Fuchsia介绍

许中兴
重德智能
xzx@droid.ac.cn

Fuchsia的基本情况

- 多年的Android, ChromeOS开发经验一方面让Google在操作系统方面积累了足够多的人才和组件，另一方面也充分认识到了Linux kernel很多的局限性

- Fuchsia是一个全新的操作系统的统称。Google挑选了一系列它认为合适的技术和组件进入这个操作系统，比如：微内核，基于能力的访问控制，Vulkan图形接口，3D桌面渲染Scenic，Flutter应用开发框架。目前支持的编程语言是：C/C++, Go, Rust, Dart

- Google2016年中放出了所有的代码，但是没有正式宣布这个项目的目标，开发社区目前有一个IRC频道进行交流

- 支持的架构是X86-64和ARM 64，支持的设备从IoT到服务器

Google为什么要从头设计全新的操作系统

- 为什么不在Android上改进，塞入各种新技术

- https://spectrum.ieee.org/aerospace/aviation/how-the-boeing-737-max-disaster-looks-to-a-software-developer

- 737的问题：机翼下空间太小，放不下大引擎。改动机翼相当于重新设计一架飞机

- 所以，改变发动机位置强行放进去，后果是改变了空气动力学特性，机头容易上扬

- 解决方案是通过修改飞控软件来弥补，飞控认为飞机在上扬，强行下压，造成失控

- 当总体设计出现问题的时候，用再多的技巧去弥补，也只会造成最终的灾难

现代通用、开放OS需要面对的方面

- 上游硬件厂商
- 下游应用开发者
- 设备友商
- 用户
- 黑客

Fuchsia解决现代OS痛点

- **原生进程沙箱，解决应用安全和分发问题（黑客）**
  - Linux: namespace, control group, unionfs => docker
- 稳定的驱动接口，硬件厂商可独立维护硬件驱动（硬件）
- 系统模块化，分层，设备厂商可以灵活定制专有系统（友商）
- 基于Vulkan和物理渲染的纯3D UI，全局光照（用户）
- Flutter应用开发框架（开发者）

Fuchsia重新思考四个Unix的基础抽象机制

- 全局文件系统
- 用户
- 进程的创建
- 系统调用

全局文件系统

- 在Unix里，存在一个全局的根文件系统
  - 它是每个进程共享的基础资源
  - 文件系统涵盖了非文件资源：/proc, /sys, ..
    - 网络是例外
- 在Fuchsia里，没有全局根文件系统
  - 文件和文件系统成为一个局部概念（局限在每个文件系统进程里），从而在进程内核数据结构里没有file
  - 用namespace来定义一个进程能够访问的资源
    - 每个name（路径）对应一个资源进程channel 的handle
  - “/“ -> root vfs service handle, “/dev” -> dev fs service handle, “/net/dns” -> DNS service handle

User

- 在Unix中，user本来是用作不同的用户登录共享服务器的机制
  - user是真正的用户
  - 后来主要用作权限控制，弱化的沙箱机制
- 在Fuchsia中，在底层(Zircon, Garnet)没有用户的概念
  - 用namespace来控制进程能够访问的资源
  - Capability-based access control
  - 从而在进程里没有uid

进程的创建

- 在Unix中，新的进程由老的进程fork而来
  - 新的进程继承父进程的全部资源
  - 一种偷懒的设计
    - Andrew Baumann, A fork() in the road, ACM Hot Topics in OS, May 2019
- 在Fuchsia中，新进程的创建需要从头开始
  - 创建process, thread
  - 父进程建立初始的namespace到资源channel handle的映射
  - 调用process_start显式的告诉内核新的进程可以跑了
- 在Fuchsia内核的process数据结构里，没有file和uid

系统调用

- Unix/Linux里，通过中断调用内核服务： int 0x80, syscall, sysenter系统调用的方式是确定的，直接的
  - 内核接口不能变
  - 可以被任意注入的代码调用
- Zircon里系统调用通过vDSO进行，意图是防止用户代码直接通过固定的中断代码调用system call，达到内核详细接口的隔离。保持 C层面的接口稳定：名字+参数。而不是内核入口汇编指令层面的稳定。
  - 注入的代码无法直接调用 vDSO里的接口，虽然加载地址固定，但是计算出入口地址很难，如果不是不可能的话
  - 内核会验证调用指令的地址，而vDSO的加载地址是固定的。并且在编译的时候会验证有限的入口符号，这些符号在编译时唯一生成，防止用户进程绕过vDSO
  - 这里主要的目的是隔离system call的调用方式，不是绝对意义上的不可注入调用
  - https://fuchsia.googlesource.com/zircon/+/master/docs/vdso.md

仿佛是专门针对漏洞利用作出的设计

- 典型的漏洞利用步骤
  - 通过系统调用fork()/exec()直接创建反向shell
  - 继承uid(或者通过获得root uid进行提权)获得泛在授权
  - 访问全局文件系统
- 在Fuchsia里，以上机制全都不存在
  - 没有固定的系统调用入口，必须动态链接
  - 创建进程时显式建立root namespace
  - 没有user，从而没有ambient authority (DAC/MAC)
  - Capability-based access control
    - 能访问的资源是父进程赋予的namespace
    - 看不到初始namespace之外的任何资源

Kernel的本质是什么

Kernel的本质不是：

- 管理硬件
- 执行特权指令
- 引导启动过程
- 处理中断

Kernel的本质是：地址空间切换

- 不同进程唯一共享内存地址空间的场合
- 切换地址空间是进入内核的标志
- 不同的进程通过共享内核地址空间来交换信息
- 切换地址空间是切换进程的关键步骤

Zircon主要内核态功能

- 虚拟内存和物理内存管理
  - vmo: virtual memory object: 包含物理页
- 进程和线程管理
- 进程间通信
  - channel

