IPADS INSTITUTE OF PARALLEL AND DISTRIBUTED SYSTEMS

上海交通大學 SHANGHAI JIAO TONG UNIVERSITY

操作系统结构

陈海波 / 夏虞斌

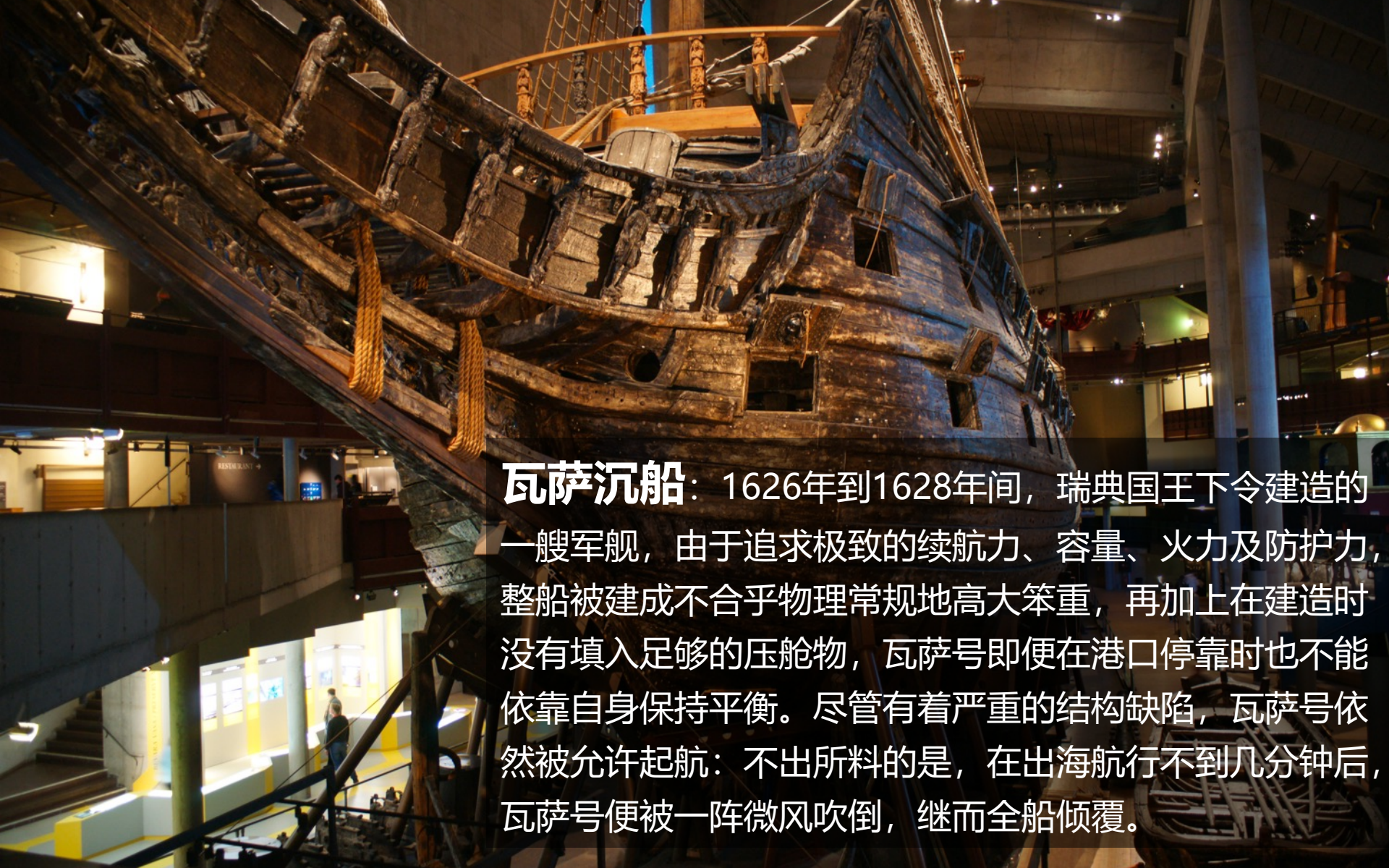
上海交通大学并行与分布式系统研究所

https://ipads.se.sjtu.edu.cn

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

复杂系统结构的重要性



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Workspace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

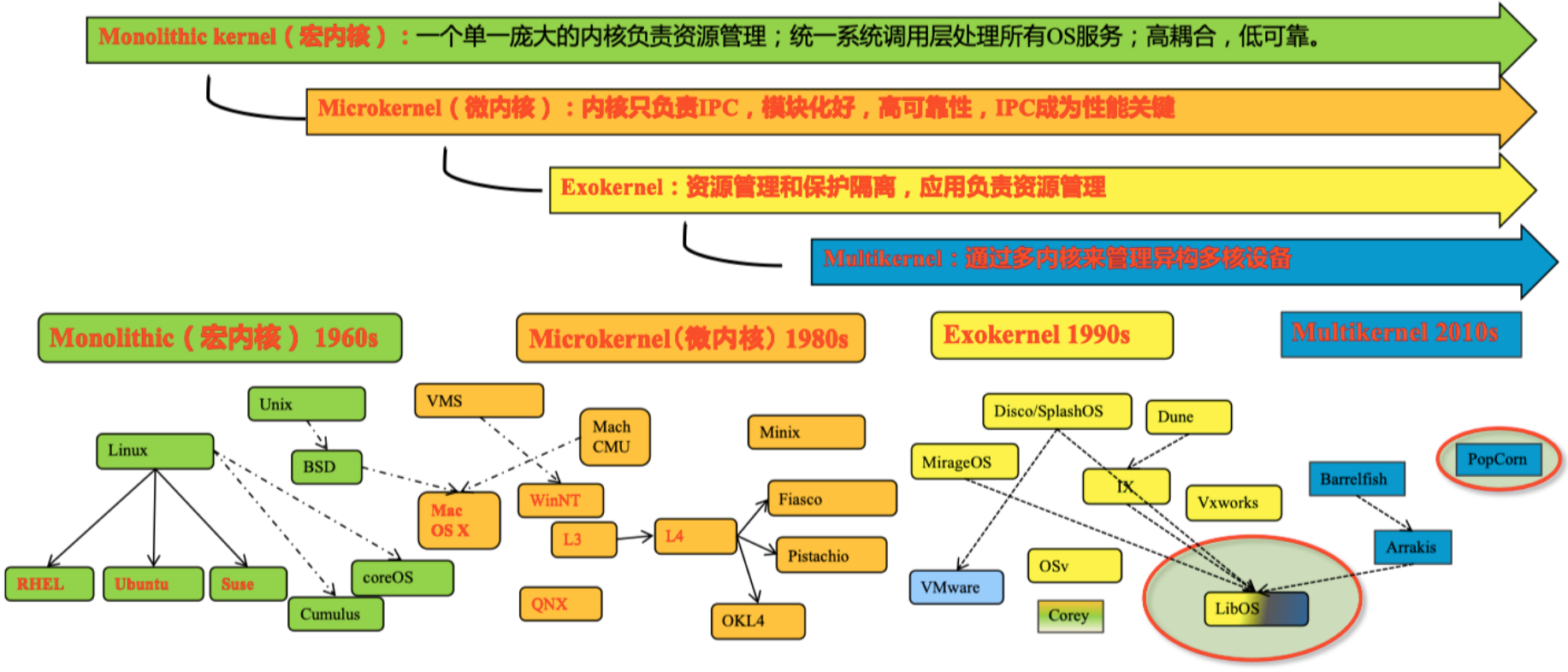
- **用户目标**
  - 方便使用
  - 容易学习
  - 功能齐全
  - 安全
  - 流畅
  - ……
- **系统目标**
  - 容易设计、实现
  - 容易维护
  - 灵活性
  - 可靠性
  - 高效性
  - ……

