



操作系统结构与组成

中国科学院大学计算机学院

2025-09-17

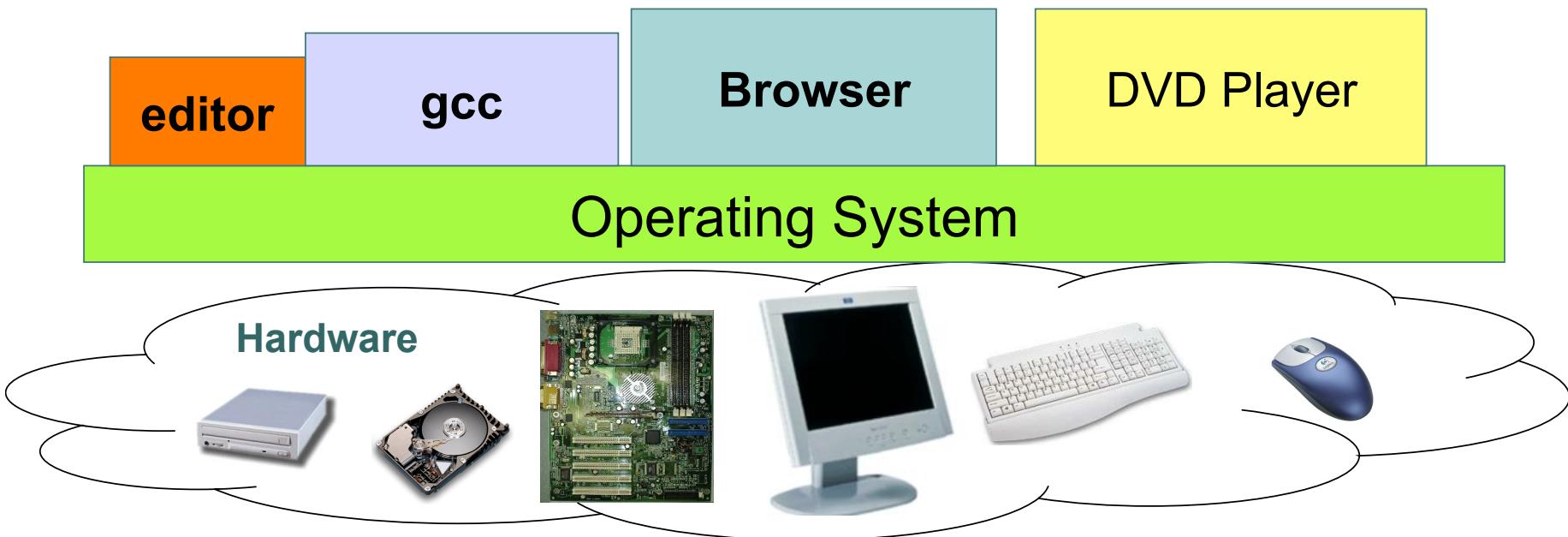




回顾Intro01：什么是操作系统？

- 承上启下

- 在应用和硬件之间的一层软件
- 对上层软件提供**硬件抽象**、实现**共用功能**和**操作接口**
- 对底层硬件进行**管理**、实现共享、保证隔离



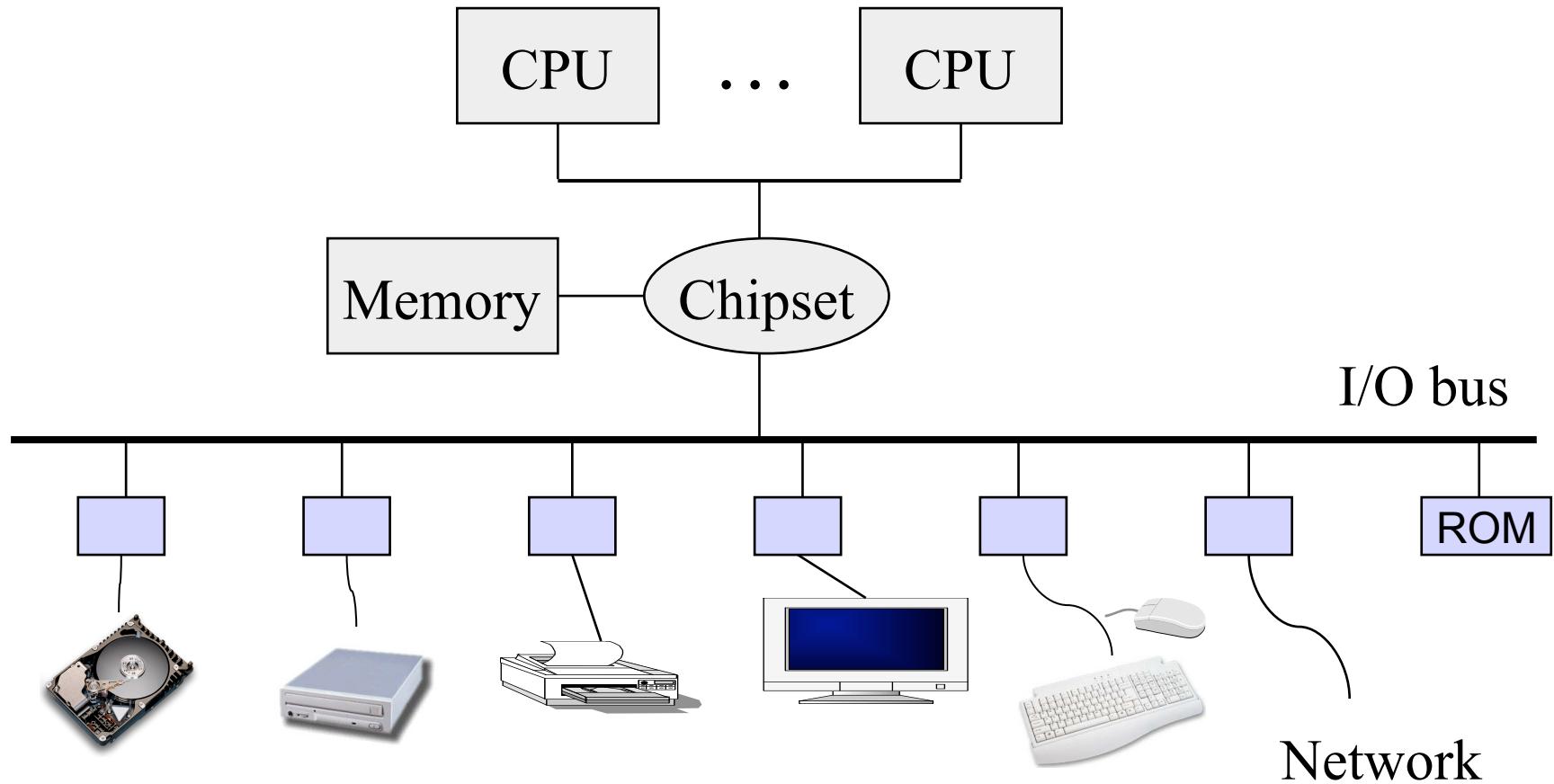


内容提要

- 操作系统结构
 - 操作系统启动
 - 内核态与用户态
 - 系统调用
 - 常见内核架构
- 操作系统组成

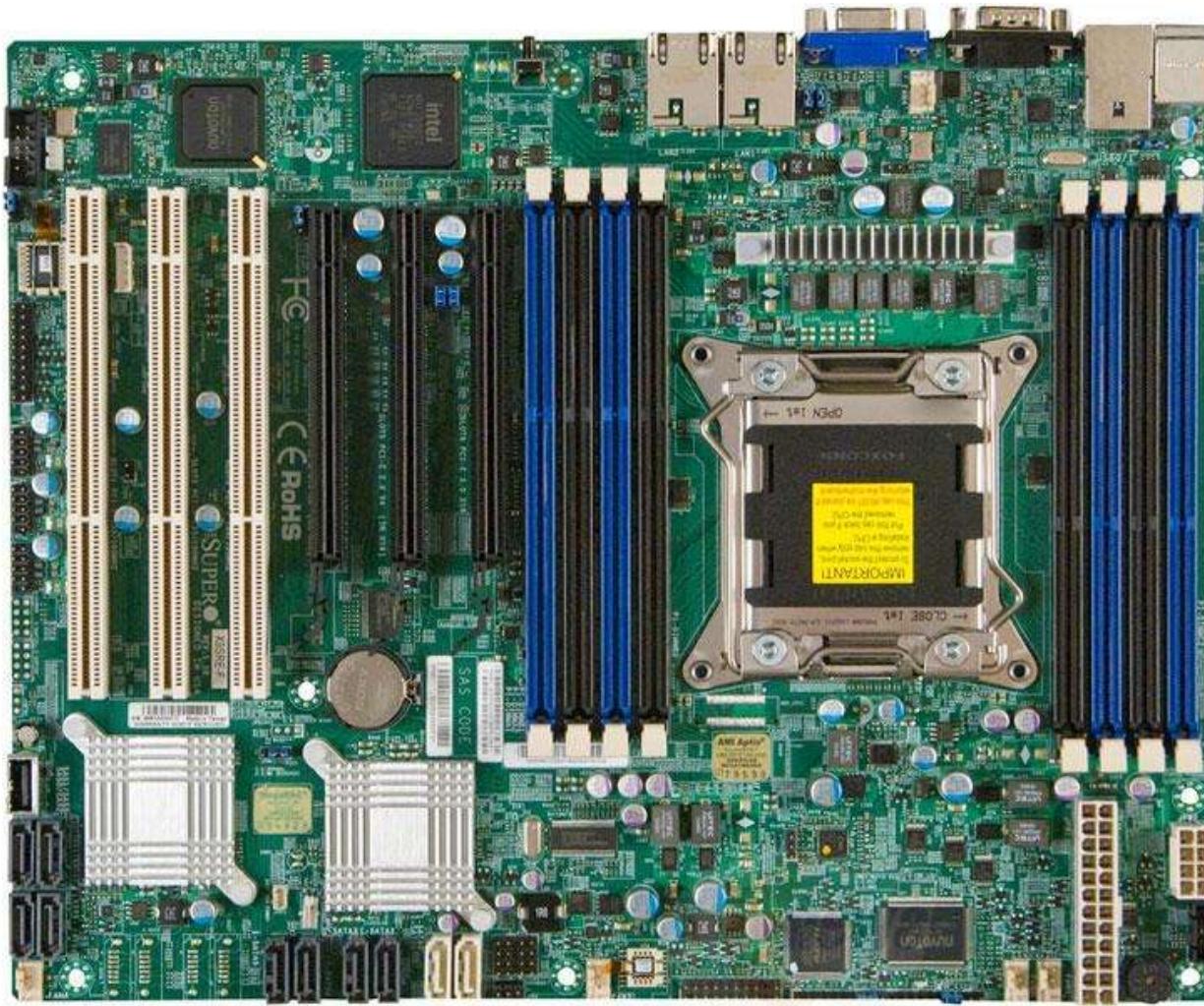


