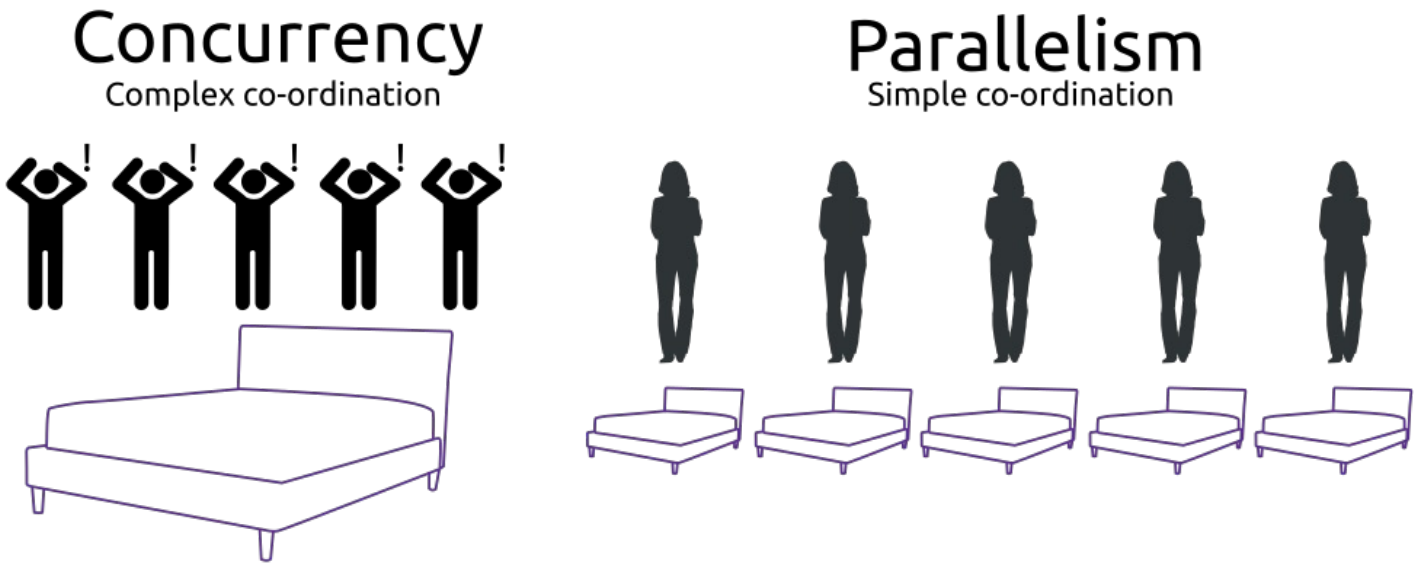
并行和并发

并发（concurrent）

- 起源于时分复用，可以在单位时间在单位CPU处理更多的任务，满足
  - 主要如何有效利用任务存在的不同占空比，增加CPU利用率
  - 任务QoS的需求，高优先级任务需求优先满足，RTOS的确定性

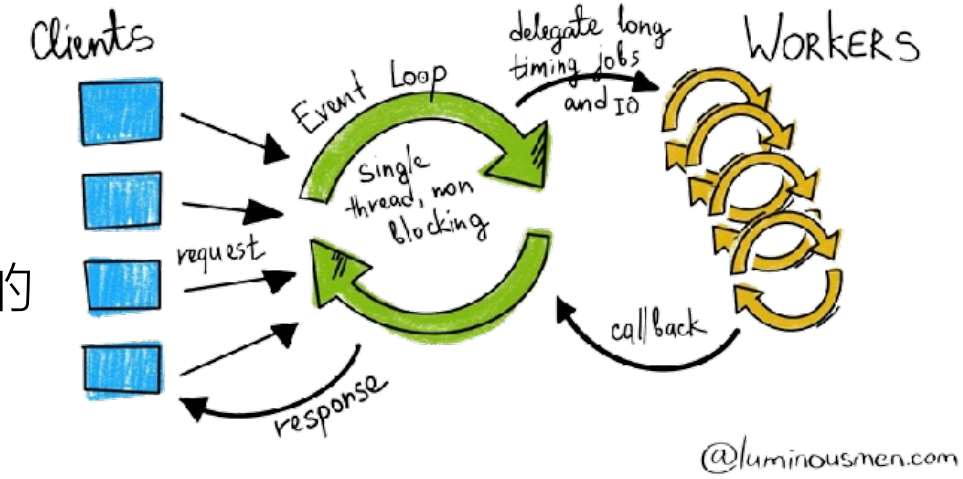
并行（parallelism）

- 起源于集群（clustering），HPC，多核处理器，网络处理器，通过投入更多的处理资源，提高单位时间的处理能力和吞吐能力
  - 数据可并行，data parallism，SIMD
  - Stateless，无状态无耦合，可以独立运行更多的程序