降低操作系统复杂性

- **重要设计原则：策略与机制的分离**
  - 策略（Policy）：要做什么 —— 相对动态
  - 机制（Mechanism）：怎么做 —— 相对静态
  - 操作系统可仅通过调整策略来适应不同应用的需求

例子	策略	机制
登录	什么用户、以什么权限登录	输入处理、策略文件管理、桌面启动加载 ...
调度	调度算法：Round-robin、Earliest Deadline First ...	调度队列、调度实体（如线程）的表示、调度中断处理 ...

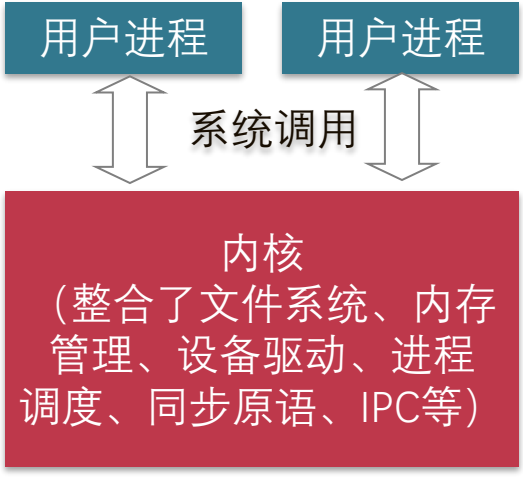
操作系统的架构及演进


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
Linux
BSD
Mach
CMU
Minix
Mac
OS X
WinNT
Fiasco
L3
L4
Pistachio
QNX
OKL4
RHEL
Ubuntu
Suse
coreOS
Cumulus
Disco/SplashOS
Dune
MirageOS
IX
Vxworks
Barrelfish
PopCorn
Arrakis
VMware
OSv
Corey
LibOS