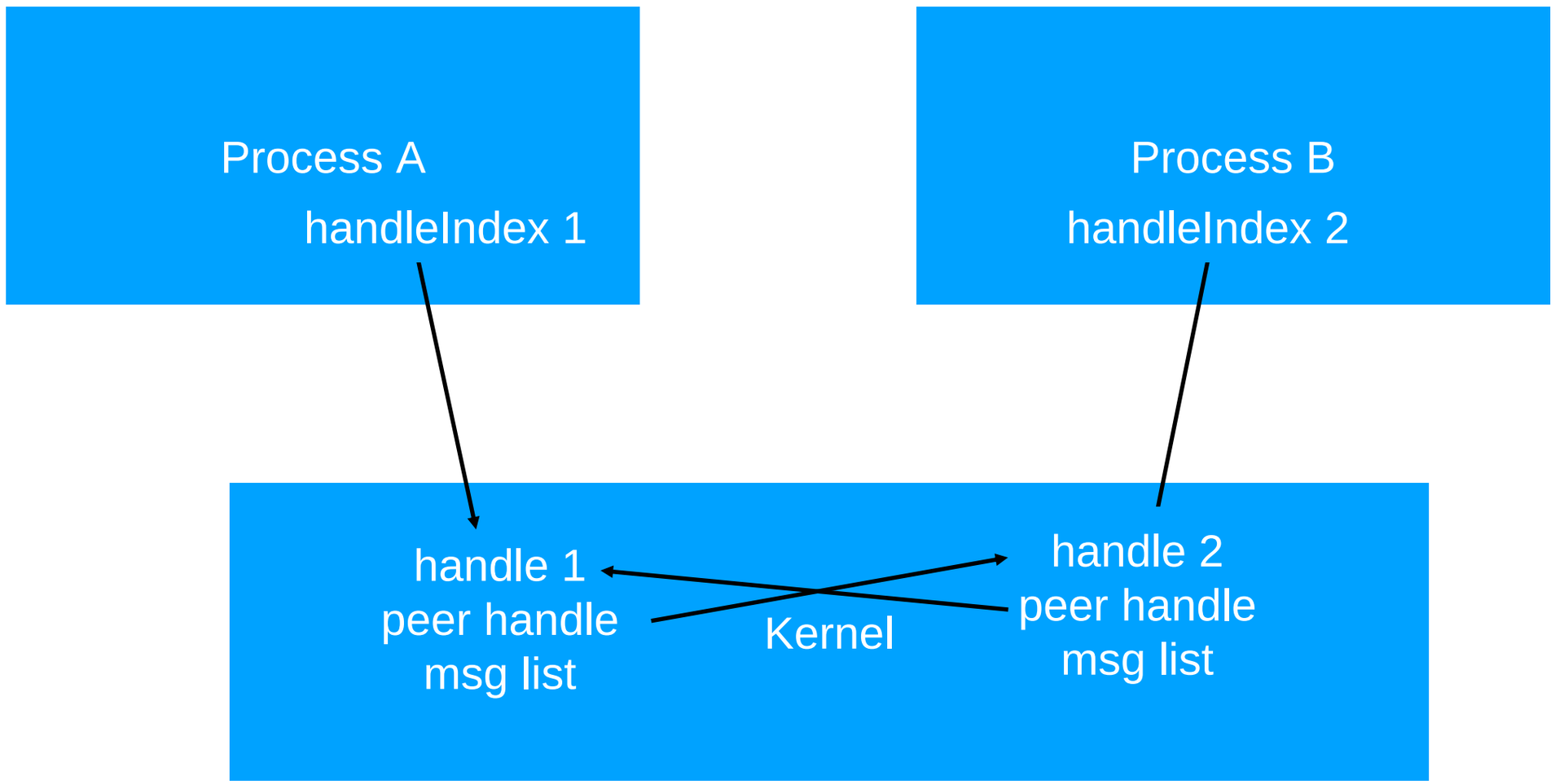
以内存为中心的设计

- Virtual memory object (vmo)
  - 代表一个内存对象，懒分配
- vmo通过向channel发送handle在进程之间传递
  - 进程拿到vmo handle,把vmo重新映射到自己的地址空间里
- Unix是以文件为中心的设计

channel

- channel是进程间通信的（唯一）机制
- 一个channel有2个handle，h1, h2，从一头写入消息，从另一头读出消息
- 一个进程在创建时有一些初始channel handle
- 要与一个服务x建立通信，进程创建一个channel，自己拿h1，把h2通过已有的channel(root_svc)发送给相应的服务，服务拿到h2，将其放到自己的事件监听循环里
  - 示意api：connectToService(root_svc, “x”, h2)
  - 比如open()，在linux里会在进程的内核数据结构里增加打开的文件描述符，不涉及到其他进程；在fuchsia里则是创建一个channel, 把远端发送给相应的服务，建立通信通道
- channel_write()把消息写到另一个进程能看到的地方。进程间是不共享内存地址空间的。只有**内核的地址空间是进程共享的**。所以channel_write()必须是一个系统调用，切换到内核地址空间里进行消息写入。一旦切换到内核地址空间里，就能看到另一个handle了。写到那个handle的消息队列里，等另外一个进程切换到内核地址空间里，就能看到消息了。

channel的实现


Process A
handleIndex 1
Process B
handleIndex 2
handle 1
peer handle
msg list
Kernel
handle 2
peer handle
msg list

