

操作系统简介

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
 - 资料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>



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
- 你觉得应当如何定义操作系统？

操作系统是在硬件和应用之间的软件层

应用

操作系统

硬件

操作系统和应用：

- 应用功能越来越多
- 操作系统沉淀越来越多功能、内涵与外延不断扩大

操作系统和硬件：

- 身体 vs. 灵魂

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)]

从 Hello World 说起

```
#include <stdio.h>

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

运行hello时，操作系统的作用？

```
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
```

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加载到 CPU 中运行？
- 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 程序同时运行的过程中如何在一个 CPU 中运行？
- ...

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一些服务

- **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 (如spinlock)
 - 操作系统提供了更高效的同步原语 (与线程切换配合 , 如pthread_mutex)
- **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)
 - Shell Pipe的例子

操作系统对应用的管理

- **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

问题

- **问题一**：如果一台机器有且只有一个应用程序，开机后自动运行且不会退出，是否还需要操作系统？
- **问题二**：如果一个应用希望自己完全控制硬件而不是使用操作系统提供的抽象，是否还需要操作系统？

操作系统=管理+服务

- 管理和服务的目标有可能存在冲突
 - 服务的目标：单个应用的运行效率最大化
 - 管理的目标：系统的资源整体利用率最大化
 - 例：单纯强调公平性的调度策略往往资源利用率低
 - 如细粒度的round-robin导致大量的上下文切换

操作系统的定义

- **操作系统的根本功能：**
 - 将有限的、离散的资源，高效地抽象为无限的、连续的资源
- **从软件角度的定义：**
 - 硬件资源虚拟化+管理功能可编程
- **从结构角度的定义：**
 - 操作系统内核+系统框架

应用与操作系统的交互：系统调用

- **什么是系统调用？**
 - 应用调用操作系统的机制，实现应用不能实现的功能
- **例如：Printf() -> write()->sys_write()**
 - `write(1, "Hello World!\n", 13)`
- **使应用调用操作系统的功能就像普通函数调用一样**

>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（其他的0代表标准输入，2代表标准错误），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)
```

系统调用过程

应用程序

```
#include <stdio.h>
int main() {
    printf("Hello World!\n");
    return 0;
}
```

用户地址空间

libc

```
write(1, "Hello World!\n", 13)
{
...
/*参数处理*/
push $__NR_write /* 第一个参数：系统调用号 */
push $1 /* 第二个参数：file descriptor */
push "Hello World!\n"
push $13
syscall (不同平台对于的指令不同，例如ARM中的svc, x86中的sysenter, RISC-V中的
ecall)
...
}
```

下陷处理

```
sys_syscall:
...
call syscall_table[__NR_write]
...
```

内核地址空间

系统调用处理

```
sys_write {
...
return error_no;
}
```

操作系统的功能：管理

- 避免一个流氓应用独占所有资源
- 方法：每10ms发生一个时钟中断
 - 决定下一个要运行的任务
- 方法：可以通过信号等打断当前任务执行 (kill -9 1951)

Rogue-1.c

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

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

操作系统的功能：管理

- 如何卡死一个OS?
 - 例: rogue-1.c 可以fork出无数的进程
- 如何解决这个问题?
 - 资源配额 : cgroup/Linux
 - 虚拟化 : 虚拟机
 - 万能方法 : 重启机器
 - 制度约束 : AppStore的程序预审准入机制

这个是很现实的...