宏内核（MONOLITHIC-KERNEL）

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2千万

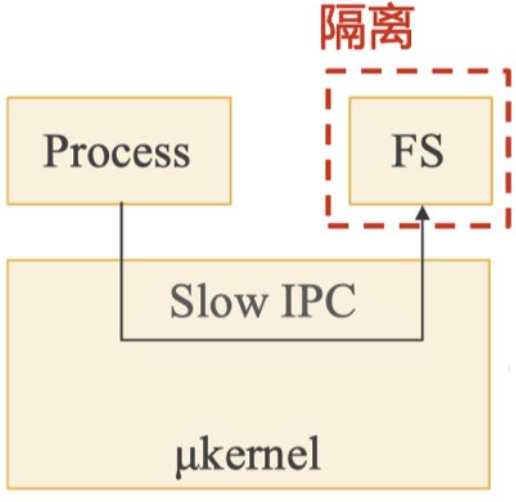
宏内核难以满足的场景

- **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

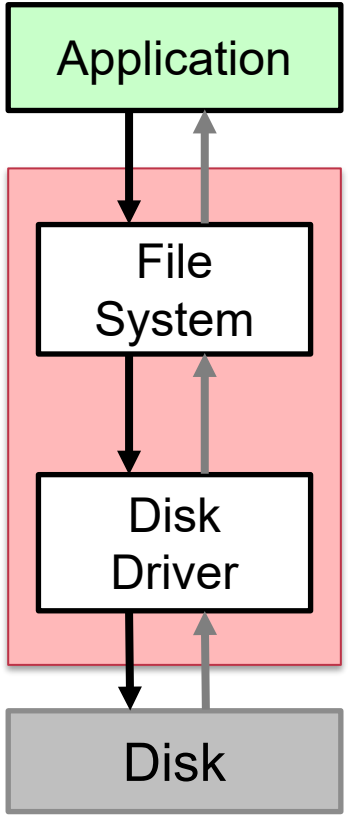
微内核（MICRO-KERNEL）

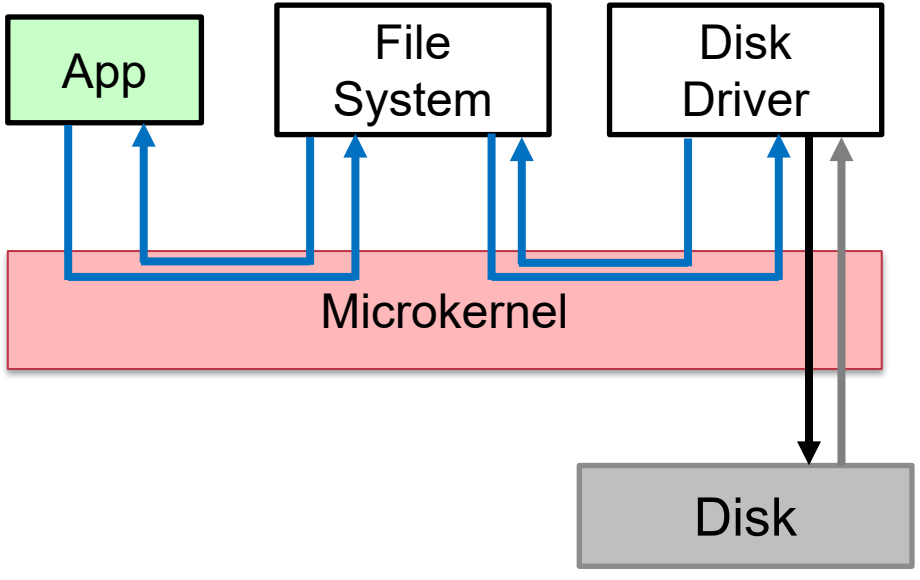
微内核的系统架构

- **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



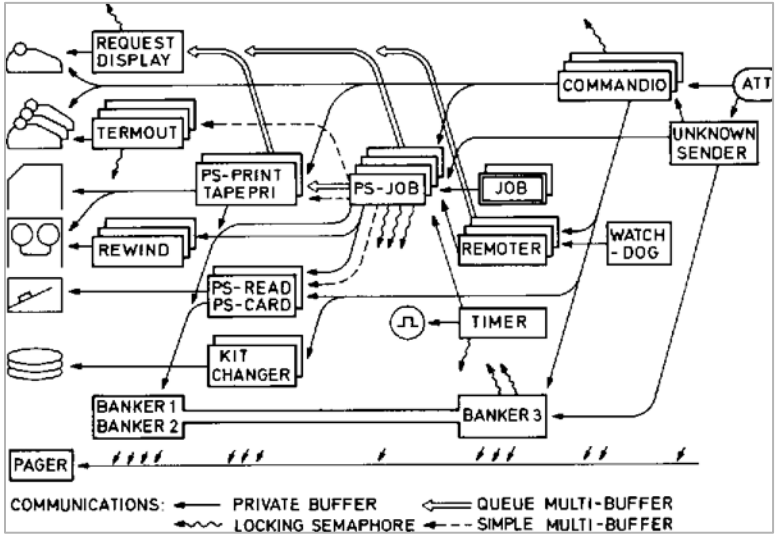
例：文件的创建


Application
File System
Disk Driver
Disk


App
File System
Disk Driver
Microkernel
Disk

微内核的历史

- **1969年，RC 4000多路编程系统**
  - 提出模块化设计，允许模块间交互
  - 提出复杂消息通信机制用于交互
  - 提出"分离策略与机制"的原则
  - 提出"管程"（Monitor）的概念
  - Per Brinch Hansen等开发者
  - 启发了后来的微内核



https://www.semanticscholar.org/paper/A-large-semaphore-based-operating-system-Lauesen/f028001d7426f7ae6a71f8deee9935a66cba4e04

Mach微内核

Rick Rashid

- **1985年，Mach 发布**
  - 由CMU开发，Rick Rashid领导
  - 对操作系统发展产生了重大影响
- **1986年，Mach 2.5**（性能比UNIX差25%）
  - 包含大量BSD的代码，如1:1的task与process映射，导致内核比UNIX更大
  - 取得了商业成功，用于NeXT，最终被苹果收购
- **1990年，Mach 3.0**（性能比UNIX差67%）
  - 规避法律风险，去掉了BSD的代码，重写了IPC以提高性能
  - 提出**"continuation"**，为用户态应用提供了更多控制
    - 允许应用自己在切换的时候保存/恢复上下文，进一步减小microkernel

Mach实现了哪些功能?

- **任务和线程管理**
  - 任务，是资源分配的基本单位；线程，是执行的基本单位
  - 对应用提供调度接口，应用程序可实现其自定义的调度策略
- **进程间通信（IPC）**：通过端口（port）进行通信
- **内存对象管理**：虚拟内存
- **系统调用重定向**：允许用户态处理系统调用
  - 支持对系统调用的功能扩展，例如，二进制翻译、跟踪、调试等

Mach还实现了哪些功能?

- **设备支持**
  - 通过IPC实现（通过port来连接设备）
  - 支持同步设备和异步设备
- **用户态的多进程**
  - 类似用户态的线程库，支持wait()/signal()等原语
  - 一个或多个用户态线程可映射到同一个内核线程
- **分布式支持**
  - 可透明地将任务与资源映射到集群中的不同节点

Mach：用户态与内核态的分工

- **Mach允许用户态代码实现Paging**
  - 应用可自己管理自己的虚拟内存
- **重定向功能（Redirection）**
  - 允许发生中断/异常时，直接执行用户的二进制
  - 这种连接不需要对内核做修改

L3/L4：极大提升IPC的性能

Jochen Liedtke

- **L4的IPC性能比Mach快20倍**
  - IPC仅传递信息
  - 使用寄存器传参，限制消息长度
  - 内核去掉了IPC的权限检查等功能，交给用户态判断
  - 系统服务的接口直接暴露给用户态，可能导致DoS攻击
- **启发了大量相关系统**
  - Pistachio、L4/MIPS、Fiasco等

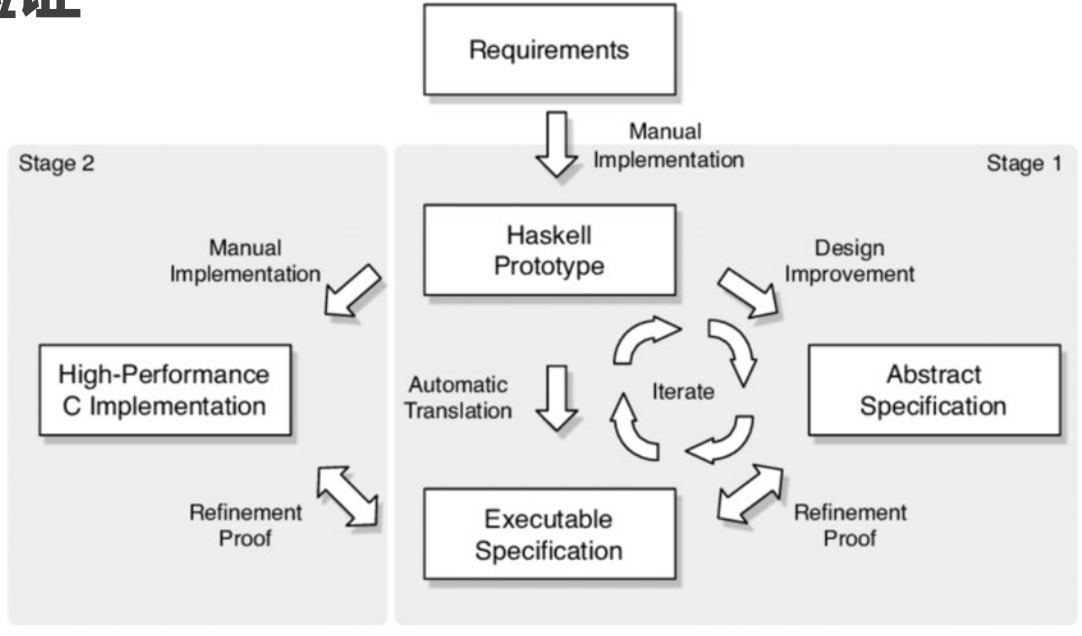
seL4：被形式化证明的微内核

Gernot Heiser

- **基于L4的微内核**
- **IPC机制：端点（endpoint）**
  - 通过Capability进行IPC的权限判断
  - Capability可被复制和传输
- **第一个完成形式化验证的内核**
  - 8700行C，bug-free!
  - 没有缓冲区溢出、空指针等错误
  - bug的定义：取决于specification

