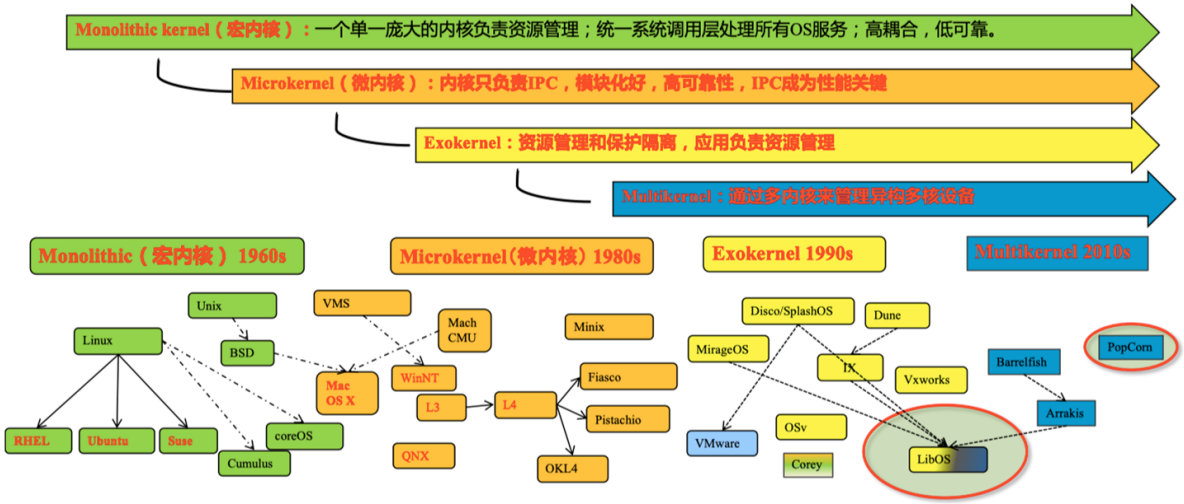
Lecture2 OS-Structure

结构的重要性

策略与机制分离

> 操作系统可仅通过调整策略来适应不同应用的需求

- 策略(Policy)：要做什么
- 机制(Mechanism)：该怎么做



宏内核

> Monolithic Kernel

- 整个系统分为内核与应用两层

  Kernel：在supervisor mode运行，集中控制所有计算资源

  应用：在user mode运行，受内核管理，使用内核服务
- 内核整合了文件系统、内存管理、设备驱动、进程调度、同步原语以及IPC等
- 优势：丰富的积淀和积累
- 劣势：

1. 安全与可靠性：模块之间没有很强的Isolation
2. 实时性支持：Too complex to 做最坏情况时延分析
3. 系统过于庞大阻碍创新

微内核

最小化内核原则

1. 将操作系统功能移动到user态，称为服务(Server)
2. 在用户模块间，使用消息传递机制通信(IPC)

微内核的历史

1. **Mach微内核**：开山鼻祖，用户态与内核态的分工
2. L3/L4：极大提升IPC性能
3. seL4：被形式化证明的微内核
4. QNX Neutrino：满足实时要求，在交通、能源、医疗与航空航天等领域
5. Google Fuchsia：使用Zircon微内核

优缺点

- Pros：易于扩展、移植，更加可靠、安全、健壮
- Cons：性能较差、生态欠缺、重用问题

混合内核架构

> 将宏内核与微内核结合

- 将需要性能的模块重新放回内核态

  MacOS/iOS: Mach微内核+BSD4.3+系统框架

  Windows NT：微内核+内核态的系统服务+系统框架
- 牺牲了一部分微内核架构带来的高可靠和隔离性

外核+库OS

> 管理与保护分离
>
> 管理：LibOS 保护：Exokernel
>
> 操作系统 = 服务应用(用户态：libOS) + 管理应用(内核态：Exokernel)

外核架构(Exokernel)

1. Exokernel不提供硬件抽象

   > 只有应用才知道最适合的抽象(end-to-end原则)

2. Exokernel不管理资源，只管理应用

   > 应用生命周期管理，将计算资源与应用绑定

库OS(LibOS)

1. 策略与机制分离：将对硬件的抽象以库的形式提供
2. 高度定制化：不同应用可以使用不同的LibOS（自定义）
3. 更高的性能：LibOS与应用其他代码之间通过函数调用交互