刚写完代码，就被开除了

只看楼主

收藏

回复



西伯利亚蓝眼睛



莽莽



```
/*
 * 获取下一天的日期
 * @return
 */
public static Date getNextDay() {
    try {
        Thread.sleep(24*60*60*1000);
    } catch (InterruptedException e) {
        e.printStackTrace();
    }
    return new Date();
}
```

是不是因为我没写注释



为什么学习操作系统

为什么学习操作系统？

- **操作系统是一个成熟的领域**
 - Windows曾经一统天下数十年
 - 让用户改变操作系统是很困难的
 - 我们是否需要新的操作系统？
- **"新的"操作系统不断出现**
 - Linux、Mac OS、Android、iOS、ROS...
 - 大疆用什么操作系统？谷歌数据中心用什么操作系统？

为什么学习操作系统？

- 操作系统是系统领域的基石
- 系统领域有大量的公司
 - 微软、谷歌、IBM、EMC、VMware、华为、阿里...
 - 谷歌的核心技术
 - 集群、GFS、MapReduce、BigTable
 - 都是系统领域的优秀工作
- 学好操作系统，**for fun and profit**
 - 优秀的公司，优秀的大学，更多的机会

图灵奖与操作系统的演变



Maurice Wilkes
1967年图灵奖



Frederick Brooks
1999年图灵奖



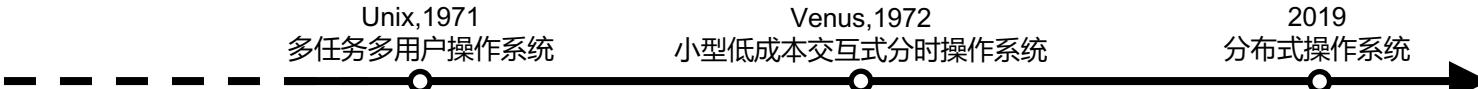
Fernando J. Corbató
1990年图灵奖



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

IBM System/360, 1964

CTSS, 1961 & Multics, 1969
分时操作系统



Unix, 1971
多任务多用户操作系统

Venus, 1972
小型低成本交互式分时操作系统

2019
分布式操作系统



Ken Thompson & Dennis Ritchie
1983年图灵奖



Barbara Liskov
2008年图灵奖



这门课程希望带给大家的能力

- 成为一个更高效的程序员
 - 更快找到并消灭bug的能力
 - 理解并调试程序性能的能力
- 理解复杂系统设计与实现的能力
- 构建一个操作系统的能力

课程的特点：抽象与具体

- **许多课程强调抽象**
 - 如：抽象数据类型、渐进分析等
- **这些抽象通常不可避免的带来限制**
 - 尤其是当bug存在的时候
 - 需要理解底层的实现细节，才能突破限制

“What I cannot create, I do not understand”
—— Richard Fynman (CIT) , Referred to by 陆奇

为什么操作系统比较难/有意思？

- 深入事情本质：直接管理硬件细节
 - 好处：实现资源的高效利用，从根本上解决问题，做一个高效程序员（降维）
 - 挑战：需要理解与处理硬件细节，硬件甚至可能出错
 - “把复杂留给自己、把简单留给用户”
 - 对比：用户态写一个Hello World和在操作系统内核中输出一个Hello World (Lab1)
- 锻炼系统架构能力
 - 将复杂问题进行抽象与化简
 - 最好的体现了M.A.L.H原则的使用
 - 计算机科学中30%的原则是从操作系统中来的 (13/41, greatprinciples.org)
- 各种问题的交互与相互影响
 - `fd = open(); ... ; fork();`

操作系统的机遇和挑战

- **新的硬件层出不穷**
 - 硬件种类越来越多：如自动驾驶、无人机、各种IoT设备等
 - 硬件互联越来越复杂：如AirDrop、摄像头和驾驶系统等
- **应用需求日新月异**
 - 对时延的要求越来越苛刻，如自动驾驶系统
 - 对资源利用率要求越来越高，如数据中心混部
- **新的硬件和应用呼唤新的抽象和管理——新的操作系统**