异步（async）

- 起源于为了利用任务，尤其是I/O任务请求和响应中间的IDLE时间，来把不同任务有效复用在一起
- 典型应用：慢速的I/O设备
  - web服务器，HTTP处理，文件系统，C10K 并发处理
  - UI前端设计，用户时间触发UI重绘，慢速网络下载
  - 异构处理器加速器调度
- 是并行和并发的一种实现方式，解决如何用写串行程序的方法写异步应用
  - 基于event-driven，nodejs，aysnc/await/future
  - 调度的对象是thread, corouting，closure
    - Apache HTTPD MPM, threading pool, select/epoll,
    - GCD 改进epoll/select，调度的还是thread
    - Nginx, nodejs event-driven, coroutine,



移动场景对并行和并发的需求

- 提升CPU使用率，面积换性能
  - 大核性能高，但是功耗也高，为传统串行应用开发
  - 小核性能低，但是可以通过增加核数适配应用计算需求
  - 传统SMP多核，为串行同构应用开发，受制于cache一致性，内存交换带宽，难以做大，
  - 未来chiplet scale，为并行异构应用开发，通过并行并发框架达到计算和内存使用的区域化（sharding）
- 优化功耗
  - 并行并发的任务调度机制可以按照任务的QoS需求提供最适合的计算资源，例如：大小核，降速，内存带宽等，优化计算效率，例如：M1的功耗远低于Intel，因为有大小核（icestrom，firestorm）调度，background任务跑在小核上，还可以降速

移动OS的并发并行软件框架

- 并发并行异步软件框架是充分发挥移动OS场景下多核多线程硬件能力的关键软件技术，重点在让开发者容易、安全地使用多核和多线程硬件，提升用户体验的同时提升硬件使用效率，包括功耗，硬件利用率
- 编程语言是移动OS的最核心的技术，并行并发框架是编程语言生态的最重要部分，移动OS的主要对标对象都有基于其语言+OS实现的非常完备的软件并发并行框架，而且在不断演进
  - iOS，Object-C GCD =》Swift GCD => Swift Async
  - Android , Java Multi-threading => Kotlin Async
  - Web/HTML5, JS async/await, React Concurrent Mode
- 并行并发框架的发展趋势
  - 从OS提供的线程抽象和管理调度演变为语言和Runtime为主的并行并发能力抽象和调度
  - 并行并发异步软件框架在不同场景出现了占据主导地位的编程范式，Goroutine，Tokio in Rust
  - 硬件配合这些范式提供加速配合能力？传统的SMP架构演变为松耦合的massive core 架构？Chiplet？
- Rust 语言为并行和并发提供了核心能力 fearless concurrent（无畏的并发）
  - Zero-cost abstraction高性能，靠静态编译解决问题， 最小化runtime，核心语言+library生态，语言仅仅提供最核心的语义支持，生态实现并行并发library，优胜劣汰
  - 开放治理结构，社区创新，不存在swift，go，kotlin社区被独家掌控
  - 系统编程语言，可以实现从firmware（SBI）+ OS + Runtime + library + language全栈优化

