IPADS
INSTITUTE OF PARALLEL AND DISTRIBUTED SYSTEMS

上海交通大学
SHANGHAI JIAO TONG UNIVERSITY

操作系统课程引言

上海交通大学 · IPADS · 古金宇

版权声明

- 本内容版权归**上海交通大学并行与分布式系统研究所**所有
- 使用者可以将全部或部分本内容免费用于非商业用途
- 使用者在使用全部或部分本内容时请注明来源
  - 内容来自：上海交通大学并行与分布式系统研究所+材料名字
- 对于不遵守此声明或者其他违法使用本内容者，将依法保留追究权
- 本内容的发布采用 Creative Commons Attribution 4.0 License
  - 完整文本：https://creativecommons.org/licenses/by/4.0/legalcode

大纲

- 为什么要学习操作系统
- 操作系统的历史
- 什么是操作系统
- 操作系统的分化
- 操作系统的不同架构
- 课程信息

为什么学习操作系统

为什么学习操作系统？三个“有助于”

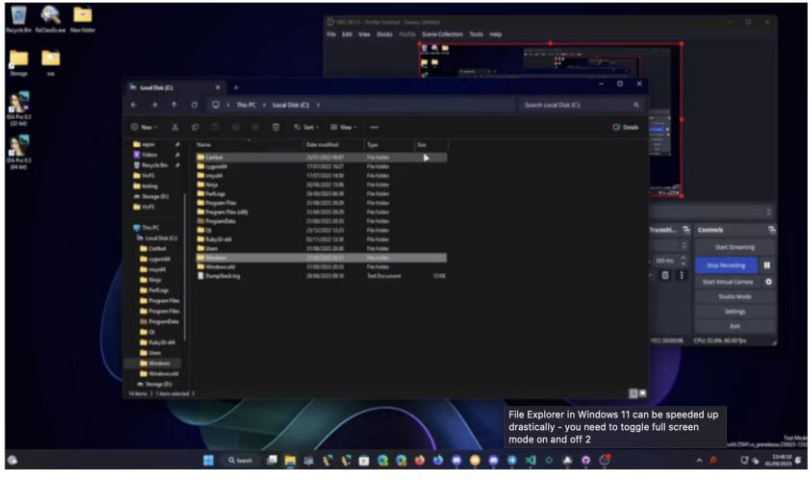
1. **有助于更好地理解底层发生了什么**
   - 有能力解释一些现象并探索更多可能
2. **有助于更好地驾驭硬件**
   - 更高效、更安全地提高生产力
3. **有助于做出更有影响力的工作**
   - 底层的创新能够影响大范围的应用

1. 有助于更好地理解底层发生了什么

File Explorer in Windows 11 can be speeded up drastically – you need to toggle full screen mode on and off

6 days ago · Robbie Elmers · Add Comment

File Explorer in Windows 11 is not particularly fast, but thanks to an unknown feature of the program, a way was found to speed it up drastically – you need to enable and disable full screen mode. This method was reported by the user of the social network X under the nickname Vivi .



Windows双击F11加速?

Linux 28击退格键直接登录?

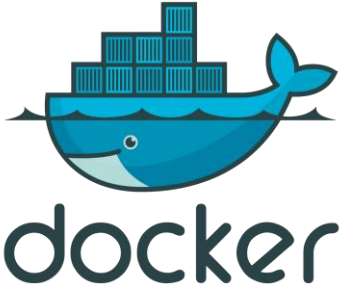
1. 有助于更好地理解底层发生了什么

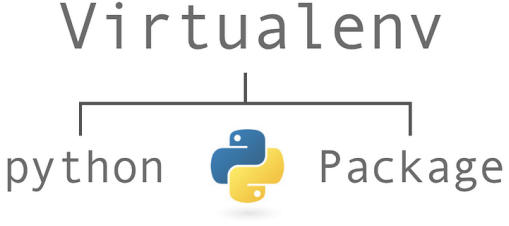
Docker vs. Virtual Environment: A Comparison and User Manual

Giovanni Solano Porras · Follow

4 min read · Jul 5

When it comes to software development and deployment, managing dependencies and isolating environments are crucial for ensuring consistency and reproducibility. Two popular solutions for achieving this are Docker and virtual environments. In this article, we will compare Docker and virtual environments, discussing their differences, benefits, and use cases. Furthermore, we will provide a comprehensive manual for using both, empowering developers to make informed choices based on their specific needs.





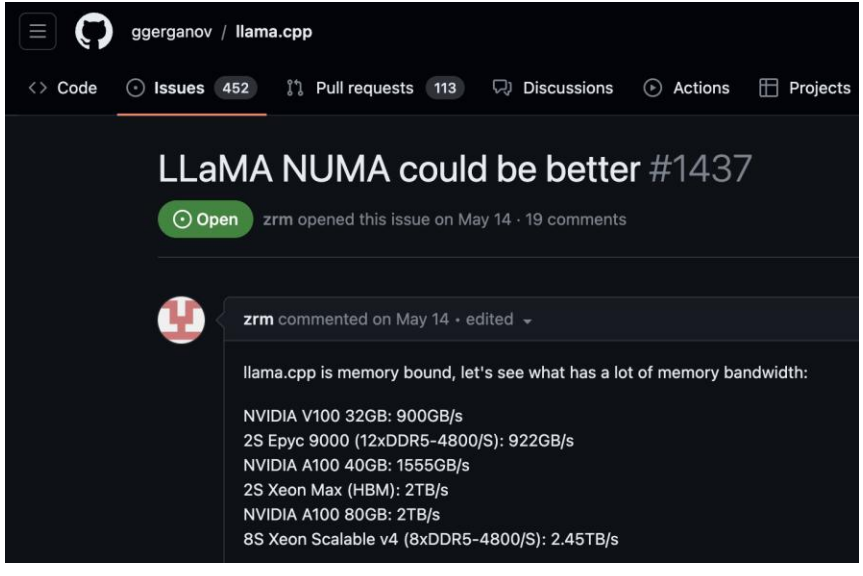
为什么Docker可以隔离不同的Python环境？什么是Namespace？是否也可以支持性能隔离？

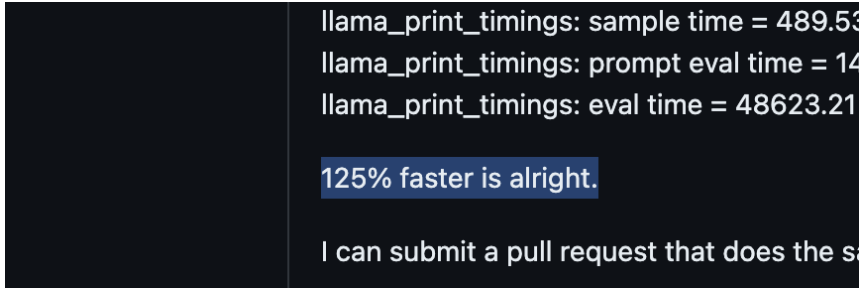
1. 有助于更好地理解底层发生了什么

```
(v) (base) [root@server48 mmocr-main]# python mmocr/utils/ocr.py demo/demo_text_ocr.jpg --print-result
Use load_from_http loader
Use load_from_http loader
/home/wht/Model/mmocr-main/v/lib/python3.7/site-packages/mmdet/datasets/utils.py:70: UserWarning: "Image
ToTensor" pipeline is replaced by "DefaultFormatBundle" for batch inference. It is recommended to manual
ly replace it in the test data pipeline in your config file.
  'data pipeline in your config file.', UserWarning)
Segmentation fault
(v) (base) [root@server48 mmocr-main]# python
Python 3.7.12 (default, Dec 22 2021, 17:18:34)
[GCC 5.5.0] on linux
Type "help", "copyright", "credits" or "license" for more information.
>>> from mmocr.utils.ocr import MMOCR
>>> ocr = MMOCR()
Use load_from_http loader
Use load_from_http loader
>>> results = ocr.readtext('demo/demo_text_ocr.jpg', print_result=True, imshow=True)
/home/wht/Model/mmocr-main/v/lib/python3.7/site-packages/mmdet/datasets/utils.py:70: UserWarning: "Image
ToTensor" pipeline is replaced by "DefaultFormatBundle" for batch inference. It is recommended to manual
ly replace it in the test data pipeline in your config file.
  'data pipeline in your config file.', UserWarning)
Segmentation fault
```

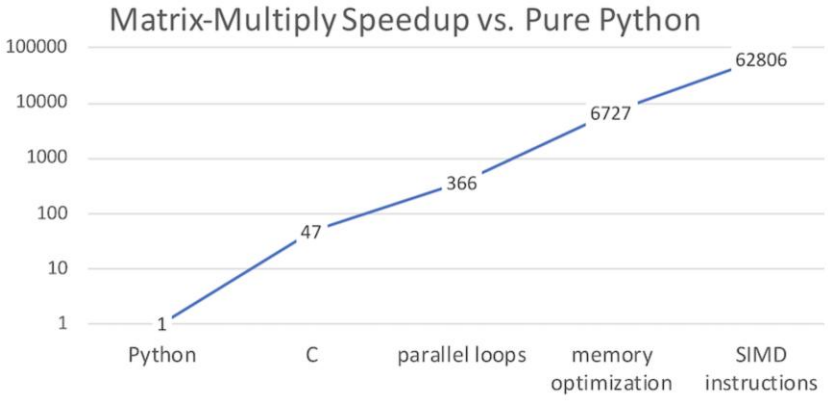
什么是Segmentation fault？为什么会发生？

2. 有助于更好地驾驭硬件





2. 有助于更好地驾驭硬件



Python矩阵乘法加速

90x speedup

#58 (comment) presents the following code:

```
for i in range(n_features):
    for n in range(n_samples):
        subgrad[i] += (- y[n] * X[n][i]) if y[n] * (np.dot(X[n], w) + b) < 1 else 0
    subgrad[i] += self.lambda1 * (-1 if w[i] < 0 else 1) + 2 * self.lambda2 * w[i]
```

Scalene proposes the following optimization:

```
# Vectorized operations to replace for loops
subgrad[:-1] = np.sum(-y[:, None] * X * (y * (X.dot(w) + b) < 1)[:, None], axis=0)
subgrad[:-1] += self.lambda1 * np.sign(w) + 2 * self.lambda2 * w
subgrad[-1] = np.sum(-y * (y * (X.dot(w) + b) < 1))
```

Scalene's proposed optimization accelerates the original code by at least 90x (89 seconds to 1 second, when running 500 iterations), and takes full advantage of multiple cores.