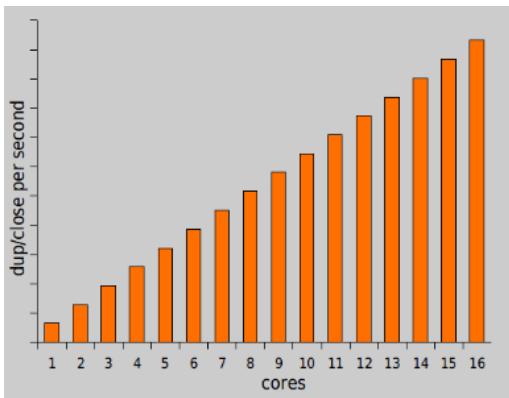


操作系统面临的挑战

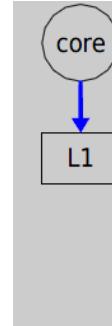
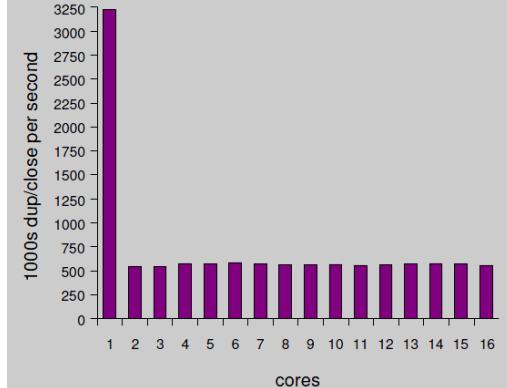
操作系统的可扩展性 (Scale-up)

例：文件描述符的可扩展性

理想性能

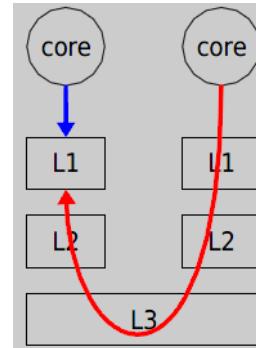


实际性能



```
fd_alloc(void) {  
    lock(fd_table);  
    fd = get_free_fd();  
    set_fd_used(fd);  
    fix_smallest_fd()  
    unlock(fd_table);  
}
```

从L1缓存读取fd，需要3个CPU周期。



```
fd_alloc(void) {  
    lock(fd_table);  
    fd = get_free_fd();  
    set_fd_used(fd);  
    fix_smallest_fd()  
    unlock(fd_table);  
}
```

现在，需要121个CPU周期！

隔离与性能的Tradeoff

- 进程间通信速度慢: 比函数调用慢16倍
 - 研究者对进程间通信的优化持续了**30+年**
 - IPC依然可占到**30%**的程序总运行时间 (e.g., Sqlite-3)

Function call



1. Prepare stack
 2. Save registers
 3. JMP
- ~30 cycles

Synchronized IPC



1. Domain* switch
 2. Message copying
- * domain means a process
- ~ 500 cycles (RISC-V)
~ 4000 cycles (4KB)

CVE-2015-8370



新浪科技 > 业界 > 正文

新闻 ▼

佟丽娅



Linux开机漏洞：连接28下Backspace可入侵系统

2015年12月21日 16:50 新浪科技 微博



CVE-2016-4484

Linux神奇漏洞：长按回车键70秒 Root权限到手

2016-11-17 21:32:29 作者：[上方文Q](#) 编辑：[上方文Q](#) 人气：18907 次 评论(18)

A- A+

让小伙伴们也看看：



21

收藏文章

一般来说获取系统Root权限是很困难的，尤其是加密系统中，但西班牙安全研究员Hector M arco、Ismael Ripoll发现，Linux系统下只需按住回车键70秒钟，就能轻松绕过系统认证，拿到Root权限，进而远程控制经过加密的Linux系统。

安全挑战 : do_brk() Bug

```
Asmlinkage unsigned long sys_brk(unsigned long brk) {  
    ...  
    /* OK, looks good – let it rip. */  
    - if (do_mmap(NULL, oldbrk, newbrk-oldbrk,  
    -                 PROT_READ|PROT_WRITE|PROT_EXEC,  
    -                 MAP_FIXED|MAP_PRIVATE, 0) != oldbrk)  
+     if (do_brk(oldbrk, newbrk-oldbrk) != oldbrk)  
        goto out;
```

