



---

# L2 操作系统结构

宋卓然

上海交通大学计算机系

[songzhuoran@sjtu.edu.cn](mailto:songzhuoran@sjtu.edu.cn)

饮水思源 · 爱国荣校



# 操作系统服务

- 操作系统提供环境以便执行程序，方便程序员与用户
- 提供用户功能的服务
  - 用户界面：几乎所有操作系统都有用户界面 (UI, user interface)
    - 命令行界面、批处理界面、图形用户界面
  - 程序执行：加载程序到内存并运行程序
  - I/O操作：程序运行可能需要I/O设备，操作系统提供设备驱动程序
  - 文件系统操作：创建、删除文件、权限管理
  - 通信：提供进程间通信渠道（共享内存、消息交换）





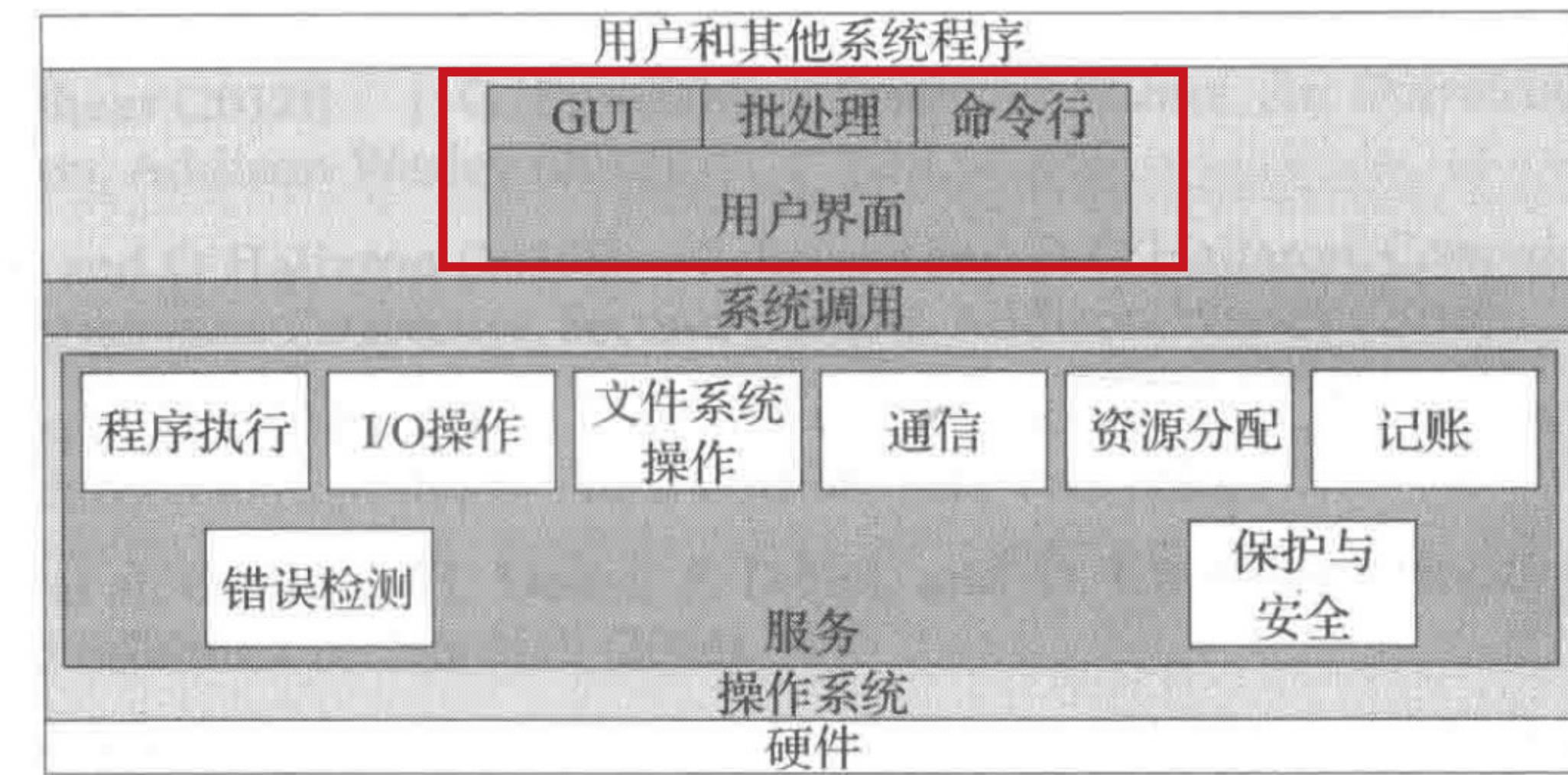
# 操作系统服务

- 提供保证系统运行效率的服务
  - 资源分配：多个用户、进程同时运行时，合理分配资源
  - 记账：记录用户使用资源的类型和数量，这种记录可以用于记账（以便向用户收费），或统计使用量
  - 保护与安全：保护不受内部、外部的干扰、威胁





# 操作系统服务





# 用户操作系统界面



- 命令行界面、命令解释程序：直接输入命令，供操作系统执行
  - 获取并执行用户的下一条命令（创建、删除、打印）
  - `rm hello.txt`

```
Last login: Thu Jul 14 08:47:01 on ttys002
iMacPro:~ pbg$ ssh root@r6181-d5-us01
root@r6181-d5-us01's password:
Last login: Thu Jul 14 06:01:11 2016 from 172.16.16.162
[root@r6181-d5-us01 ~]# uptime
 06:57:48 up 16 days, 10:52,  3 users,  load average: 129.52, 80.33, 56.55
[root@r6181-d5-us01 ~]# df -kh
Filesystem      Size  Used Avail Use% Mounted on
/dev/mapper/vg_ks-lv_root
                  50G   19G   28G  41% /
tmpfs           127G  520K  127G   1% /dev/shm