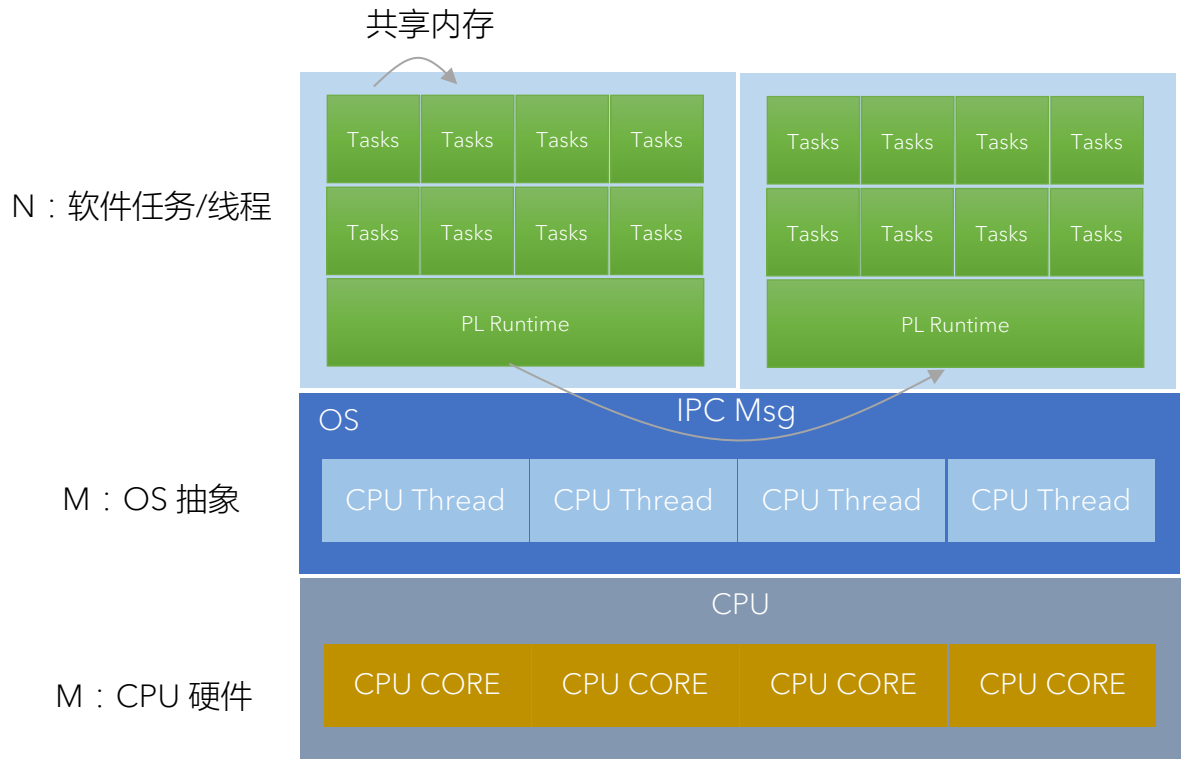
移动OS 并行并发场景和生态

- 前端UI
  - 慢速I/O：多点触屏，鼠标，键盘
  - Pattern: MVC, model view controller; ECS – Entity Component System；定义输入=》状态修改 =〉UI重绘流程
  - Declarative UI, React, swiftUI
  - Async编程方式, React Concurrent, GCD
- 2D图像渲染
  - 基于retained mode，大部分时间只有小部分区域需要重新渲染
  - 不同的windows/layer单独线程渲染，GPU composition，不需要redraw every pixel
  - Async graphics API，Vulcan/Metal/WebGPU
  - Servos实现的并行并发向量图，字体渲染
- 3D 渲染
  - immediate mode，per pixel rendering@60fps
  - 主要取决于GPU的并行并发实现
  - 物理模型计算加速，Raytracing，PBR
  - Cloud Gaming 是另一条思路（原神 90%）
- I/O后台任务
  - 网络，文件系统，Sensor 数据读取：Tokio
  - Database：GraphQL
- 计算密集型任务
  - 视频，音频编解码，AI/ML模型，物理引擎计算，图像/照片处理
  - 异构处理器 ISP/DSP/GPU/NPU加速
  - 声明式并行并发计算框架：TVM，ONNX，OpenCL/SYSCL, RenderScript, TaskGraph

并行和并发的实现痛点

- 工具/方法论问题
  - 问题的形式化描述 =》代码实现 =〉不同逻辑执行链的触发
  - Edward Lee “The Problem with Threads”，经过专业工具检查和专家开发的代码，4年后发生deadlock问题
- 并发并行编程问题
  - 传统的并行并发元语：mutex，semaphore，messaging，thread-safe library，lock-free data structure
  - 开发者需要有数据、算法、控制流的并行专业知识，而且难以形式化证明， 难以调试
  - 普通开发者因为缺乏语言和工具支持，大量的代码可以并行化但是写成了串行
  - 大量存量系统代码是单CPU时代开发的，是串行模式的

并行和并发



- OS对CPU做1:1抽象，把CPU核/线程映射为CPU线程资源
- 不同用户的任务通过OS进程隔离，进程通过IPC消息通讯，通过OS调度策略来管理不同用户的任务
- 编程语言提供面向用户的task概念，task共享堆栈和内存空间，PL runtime负责用户task的调度
- 通过用户进程可以实现并发并行，避免data race，例如：nodejs，python，但是受制于IPC通信的开销和OS调度策略，适合无状态的任务并发，例如：web服务器
- 语言runtim提供task级别的并行和并发，从OS获得thread pool，通过自己的任务调度策略和内存共享机制，实现task的并行和并发，因为任务共享内存空间和应用context，任务可以进行高带宽通信，任务切换代价低