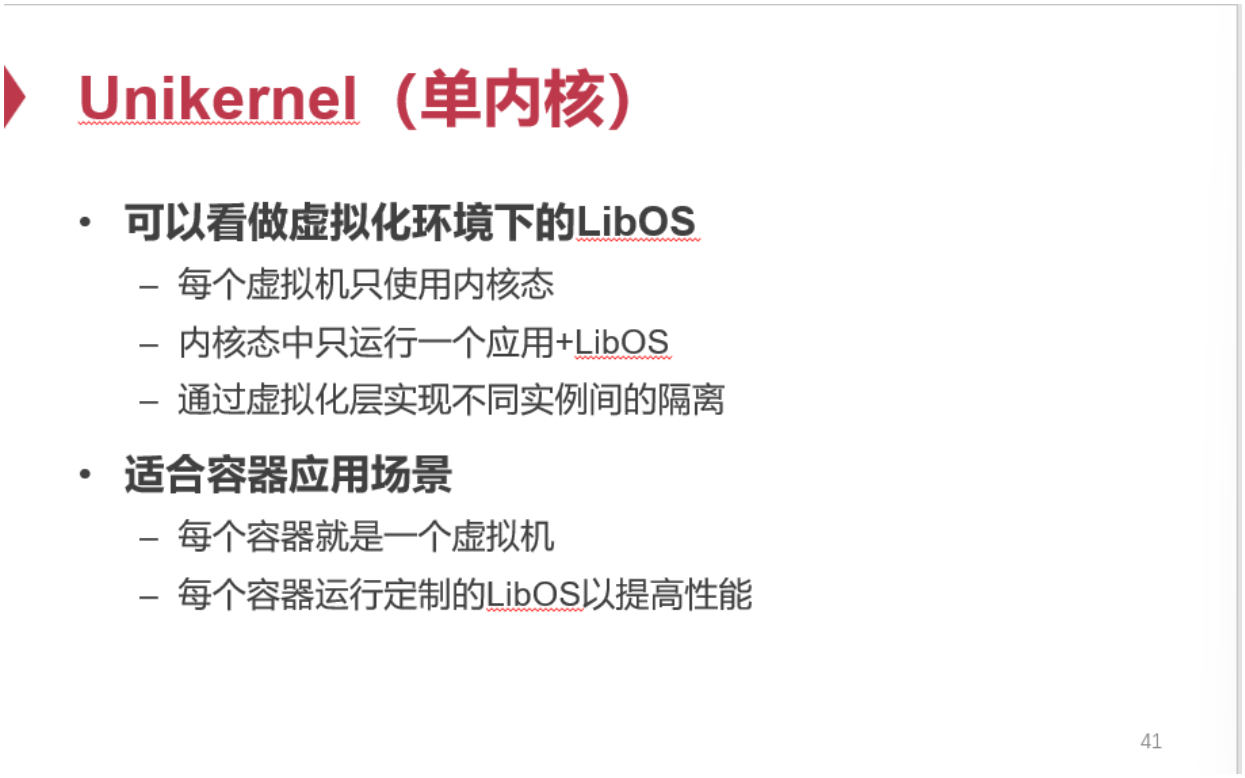
一些机制

1. 安全绑定

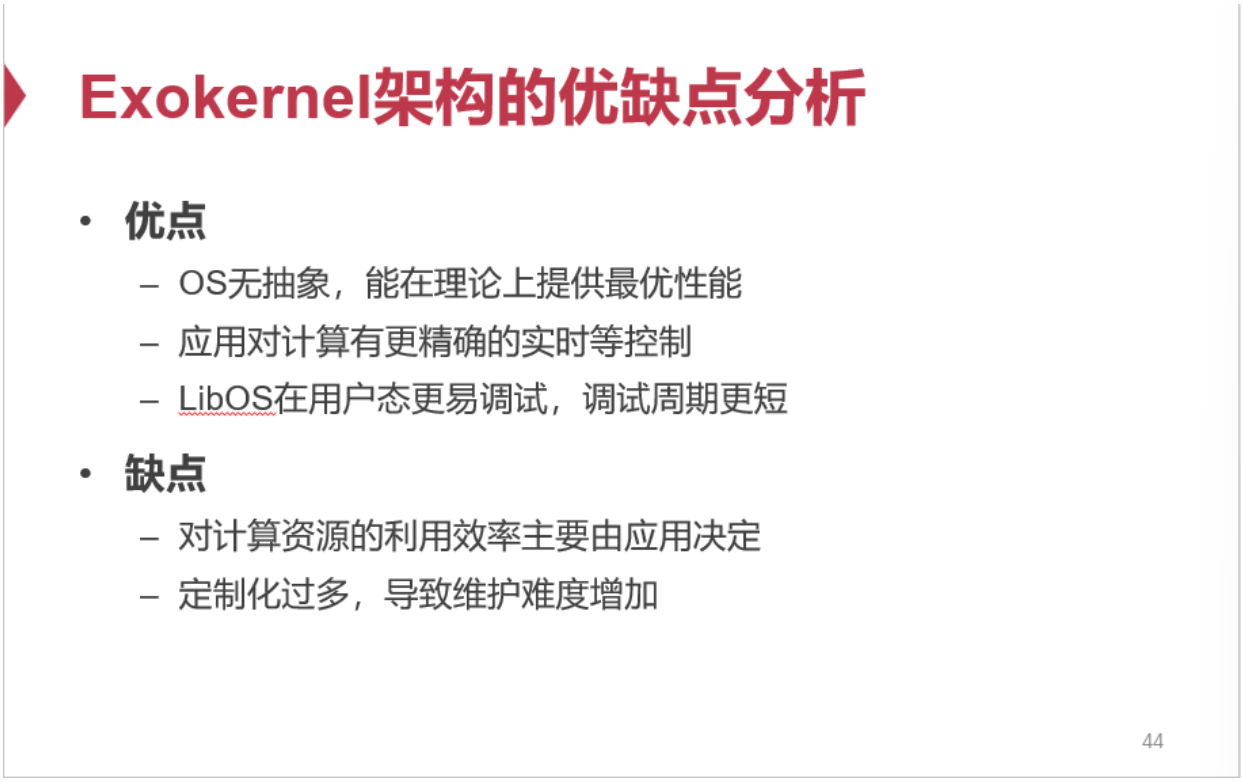
> 将LibOS与计算资源绑定（可用性&隔离性）

2. 显示资源回收&中止协议
    1. Exokenel与应用之间的协议：显示告知应用资源分配的情况，在租期解释之前应主动归还资源
    2. 若应用不归还资源，则强制中止（主动解除资源与应用之间的绑定关系）

Unikernel(单内核)



Exokernel的优缺点分析



Q1: 三种架构性能?

A1：外核>宏内核>微内核

Q2:外核和微内核的区别?