系统调用vDSO

- 内核映像还嵌入了一个vDSO，包含了系统调用入口
- 这个vDSO被映射到每个进程的内存地址空间里
- 它本身是ELF shared object文件格式，但是又不是以文件形态存在，所以叫做vDSO
- Linux kernel也用这种方式实现了一些简单的系统调用，比如getdaytime()。但是Zircon并不是为了避免切换内核态，而是把系统调用的接口用vDSO进行隔离。

vDSO的好处

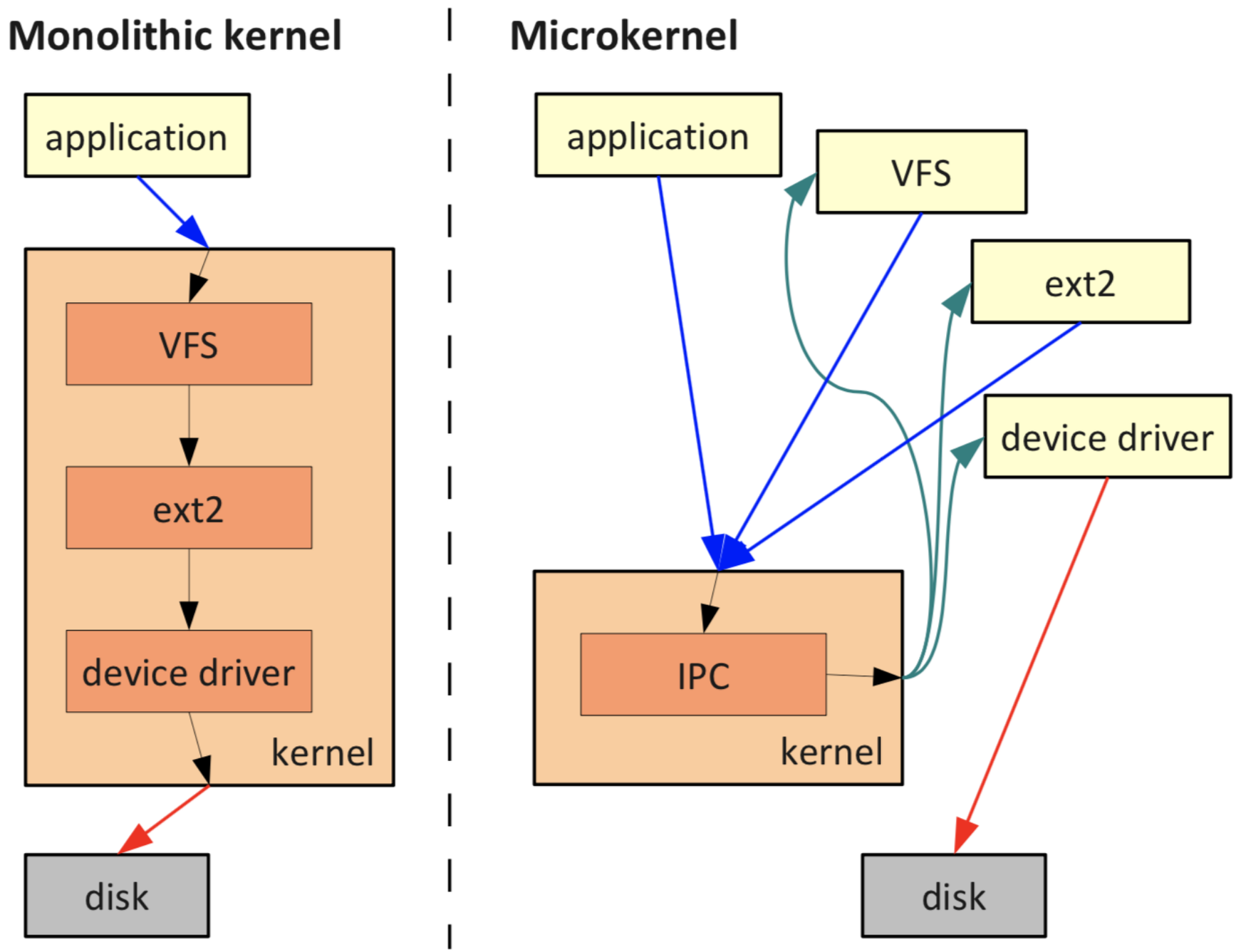
vDSO 最大的优势在于保证二进制接口（ABI）兼容性的同时，不干扰升级系统调用的实现。Linux 的系统调用 ABI 是基于处理器中断的，每个系统调用都有自己的系统调用编号，这么多年来除了 syscall/sysenter 并没有本质上的升级。这是因为相当多的预编译好的程序会不管三七二十一去调用中断。一旦 Linux 不再支持基于中断的系统调用接口，那这些现有的应用程序除非返回源代码重新编译，否则无法正常运行。Fuchsia 非常重视 ABI 兼容性，但同时也不希望放弃将来优化系统调用的机会。这类问题可以通过多加一层间接调用来解决（Fundamental theorem of software engineering）。

Fuchsia 里所有的系统调用都必须由 vDSO 发起。用户代码要想调用系统函数就要先动态链接上 vDSO，再呼叫 vDSO 中的某个函数，再由那个函数去调用系统。这样一来用户程序跟内核接口完全解耦，下一版、Fuchsia 无论是要改变系统调用的寄存器顺序，还是改换效率更高的调用约定，都可以通过更新 vDSO 来无缝兼容现有的一切应用。

另外，在 Linux 里遇到缓冲区溢出 bug，攻击者能够注射一段调用系统函数的代码，从而拿到 shell。但是在 Fuchsia 里因为系统调用来源强制限制为 vDSO，而 vDSO 所在的内存页面永远是只读页面，注入的代码并不能调用任何系统函数，形成了 0-day 漏洞前方的最后一道防线。(内核会检查入口指令的地址，而vDSO映射的地址是固定的)