seL4：被形式化证明的微内核

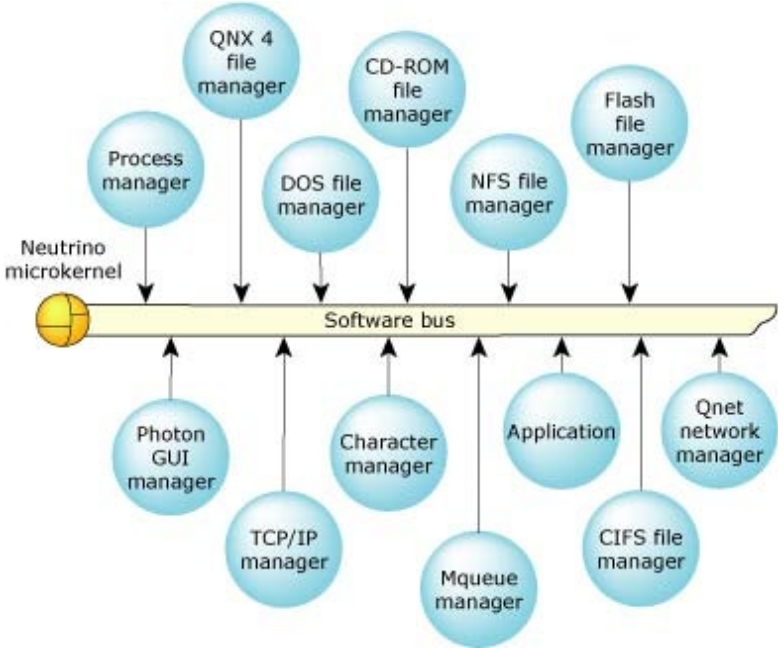
- **对C的限制，以方便验证**
  - 栈变量不得取引用
    必要时用全局变量
  - 不使用函数指针
  - 不适用union
- **用Haskell构造原型**
  - 用于验证
  - 再手动转换为C



Comprehensive Formal Verification of an OS Microkernel, 2004

QNX Neutrino

- **QNX：Quick UNIX**
  - 使用Neutrino微内核
  - 1980年发布
  - 2004年被Harman国际收购
  - 2010被黑莓收购
- **满足实时要求**
  - 广泛用于交通、能源、医疗、航天航空领域，如波音



http://www.qnx.com/

Google Fuchsia

- **Google开发的全新OS**
  - 试图覆盖多个领域，具体用途未知
- **使用Zircon微内核**
  - 仅提供IPC、进程管理、地址空间管理等功能