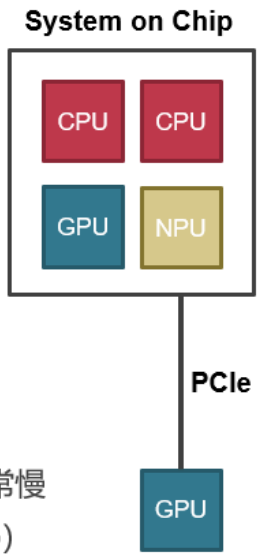
A2：外核是库方式调用硬件，微内核是通过IPC机制调用硬件接口。而且微内核中filesystem只有一份，由于外核中一个应用对应于一种LibOS，filesystem可能有多份，可能会比较占用内存

多内核/复内核(Multi-kernel)

背景：多核与异构

1. OS内部维护很多的共享状态
2. GPU等设备越来越多

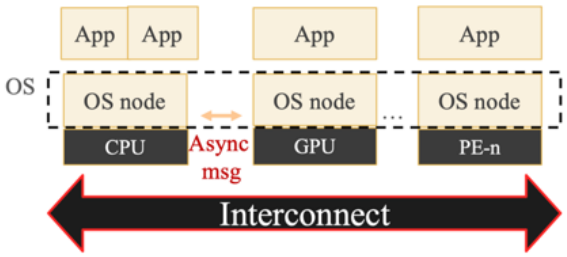
多内核/复内核（Multikernel）



- **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



不同架构对比

不同操作系统架构的对比