典型的计算机系统硬件结构



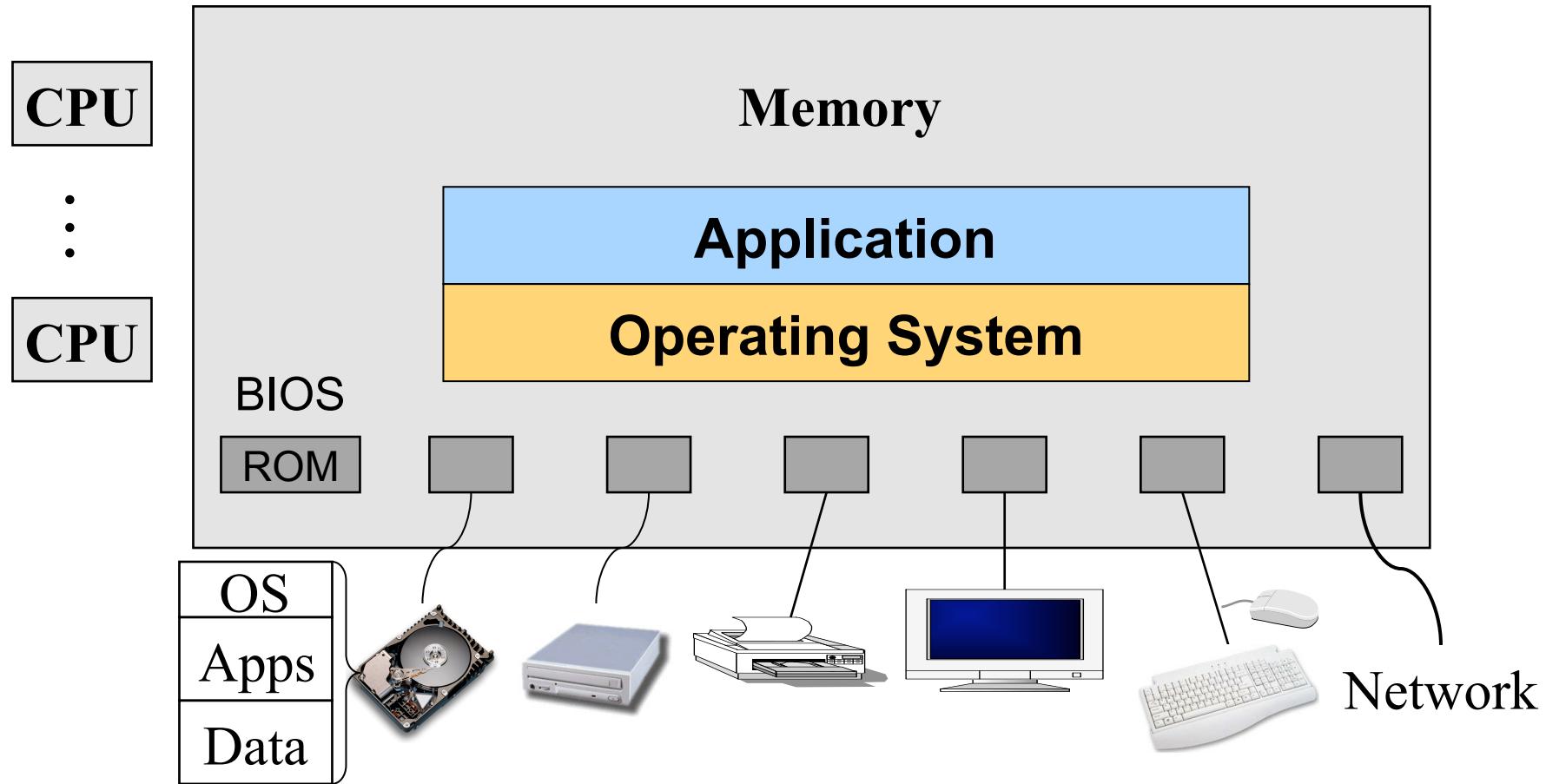


典型的计算机硬件主板





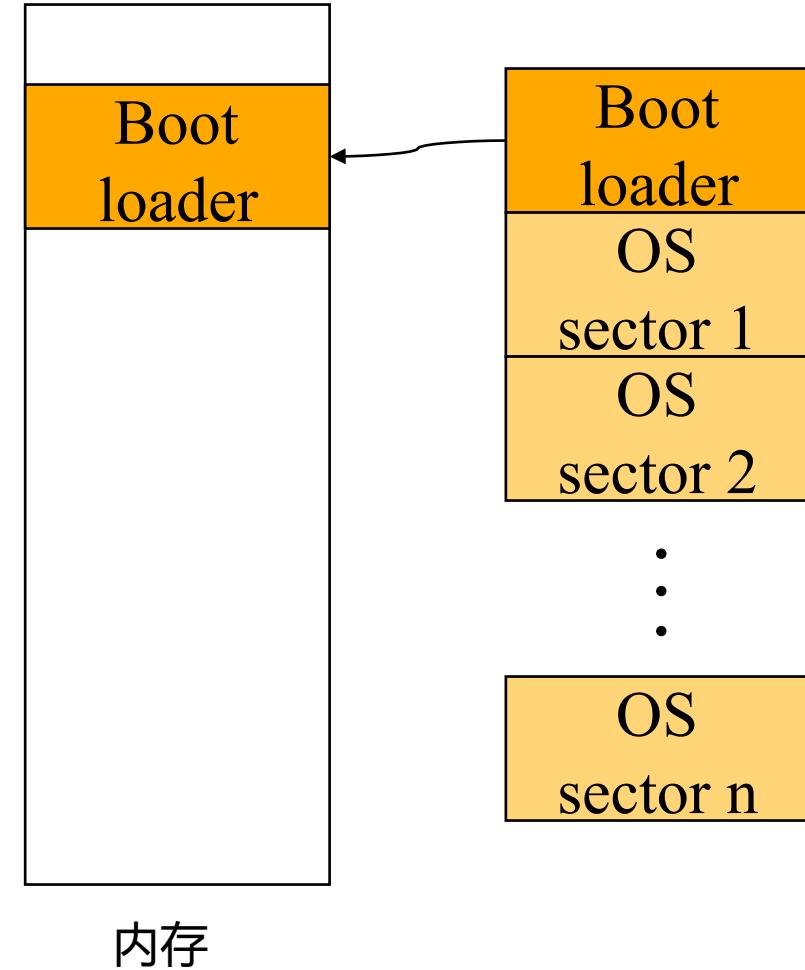
典型的计算机系统





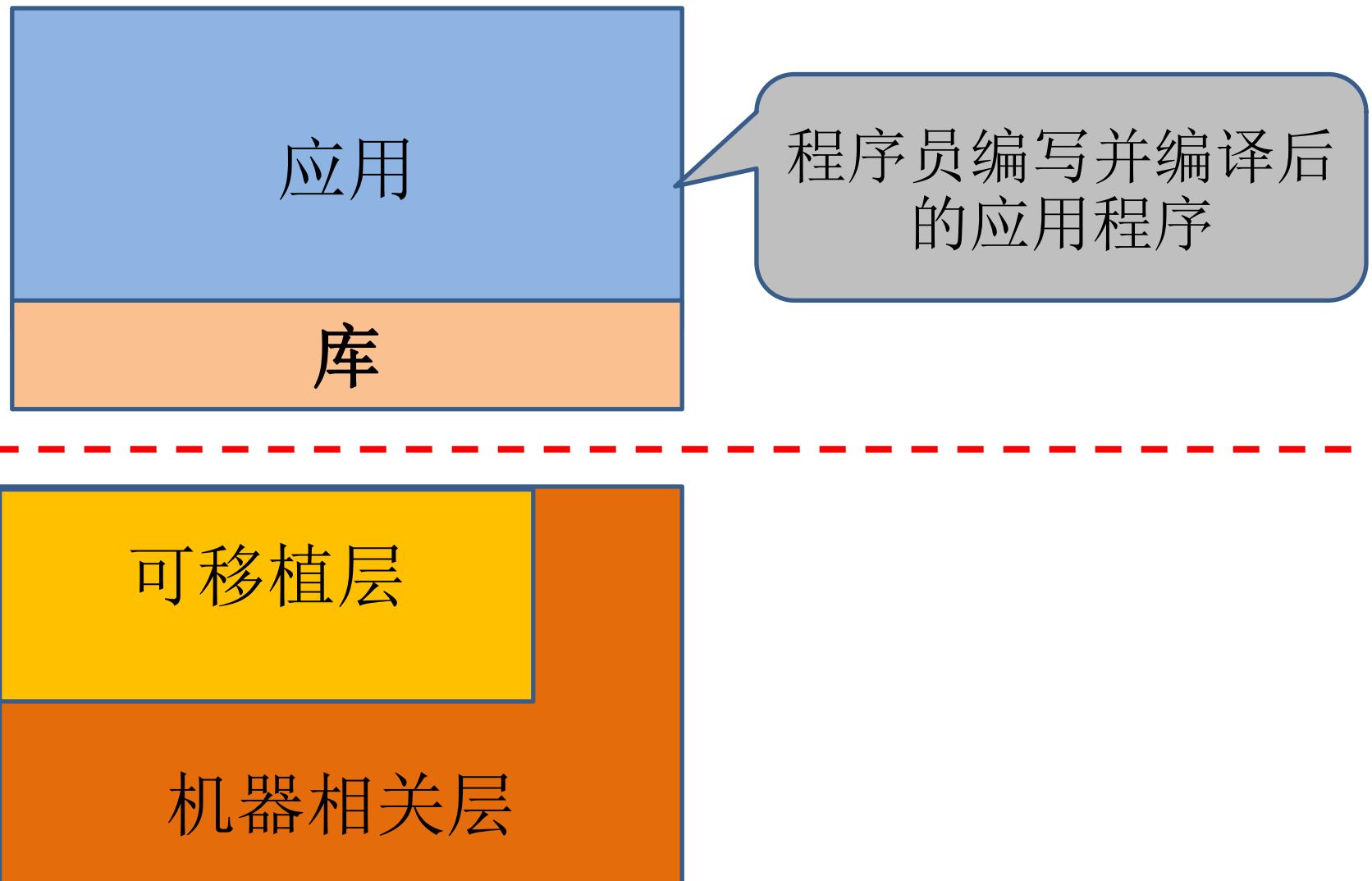
操作系统启动

- 计算机上电
- 处理器Reset
 - 设置到初始状态
 - 跳转到ROM代码 (BIOS)
 - 初始化启动所需最少的设备
- 从持久存储加载BootLoader
- 跳转到BootLoader执行
- 加载OS其余的部分
- 初始化OS并运行