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/

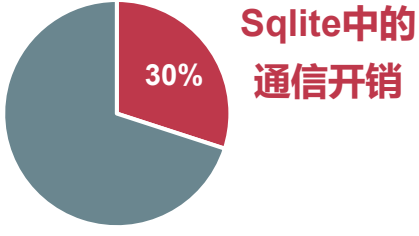
微内核的优缺点分析

- **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**

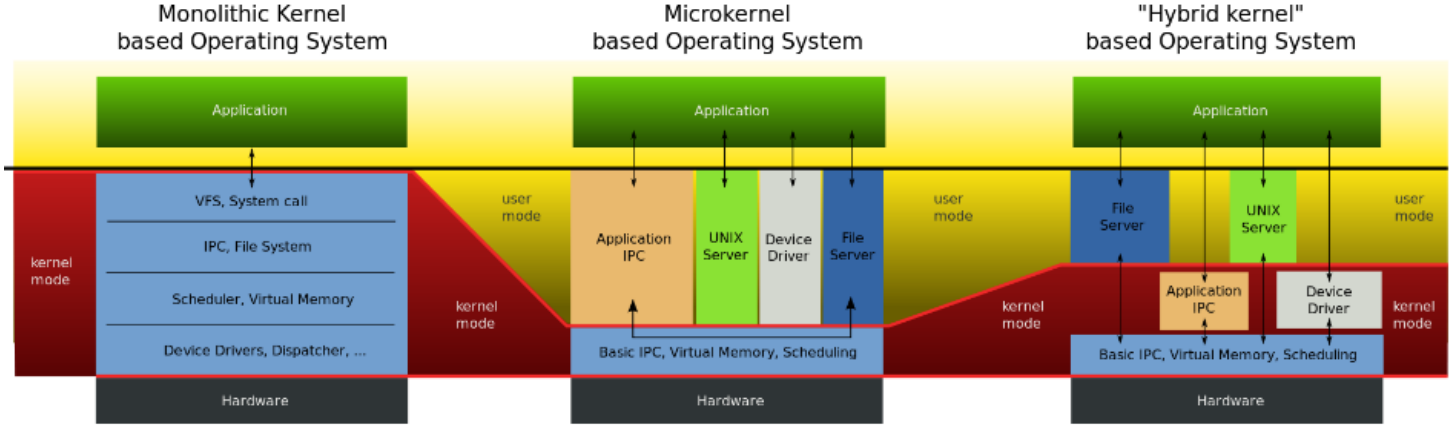
微内核的优缺点分析

- **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



混合内核架构

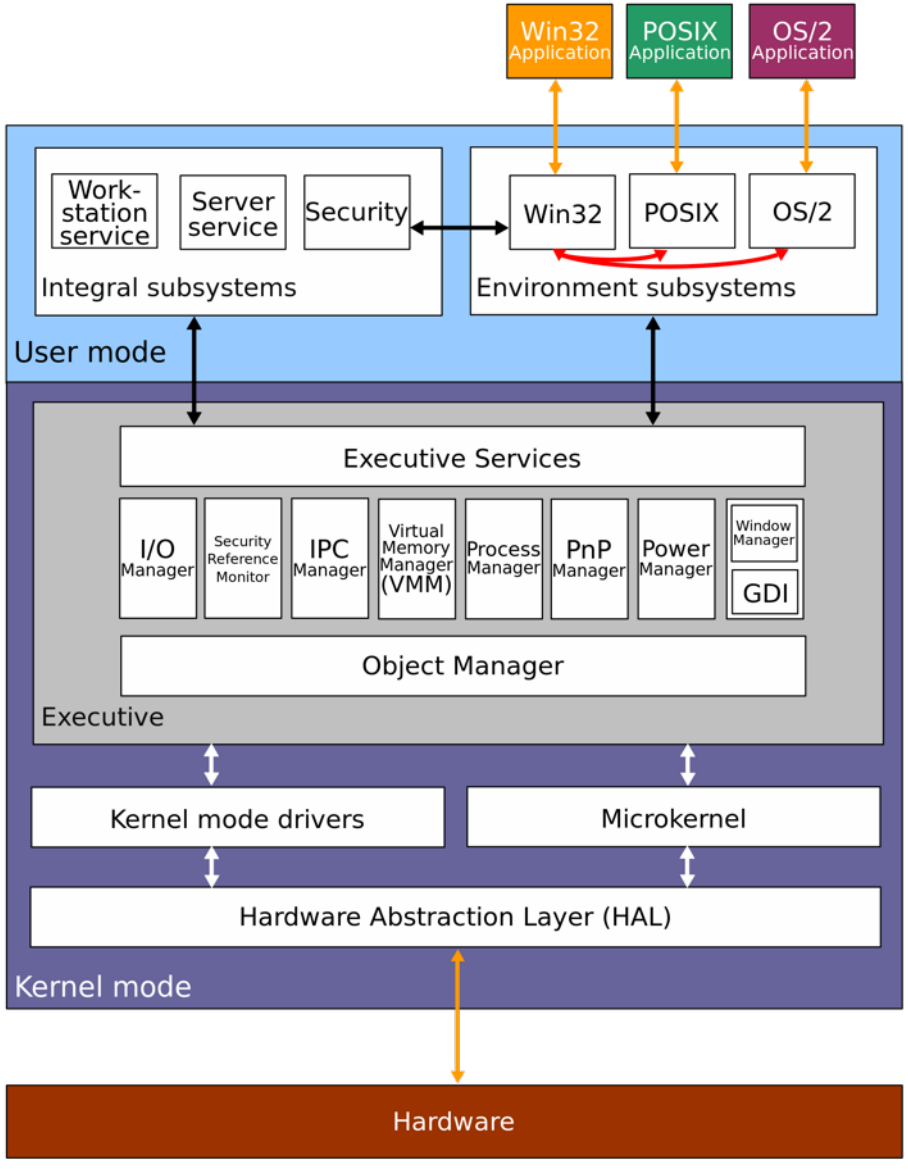
- **宏内核与微内核的结合**
  - 将需要性能的模块重新放回内核态
  - 例：macOS / iOS = Mach微内核 + BSD 4.3 + 系统框架
  - 例：Windows NT = 微内核 + 内核态的系统服务 + 系统框架



https://zh.wikipedia.org/wiki/微内核

Windows NT

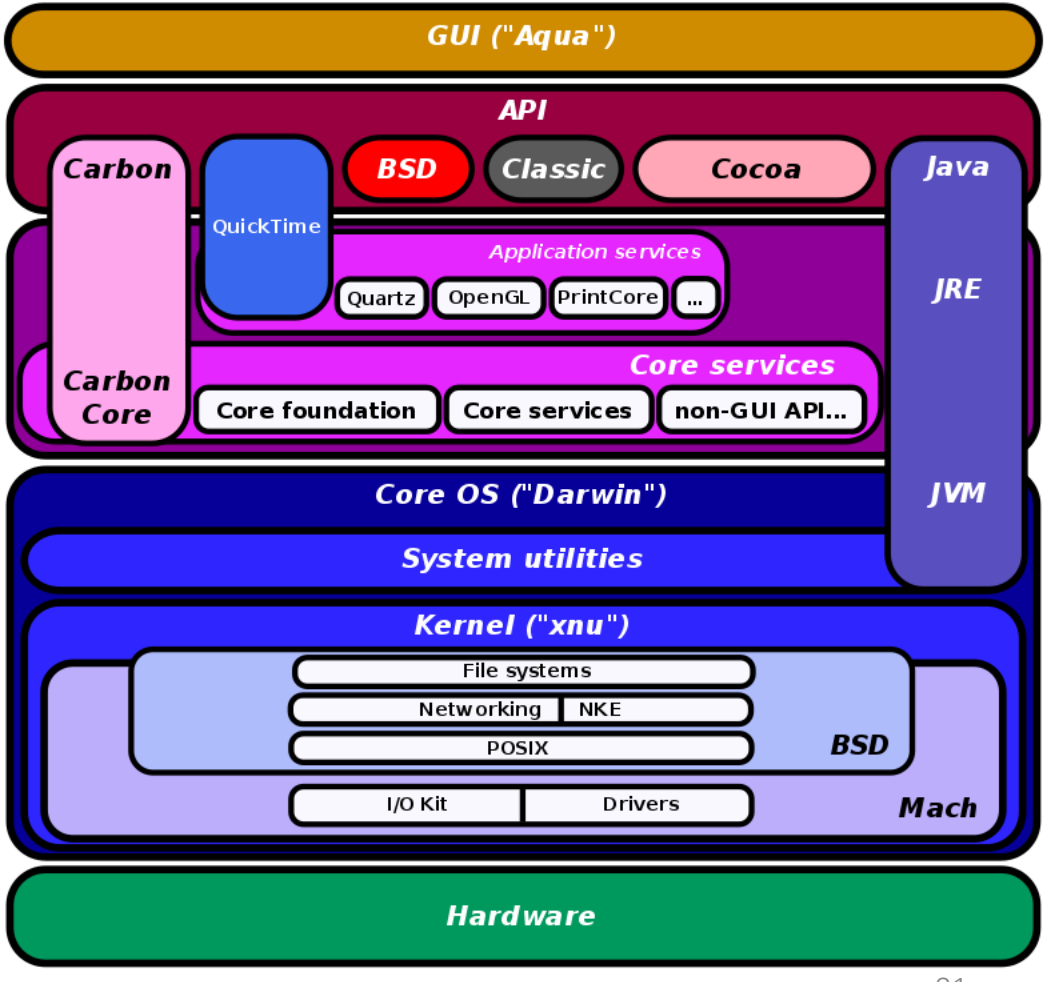
- **Integral子系统（用户态）**
  - 负责处理I/O、对象管理、安全、进程等
- **环境子系统（用户态）**
  - POSIX
- **Executive（内核态）**
  - 为用户态子系统提供服务
- **Microkernel**
  - 提供进程间同步等功能