问题本质是N个任务负载影射到M资源的调度问题

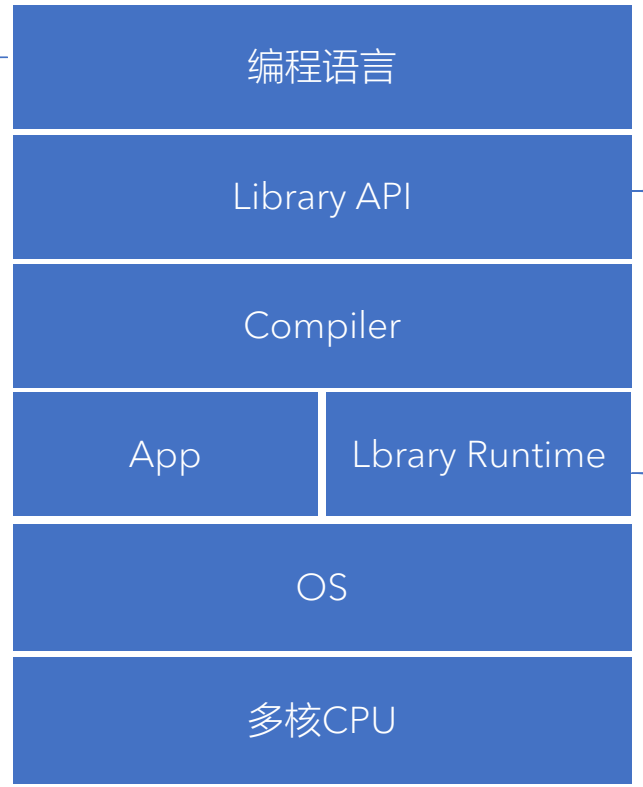
- 任务的颗粒度（时间、workload）越小越容易增加调度的机会，fine grained，类似FaaS的理念
- 任务可随时打断越容易增加调度的机会，stateless
- 任务的QoS越明确，越容易实现最佳资源利用率，可以减少消峰填谷

全栈解决并行和并发问题

人机交互界面，描述问题的表达能力，内置设计约束，例如：Rust的内存安全检查，python/js 的单进程，erlang的function programing实现并发并行。现代语言都内置mutex，async，channel等支持并行并发的语义。

编译器，主要衡量指标是指令翻译的效率。编译器IR可以实现数据流和控制流优化，实现对并行并发语法糖的code gen，例如：把future翻译为state machine

OS实现对CPU核的抽象，提供thread软件对象和对thread的管理调度能力，提供良好的thread context隔离，同时带来context switch的开销；内核是同步实现，设备驱动也是同步实现。



针对领域问题（domain）提供扩展语言能力的API，强化设计模式，大大简化面向领域问题的应用设计难度，是语言成功的关键。

Runtime链接到用户应用，并行并发的runtime具备

- OS向Runtime提供线程池，软件task由Runtime全权管理
- 提供任务生命周期管理，任务调度，任务间通信等核心能力
- 提供async的内核API，例如：网络、文件系统
- 提供高性能的异步并行库，例如：http库
- 提供串并转化能力

Runtime接管了OS的并发并行的调度管理能力，对thread实现了虚拟化，N：M model提升CPU利用率，改善功耗，增加应用性能（UI顺滑，网络吞吐量，计算负载）

为什么PL对并行并发能力重要？

- 大量的代码可以写成并发并行，主要取决于开发者，代码一旦写成了串行，很难并行化，因为用户的问题的逻辑难以传递给下游的工具-runtime，compiler
- 要解决问题，需要给用户好的工具，让用户更好的传递要解决的问题，而不是解决问题的方法，所以声明式编程可以优化的空间大于imperative方式，但是存在取舍，基于机器的优化还难以完全取代开发者，例如TF vs Pytorch，UML自动编程，某些领域难以抽象，例如：用户逻辑，还是需要用户写
- Human in the loop 是必须的，精英工程师解决困难的问题，变为library和runtime，一般开发者使用声明式并行并发框架
- 尊重domain问题的生态，重要的domain都有激烈的竞争，in house方案最大的弱点在生态

典型语言并发并行比较

	GCD	Goroutine	Rust
设计需求	POSIX thread用户体验差，多核处理器商用，需要简化并发并行开发，以提升用户体验，提升桌面和移动处理器处理效率，提升UI为核心的用户体验 在Obj C增加Block（closure），替换Kernel的select为Kqueue，Pthread为Pthread-Workqueue	需要给C程序员Python的开发体验，C/C++并行并发开发能力工具落后 主要面向云计算后端业务，网络等高吞吐量业务 声明式的并行并发编写方法，由精英工程师优化runtime	C/C++在多核处理器上开发并行和浏览器内核易出错，需要同时保证安全和性能，语言设计内存onwershhip和borrow checker可以安全使用内存，语言设计内置并行和并发的能力，不安全的用法无无法通过编译，存在known unsafe，可以通过精英工程师编程、反复测试来解决，不存在unkown unsafe场景，的通过library机制来支持不同的高层并行并发框架，社区竞争
Memory sharing	Yes	Sharing by communicating，Channel是主推的模式	语言核心lib提供基础channe和mutex，生态实现不同的设计模式
语言层面声明式的串并转化	NO	Go	通过生态library提供，例如：Rayon
Task queue w/ QoS	5 levels	N/A	可以在并发并行库支持
社区	libgcd是swift核心库，Apple 把控	Golang Google把控	开放治理，语言支持ATOMIC，Send/Sync Traits等并发并行机制，社区库支持Rayon声明式并发和Crossbeam并行算法库实现