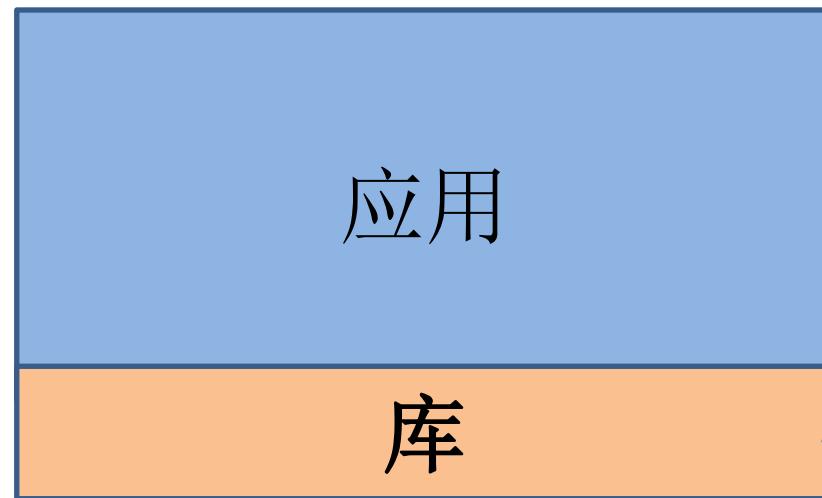


典型的UNIX操作系统结构





典型的UNIX操作系统结构



- 精心设计的代码
- 预编译好的对象
- 通过头文件定义
- 通过链接器引入
- 类似函数调用
- 程序加载时必须定位



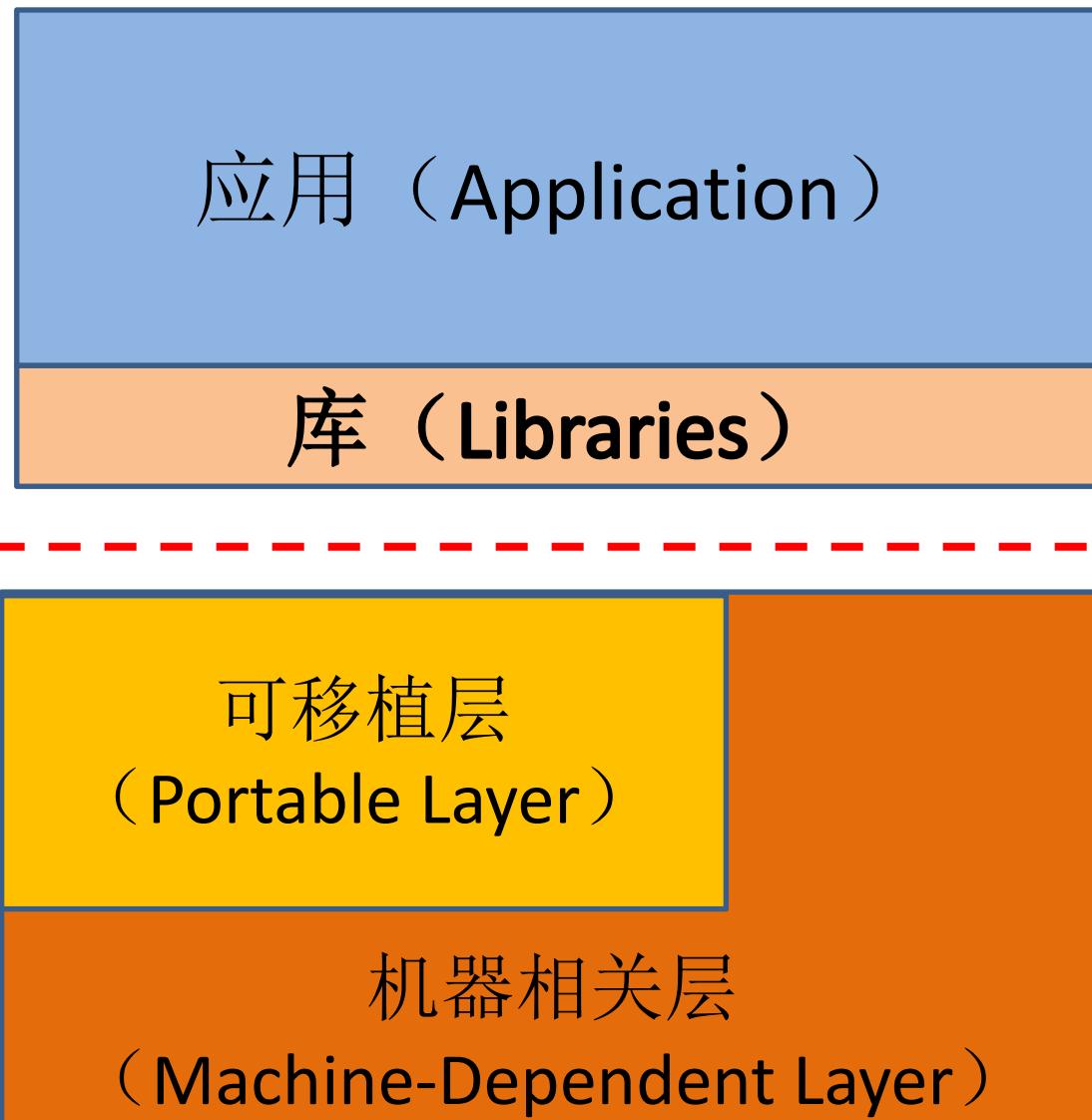


glibc库示例

- stdio.h
 - 标准输入输出 : printf, scanf, fopen, fclose, fread, fwrite 等
- stdlib.h
 - 通用工具函数: malloc, free, exit等
- string.h
 - 字符串处理 : strcpy, strcat, strlen, strcmp, memcpy 等
- unistd.h
 - 操作系统服务访问接口 : read, write, close等
- math.h, time.h等



典型的UNIX操作系统结构





为什么需要区分用户态和内核态

- 审视操作系统的功能范围
 - 应用程序生命周期管理、调度与切换
 - 内存资源分配、回收
 - 外部设备访问和控制
- 审视应用程序的操作边界
 - 应用程序A执行开关中断操作，是否允许？
 - 应用程序A修改OS的代码和数据结构，是否允许？
 - 应用程序A写入磁盘的任意位置，是否允许？



为什么需要区分用户态和内核态

- 操作系统设计的诉求
 - 能操作计算机系统的各类资源，同时让应用程序不能随意访问计算机资源
 - 操作系统自身不能受到应用程序的干扰
 - 隔离是操作系统设计的一个基本诉求
 - CPU指令执行隔离