- Purpose of do_brk() was to speed up anonymous mmaps from sys_brk()
- do_mmap() checked things properly, do_brk() removed almost all checking

The do_brk() Fix

```
if (!len)
    return addr;

+ if ((addr + len)) > TASK_SIZE || (addr + len) < addr
+ return -EINVAL;
+
```

- The TASK_SIZE is typically set to 0xc0000000 in 32-bit OS, the start of kernel memory

新硬件的机遇：非易失性内存

3D XPoint™ TECHNOLOGY

MEMORY

Technology	Latency	Size of Data
SRAM	1X	1X
DRAM	~10X	~100X
3D XPoint™	~100X	~1,000X

STORAGE

Technology	Latency	Size of Data
NAND	~100,000X	~1,000X
HDD	~10 MillionX	~10,000 X

The diagram illustrates the performance hierarchy of various memory and storage technologies. It uses a color gradient from red (highest latency) to green (lowest latency) to represent increasing performance. SRAM is at the top left, followed by DRAM, then 3D XPoint™ in the center, then NAND, and finally HDD at the bottom right. Each technology is accompanied by a representative image: a die for SRAM, a chip for DRAM, a 3D XPoint™ stack, a SSD for NAND, and a hard drive for HDD.

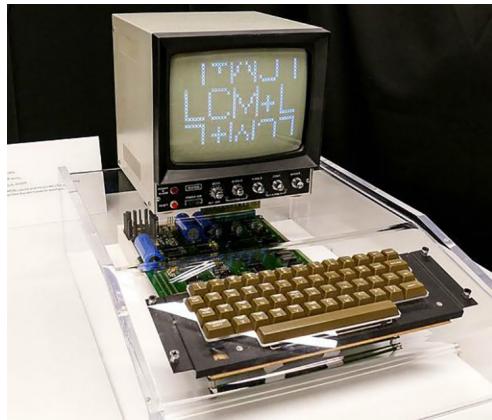
新硬件的机遇：非易失性内存

- 操作系统的抽象需要适应新的硬件特性
- 非易失性内存改变了操作系统的资源管理方式
 - 例如：奔溃一致性保证（fsck是否依然适用？）
- 提供更好的I/O性能



案例：APPLE操作系统技术演进

1976年：Apple I



Apple发布第一个桌面计算机Apple I

Apple I 的OS是最便宜、最快、最可靠、完全无bug的！

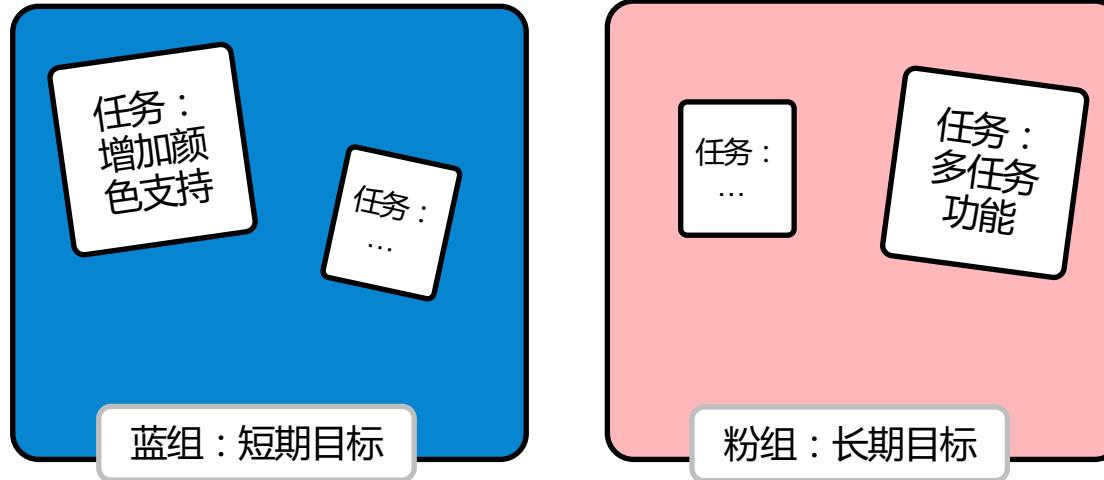
Apple I 没有OS!