/dev/sda1        477M   71M   381M  16% /boot
/dev/dssd0000    1.0T  480G  545G  47% /dssd_xfs
tcp://192.168.150.1:3334/orangesfs
                  12T  5.7T  6.4T  47% /mnt/orangesfs
/dev/gpfs-test   23T  1.1T   22T   5% /mnt/gpfs
[root@r6181-d5-us01 ~]#
[root@r6181-d5-us01 ~]# ps aux | sort -nrk 3,3 | head -n 5
root      97653 11.2  6.6 42665344 17520636 ?  S<Ll Jul13 166:23 /usr/lpp/mmfs/bin/mmfsd
root      69849  6.6  0.0     0     0 ?      S    Jul12 181:54 [vpthread-1-1]
root      69850  6.4  0.0     0     0 ?      S    Jul12 177:42 [vpthread-1-2]
root      3829  3.0  0.0     0     0 ?      S    Jun27 730:04 [rp_thread 7:0]
root      3826  3.0  0.0     0     0 ?      S    Jun27 728:08 [rp_thread 6:0]
[root@r6181-d5-us01 ~]# ls -l /usr/lpp/mmfs/bin/mmfsd
-r-x----- 1 root root 20667161 Jun  3  2015 /usr/lpp/mmfs/bin/mmfsd
[root@r6181-d5-us01 ~]#
```





# 用户操作系统界面



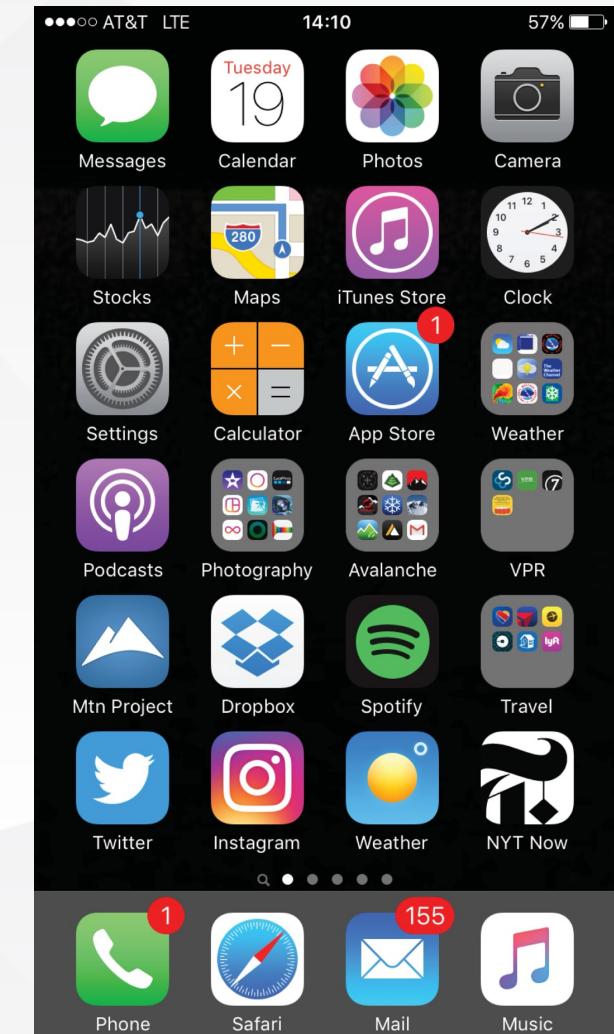
- 这些命令的实现有两种常用方法
  - 命令解释程序本身包含代码以执行这些命令
    - 例如，删除文件的命令可让命令解释程序跳转到相应的代码段，以设置参数并执行相应系统调用
  - 通过系统程序实现大多数命令，命令解释程序不必理解命令
    - `rm hello.txt`
    - 更灵活、轻量
- 程序员更偏好命令行的界面，因为效率更高，不需要层层嵌套