 - 内存访问隔离



如何支持用户态和内核态

- CPU具有不同特权级
 - CPU设计有特权寄存器
 - MIPS : Coprocessor 0系列寄存器 , CP0_Status, CP0_Cause等
 - RISC-V : CSR系列寄存器 , mstatus , mcause , sstatus , scause等
 - 特权寄存器用于管理机器状态、中断、内存访问等
 - 特权寄存器需要使用特权指令在特定的特权级别进行访问
 - MIPS : mfc0 , mtc0等
 - RISC-V : csrr , csrw等
 - 在非特权级执行特权指令，会触发异常

特权级别	运行
M态	BIOS
S态	操作系统
U态	应用程序

RISC-V指令集

高4位值	运行级别
00	所有态
01	S态和M态
10	M态

RISC-V CSR寄存器高4位含义



内核态的保护机制

- 利用处理器特权级实现CPU指令执行隔离
- 利用页表机制实现内存访问隔离（后续介绍）

非特权级（用户态）

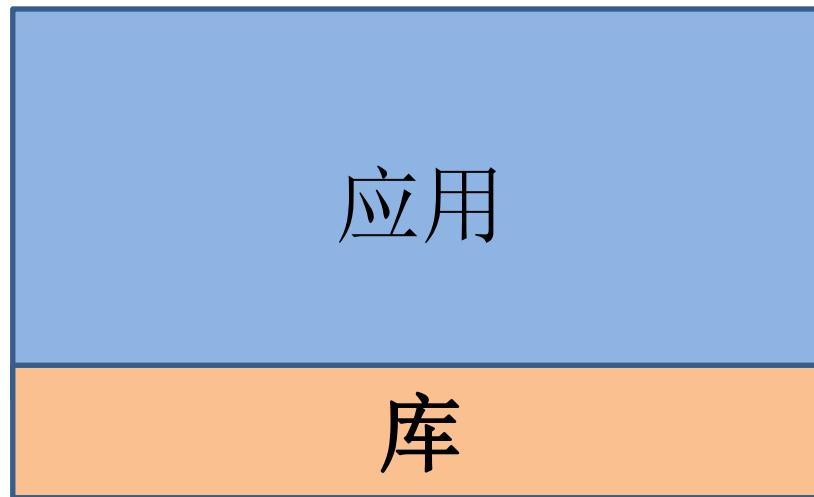
- 常规指令
- 访问用户内存

特权级（内核态）

- 常规指令
- 特权指令
- 访问用户内存
- 访问内核内存



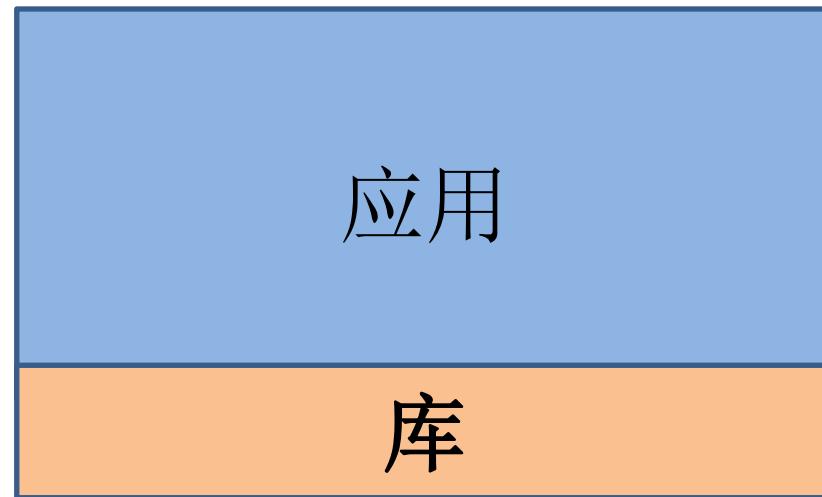
典型的UNIX操作系统结构



系统调用功能的集合



典型的UNIX操作系统结构



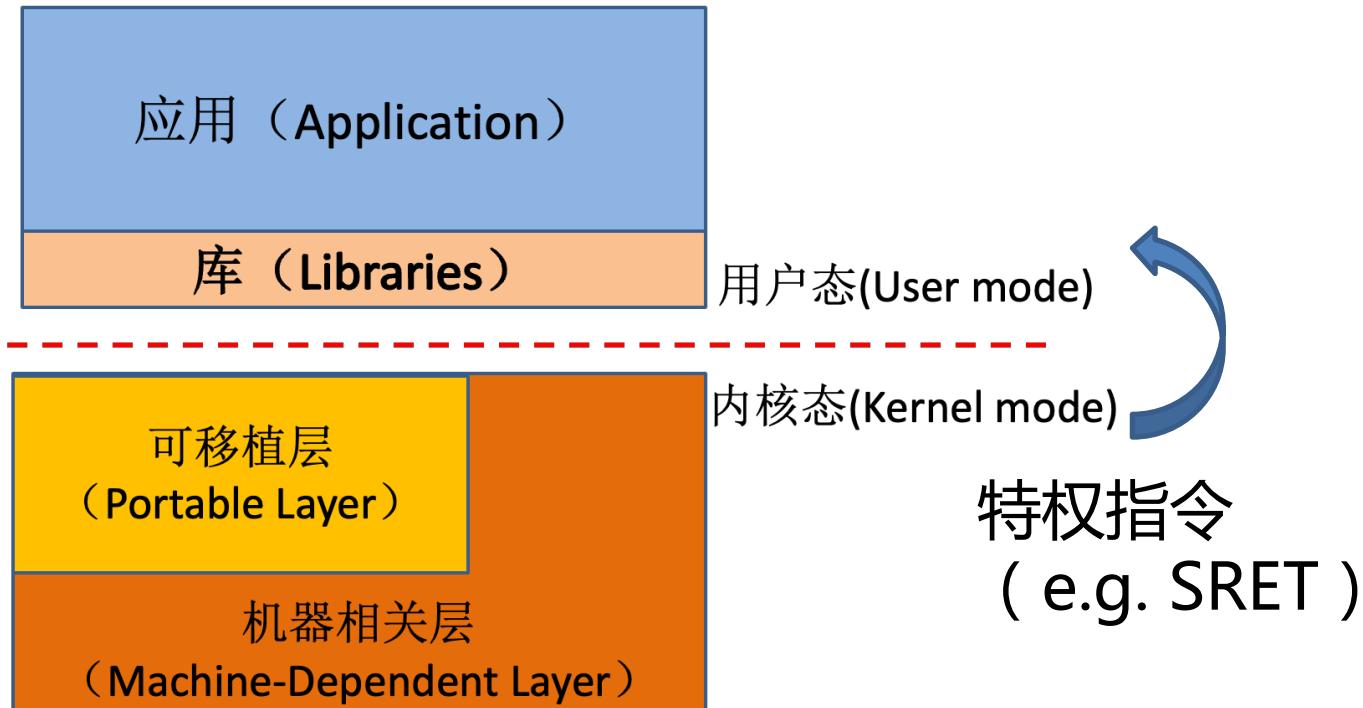
- 启动
- 初始化
- 中断和例外
- I/O设备驱动
- 内存管理
- 处理器调度
- 模式切换