The cheapest, fastest, and most reliable components of a computer system are those that aren't there.

— Gordon Bell

Image from <https://media.techeblog.com/images/apple-1-computer-steve-jobs-wozniak.jpg>

1987年：蓝组、粉组和 Mac OS 7

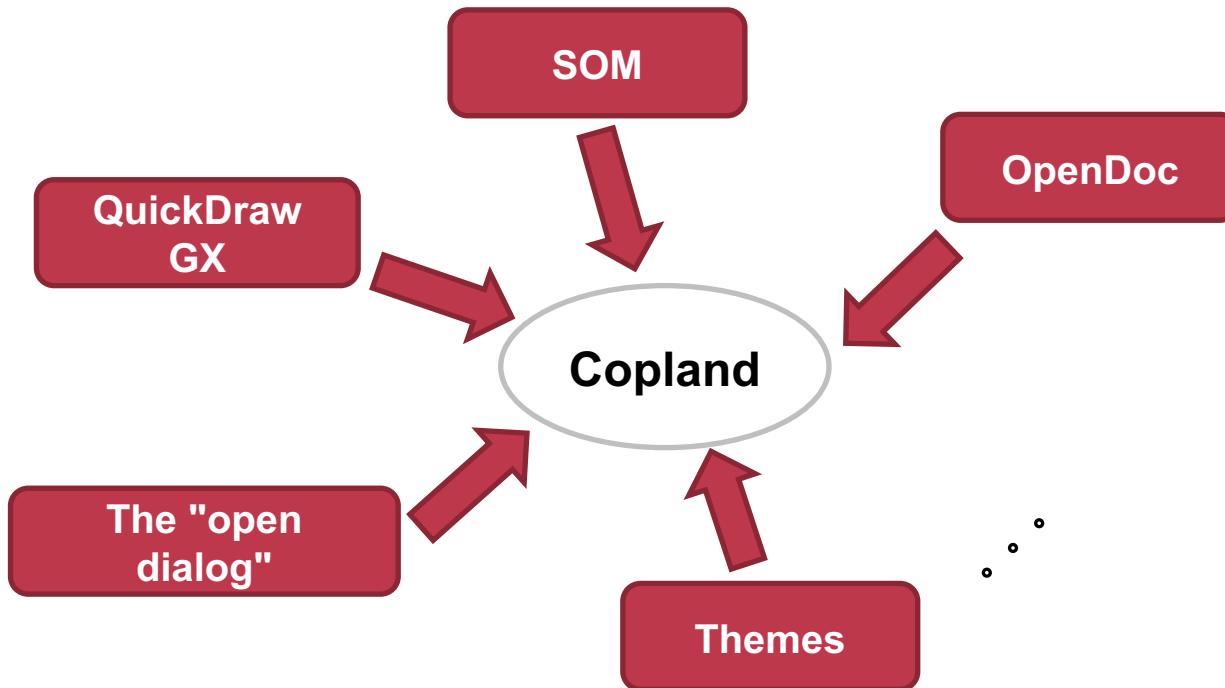


1991.5，蓝组顺利按时完成开发任务，Mac OS 7 发布

Timely

区分长短期目标，迭代增量式开发，系统成功上线

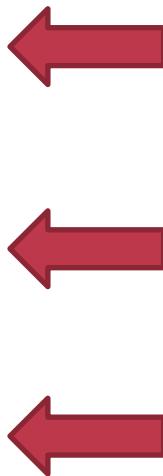
1994年 : Copland



!Simple

Do one thing well!
太多的“核心组件”被加入到Copland，导致项目最终失败

1998年：HFS+



压缩功能

Journaling支持

全盘加密
(FileVault2)