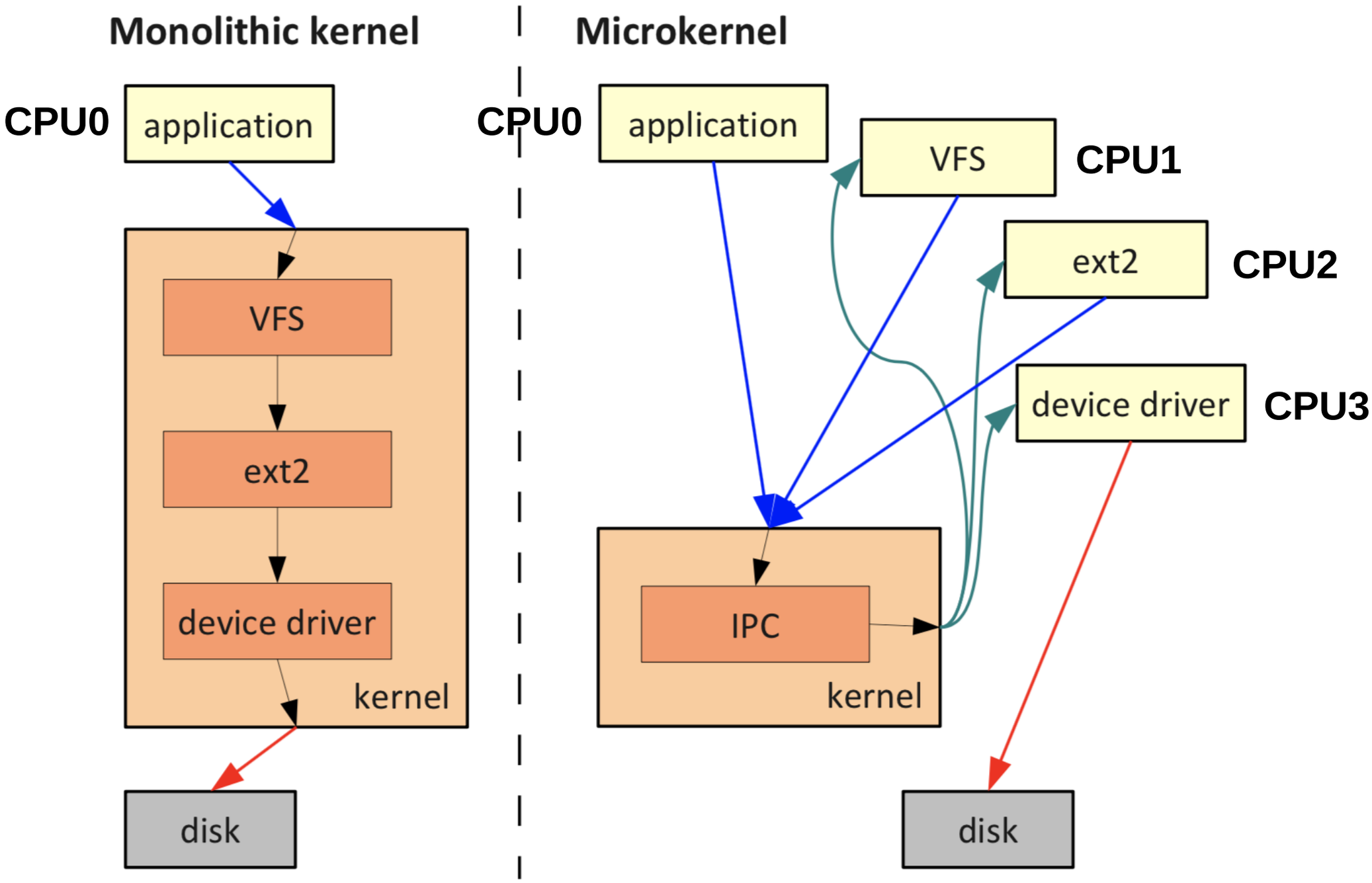
微内核

- Linus vs Tanenbaum的论战
  - Tanenbaum: Linux是七十年代的技术。在1991年写宏内核是错误的。争论早就结束了，微内核已经赢了。我是教授，Minix只是我的hobby，所以别拿Minix说事。
  - Linus: Linux比你写的Minix强多了。微内核只是你们学术界的玩具，我看过所有的关于微内核效率优化的论文，它们实际上只是在重复宏内核早就用过的技巧
- Mach, Hurd
- Performance overhead
  - Context switching (user space <=> kernel space)
  - Thread scheduling


Monolithic kernel
application
VFS
ext2
device driver
kernel
disk
Microkernel
application
VFS
ext2
device driver
IPC
kernel
disk

Overhead: single core case

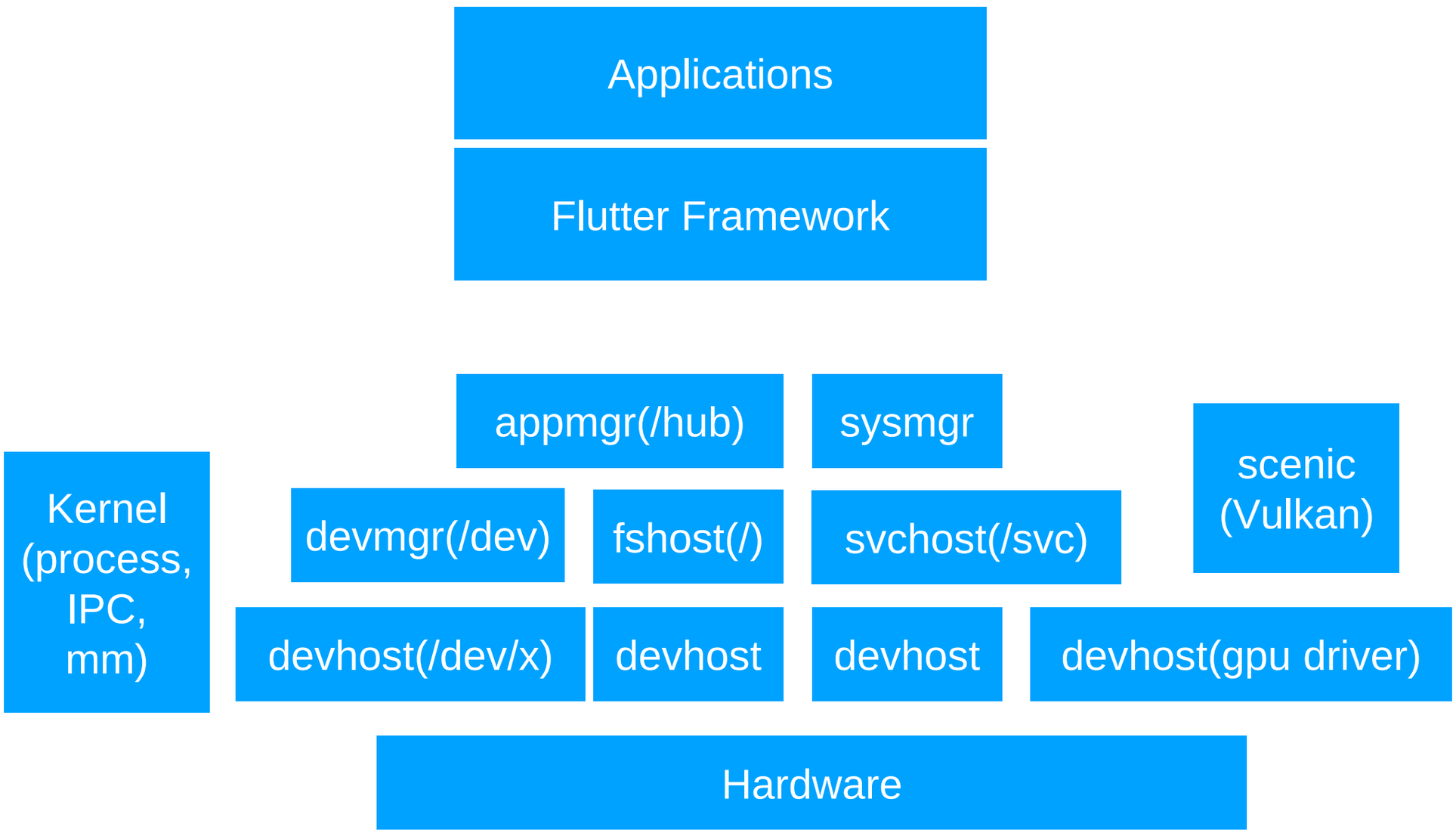
- Monolithic
  - CPU0
    - Context switching: 2
- Micro
  - CPU0
    - Context switching: 8
    - Thread scheduling: 4