Python矩阵运算优化

3. 有助于做出更有影响力的工作

鸿蒙Next有什么不同？为何不再兼容Android？有哪些意义？

共同推进开源鸿蒙OpenHarmony生态

技术指导委员会主席

陈海波

OpenHarmony项目群TSC主席

技术指导委员会成员

臧斌宇

OpenHarmony项目群TSC成员

技术支撑组成员

夏虞斌

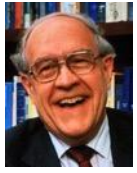
OpenHarmony技术指导委员会
安全及机密计算TSG成员

全国首个OpenHarmony高校技术俱乐部

图灵奖与操作系统

Maurice Wilkes
1967年图灵奖

EDSAC,1949
Multi-programming
第一台存储程序式电子计算机



Frederick Brooks
1999年图灵奖

IBM System/360,1964

Fernando J. Corbató
1990年图灵奖

CTSS,1961 & Multics,1969
分时操作系统

Unix,1971
多任务多用户操作系统

Ken Thompson & Dennis Ritchie
1983年图灵奖

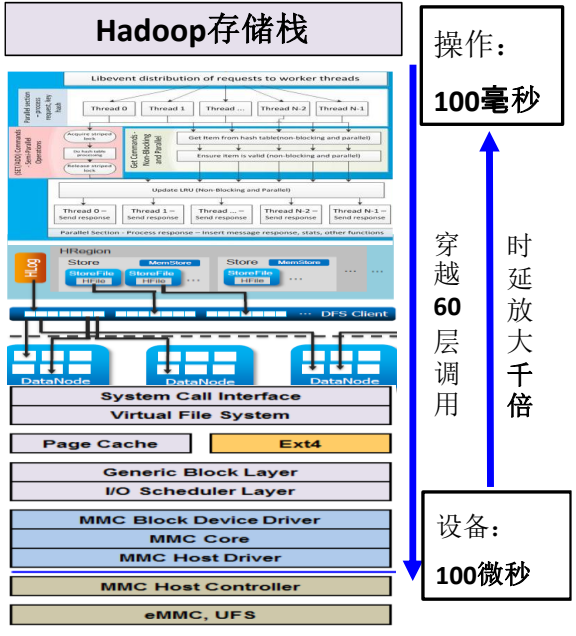
Venus,1972
小型低成本交互式分时操作系统

Barbara Liskov
2008年图灵奖

2019
分布式操作系统

计算机硬件在新应用需求下迅猛发展

- **计算:** 从通用计算走向领域计算，各种xPU不断繁荣
  - GPU、TPU、NPU、IPU等支撑人工智能算力需求
- **存储:** 智能存储，存算一体，非易失内存（SCM），内存与持久存储走向融合
- 数据中心**网络**：Infiniband等网络走向**纳秒级**时延
- 广域**网络**：5G大连接、**低时延**、高可靠使能新型高吞吐、低时延广域计算

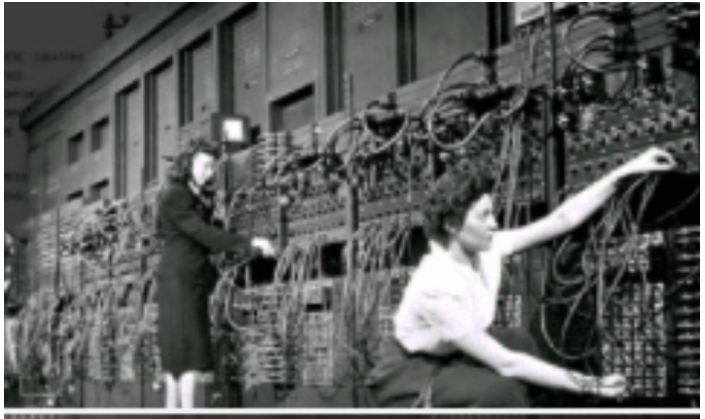


新型硬件的发展，需要新的操作系统抽象与设计来充分释放算力

操作系统的简单历史

计算机诞生时的操作员

1946年2月14日，世界上第一台通用计算机ENIAC在美国宾夕法尼亚大学诞生，请在这个特殊的节日多陪陪你的电脑



批处理操作系统：GM-NAA I/O

- **Robert L. Patrick和Owen Mock于1956年建设**
  - 运行在IBM 704上
  - 主要功能：批处理运行任务

通用操作系统：OS/360

- **IBM System/360 OS，1964**
  - 首个通用操作系统，首次将操作系统与计算机分离
  - ISA：指令集架构
  - 架构师：Gene Amdahl (Amdahl’s Law)
  - 项目经理：Fred Brooks (《人月神话》，1999年图灵奖得主)

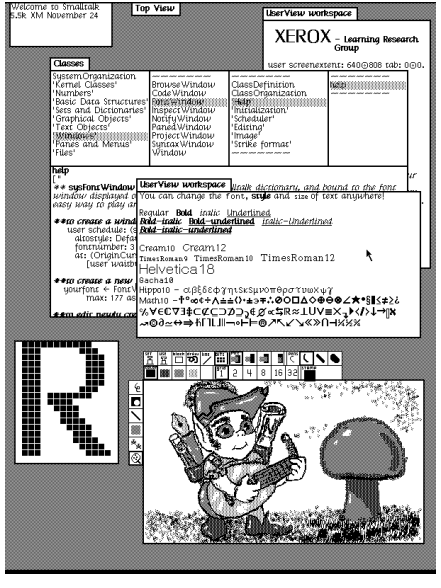
分时与多任务：Multics/Unix/Linux

Multics: Fernando Corbató (1990年图灵奖) MIT/GE, 1964
分时，文件系统，动态链接等

Unix: Ken Thompson, Dennis Ritchie (1983年图灵奖), 1969
Shell，层次化文件系统

Linux: Linus Torvalds, 1991
最流行的开源操作系统

图形界面： Xerox Alto/MacOS/Windows



Xerox Alto （1973）：第一个图形化操作系统，首次使用鼠标 （ Chuck Thacker，2009年图灵奖）

Mac OS (Apple LISA, 1983):1979年乔布斯访问Xerox PARC，意识到GUI的重要性，买下了GUI进行研究

Windows 1.0 （1985）：基于图形界面的操作系统

Credit: https://zh.wikipedia.org/wiki/Xerox_Alto

现代操作系统

今天，操作系统空前繁荣

这些设备中，都有操作系统么？

操作系统的定义

什么是操作系统?

- **你觉得以下哪些属于操作系统?**
  - A. Windows 10所包含的所有软件
  - B. Linux内核以及所有设备的驱动
  - C. 在Macbook上下载安装的第三方NTFS文件系统
  - D. 华为Mate 30出厂时所有的软件
  - E. 大疆无人机出厂时所有的软件
  - F. 火星车上运行的软件
- **你觉得应当如何定义操作系统?**

操作系统：在硬件和应用之间的软件层

操作系统和应用的关系

- 服务应用：如提供硬件操作
- 管理应用：如加载、调度等

操作系统和硬件的关系

- 管理硬件：操作硬件以完成功能
- 抽象硬件：应用不关心硬件差异

“操作系统是管理硬件资源、控制程序运行、改善人机界面和为应用软件提供支持的一种系统软件。”

《计算机百科全书(第2版)》

“操作系统将有限的、离散的资源，高效地抽象为无限的、连续的资源。”