# 图形用户界面

- 交互式界面：图形用户界面（GUI）
- 采用桌面的概念
- 利用指针定位屏幕上面图标
- 触摸屏，利用手势点击操作
- 语音控制
- 普通用户友好，使用方便





# 操作系统服务





# 系统调用

- 系统调用提供操作系统服务接口
- 调用通常以C或C++编写
- 对程序员隐藏复杂的系统调用，只需要程序员根据应用编程接口（API）设计程序
- 三种常用的API
  - Window API
  - POSIX API (适用于UNIX、Linux、Mac OS X)
  - Java API

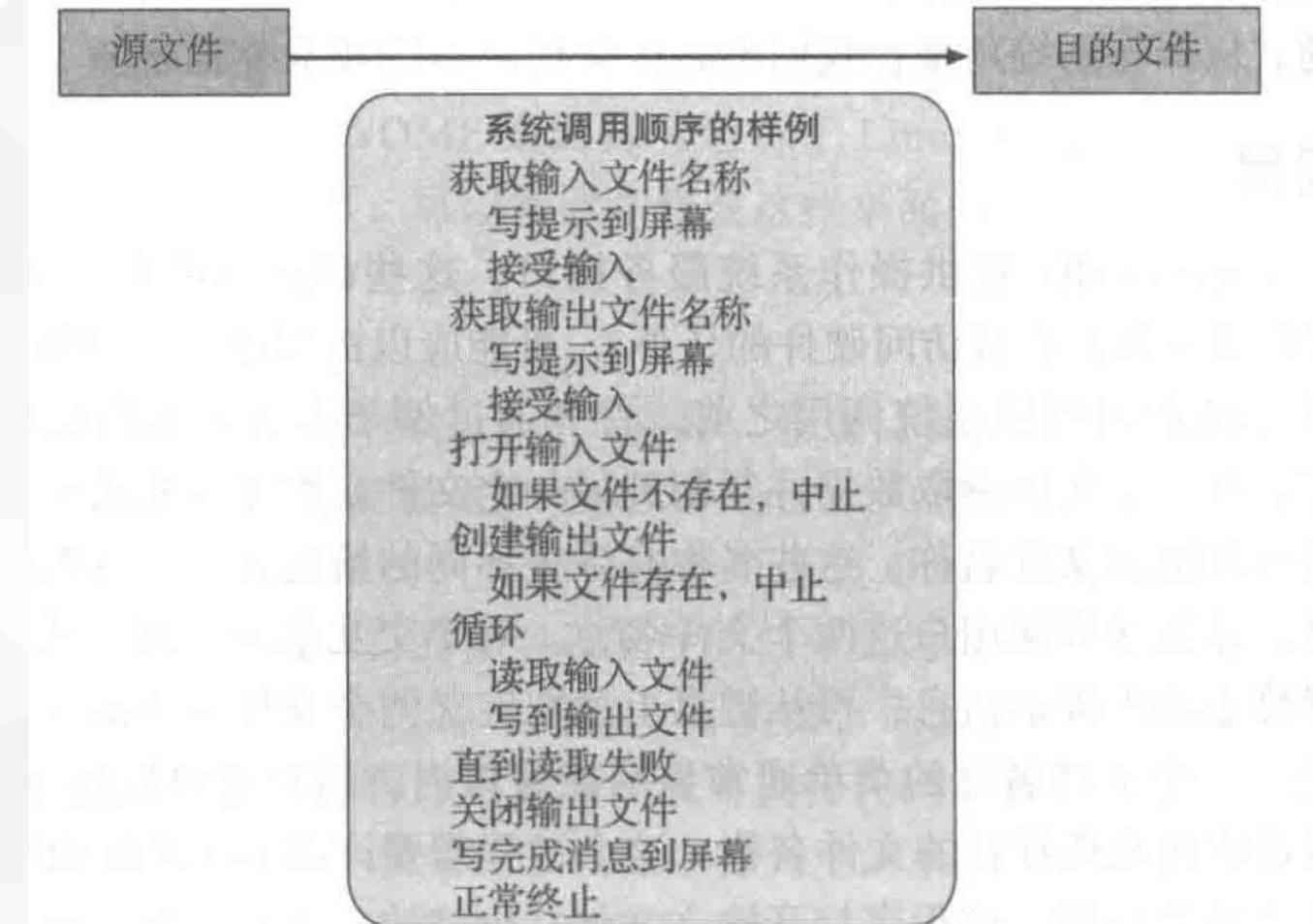




# 系统调用-实例



- 利用系统调用实现把文件A的内容复制到文件B中





# 标准API的例子



- 使用API编程有利于提高程序的可移植性
- 读取文件，返回读取的字节数
- 头文件unistd.h
- 传入参数：
  - int fd: 要读的文件描述符
  - void \*fd: 数据要被读到的缓冲区
  - size\_t count: 读取的最大字节数

```
#include <unistd.h>
ssize_t      read(int fd, void *buf, size_t count)
```