Monolithic kernel
CPU0
application
VFS
ext2
device driver
kernel
disk
Microkernel
CPU0
application
VFS
CPU1
ext2
CPU2
device driver
CPU3
IPC
kernel
disk

Overhead: multicore case

- Monolithic
  - CPU0
    - Context switching: 2
- Micro
  - CPU0
    - Context switching: 2
  - CPU1
    - Context switching: 2
  - CPU2
    - Context switching: 2
  - CPU3
    - Context switching: 2

Fuchsia架构


Applications
Flutter Framework
Kernel (process, IPC, mm)
appmgr(/hub)
sysmgr
scenic (Vulkan)
devmgr(/dev)
fshost(/)
svchost(/svc)
devhost(/dev/x)
devhost
devhost
devhost(gpu driver)
Hardware

Fuchsia在各个平台上的可能的优势

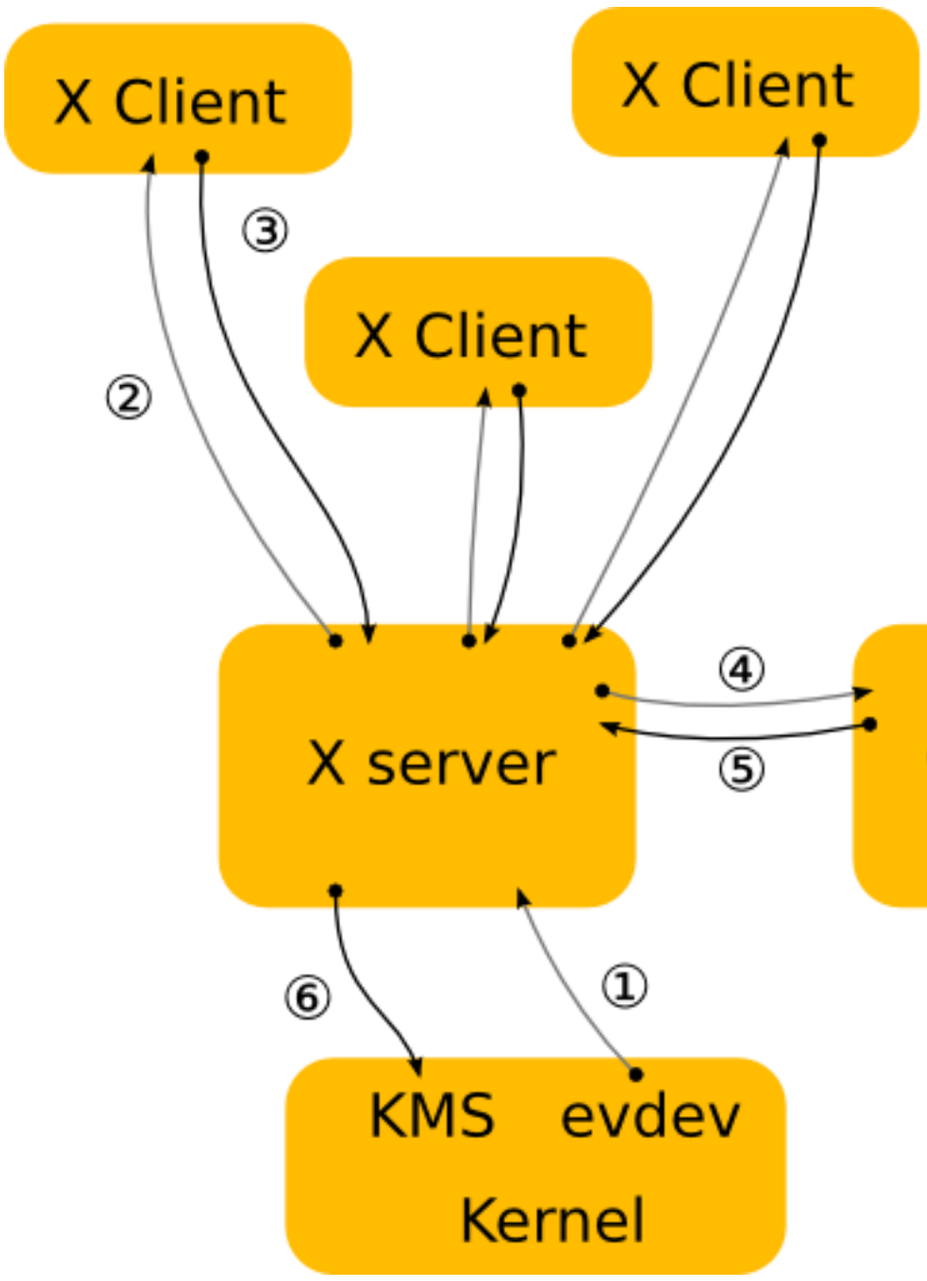