《操作系统：原理与实现》

操作系统的职责

服务应用

管理硬件

请求OS显示一帧画面

应用

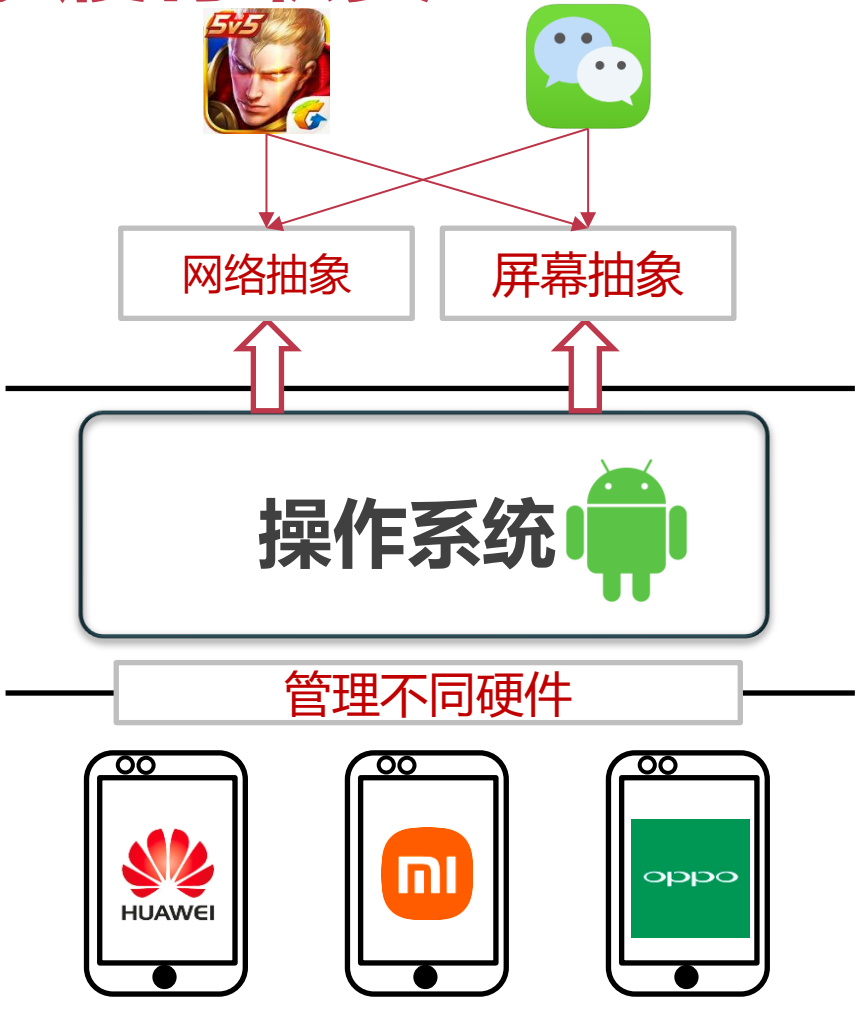
OS向GPU发送渲染指令、
更新屏幕内容

OS

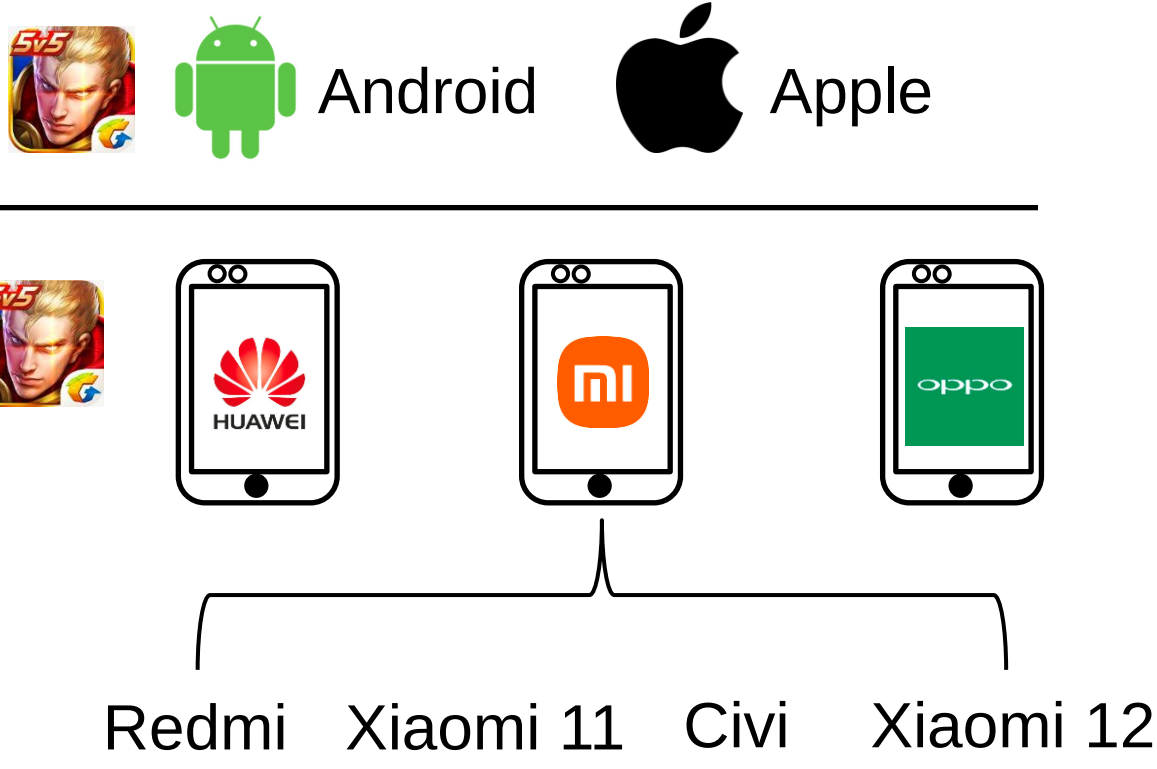

下局一起开黑
应用
应用
请求OS发送消息
OS
OS把消息封装成网络包并发送出去
OS负责调度不同的应用程序、并且协调它们对于CPU、屏幕、网络等硬件的使用

操作系统通过抽象履行职责

硬件抽象：
向应用程序屏蔽硬件实现细节



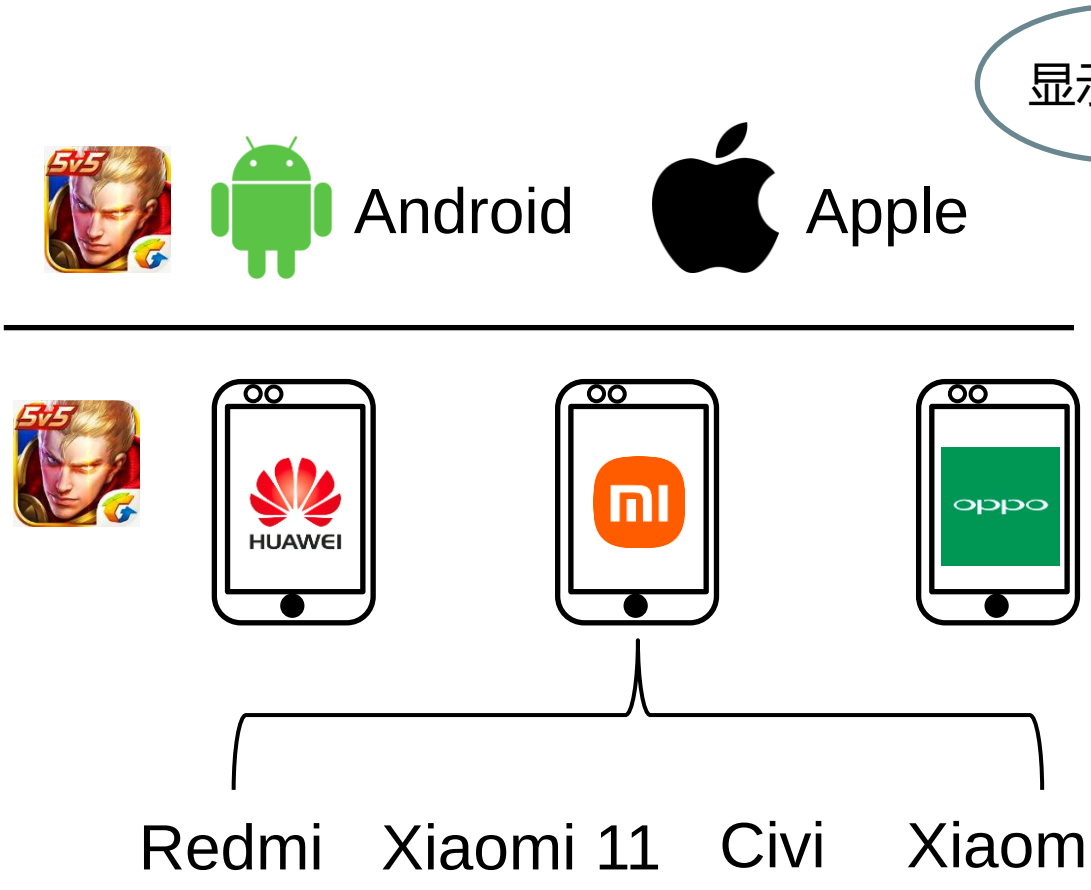
如果没有操作系统



面向操作系统开发应用

面向各款手机开发应用

如果没有操作系统



协调不同的应用程序

究竟什么是操作系统?

- **“操作系统”并没有严格的唯一定义**
  - 是一个相对概念：相对“应用”而言
  - 操作系统也可包含运行在用户态的框架（framework）
- **例如：SSL库、Dalvik（Java虚拟机）是否属于操作系统?**
  - 对Android来说，属于操作系统框架层；App属于应用
  - 对Linux来说，不属于操作系统；hello属于应用
- **我们课程中的操作系统：以hello作为典型应用**

从 Hello World 说起

```c
#include <stdio.h>

int main()
{
    printf("Hello World!\n");
    return 0;
}
```

```
bash$ gcc hello.c -o hello

# 运行一个hello world程序
bash$./hello
Hello World!

# 同时启动两个hello world程序
bash$ ./hello & ./hello
[1] 144
Hello World!
Hello World!
[1]+ Done       ./hello
```

运行hello时，操作系统的作用?

操作系统考虑的一些问题

- hello 这个可执行文件存储在什么位置？是如何存储的？
- hello 这个可执行文件是如何从硬盘加载到内存中运行的？
- hello 这个可执行文件是如何将"Hello World!"这行字输出到屏幕？
- 两个 hello 程序是如何同时运行在一个 CPU 上的？
- 两个 hello 程序之间如果互相通信该怎么做？
- 如果其中一个 hello 出bug了，如何保证另一个能不受影响正常运行？
- …

操作系统需要：1、服务应用；2、管理应用

操作系统为应用提供的服务（部分）

- **为应用提供计算资源的抽象**
  - CPU：进程/线程，数量不受物理CPU的限制
  - 内存：虚拟内存，大小不受物理内存的限制
  - I/O设备：将各种设备统一抽象为文件，提供统一接口
- **为应用提供线程间的同步**
  - 应用可以实现自己的同步原语（lock）
  - 操作系统提供了更高效的同步原语（与线程切换配合，如futex）
- **为应用提供进程间的通信**
  - 应用可以利用网络进行进程间通信（如loopback设备）
  - 操作系统提供了更高效的本地通信机制（具有更丰富的语义，如pipe）

操作系统对应用的管理（部分）

- **应用生命周期的管理**
  - 应用的加载、迁移、销毁等操作
- **计算资源的分配**
  - CPU：线程的调度机制
  - 内存：物理内存的分配
  - I/O设备：设备的复用与分配
- **安全与隔离**
  - 应用程序内部：访问控制机制
  - 应用程序之间：隔离机制，包括错误隔离和性能隔离

操作系统的功能：管理

- **问：如何避免一个流氓应用独占CPU资源?**

```
rogue.c

int main () {
    while (1);
}
```

- **方法-1：每10ms发生一个时钟中断（时间片）**
  - 调度器决定下一个要运行的任务
- **方法-2：可通过信号等打断当前任务执行**
  - 如：kill -9 1951

操作系统的功能：管理

```
rogue-1.c

int main () {
    while (1)
      fork();
}
```

- **问：如何通过3行代码卡死一个OS?**
  - 例：rogue-1.c 可以fork出无数的进程

- **如何解决这个问题?**
  - 方法一：万能方法——重启
  - 方法二：将代码运行在虚拟机中，虚拟机外的应用不受影响
  - 方法三：Linux cgroup （docker），预先设置应用最大占用资源

