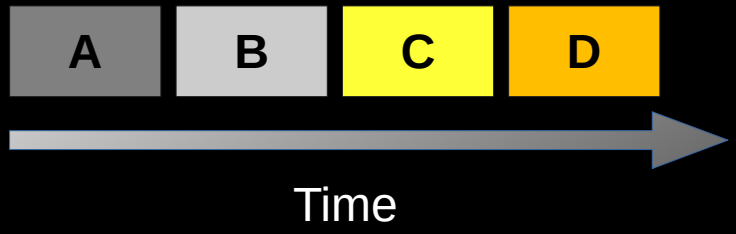
async_call.cpp

std::async(…) - oversubscription

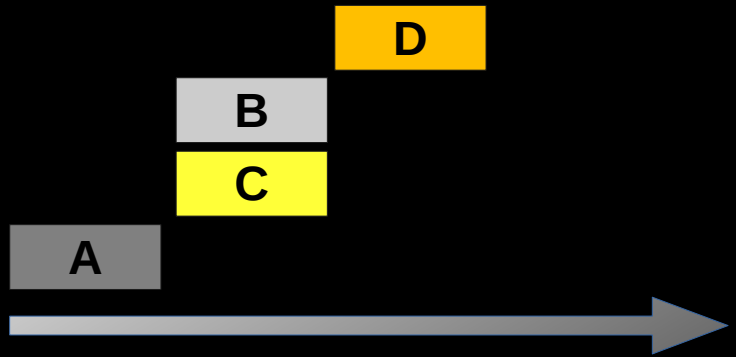
- I/O – not so dangerous…
  - os classifies them correctly (i/o intensive process/thread)
  - waiting threads, sleeping (no cpu consumption)
  - natural barrier: bandwidth/latency
- Computational – not recommended
  - scheduler overload
  - easy full system utilize
  - dangerous for one core systems

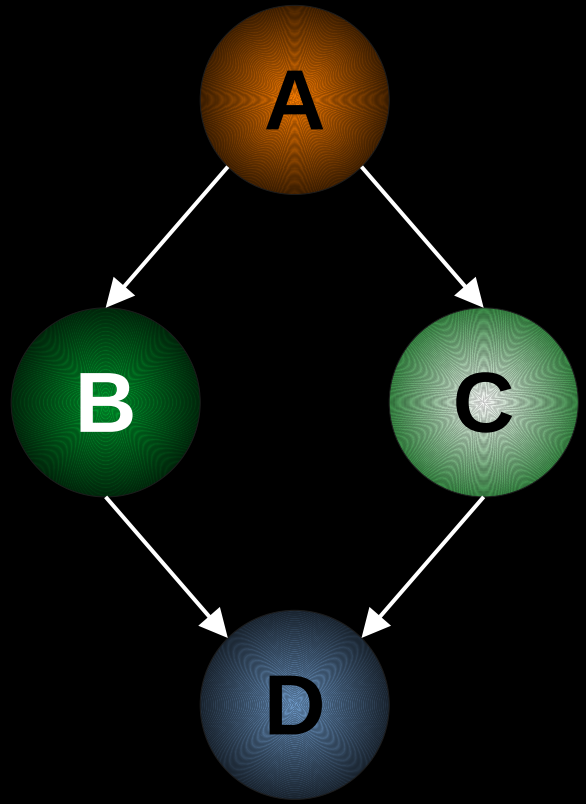
std::async graph?

- std::luanch::deferred



- std::launch::async





async_graph.cpp

std::async – in practice

- Pros:
  - Simple
  - Work controller
  - Containerization possible (via future)
  - Simple async synchronization
  - Easy progress control (future)
- Cons:
  - Difficult intra-dependency
  - Start control? … no...
  - Implicit dependencies between executions
  - Imprecise progress control
  - Easy oversubscription