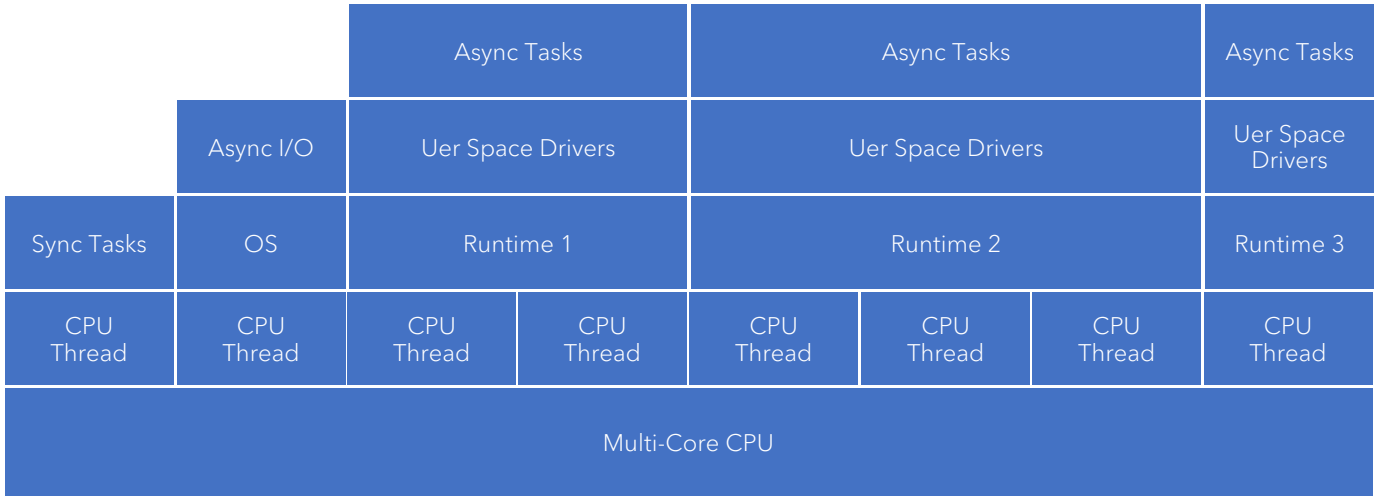
GCD

- GCD初衷
  - 解决pthread用户体验差，难以使用，为此objc加入block，实现closure
  - GCD使用Kqueue和Pthread-Workqueue替换select/poll和Pthread，提高多线程性能
  - Swift之后重写，成为corelibrary
- GCD 成果
  - UI线程优先级最高，Preemmptive threading model，响应速度快，用户体验好
  - 基于QoS的调度策略，可以把background任务调度到小核，降速跑
- GCD问题
  - Async难以使用，objc是80年代语言，swift 封闭社区，难以坚持设计原则，Chirs Lattner离开，例如：rust zero-overhead abstraction，only pay for what you use
  - 使用场景不明确，async不适合所有的多任务，但是Apple坚持async，一旦call chain内有一个async，所有的function都必须是async
  - 性能下降，数据同步的开销大，context switch开销大，使用锁机制，改成sync后，更好维护，性能更高
- GCD 逐步会被Swift async/await替换
  - "GCD has been mostly replaced by Swift's new built-in async API as of WWDC 2021"