用户态-内核态切换

- 特权级和非特权级切换

- 硬件中断
- 系统调用





异常

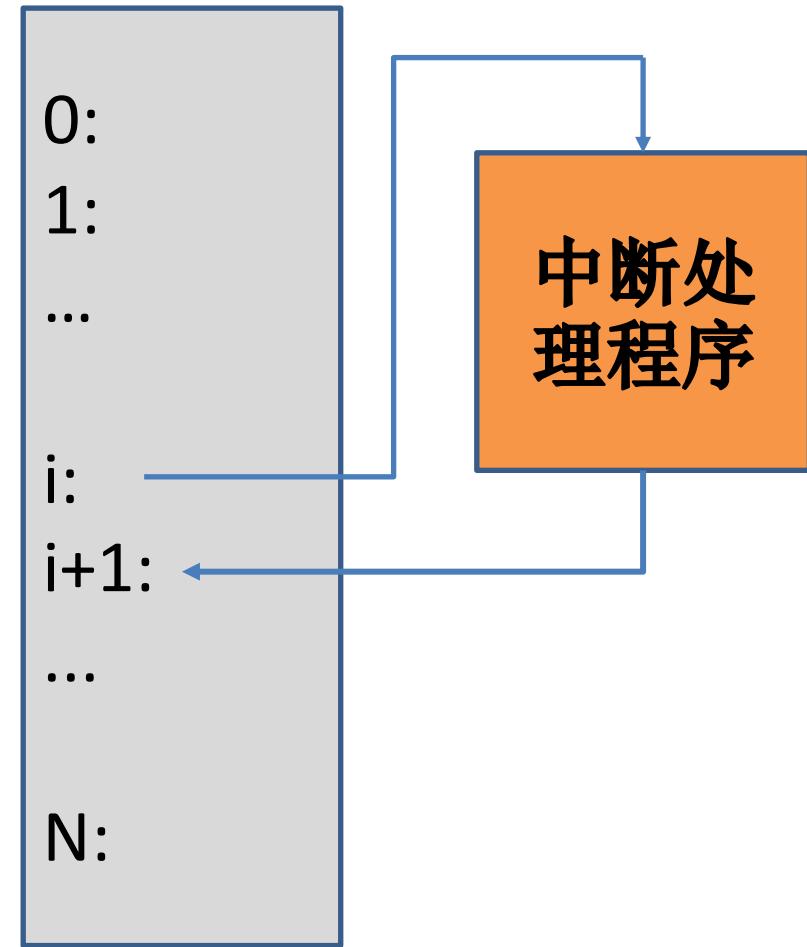
- 异常 (Exceptions)

- 导致CPU暂停当前指令 $I_{current}$ 执行，转去执行特定处理程序的事件（控制流转移）
- 事件可能和当前执行的指令无关（硬件中断），也可能和当前执行的指令有关，即由当前正在执行的进程产生（Fault, Trap, Abort）
- 特定处理程序完成处理后，控制权变化：
 - 控制权返回给：异常发生时正在执行的指令 $I_{current}$
 - 控制权返回给：若未发生异常将会执行的下一条指令 I_{next}
 - 终止被中断的程序



硬件中断

- 硬件中断 (Hardware Interrupt)
 - 由外部事件 (外部设备) 触发
 - 例如 , 时钟中断 , 硬盘读写请求完成 , 移动鼠标 , 键盘输入
 - 硬件中断产生与当前正在执行的进程无关
 - 硬件中断可以被关闭
 - 例如 x86 架构 EFLAGS 寄存器 bit 9 (Interrupt enable Flag)
- 中断处理程序 (Interrupt Service Routines , ISR)
 - 中断处理程序运行在内核
- 最终恢复被中断的进程





中断向量表

- 操作系统中用于管理中断的关键数据结构，保存在内存中的固定位置
- 表中每个条目包含一个中断处理程序的入口地址，指向实际的中断服务例程

中断号	中断服务例程入口地址	
IRQ0	0x10002000 (时钟中断处理程序)	
IRQ1	0x10003000 (键盘中断处理程序)	
IRQ2	0x10004000 (串口1中断处理程序)	
IRQ3	0x10005000 (串口2中断处理程序)	
...	...	
IRQn	0x1000XXXX (其他中断处理程序)	