操作系统的分化

应用与操作系统的解耦

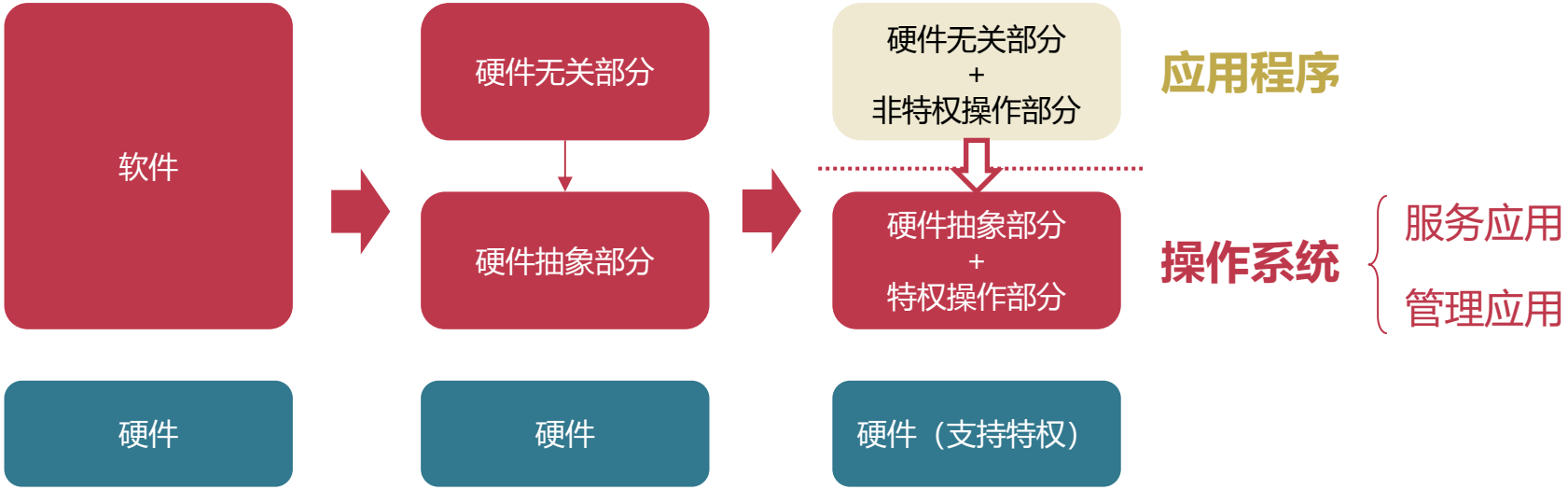
- **在最早的时候，并不区分应用与操作系统**
  - 所有功能都在一起，负责开发的也是同一批人
- **随着功能越来越多，开始出现分化和分工**
  - 有些底层功能被频繁复用，如对存储设备的操作
  - 这些常用的功能形成了特定的模块，通过接口与其他模块交互
    - 例如：存储模块，可将存储抽象成一个大数组
  - 但这并不一定需要操作系统——比如，可以以库的方式存在
- **那么，为什么不能把操作系统作为一个库呢?**

特权级的必要性

- **多道（Multi-programming）操作系统的诞生**
  - 通过时分复用计算资源的方式，在一台计算机上同时运行多个应用程序
- **新的问题：如何保证不同应用间的隔离?**
  - 如果所有的应用均能完全控制硬件计算资源，则会导致混乱
    - 例如：某个应用希望关机，某个应用希望格式化硬盘
  - 因此必须先让应用降权，不允许直接改变全局的系统状态
    - 例如：中断是否打开
- **方案：必须要有不同的权限级——至少两种权限**
  - 低权限：不允许改变全局系统状态，用来运行应用
  - 高权限：集中运行能改变全局系统状态的操作，形成了操作系统

特权操作

操作系统的分化


软件
硬件无关部分
硬件抽象部分
硬件
硬件
硬件无关部分
+
非特权操作部分
硬件抽象部分
+
特权操作部分
硬件（支持特权）
应用程序
操作系统
服务应用
管理应用

对硬件（CPU）的要求

- **CPU对软件提供的接口称为ISA**
  - ISA: Instruction Set Architecture
  - 包含指令、寄存器等软件可见、可操作的接口
- **CPU相应分化出两个模式：非特权模式和特权模式**
  - 非特权模式ISA：应用可使用的指令和寄存器
    - 包括各种运算指令、通用寄存器等
  - 特权模式ISA：只有操作系统才可使用的指令和寄存器
    - 包括各种特权指令、系统寄存器等
  - 从上到下的切换过程通常称为陷入（trap）

非特权部分与特权部分的交互

- **系统调用（System Call）**
  - 应用调用操作系统的机制，实现应用不能实现的功能
  - 应用通过CPU的陷入机制进行模式切换（非特权→特权）
    - 有多种陷入方式，包括特定指令、异常、硬件中断等
  - 操作系统内核通过特定的硬件指令返回应用（特权→非特权）
    - 操作系统**主动**返回：如ARM的eret
- **应用调用操作系统的功能，就像调用普通函数一样**
  - 例如：库函数 printf() ->系统调用 write() ->内核实现 sys_write()
    - `write(1, "Hello World!\n", 13)`

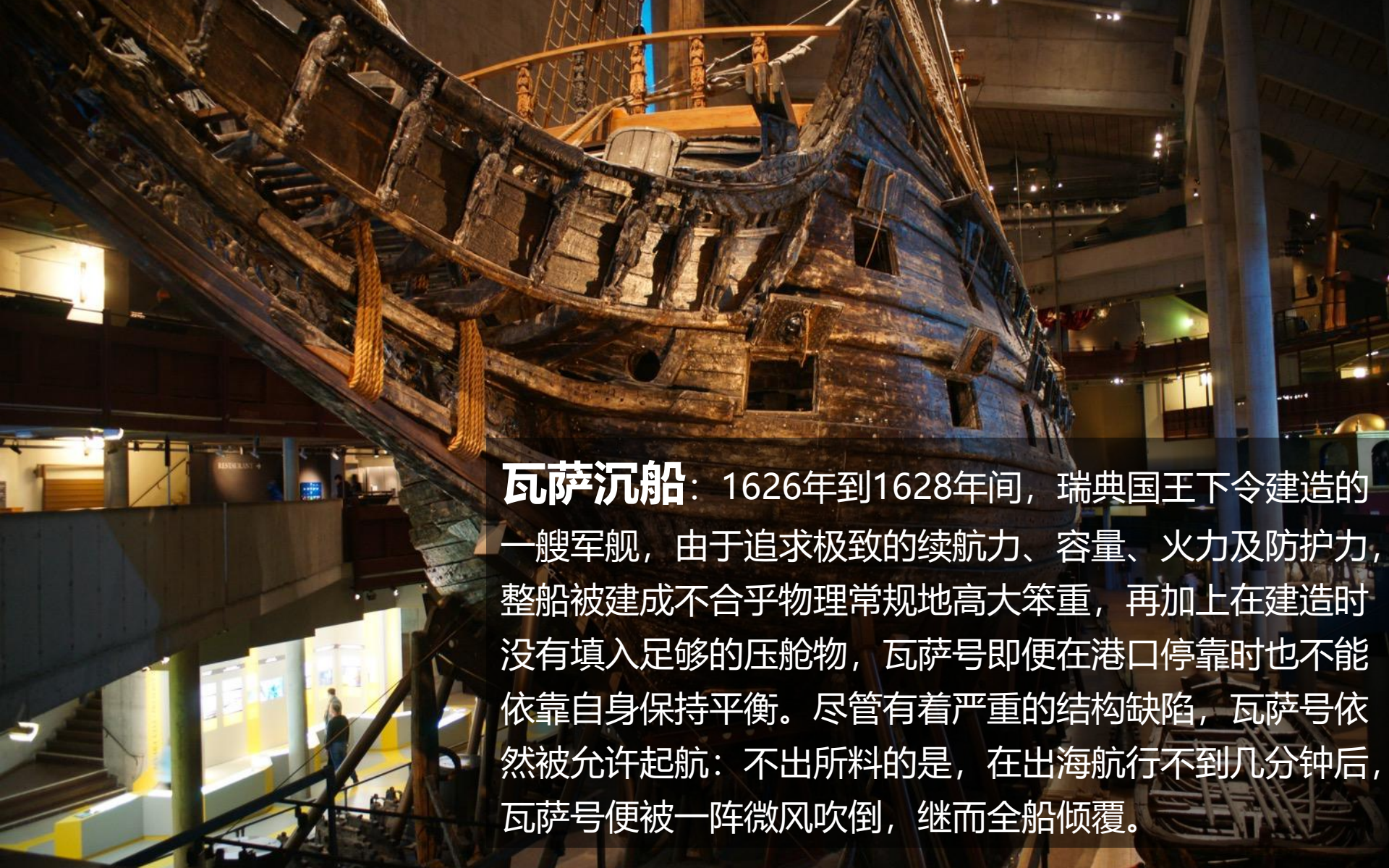
Hello运行中的系统调用（strace）

```
/* 运行hello程序 */
execve("./hello", ["./hello"], 0x7ffed5a79e80 /* 64 vars */) = 0
...

/*将``Hello World!\n''写到标准输出中，在这里，1代表标准输出，
  13代表一共写了13个字符。*/
write(1, "Hello World!\n", 13Hello World!) = 13

/* 执行结束后，hello程序退出*/
exit_group(0)
```

操作系统的不同架构



瓦萨沉船：1626年到1628年间，瑞典国王下令建造的一艘军舰，由于追求极致的续航力、容量、火力及防护力，整船被建成不合乎物理常规地高大笨重，再加上在建造时没有填入足够的压舱物，瓦萨号即便在港口停靠时也不能依靠自身保持平衡。尽管有着严重的结构缺陷，瓦萨号依然被允许起航：不出所料的是，在出海航行不到几分钟后，瓦萨号便被一阵微风吹倒，继而全船倾覆。