- 在服务器平台上，原生的进程沙箱机制将带来新的安全特性和容器机制

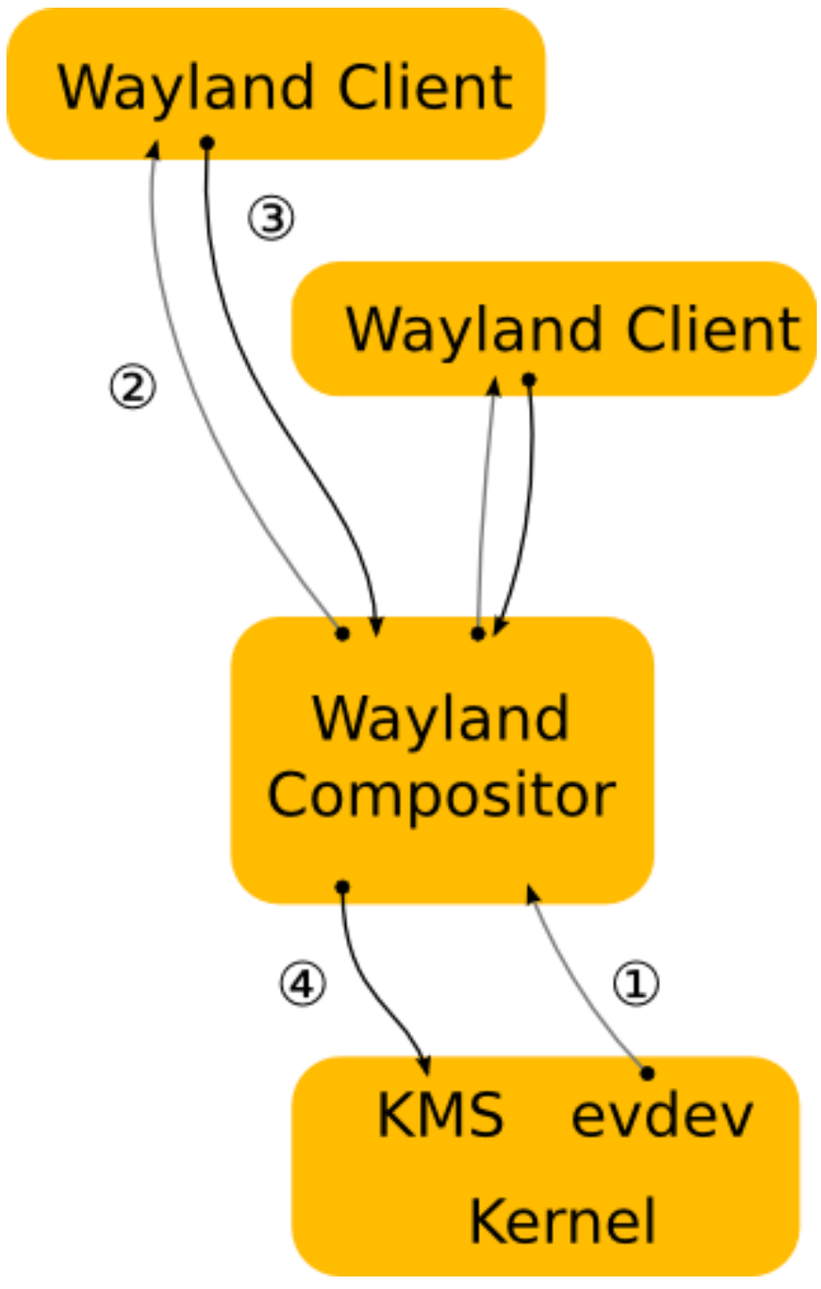
- 在桌面平台上，类似于游戏3D引擎pipeline的图形栈以及毫无遗产负担的实现将使电子娱乐应用变得更为高效；无缝兼容庞大的Android生态

- 在移动平台上，系统的模块化方便第三方设备厂商的全面定制，驱动框架方便硬件厂商编写和维护私有驱动

Linux桌面的现状

- Ubuntu 18.10很精致，X + OpenGL/Vulkan + QT比较稳定可靠了
- X Window 诞生于1984年，越来越多的功能被加入到X stack里
- 随着技术的发展，越来越多的功能从X中移到内核或者是其他专门的库中：kms, cairo, opengl/vulkan DRI2, fontconfig/freetype
- X的问题：window manager也作为一个client，导致效率低，容易出bug，如果不遵循ICCCM的话
- Wayland 是一个融合了compositor的display server, 去掉了X里的过时功能，比如绘制
- 但是Nvidia和其余厂商还没有就Wayland里的buffer管理达成一致，导致nv还不支持wayland
- 注：在X里，OpenGL/Vulkan是嵌在window里的；在Fuchsia里，整个桌面的绘制是交给Vulkan的