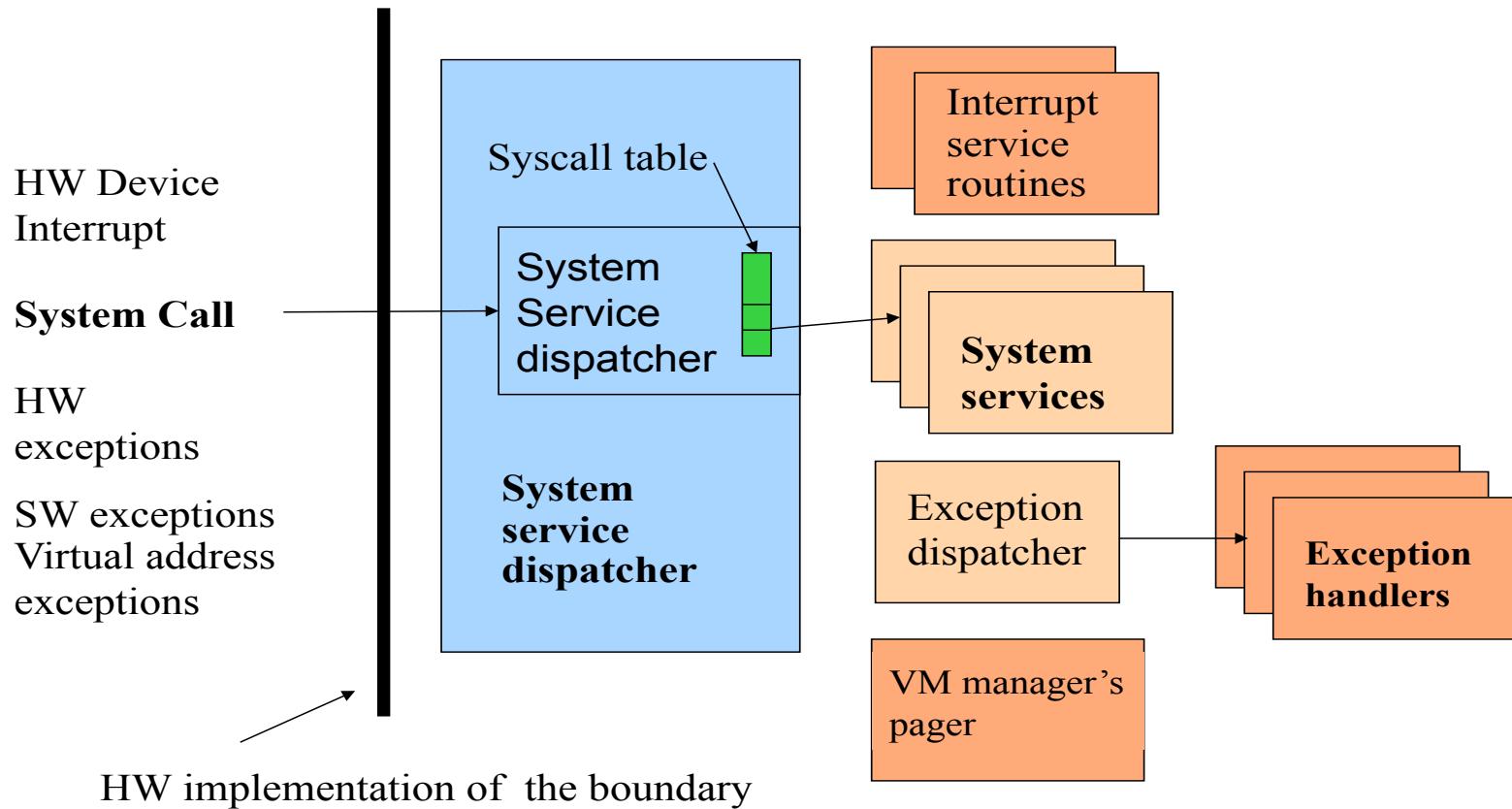
其他异常类型

- 出错 (Fault)
 - 发生了潜在可以恢复的错误。若异常处理程序能修复，则重新执行触发出错的指令；若无法修复，则终止当前程序
 - 例如， Division Error, General Protection Fault, Page Fault
- 陷入 (Trap)
 - 有特定目的的异常，异常处理完成后，执行触发异常指令的下一条指令
 - 例如，系统调用（提供应用程序和内核间的调用），断点异常（用于断点调试）
- 终止 (Abort)
 - 不可回复的错误造成的异常，操作系统终止应用程序
 - 例如，硬件错误，内存校验错误



系统调用

- 编程者用相应指令触发异常，将控制器交由操作系统，进行处理（即提供OS服务）





如何触发系统调用

- INT 0x80指令
 - 软件中断，常用于32位操作系统
- syscall指令
 - 专用系统调用指令，用于64位操作系统
 - 相比INT 0x80指令，执行效率高



如何触发系统调用

- 系统调用示例

```
section .data
msg:
    .ascii "Hello OS!\n"

section .text

    global _start
_start:
    ; write(1, msg, len)
    mov rax, 1          ; sys_write
    mov rdi, 1          ; stdout
    mov rsi, msg        ; message address
    mov rdx, len        ; message length
    syscall

    ; exit(0)
    mov rax, 60         ; sys_exit
    mov rdi, 0          ; exit code
    syscall
```

64位x86架构下系统调用示例

```
section .data
msg:
    .ascii "Hello OS!\n"

section .text

    global _start
_start:
    ; write(1, msg, len)
    mov eax, 4          ; sys_write
    mov ebx, 1          ; stdout
    mov ecx, msg        ; message address
    mov edx, len        ; message length
    int 0x80

    ; exit(0)
    mov eax, 1          ; sys_exit
    mov ebx, 0          ; exit code
    int 0x80
```

32位x86架构下系统调用示例



如何实现系统调用

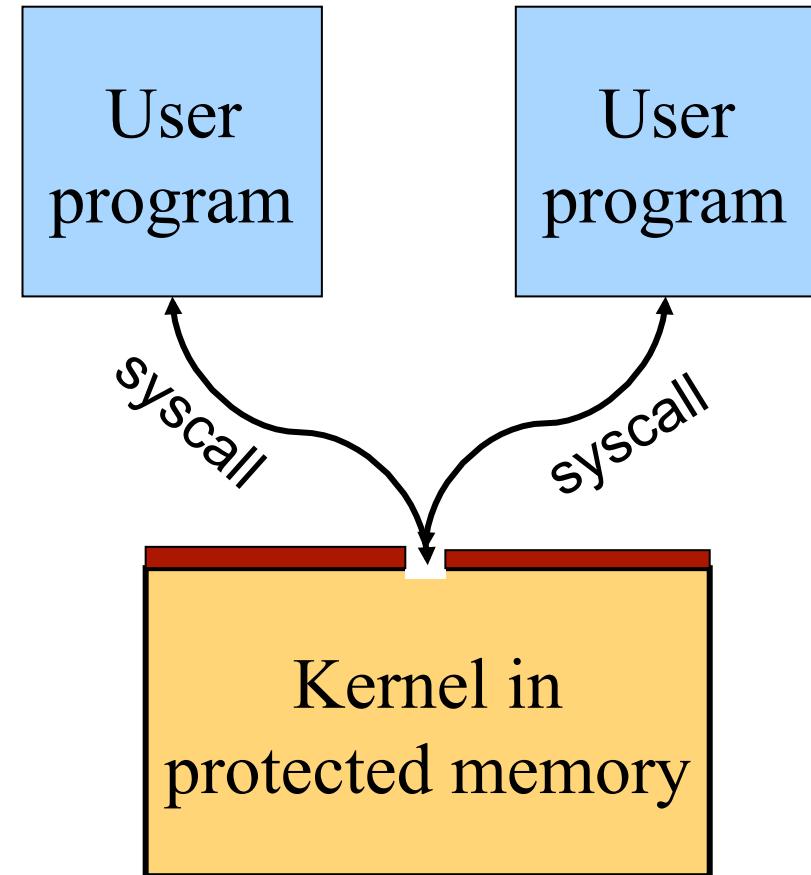
- 系统调用的内核实现函数

```
asmlinkage long sys_getpid(void)
{
    return current-> tgid;
}
```



系统调用的执行机制

- 系统调用
 - 关键、核心功能由内核完成
- 过程
 - 应用程序调用C库函数
 - C库函数调用系统调用，并传递参数
 - 系统模式从用户态切换到内核态
 - 内核中的系统调用实现函数执行系统调用功能
 - 返回结果，模式切换到用户态





系统调用的执行机制

- 系统调用执行流程

EntryPoint:

switch to kernel stack

save context

check R_0

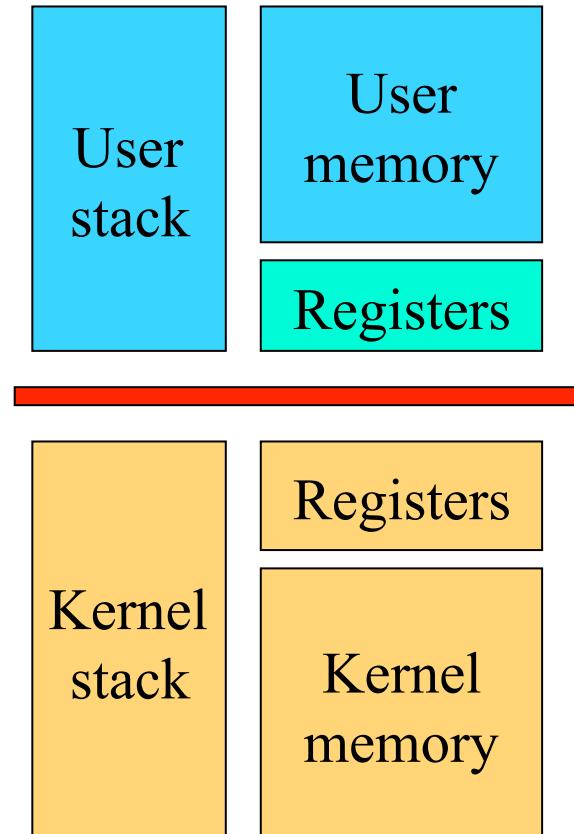
call the real code pointed by R_0

place result in R_{result}

restore context

switch to user stack

iret (change to user mode and return)



(Assume passing parameters in registers)



系统调用种类

- 操作系统的API
 - 应用和操作系统之间的接口
- 种类：
 - 进程管理: sys_fork, sys_execve, sys_exit, sys_getpid ...
 - 内存管理: sys_brk
 - 文件管理: sys_write, sys_read, sys_open, sys_close ...
 - 设备管理: sys_ioctl
 - 通信: sys_signal, sys_pipe



常见的系统调用使用方法

- 直接调用对应C库函数，由库函数进一步调用系统调用
- 使用通用的C库函数syscall
 - long syscall(long sys_number, ...)
- 使用内联汇编和INT 0x80/syscall指令调用

```
int read(int fd,char* buf,int size)
{
```

 move fd,buf,size to R₁ R₂ R₃

 move READ to R₀

 int \$0x80

 move result to R_{result}

```
}
```



如何向内核态传递参数

- 寄存器传参
 - 系统调用参数被加载到寄存器
 - 例如，RISC-V a0-a7寄存器可以用于传参
 - 可用寄存器个数 vs. 系统调用参数个数
 - 扩充知识点：ABI (Application Binary Interface)
 - 应用程序二进制接口规范
 - 定义应用程序如何与库、操作系统进行交互
 - 函数调用、系统调用约定：通过栈还是寄存器传参，哪些寄存器可以传参
 - 实现二进制兼容：由不同编译器或在不同平台上编译的应用程序，遵守ABI规定即可以跨平台运行