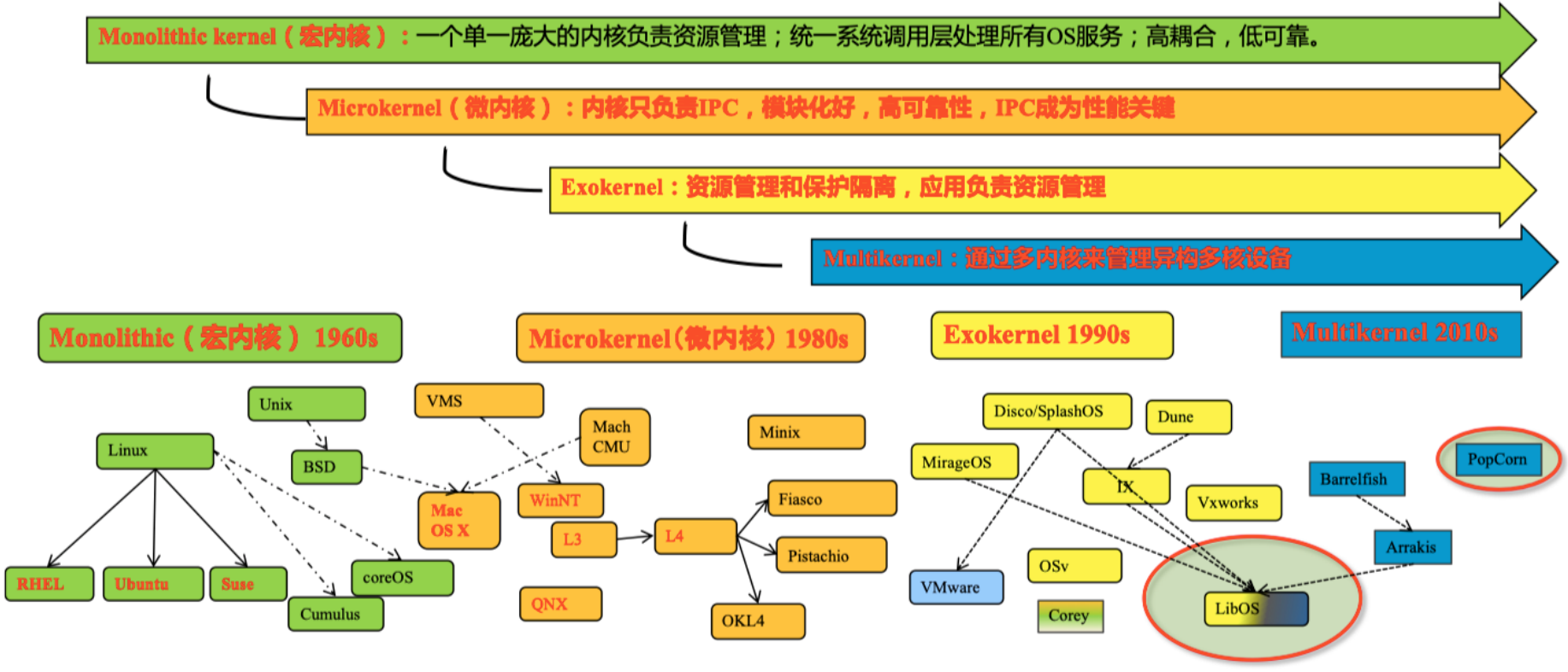
操作系统复杂性与结构

- **操作系统中的"瓦萨号"**
  - 1991-1995年，IBM投入20亿美元打造Workplace操作系统
  - 目标过于宏伟，系统过于复杂，导致项目失败
  - 间接导致IBM全力投入扶植Linux操作系统
- **复杂系统的构建必须考虑其内部结构**
  - 不同目标之间往往存在冲突
  - 不同需求之间需要进行权衡

操作系统的不同目标

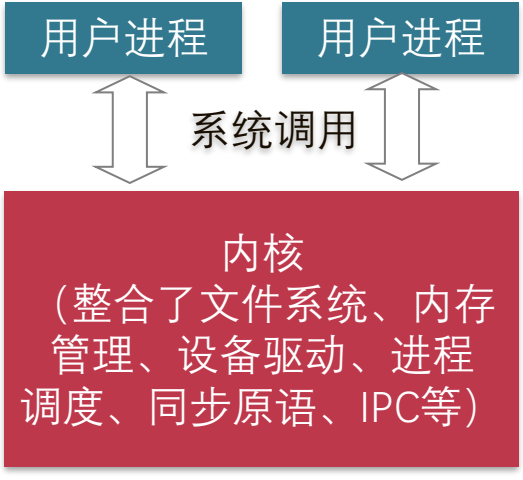
- **用户目标**
  - 方便使用
  - 容易学习
  - 功能齐全
  - 安全
  - 流畅
  - ……
- **系统目标**
  - 容易设计、实现
  - 容易维护
  - 灵活性
  - 可靠性
  - 高效性
  - ……

操作系统的架构及演进


Monolithic kernel（宏内核）：一个单一庞大的内核负责资源管理；统一系统调用层处理所有OS服务；高耦合，低可靠。
Microkernel（微内核）：内核只负责IPC，模块化好，高可靠性，IPC成为性能关键
Exokernel：资源管理和保护隔离，应用负责资源管理
Multikernel：通过多内核来管理异构多核设备
Monolithic（宏内核）1960s
Microkernel(微内核）1980s
Exokernel 1990s
Multikernel 2010s
Unix
VMS
Mach
CMU
Linux
BSD
Mac
OS X
WinNT
L3
L4
Minix
Fiasco
Pistachio
QNX
OKL4
RHEL
Ubuntu
Suse
coreOS
Cumulus
Disco/SplashOS
Dune
MirageOS
IX
Vxworks
Barrelfish
PopCorn
Arrakis
VMware
OSv
Corey
LibOS

1、宏内核（Monolithic Kernel）

- **整个系统分为内核与应用两层**
  - 内核：运行在特权级，集中控制所有计算资源
  - 应用：运行在非特权级，受内核管理，使用内核服务



宏内核的优缺点分析

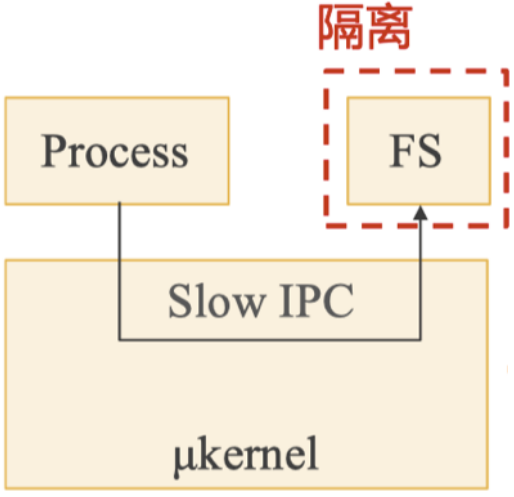
- **宏内核拥有丰富的沉淀和积累**
  - 拥有巨大的统一的社区和生态
  - 针对不同场景优化了30年
- **宏内核的结构性缺陷**
  - 安全性与可靠性问题：模块之间没有很强的隔离机制
  - 实时性支持：系统太复杂导致无法做最坏情况时延分析
  - 系统过于庞大而阻碍了创新：Linux代码行数已经过2800万

宏内核难以满足的场景

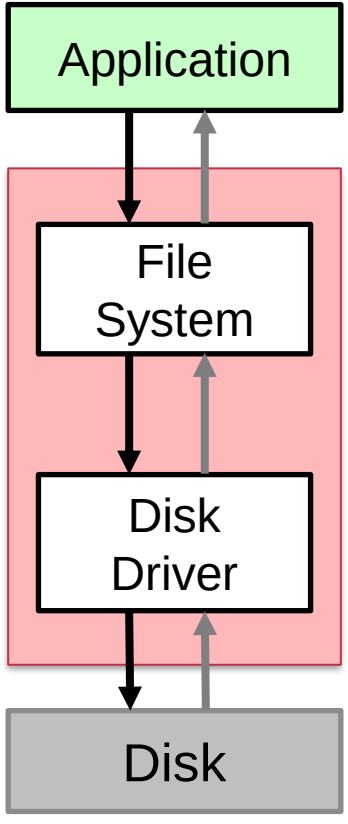
- **向上向下的扩展**
  - 很难去剪裁/扩展一个宏内核系统支持从KB级别到TB级别的场景
- **硬件异构性**
  - 很难长期支持一些定制化的方式去解决一些特定问题
- **功能安全**
  - 一个广泛共识：Linux无法通过汽车安全完整性认证（ASIL-D）
- **信息安全**
  - 单点错误会导致整个系统出错，而现在有数百个安全问题（CVE）
- **确定性时延**
  - Linux花费10+年合并实时补丁，目前依然不确定是否能支持确定性时延

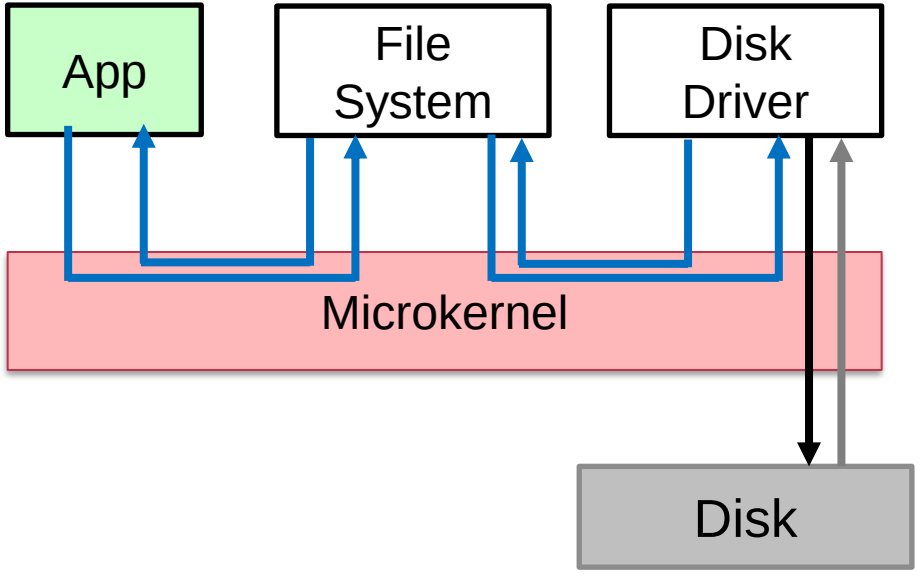
2、微内核的系统架构

- **设计原则：最小化内核功能**
  - 将操作系统功能移到用户态，称为"服务"（Server）
  - 在用户模块之间，使用消息传递机制通信



例：文件的创建


Application
File System
Disk Driver
Disk


App
File System
Disk Driver
Microkernel
Disk

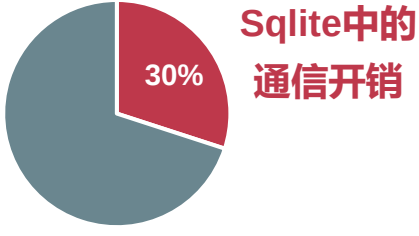
微内核的优缺点分析

- **优点**
  - 易于扩展：直接添加一个用户进程即可为操作系统增加服务
  - 易于移植：大部分模块与底层硬件无关
  - 更加可靠：在内核模式运行的代码量大大减少
  - 更加安全：即使存在漏洞，服务与服务之间存在进程粒度隔离
  - 更加健壮：单个模块出现问题不会影响到系统整体