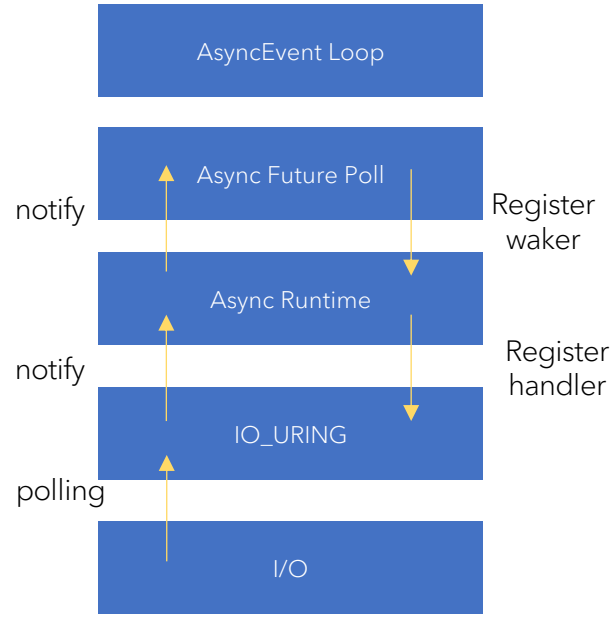
关键趋势：全栈异步化的并发并行框架

用户态Async Task模式

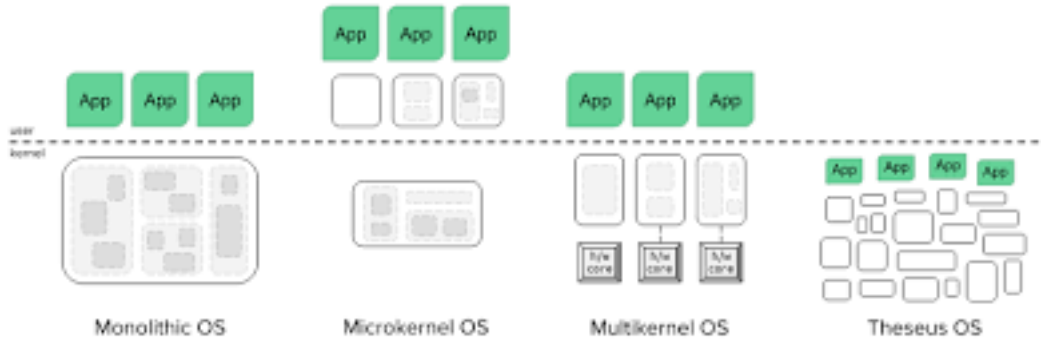
- 传统的OS的multi-threading设计 context switching (5-7us)，threading pooling 代价太高，现代硬件处理能力很强，使用coroutine、green task等软件线程，和异步kernel API，结合可信语言（context switch也是隔离的需求），可以达到更高的处理效率
- async/awaits 使用逐渐标准化，成为前端并发编程标准，JS/Rust/Kotlin/Swift
- Task只有100s byte 内存，采用cooperative scheduling，不需要保护context，开销小
- 可以mapping到多个OS/硬件线程，可以按照应用需求创建cooperative或者preemptive scheduling
- 可以强化各种内存共享方式，例如：channel，减少锁的使用
- 用户态创新，domain specific async library

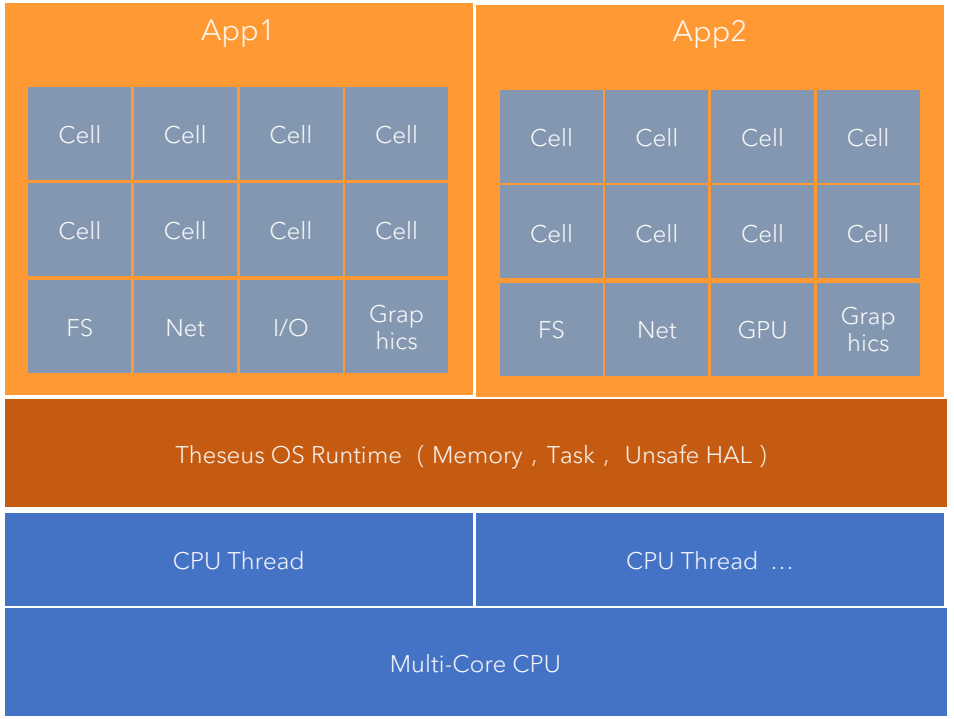


- IO_URING加入内核，使得async I/O使用更加方便，基于此大量设备驱动重写为async方式
- 设备中断代价太高，基于IO-URING实现高效率的polling
- eBPF提供了内核编程能力，可以进一步在内核实现用户逻辑
- Async function的waker可以直接注册到内核



Theseus OS: Zero-cost Isolation





Theseus OS 实现 Single Address Space和Single Priviledge Level的新型OS

- 基于Rust Compiler的内存ownership能力，保证基于crate构建的App实现内存隔离
- 以crate为基础的基本运算单元cell，可以不停机动态加载和升级，对关键业务领域非常有价值
- 用户态的驱动程序，LibOS
- Theseus Runtime实现内存的管理和任务调度
- 通过WASM集成第三方库和App