如何向内核态传递参数

- 内存向量(数组)传参
 - 系统调用参数个数多于寄存器数量，或者参数太大不适合使用寄存器
 - 参数保存在位于用户地址空间的数组中
 - 寄存器记录数组的起始地址
 - 内核用`copy_from_user`获取用户地址空间的参数



系统调用返回

- 系统调用
 - 通过寄存器返回结果，例如X86 eax寄存器
 - 怎么把错误返回给调用者
 - 将错误码保存在全局变量errno
 - perror函数读取errno



系统调用 vs. 库函数调用

- 以内存管理为例
- 内核
 - 分配带硬件保护的页面
 - 分配一大块(多个页面)给库
 - 不关心小粒度的分配
 - sbrk()/brk()/mmap()
- 库
 - 提供 malloc/free函数用于分配/释放
 - 应用使用这些函数细粒度管理内存
 - 当页面用完后，库函数会向内核批量请求更多的页面



系统调用 vs. 库函数调用

- 系统调用
 - 陷入内核，在内核态执行具体功能
 - 依赖于操作系统的实现
 - 伴随着内核态和用户态切换
- 库函数调用
 - 在进程用户态空间执行功能
 - 过程调用
- 为什么还需要库函数？

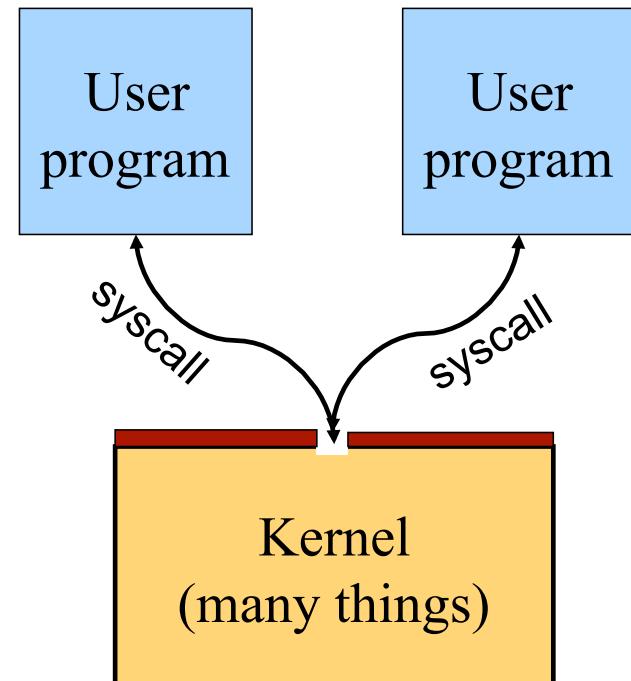


内核架构 - 宏内核 (Monolithic)

- 在内核态实现操作系统所有功能
- 应用通过调用系统调用使用操作系统提供的功能
- 例子：
 - Linux
 - BSD Unix
 - Windows
- 好处：
 - 内核所有函数共享地址空间
 - 内核模块相互调用方便高效
- 缺点：
 - 不稳定：模块crash → 系统crash
 - 不灵活：新增模块 → 内核编程

The UNIX Time-Sharing System

Dennis M. Ritchie and Ken Thompson
Bell Laboratories



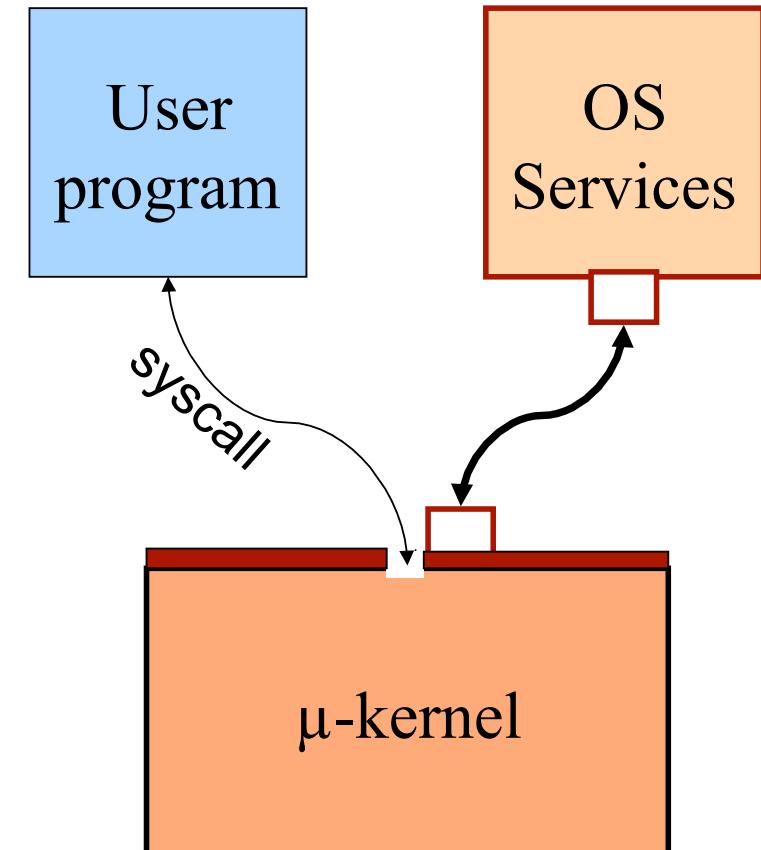


内核架构 -- 微内核 (microkernel)

- 操作系统部分功能服务作为用户态的常规进程
- 应用通过消息获取服务进程的服务
- 例子：
 - Mach (CMU)
 - MINIX(Andrew Tanenbaum)
- 好处：
 - 开发灵活
 - 故障隔离
- 缺点：
 - 应用程序和OS服务通信效率低
 - 用户态程序漏洞多，保护机制不完整

CMU Mach Project

<http://www.cs.cmu.edu/afs/cs/project/mach/public/www/mach.html>



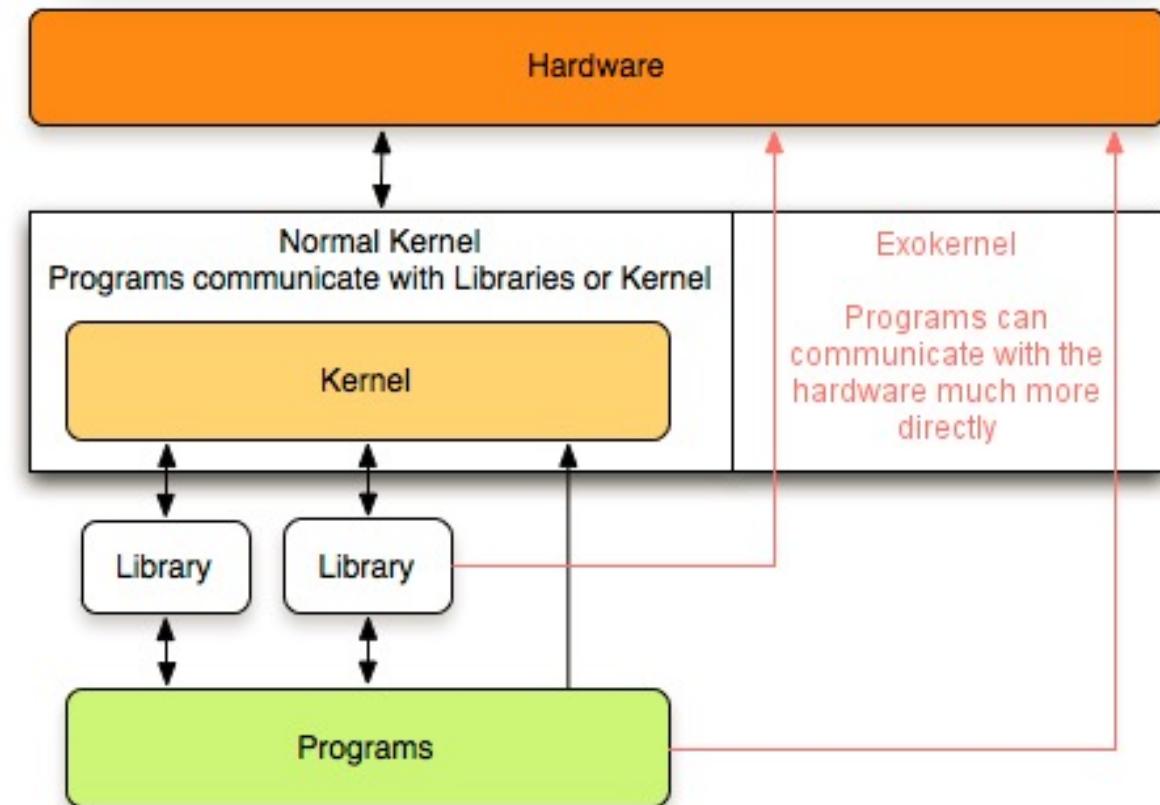


内核架构 - 库操作系统 (LibOS)