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

外核+库OS（EXOKERNEL + LIBOS）

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

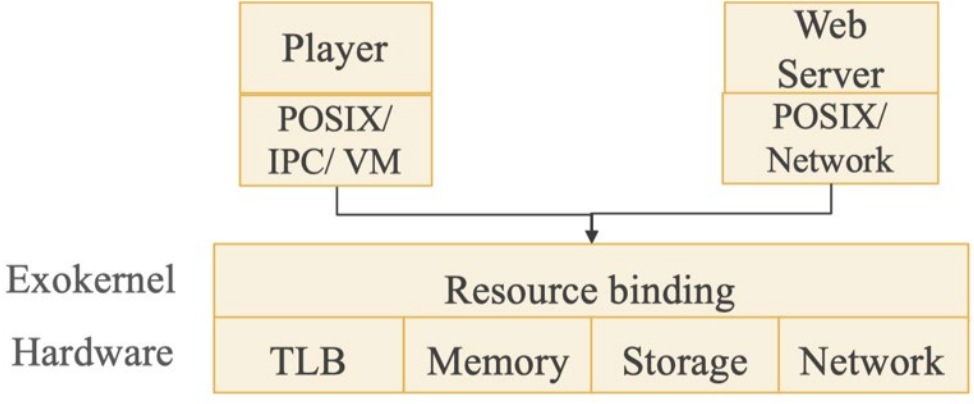
内核态：Exokernel

- **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



Exokernel架构的设计

- **外核的功能**
  - 追踪计算资源的拥有权
  - 保证资源的保护
  - 回收对资源的访问权
- **对应的三个技术**
  - 安全绑定（Secure binding）
  - 显式回收（Visible revocation）
  - 中止协议（Abort protocol）

设计原则："将管理与保护分离"

安全绑定

- **将LibOS与计算资源绑定**
  - 可用性：允许某个LibOS访问某些计算资源（如物理内存）
  - 隔离性：防止这些计算资源被其他LibOS访问
- **Software caching**
  - 例：利用software TLB保证LibOS只使用了自己的物理内存
    - LibOS可直接修改页表，因此有可能会将自己的页表指向其他LibOS的屋里页
    - Software TLB是软件可控的TLB，MIPS等处理器支持
    - 每次发生TLB miss时，由Exokernel负责遍历页表并填写对应的TLB项
    - Exokernel可在填写TLB项时检查LibOS对内存的使用是否合法

显式资源回收与中止协议

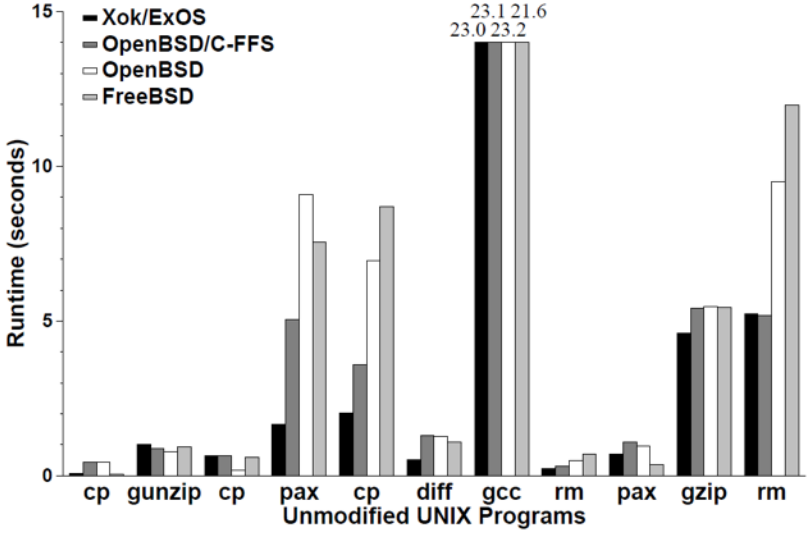
- **Exokernel与应用之间的协议**
  - Exokernel显式告知应用资源的分配情况
  - 应用在租期结束之前主动归还资源
- **若应用不归还资源，则强制中止**
  - Exokernel拥有对资源的控制权
  - 主动解除资源与应用间的绑定关系

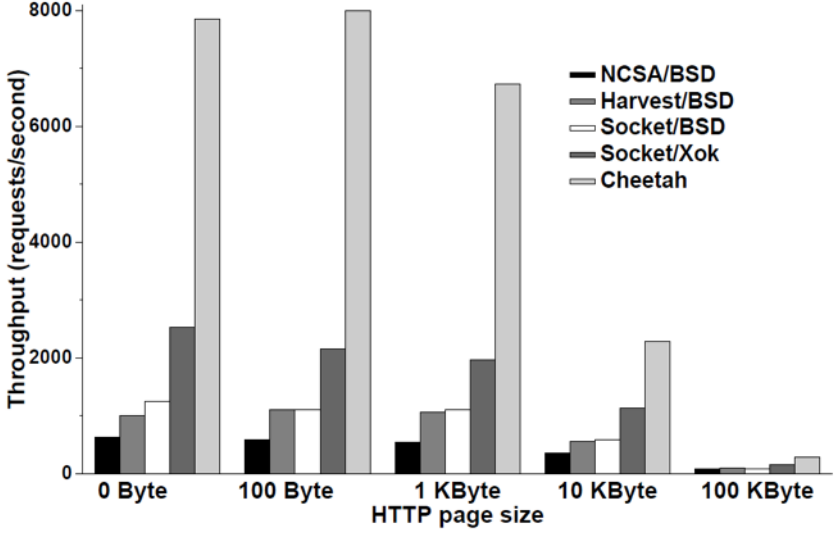
例：Exokernel对磁盘的管理

- **应用程序**
  - 管理磁盘块设备的page cache
- **Exokernel**
  - 允许应用之间安全的共享page cache

Exokernel的性能提升

- 未修改应用性能最多提升4x，定制化应用性能最多提升8x





数据来源：Application Performance and Flexibility on Exokernel Systems, SOSP'97

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

Unikernel的部分开源项目

- **Rumprun**
  - POSIX接口，BSD兼容的运行时环境
  - 运行在Xen虚拟化平台之上
- **Drawbridge**
  - 来自微软，兼容Win32接口的运行时环境
- **OSv**
  - 与Linux兼容的应用环境，单地址空间

Linux as a LibOS?