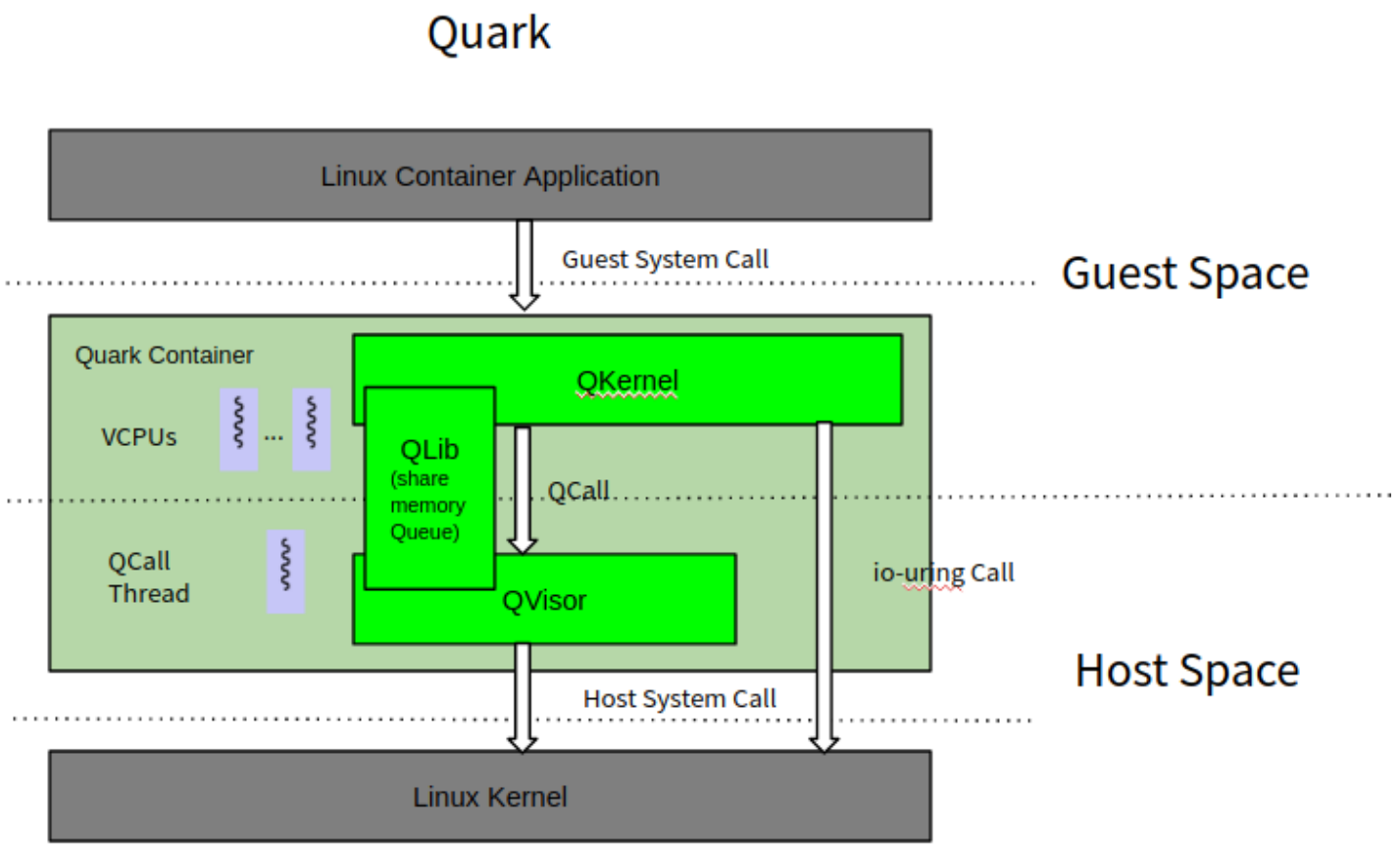
优势

- 性能，zero-copy内存使用，不存在kernel和user space context switch
- 安全，编译器保证不存在内存越界访问

挑战

- 从头构建OS生态，大量硬件，存量驱动
- 应用需要使用Rust编写（WASM可以缓解）

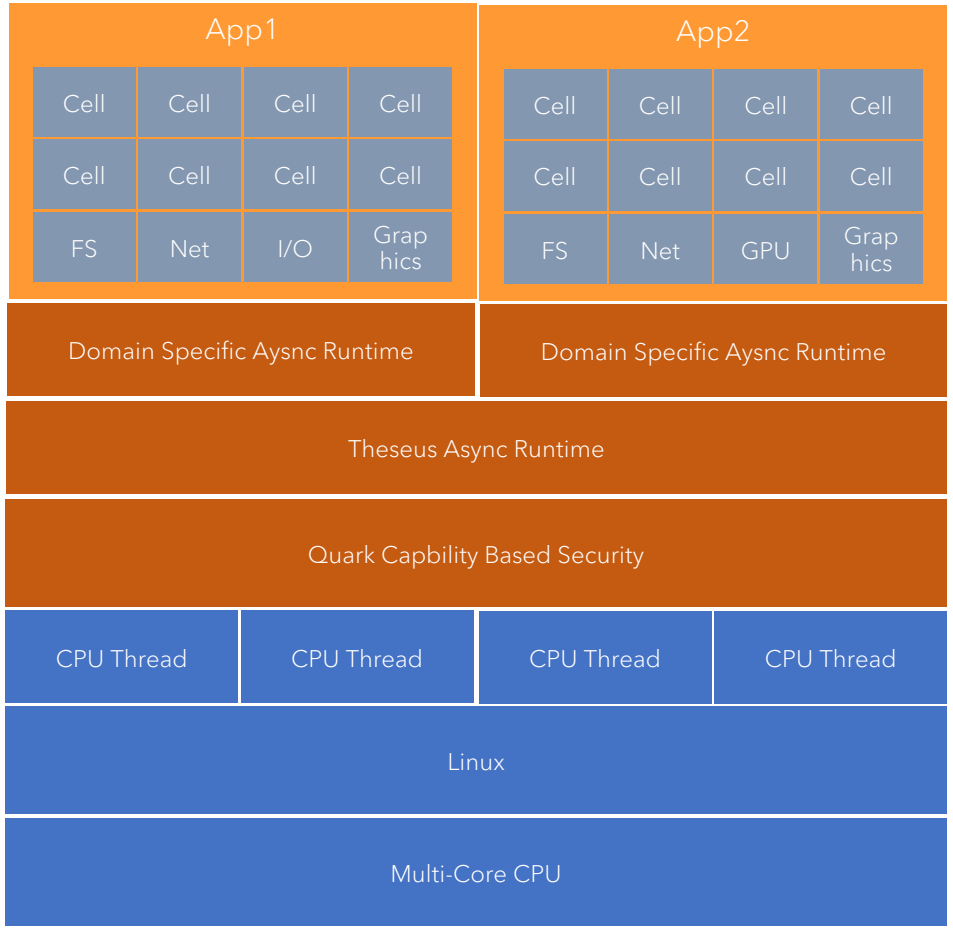
Quark 基于Rust实现的I/O虚拟化



- 场景和gVisor，Runc一样，解决容器（广义，可以是docker container，也可以是WASM）直接访问OS API带来的潜在安全性，在container和OS之间构筑一个隔离层
- 这个层面可以proxy API访问，而且可以实施各种安全策略，例如：capability based
- Quark 利用io-uring实现了全异步的I/O访问，只有设备open/close需要管理，数据传输是pass through模式

https://github.com/QuarkContainer/Quark

Async App Runtime for Mobile OS



Theseus Async Runtime

- 可以对实现App分组，对组内App实现统一的管理策略，例如：capability
- 实现了基于crates的细粒度生命周期管理策略
- 可以集成不同的asyc 框架，实现对不同use case的支持
- 用户态Async驱动和协议栈，配合IO_URING实现全异步访问
- 集成Quark实现crate对host OS API访问的隔离和基于capability的管理

Domain Specific Aysnc Runtime

- 基于use case生态产生的winning 框架，例如：tokio在网络领域，yew在前端