- **上世纪80/90年代，"微内核"一度成为下一代操作系统的代名词**

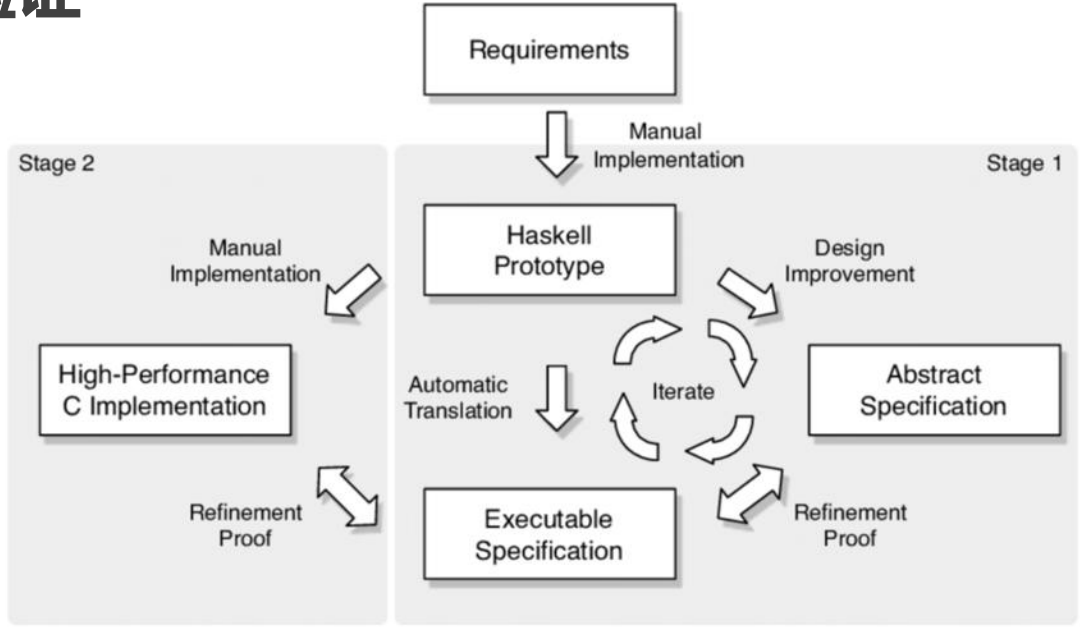
微内核的优缺点分析

- **缺点**
  - 性能较差：内核中的模块交互由函数调用变成了进程间通信
  - 生态欠缺：尚未形成像Linux一样具有广泛开发者的社区
  - 重用问题：重用宏内核操作系统提供兼容性，带来新问题



seL4：被形式化证明的微内核

- **对C的限制，以方便验证**
  - 栈变量不得取引用
    必要时用全局变量
  - 不使用函数指针
  - 不适用union
- **用Haskell构造原型**
  - 用于验证
  - 再手动转换为C



Comprehensive Formal Verification of an OS Microkernel, 2004

MINIX

- **教学用的微内核**
  - 阿姆斯特丹自由大学，Andrew Tanenbaum教授
- **被用于Intel的ME（管理引擎）模块**
  - 也许是世界上用的最多的操作系统...

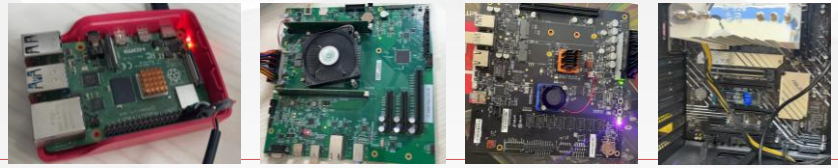
Andrew Tanenbaum

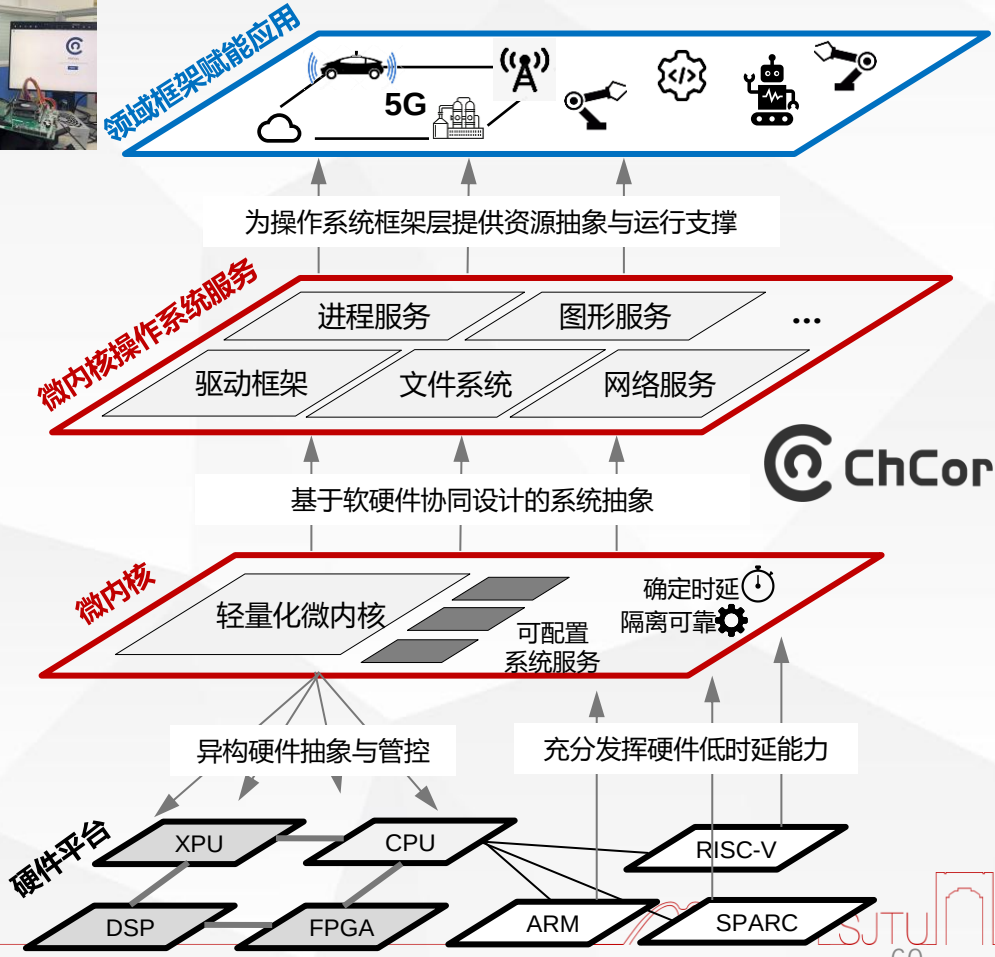
https://www.zdnet.com/article/minix-intels-hidden-in-chip-operating-system/

微内核操作系统ChCore（课程实验裁剪于此）

自主微内核ChCore

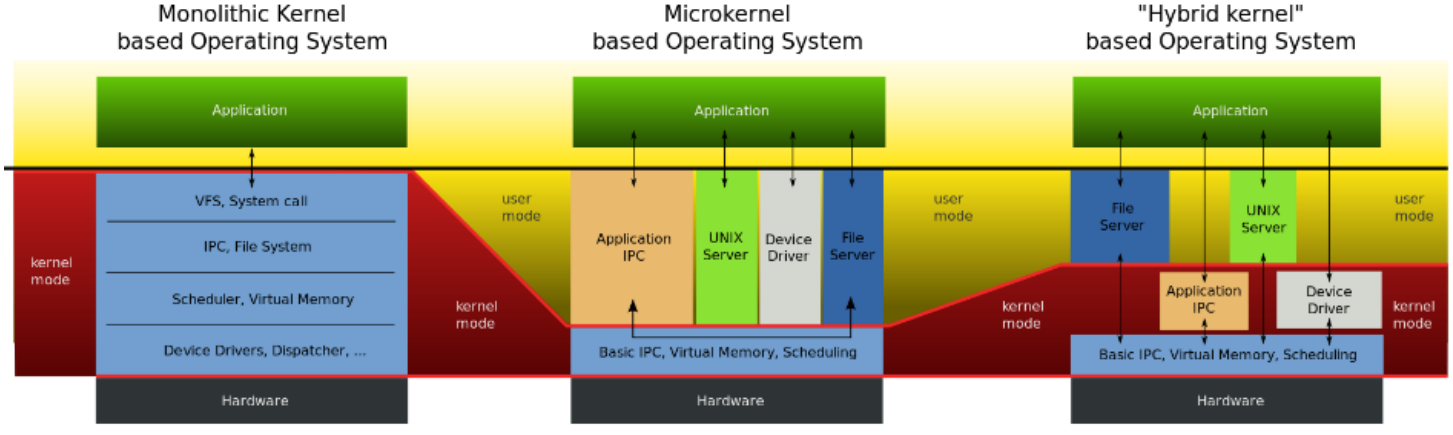
- 初步形成微内核+系统服务+系统框架三层架构，可配置性与扩展性良好
- **应用兼容**：支持C、C++、Python语言开发、支持主要的POSIX接口
- **体系结构**：支持32/64位，支持ARMv8、x86-64、RISC-V、SPARC-V8
- **学术成果**：提出10倍性能提升的IPC设计和微内核高可靠机制 (ATC20, ICDCS21, ATC22)





2.5、混合内核架构

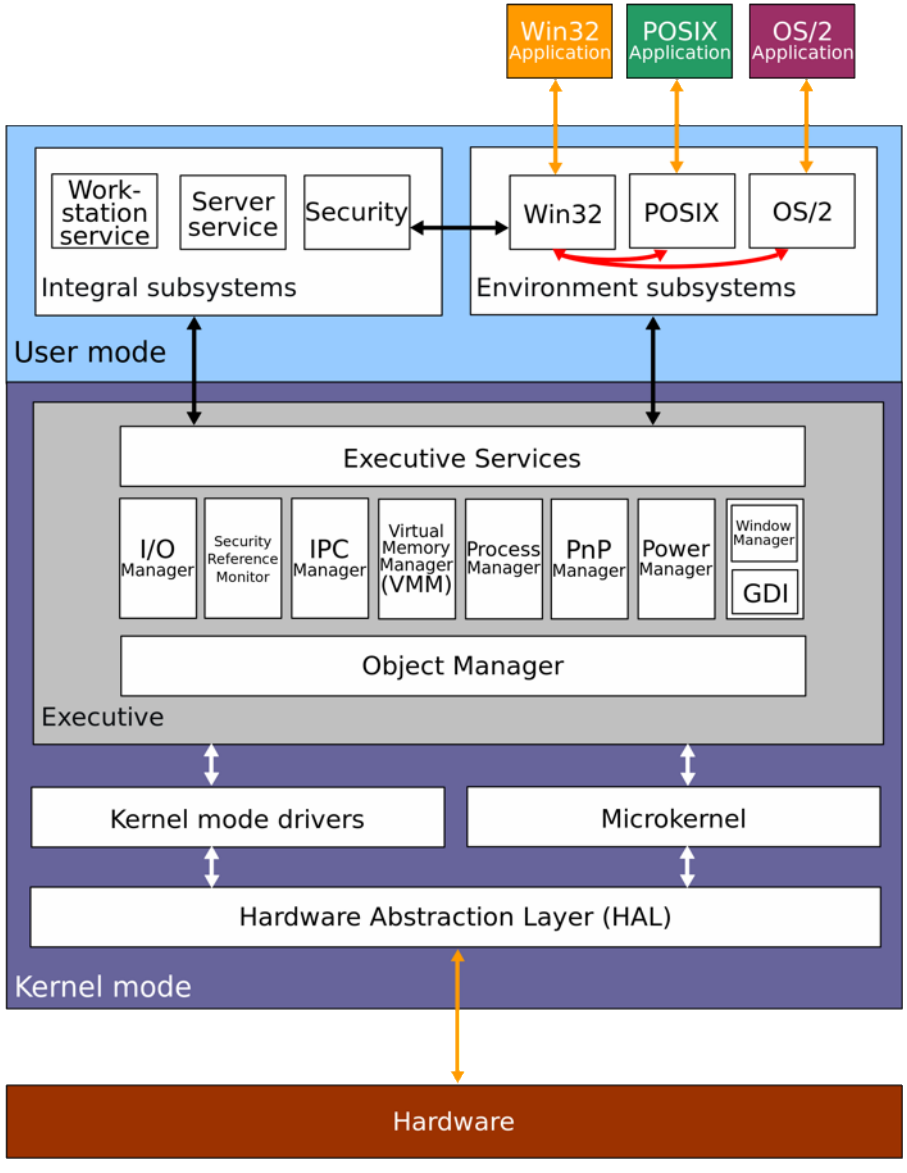
- **宏内核与微内核的结合**
  - 将需要性能的模块重新放回内核态
  - macOS / iOS：Mach微内核 + BSD 4.3 + 系统框架
  - Windows NT：微内核 + 内核态的系统服务 + 系统框架