- **将Linux作为LibOS或Unikernel**
  - 例：LKL - Linux kernel library (https://github.com/lkl)
  - 将系统调用变为普通函数调用
  - 可提供一定的兼容性，有效避免重复开发
- **许多新问题**
  - Linux是否适合作为LibOS/unikernel?
  - fork()如何处理?
  - 尚在探索阶段

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

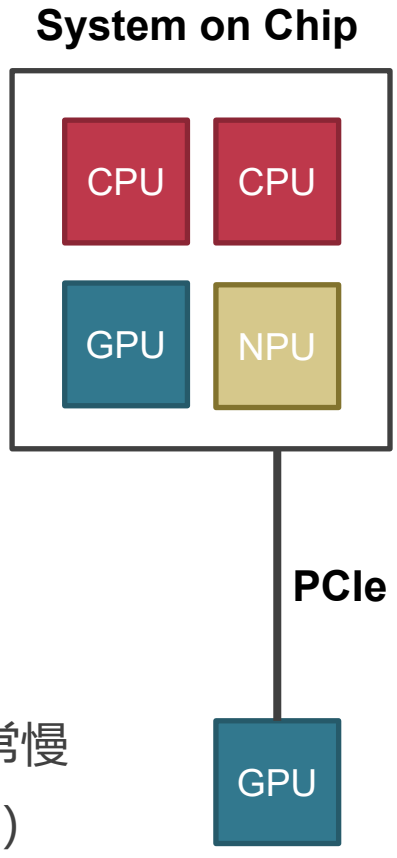
Exokernel架构的优缺点分析

- **Customer-Support:**
  - "*Extensibility has its problems. For example, it makes the customer-support issues a lot more complicated, because you no longer know which OS each of your customers is running*" (Milojicic, 1999).