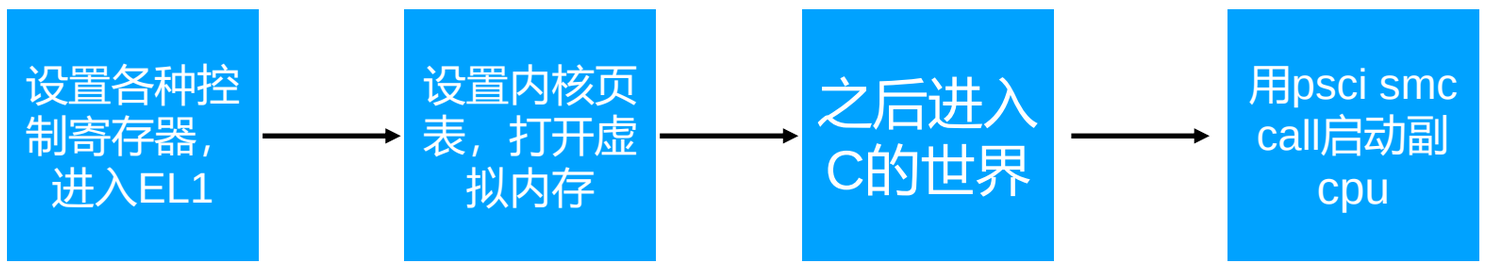

X Client
X Client
X Client
③
②
X server
④
⑤
Compositor
⑥
①
KMS
evdev
Kernel

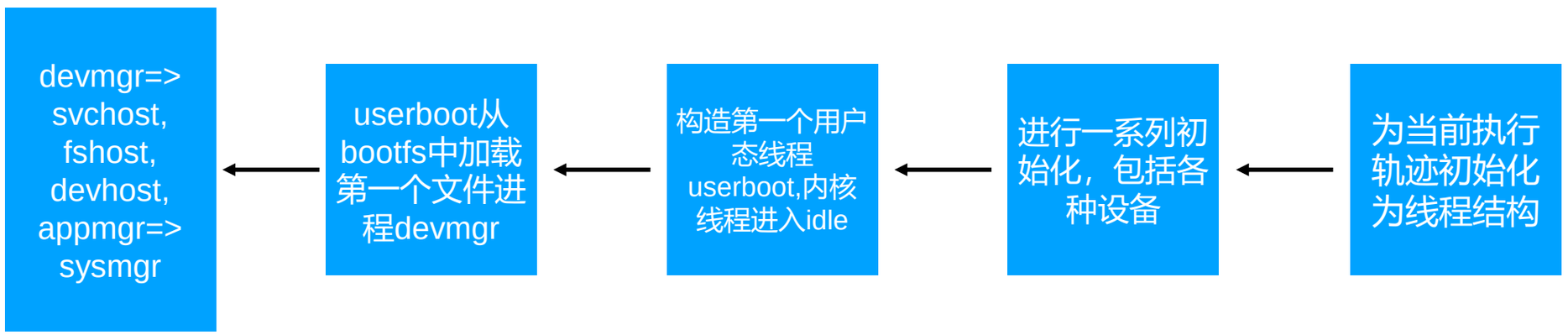

Wayland Client
Wayland Client
③
②
Wayland
Compositor
④
①
KMS
evdev
Kernel

Fuchsia分层

- Fuchsia是一个像Lego玩具一样组装起来的操作系统
  - 谷歌在设计时已经考虑了其他厂商可能会深度定制适配自己产品的操作系统，所以模块化做得比Android彻底很多
  - 厂商的深度定制可以从以下任意一层开始
    - Zircon: 微内核，基础服务进程（设备管理器，核心设备驱动，libc, 进程间通信接口库fidl)
    - Garnet: 设备层面的系统服务：软件安装，通信，媒体，图形，包管理，更新系统等
    - Peridot: 用户体验的基础设施层：模块，用户，存储服务，等等
    - Topaz: 系统的基础应用，Web, Dart, Flutter
    - 这些名字来自于Steven Universe

Fuchsia启动流程





Zircon内核线程比较少，主要是dpc thread (deferred procedure call)

上面的符号代表虚拟地址(-4G之上)

__code_start=-4G,
KERNEL_BASE

_start, IMAGE_ELF_ENTRY

__code_end

boot_cpu_kstack

__bss_start

tt_trampoline

boot_cpu_kstack_end

sp(third)

_end

0

64k

64k+80

stack_end,
sp_el1,
sp(first)

kernel

_bootdata_file_header

_bootdata_kernel_header

_bootdata_kernel_payload

kernel_image

sp(second)

下面的符号代表物理地址(0开始)

第一个用户态进程的创建

- 之前的微内核一般需要实现一个基本的文件系统加载功能在内核里，然后加载第一个用户进程文件，之后就不再使用内核里的文件系统功能

- Zircon把第一个用户态进程的ELF文件嵌入进内核映像里，这样就不需要从文件系统里加载了。

Zircon用户态

- devmgr, devhost, svchost, fshost
- appmgr
- sysmgr
- Fuchsia定义了一套稳定的DDK接口，硬件厂商开发自己的闭源驱动的方便性大大提高了。因为Linux kernel是拒绝提供稳定的内核内部驱动接口的。要想被官方维护，就得放进内核里，否则只能自己跟着内核去改接口。
- 内核不提供POSIX支持，用户层可以模拟一部分POSIX接口

Kernel Address Space Layout Randomization

- ELF的加载位置是随机的，并不是遵守ELF program header里规定的v_addr

- 会在加载时对符号地址进行修正

Fuchsia目前的运行环境

- Qemu
  - 最方便的环境，没有GUI
- Intel NUC
  - 目前最好的测试环境，有GUI
- Khadas Vim2
  - Google内部开发用的板子

Qemu

- 在Qemu中可以直接运行
  - booloader加载到0x40080000
  - 内核加载到0x40090000
  - ramdisk加载到0x48000000
  - 0x40000000-0x40080000之间是FDT flattened device tree

Intel NUC

- 开发机启动paving服务，会将整个Fuchsia操作系统刷到NUC上。

- 启动zircon到zedboot模式，会直接连接开发机

Khadas Vim2开发板

- Amlogic S912 SoC
  - Quad Core A53
  - Mali-T450MP5 GPU
- 3G DDR4
- 64G eMMC storage
- HDMI, USB-C, USB 2.0, TF Card, Ethernet, WiFi, Bluetooth