https://zh.wikipedia.org/wiki/微内核

Windows NT

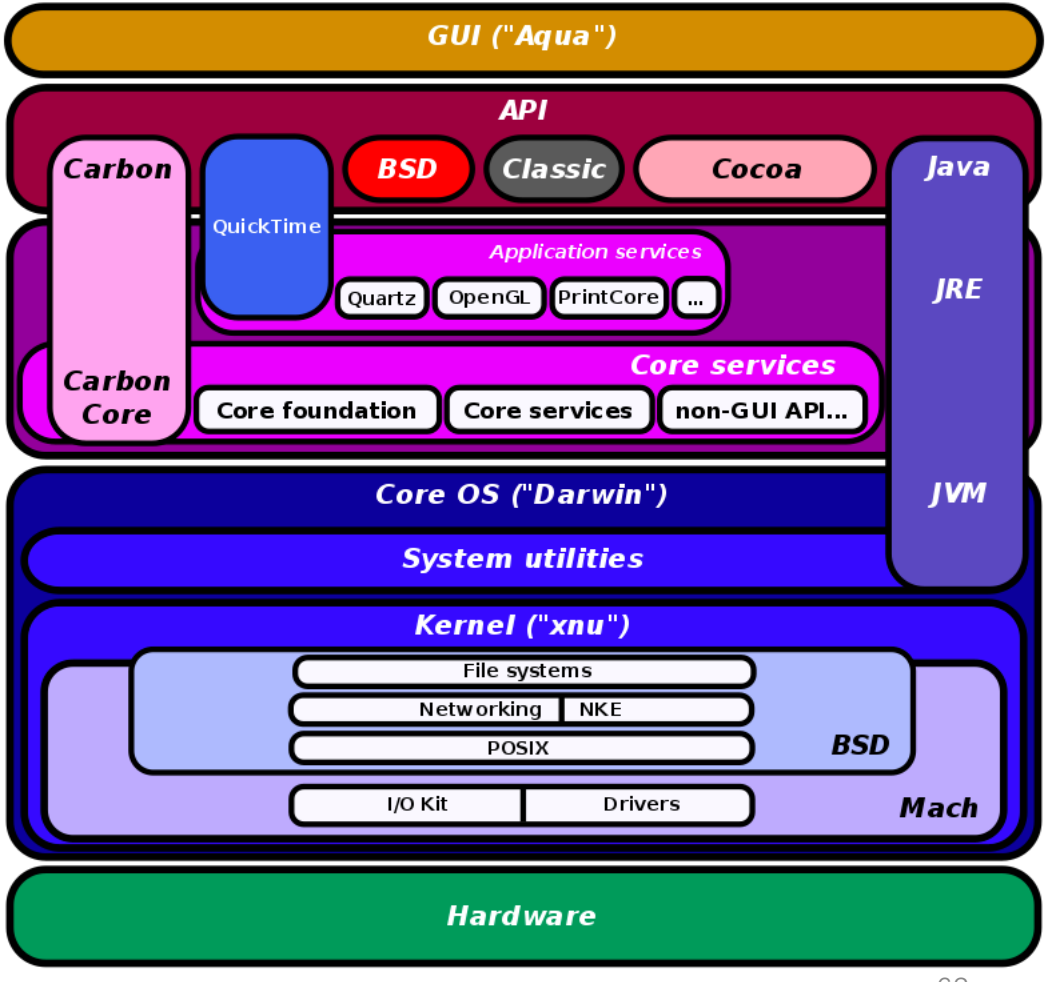
- **Integral子系统（用户态）**
  - 负责处理I/O、对象管理、安全、进程等
- **环境子系统（用户态）**
  - POSIX
- **Executive（内核态）**
  - 为用户态子系统提供服务
- **Microkernel**
  - 提供进程间同步等功能



https://en.wikipedia.org/wiki/Architecture_of_Windows_NT

macOS

- **XNU内核**
  - 基于Mach-2.5打造
  - BSD代码提供文件系统、网络、POSIX接口等
- **macOS与iOS**



https://en.wikipedia.org/wiki/Architecture_of_macOS

3、外核架构（Exokernel）

- **Exokernel 不提供硬件抽象**
  - "只要内核提供抽象，就不能实现性能最大化"
  - 只有应用才知道最适合的抽象（end-to-end原则）
- **Exokernel 不管理资源，只管理应用**
  - 负责将计算资源与应用的绑定，以及资源的回收
  - 保证多个应用之间的隔离

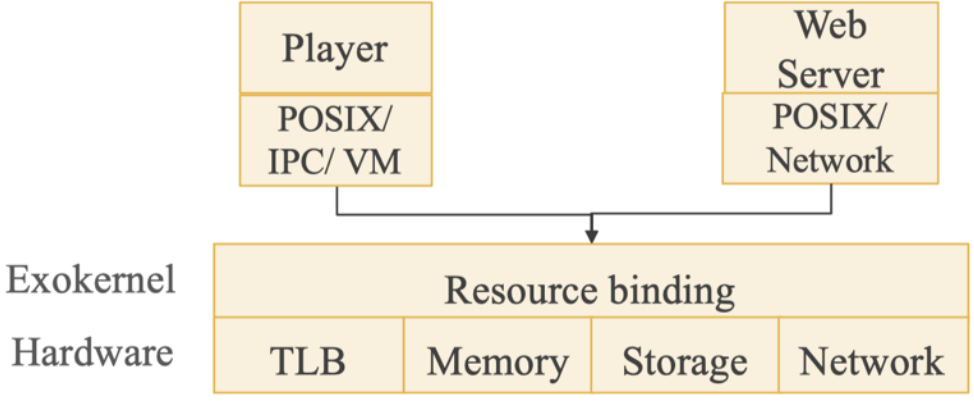
内核态：Exokernel

- **回顾：操作系统 = 服务应用 + 管理应用**

用户态：libOS

Exokernel + LibOS

- **库OS（LibOS)**
  - 策略与机制分离：将对硬件的抽象以库的形式提供
  - 高度定制化：不同应用可使用不同的LibOS，或完全自定义
  - 更高性能：LibOS与应用其他代码之间通过函数调用直接交互



Unikernel（单内核）

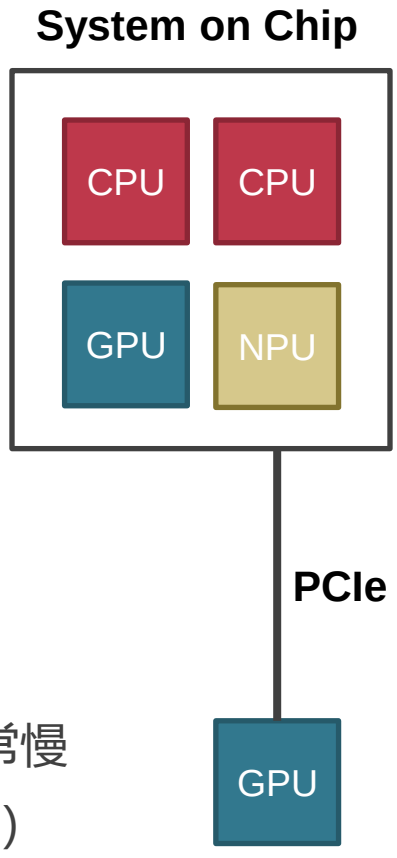
- **虚拟化环境下的LibOS**
  - 每个虚拟机只使用内核态
  - 内核态中只运行一个应用+LibOS
  - 通过虚拟化层实现不同实例间的隔离
- **适合容器等新的应用场景**
  - 每个容器就是一个虚拟机
  - 每个容器运行定制的LibOS以提高性能

Exokernel架构的优缺点分析

- **优点**
  - OS无抽象，能在理论上提供最优性能
  - 应用对计算有更精确的实时等控制
  - LibOS在用户态更易调试，调试周期更短
- **缺点**
  - 对计算资源的利用效率主要由应用决定
  - 定制化过多，导致维护难度增加

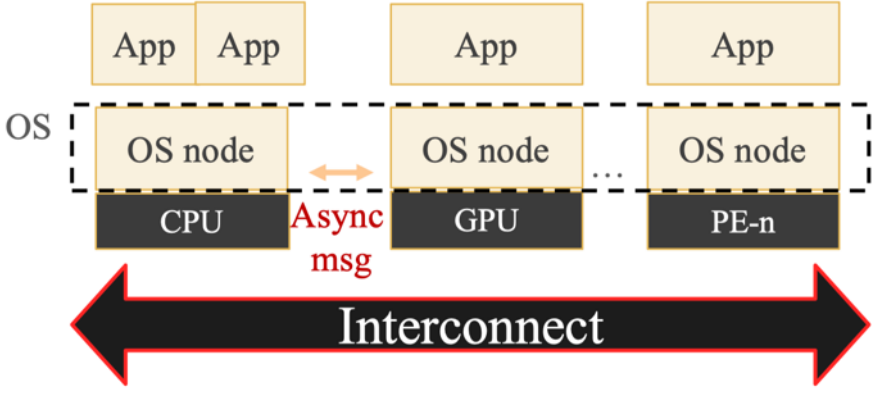
4、多内核/复内核（Multikernel）

- **背景：多核与异构**
  - OS内部维护很多共享状态
    - Cache一致性的保证越来越难
    - 可扩展性非常差，核数增多，性能不升反降
  - GPU等设备越来越多
    - 设备本身越来越智能——设备有自己的CPU
    - 通过PCIe连接，主CPU与设备CPU之间通信非常慢
    - 通过系统总线连接，异构SoC（System on Chip）