多内核/复内核（MULTI-KERNEL）

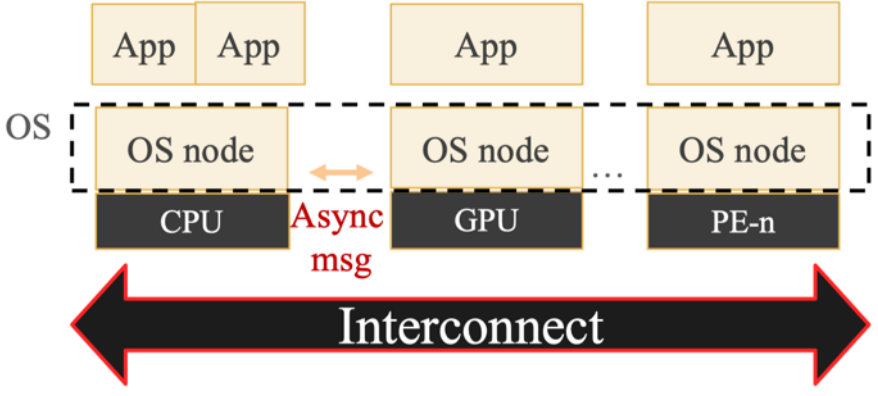
多内核/复内核（Multikernel）

- **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

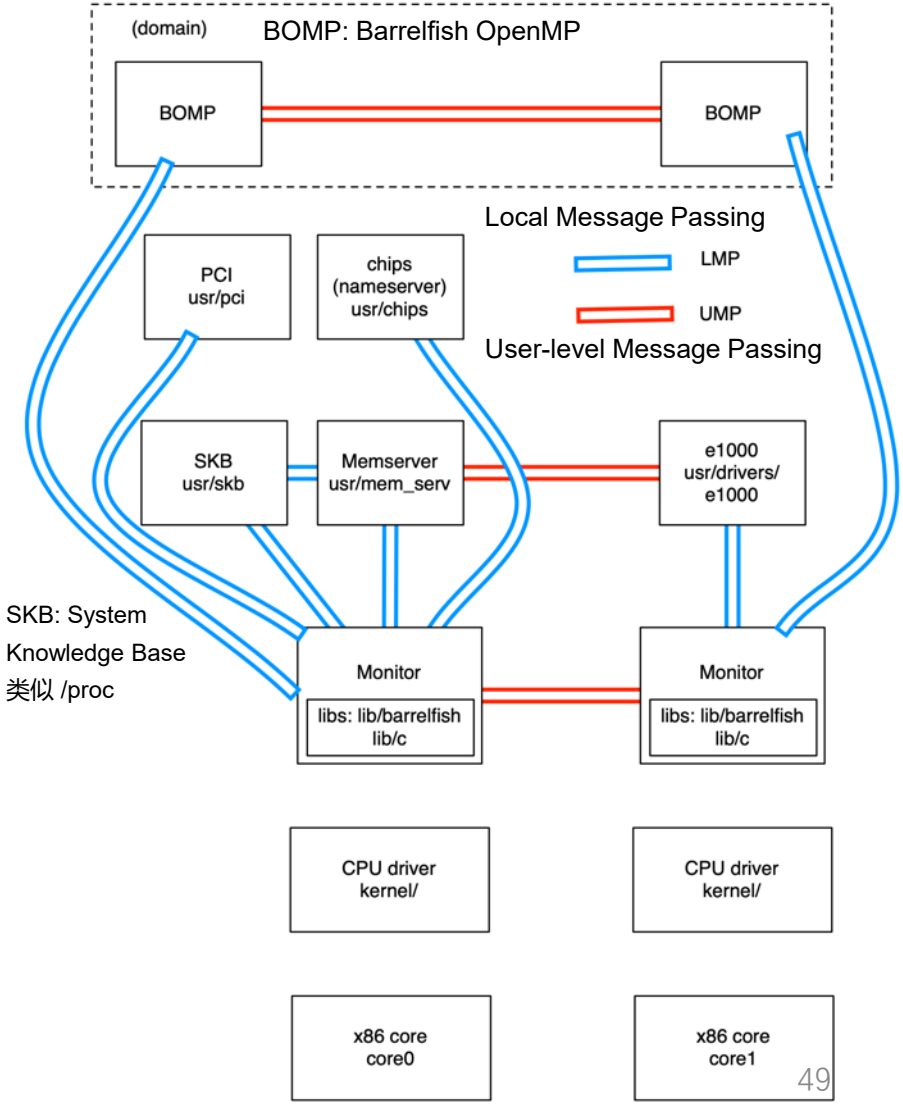


Barrelfish Multikernel

- **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

Barrelfish Structure

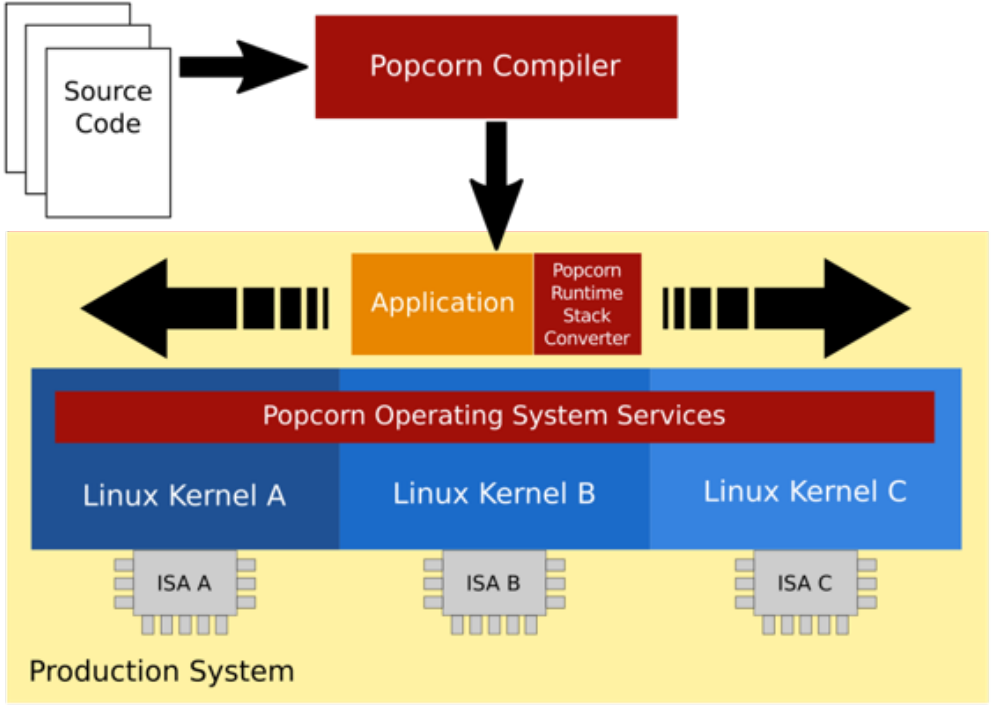
- **内核：每个core对应一个**
  - 类似"CPU驱动"，适应不同CPU
  - 负责执行系统调用，处理中断/异常
  - 事件触发，单线程，不可中断
  - 内核调度并运行"Dispatcher"
- **Dispatcher**
  - 类似线程
  - 多个Dispatcher组成一个Domain
- **Domain**
  - 类似进程



http://www.barrelfish.org/documentation.html

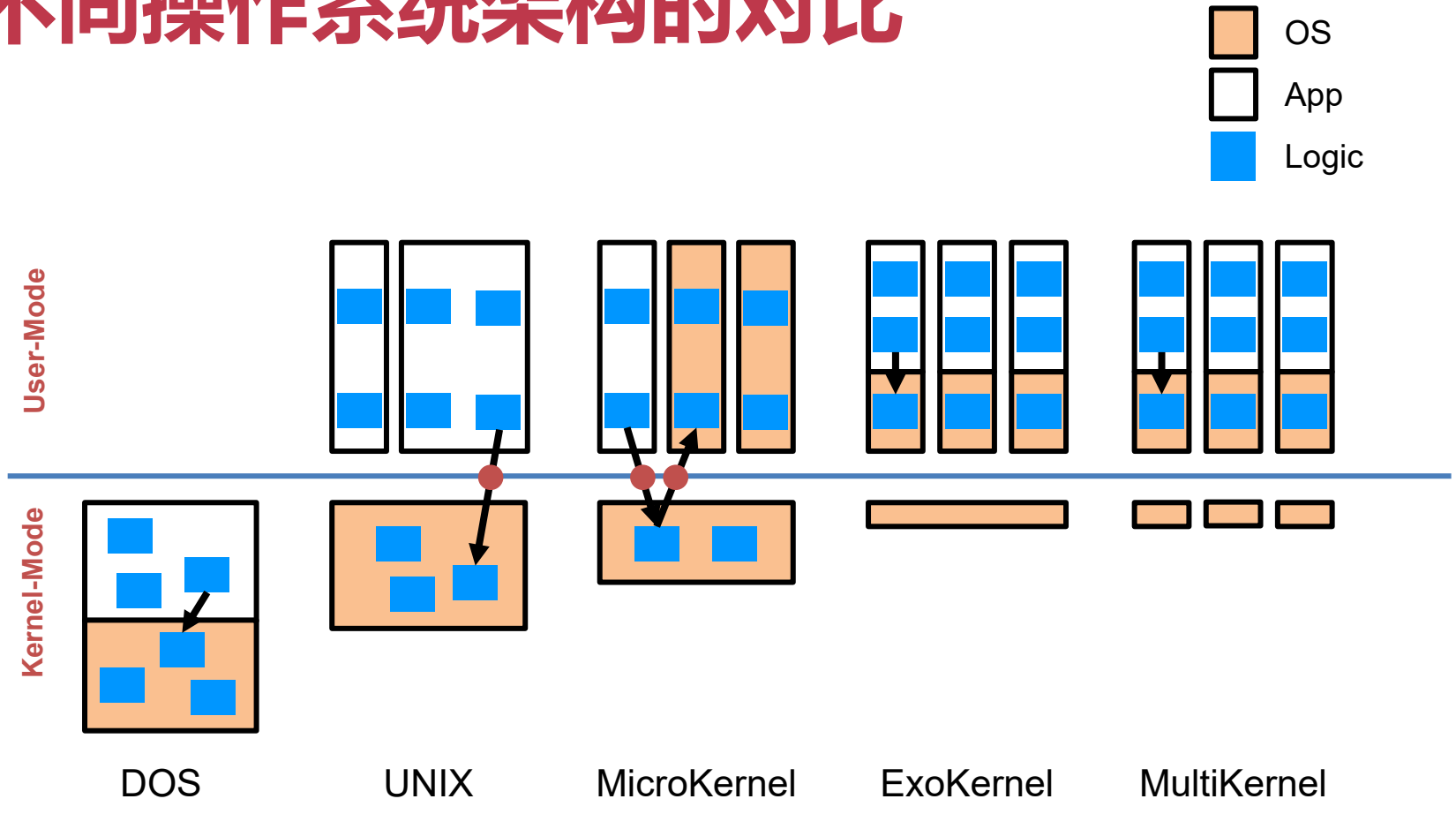
Popcorn Linux

- **支持异构体系结构**
  - ARM、x86等
- **多个Linux内核副本**
  - 一套代码编译不同副本
  - 不同ISA不同副本
  - 多个副本同时向上提供OS服务



小结

不同操作系统架构的对比



操作系统结构的演进与生态

- **系统软件需要一条演进之路**
  - 尽可能集成现有的POSIX API/Linux ABI
  - 避免棘手的系统调用（如fork）
  - 避免不可扩展的POSIX API
- **系统软件一直在不断演化**
  - 例：Linux Userspace I/O (UIO)，向微内核近了一步
  - 单节点下也存在更多的分布式、低时延的可编程设备
  - 非易失性内存的出现可能推动存储层次在OS中的完全改革