- 应用程序直接通过库与底层硬件交互
- 库函数完成OS功能
- 例子：
 - Exokernel
 - ExOS
- 好处
 - 效率高
- 缺点
 - 通用性差
 - 无法和其他程序共享资源

Exokernel: An Operating System Architecture for Application-Level Resource Management

Dawson R. Engler, M. Frans Kaashoek, and James O'Toole Jr.
M.I.T. Laboratory for Computer Science





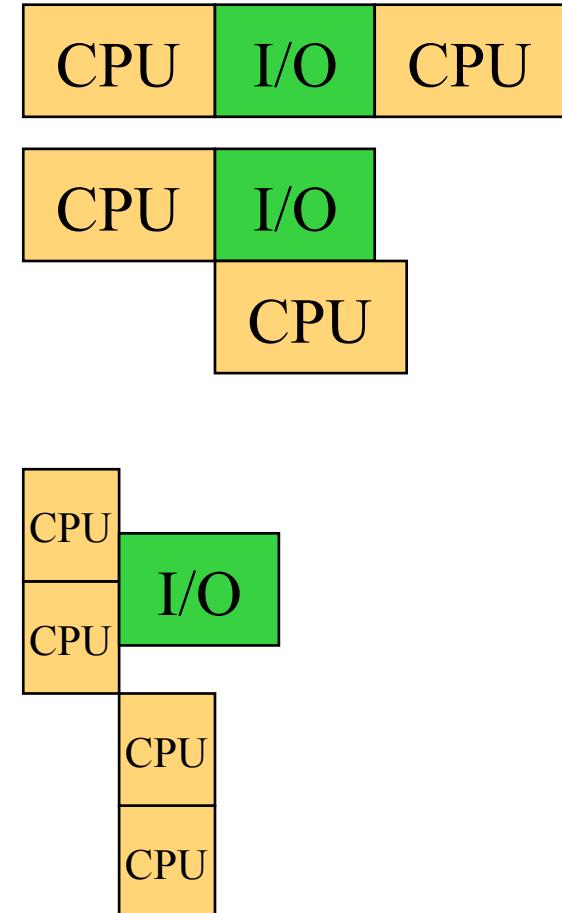
内容提要

- 操作系统结构
- 操作系统组成



处理器管理

- 目标：
 - I/O和计算重叠
 - 分时复用
 - 多个CPU的分配
- 研究问题：
 - 不能浪费CPU：调度
 - 同步和互斥
 - 公平、无死锁





内存管理

- 目标

- 支持程序的运行
 - 分配和管理
 - 与持久化存储交换数据

- 研究问题

- 访问效率和便捷性
 - 公平共享
 - 数据访问保护

Register: 1x

L1 cache: 2-4x

L2 cache: ~10x

L3 cache: ~50x

DRAM: ~200-500x

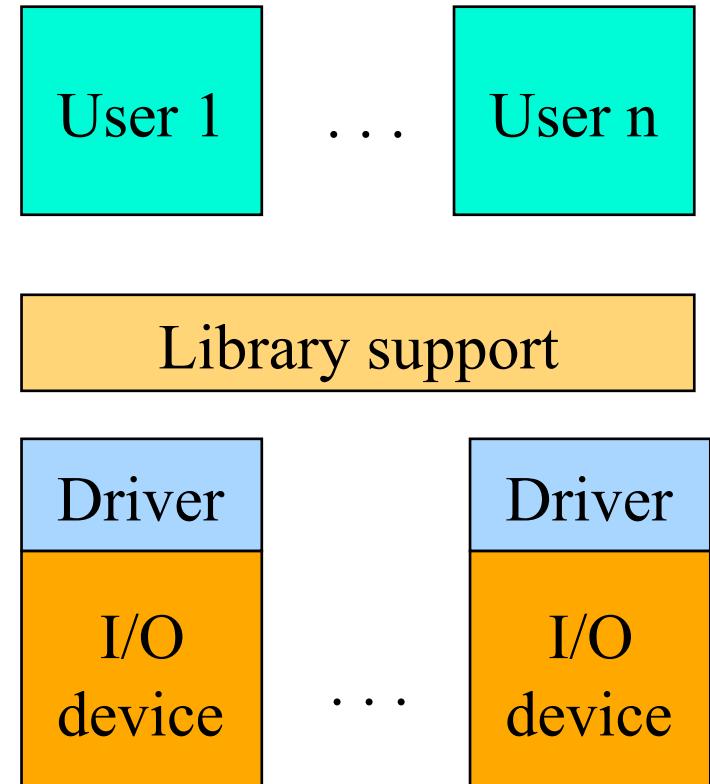
Disks: ~30M x

Archive storage: >1000M x



I/O设备管理

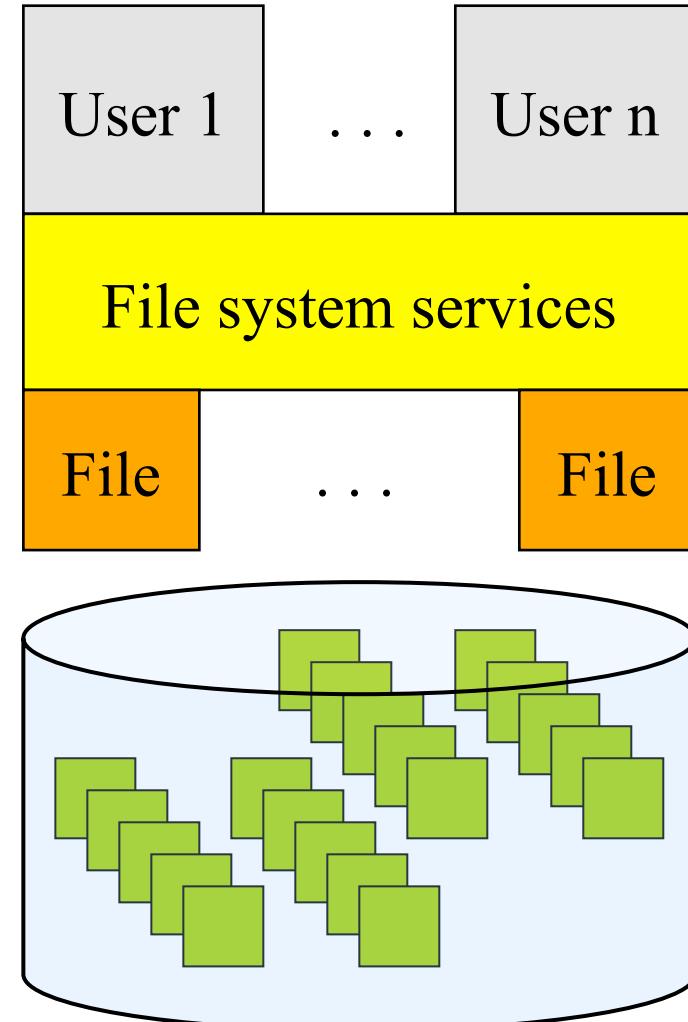
- 目标
 - 设备和应用之间的交互
 - 可动态插拔设备
- 研究问题
 - 访问效率
 - 驱动稳定性
 - 安全保护





文件系统

- 目标
 - 管理磁盘空间
 - 映射文件和磁盘块
 - 保证数据可靠性保证
- 研究问题
 - 数据访问效率
 - 数据可靠性
 - 数据安全保护





窗口系统 (GUI)

- 目标
 - 人机交互
 - 检查和管理系统的接口
- 研究问题：
 - 多方式输入、输出





总结

- 操作系统结构
 - Bootloader
 - 内核态和用户态的区分
 - 安全保护的需求
 - 用户程序、库、可移植层、机器相关层
 - 内核态和用户态切换
 - 硬件中断
 - Fault
 - Trap : 系统调用
 - Abort
 - 内核架构 : 宏内核、微内核



总结

- 操作系统的组成
 - 进程管理
 - 内存管理
 - 文件系统
 - I/O设备管理
 - 用户接口