Timely

Dependable

Do it again. Make copies. Find consensus.
逐步增加功能，增强可靠性和安全性

2007年：iPhone OS 1 (iOS 1)



Aqua

SpringBoard

Application Frameworks

Application Frameworks

Core Frameworks

Core Frameworks

Darwin

Darwin

Adaptable

需求的改变：为支持终端设备的从MacOS中衍生出
iOS

2007年： LLVM-Clang

```
struct foo { int x; }  
typedef int bar;
```

GCC:

t.c:3: error: two or more data types in declaration specifiers

Clang:

t.c:1:22: error: expected ';' after struct

```
#include <inttypes.h>  
int64 x;
```

GCC:

t.c:2: error: expected '=', ',', ';', 'asm' or '__attribute__' before 'x'

Clang:

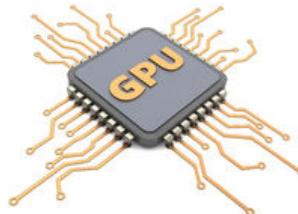
t.c:2:1: error: unknown type name 'int64'; did you mean 'int64_t'?
int64 x;^~~~~~int64_t

Simple

Yummy

Clang报错简单且对用户更加友好

2008年：Open CL



多核架构
浮点计算效率比传统 CPU 更高

编程复杂 → 需要新一代统一的 GPU 编程语言



OpenCL

第一个异构系统的通用并行编程的开放标准
统一的编程环境
编写的 GPU 程序比单核 CPU 能快上数十至数百倍

Efficient

Adaptable

根据硬件发展而改进，充分利用硬件性能

2010年：从iPhone到Apple TV



iPhone 1 (2007)



iPod Touch 2 (2008)



iPad (2010)



Apple TV (2010)