- 因为Rust良好的composable的能力（traits，crate），可以最大限度利用
- 差异化针对问题的易用性，Runtime调度策略等

Theseus Async Runtime vs Linux

	内存管理	进程管理	设备驱动	I/O虚拟化
Linux	进程隔离	进程/线程调度	内核态	Namespace cgroup
Thesueus OS	Rust静态分配	单一线程池 基于Cell管理	用户态library	用户态library
Domain Specific Aysnc Runtime		Runtime调度管理	用户态library	用户态library
Quark				capability的I/O security

并发并行软件框架和硬件co-design

- PL的并行和并发能力和PL的设计强相关
  - 语义层面，并发并行库生态
  - 受制于PL的治理模式，go – google， swift – apple，C# - Microsoft，都有自己的核心应用场景
- 硬件设计需要PL设计配合
  - 并发并行的软件生态重构硬件架构，SMP模式转为AMP模式，scaling up 转变为 scale out，云原生模式进入芯片
  - 原来在OS配合，但是趋势是user space driver，runtime做厚，OS变薄，例如Intel的用户态中断可以减少IPC发送的overhead，Rust可以安全共享内存
  - PIE理论，硬件提供performance，软件提供isolation和efficiency
- 为什么选择Rust
  - 下一代系统编程语言，即将进入Linux Kernel，主要用于写基础软件
  - zero 理念和模块化构建支持系统编程，zero-overhead abstraction，zero-overhead isolation
  - 最新的软件技术的成果，语言设计+编译器+丰富的生态
  - 开放治理模式，Don’t pick up winner