Multikernel的设计

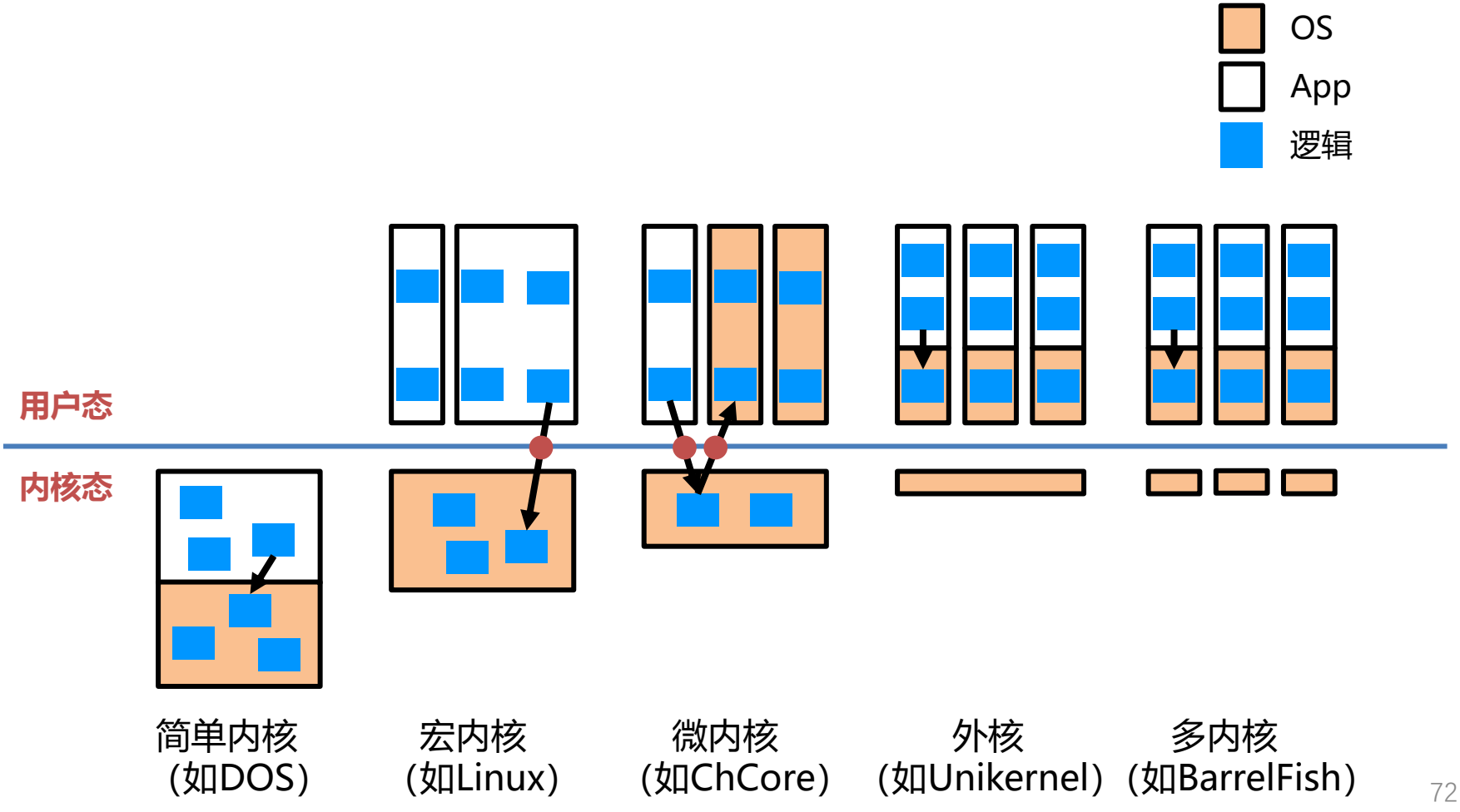
- **Multikernel的思路**
  - 默认的状态是划分而不是共享
  - 维持多份状态的copy而不是共享一份状态
  - 显式的核间通信机制
- **Multikernel的设计**
  - 在每个core上运行一个小内核
    - 包括CPU、GPU等
  - OS整体是一个分布式系统
  - 应用程序依然运行在OS之上



Barrelfish Multikernel

- **Barrelfish操作系统**
  - 来自ETH Zurich和微软研究院
  - 支持异构CPU
  - 在CPU核与节点之间提供通用异构消息抽象
  - 大约10,000行C，500行汇编代码

不同操作系统架构的对比

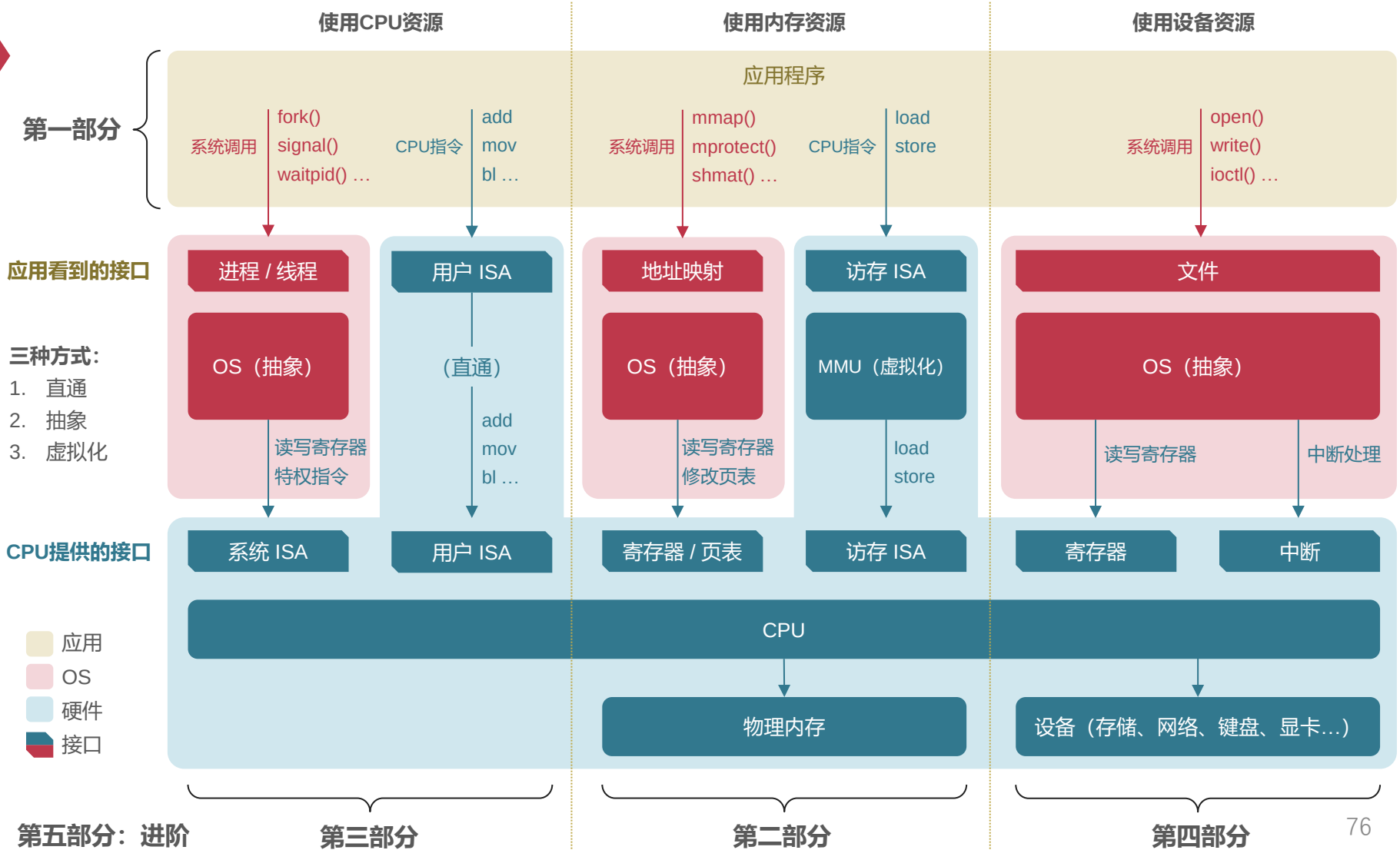


课程信息

从应用的视角看操作系统

- **注意：常见的误解**
  - 应用只是逻辑上运行在操作系统上层
  - 但并不是“由操作系统来运行应用”
  - 实际上，应用的每一行代码都运行在CPU上
  - 操作系统的每一行代码，也都运行在CPU上

- **让一个硬件支持多个应用的三种形式**
  - **直通**：将硬件接口直接暴露给应用
  - **虚拟化**：向应用提供的接口与硬件原有接口一样，但允许多个应用使用
  - **抽象**：提供的接口与硬件接口不一样，操作系统负责转化


使用CPU资源
使用内存资源
使用设备资源
第一部分
应用程序
系统调用
fork()
signal()
waitpid() …
CPU指令
add
mov
bl …
系统调用
mmap()
mprotect()
shmat() …
CPU指令
load
store
系统调用
open()
write()
ioctl() …
应用看到的接口
进程 / 线程
用户 ISA
地址映射
访存 ISA
文件
三种方式：
1. 直通
2. 抽象
3. 虚拟化
OS（抽象）
（直通）
OS（抽象）
MMU（虚拟化）
OS（抽象）
读写寄存器
特权指令
add
mov
bl …
读写寄存器
修改页表
load
store
读写寄存器
中断处理
CPU提供的接口
系统 ISA
用户 ISA
寄存器 / 页表
访存 ISA
寄存器
中断
CPU
物理内存
设备（存储、网络、键盘、显卡…）
应用
OS
硬件
接口
第五部分：进阶
第三部分
第二部分
第四部分


欢迎进入
操作系统的世界