Vim2的启动

- Arm Trusted Firmware:
  - BL1 in ROM
  - Custom u-boot: BL2 + BL30 +BL31 + BL32 + u-boot(BL33)
    - 其中bl2,bl30,bl31,bl32都是amlogic提供的binary
- bl33从emmc offset 0x50200处开始，加载到内存16MB处执行。
- 使用fastboot协议可以用usb-c将zircon kernel写入boot分区
- https://github.com/mikevoydanoff/u-boot

系统软件研发能力的获得

- 系统软件与应用软件不同
  - 有大量的缄默知识，长期积累的know-how
  - 工具链：gcc, ld, as, clang, ELF,
  - 微处理器：X86, ARM,
  - 周边设备：UEFI, ACPI, APIC, PCIE, USB, SATA, AHCI, GPU …
  - 知识存在于代码中，没有系统化的know-how文档，硬件标准文档一般都是1000+页
  - 写玩具系统容易，产品级的设计非常困难：支持海量的设备，应用，负载
- 要经过以下四个阶段
  - 模仿
  - 理解
  - 掌握
  - 创新

总结

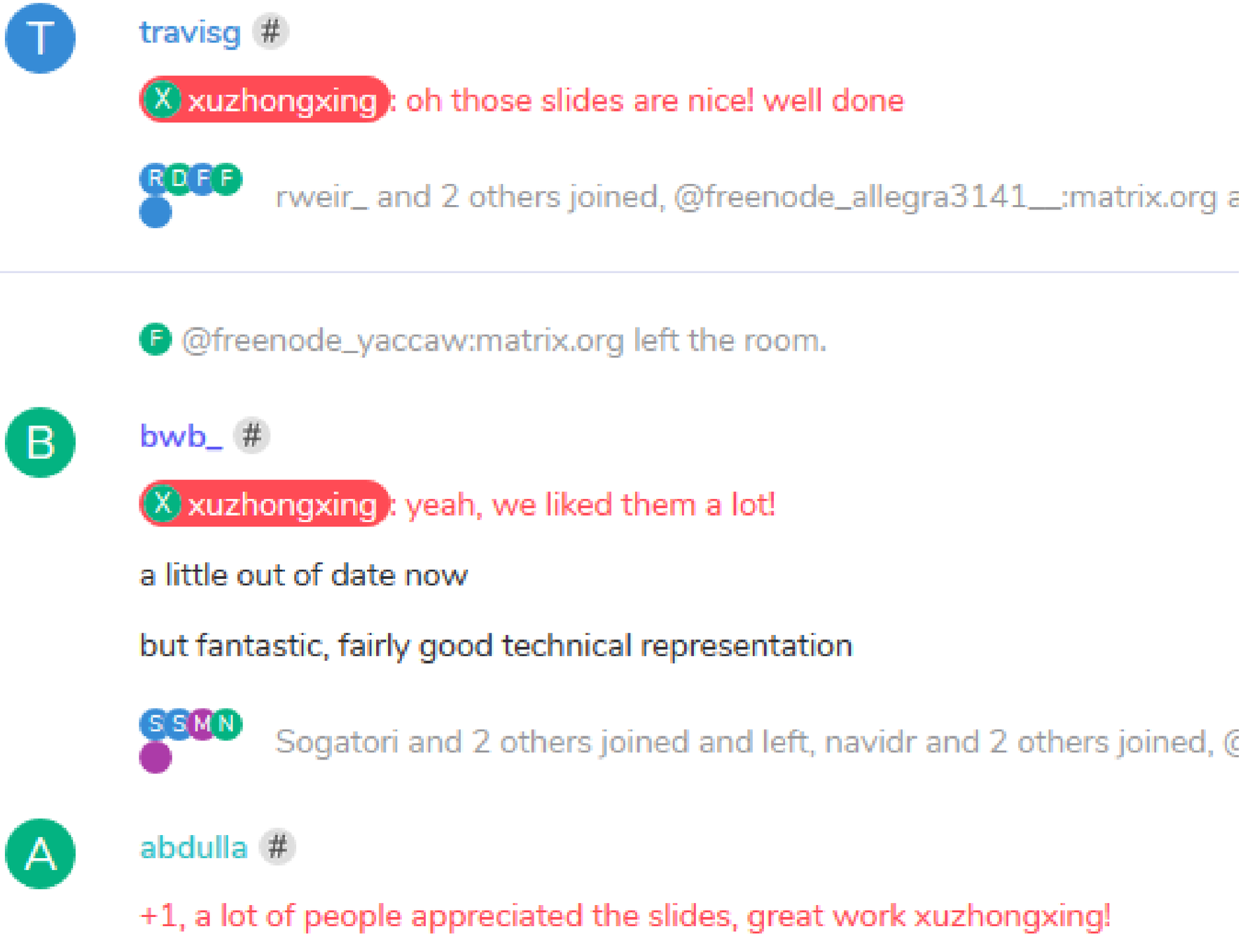
- Fuchsia重新思考了操作系统设计的各个方面，是一次难得的从头开始的机会

- 在未来Fuchsia会成为一个非常重要的操作系统

Hacker News上的讨论

- https://news.ycombinator.com/item?id=19485121

IRC上Fuchsia开发者的反馈



笔记

- https://github.com/xuzhongxing/fuchsia-notes

世界需要新的操作系统

- 每一个大型软件系统都值得尊重
- Windows老迈龙钟，历史负担太重，微软自己的创新Midori胎死腹中，因为无法承受在新的框架中重新实现一遍Windows的全部功能，只能在原地进行重构
- Linux里大部分开发人员只关心服务器的世界，就像一个专注于在甲板下面锅炉房里干活的锅炉工
- MacOS, iOS封闭在苹果的硬件生态里
- Android为了弥补Linux的缺点打上了一个厚厚的中间层，不断在做着妥协
- GNU Hurd作为GNU项目“最后的组件”一直未能产品化，原因是“微内核消息传递机制debug太困难”?
- Unix的后继者Plan 9于2002年发布了最后一个版本，它的余热随着作者融入了Go