Adaptable

iOS适配到各种不同的硬件平台

2016年：APFS

高效文件复制

Efficient

针对SSD优化

Adaptable

数据完整性校验和加密

Security

Efficient

Adaptable

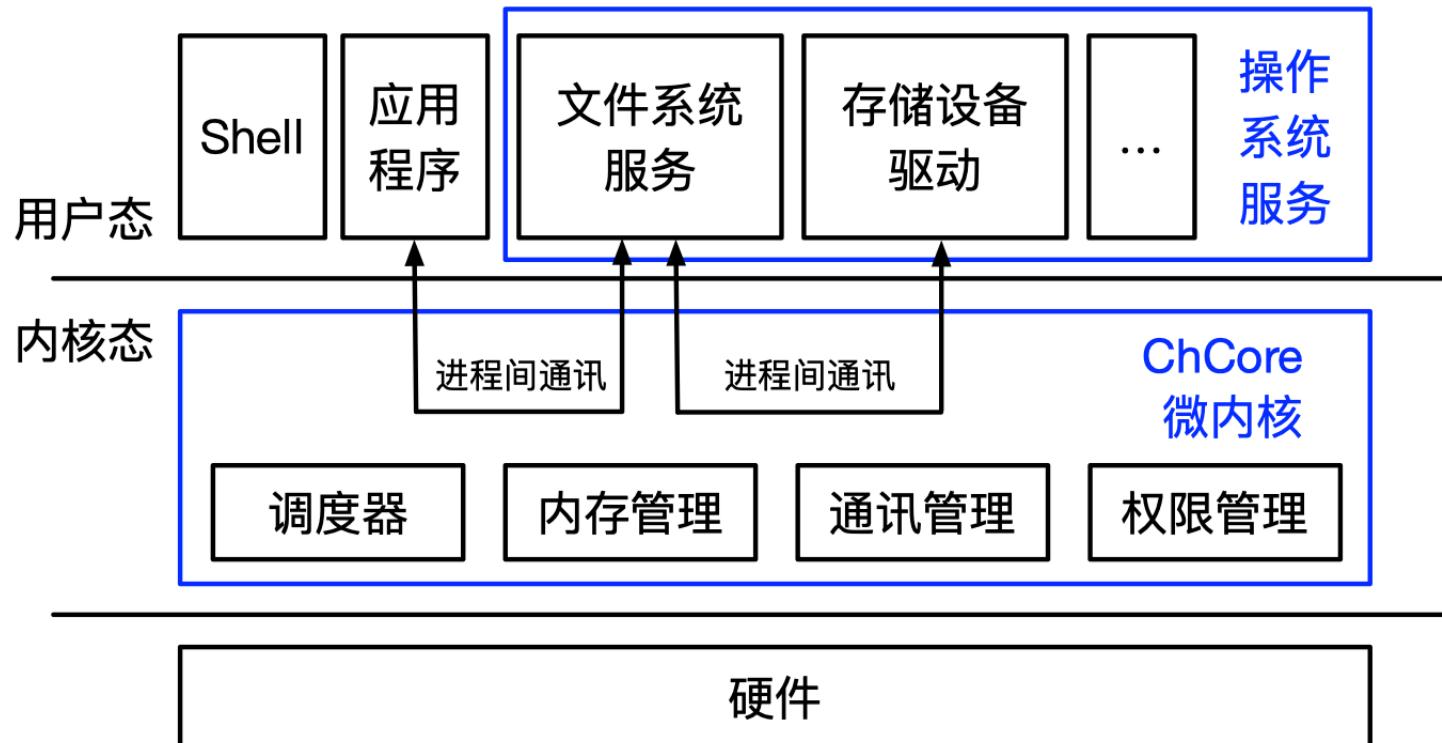
Dependable

针对新硬件优化、注重安全
和可靠性



CHCORE简介

ChCore采用了微内核架构



宏内核与微内核的崩溃概率比较

假设：

总模块数：100

单位时间内一个模块崩溃的概率：1%

概率比较：

宏内核

内核态模块数：100

单位时间内**宏内核正常工作**的概率：

$$(1 - 1\%)^{100} \approx 36.6\%$$

微内核

内核态：10

用户态：90

单位时间内**微内核正常工作**的概率：

$$(1 - 1\%)^{10} \approx 90.4\%$$

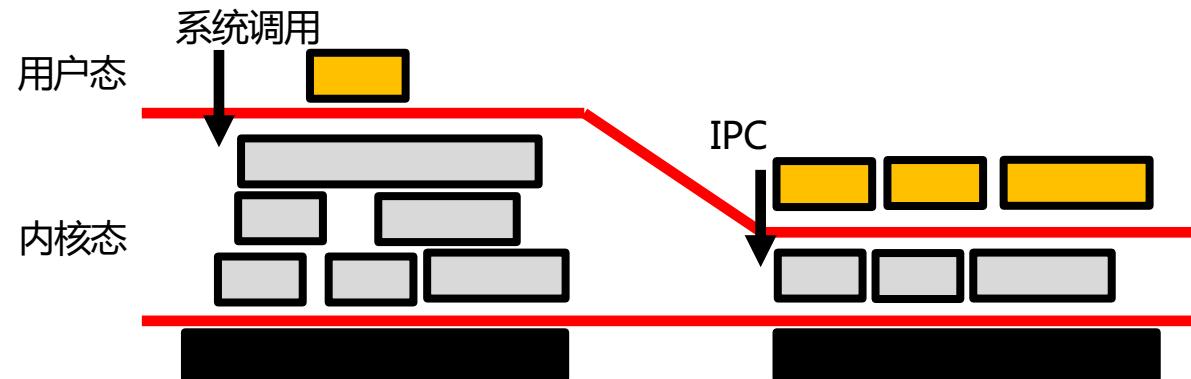
微内核更利于形式化验证，提升内核安全性

相比于宏内核，微内核有以下特点：

在内核态执行的代码量少得多

依靠IPC通信

更便于形式化验证来辅助提升内核的安全性



下次课内容

- 硬件结构与ARM64的系统接口