返回值      函数名称      参数

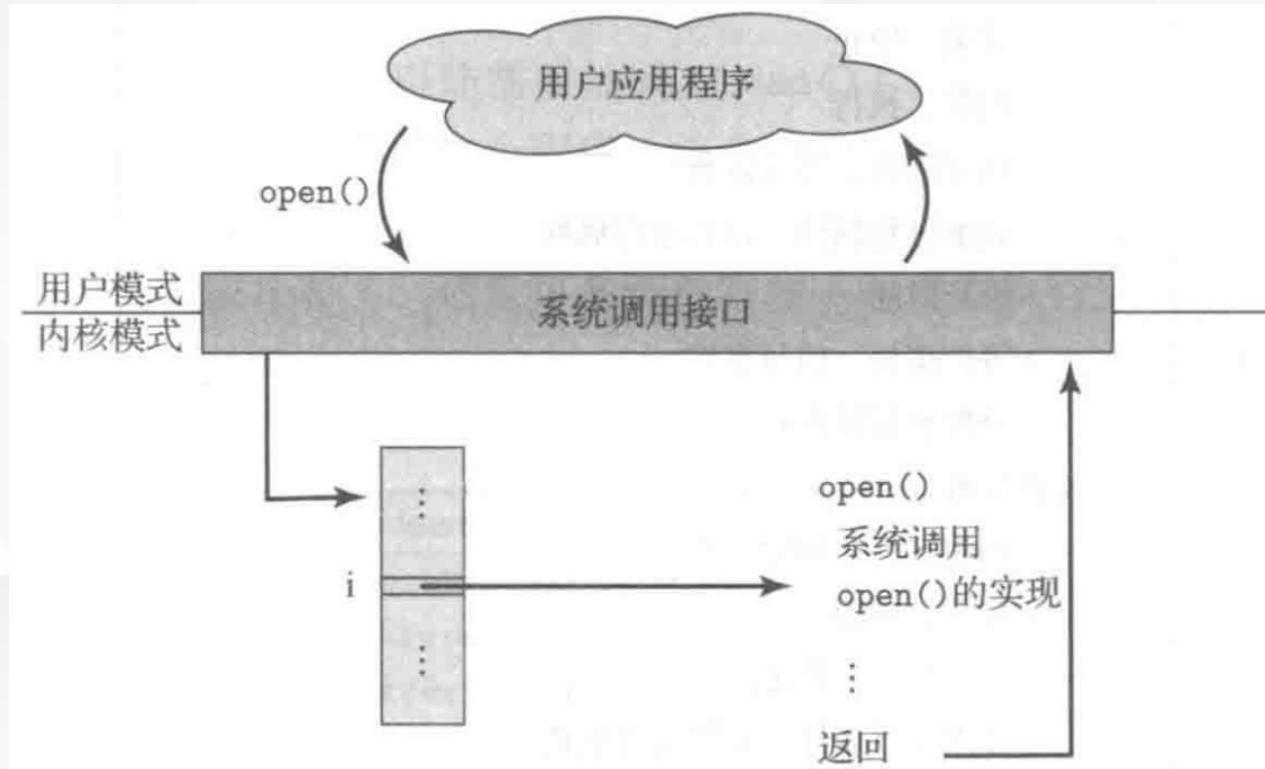




# 系统调用实现



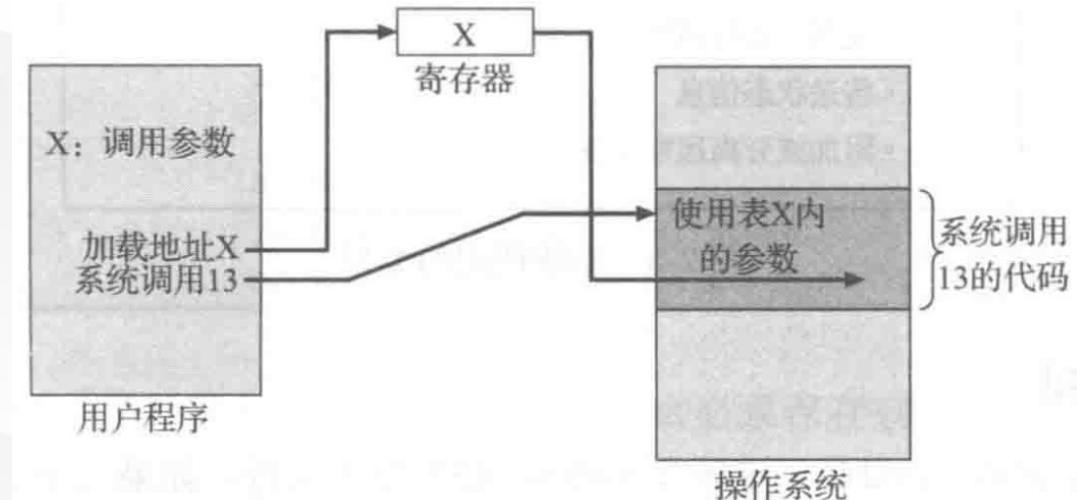
- 对大多数程序设计语言，运行时支持系统提供**系统调用接口**，以链接到操作系统的系统调用
- 系统调用接口截取API函数的调用，并调用操作系统中的所需系统调用





# 系统调用参数传递

- 通过寄存器传递参数
- 当参数量大于寄存器数量时，可将参数存放在内存，寄存器指向内存地址
- 利用堆栈传递参数
  - 压入
  - 弹出
- 利用栈不限制传递参数的数量或长度





# 系统调用的类型



- 进程控制
  - 结束、中止
  - 加载、执行
  - 创建进程、终止进程
  - 获取进程属性
  - 等待时间
  - 分配和释放内存





# 系统调用的类型

- 文件管理
  - 创建、删除文件
  - 读、写
  - 获取文件属性
- 设备管理
  - 请求设备、释放设备
  - 读、写、重新定位
  - 获取设备属性





# 系统调用的类型

- 信息维护
  - 获取时间、日期
  - 获取系统数据、设置系统数据
  - 获取进程、文件、设备属性
- 通信
  - 创建、删除通信连接
  - 发射、接受数据
  - 传送状态信息





# 系统调用-实例



	Windows	UNIX
进程控制	CreateProcess() ExitProcess() WaitForSingleObject()	fork() exit() wait()
文件管理	CreateFile() ReadFile() WriteFile() CloseHandle()	open() read() write() close()
设备管理	SetConsoleMode() ReadConsole() WriteConsole()	ioctl() read() write()
信息维护	GetCurrentProcessID() SetTimer() Sleep()	getpid() alarm() sleep()
通信	CreatePipe() CreateFileMapping() MapViewOfFile()	pipe() shm_open() mmap()
保护	SetFileSecurity() InitializeSecurityDescriptor() SetSecurityDescriptorGroup()	chmod() umask() chown()

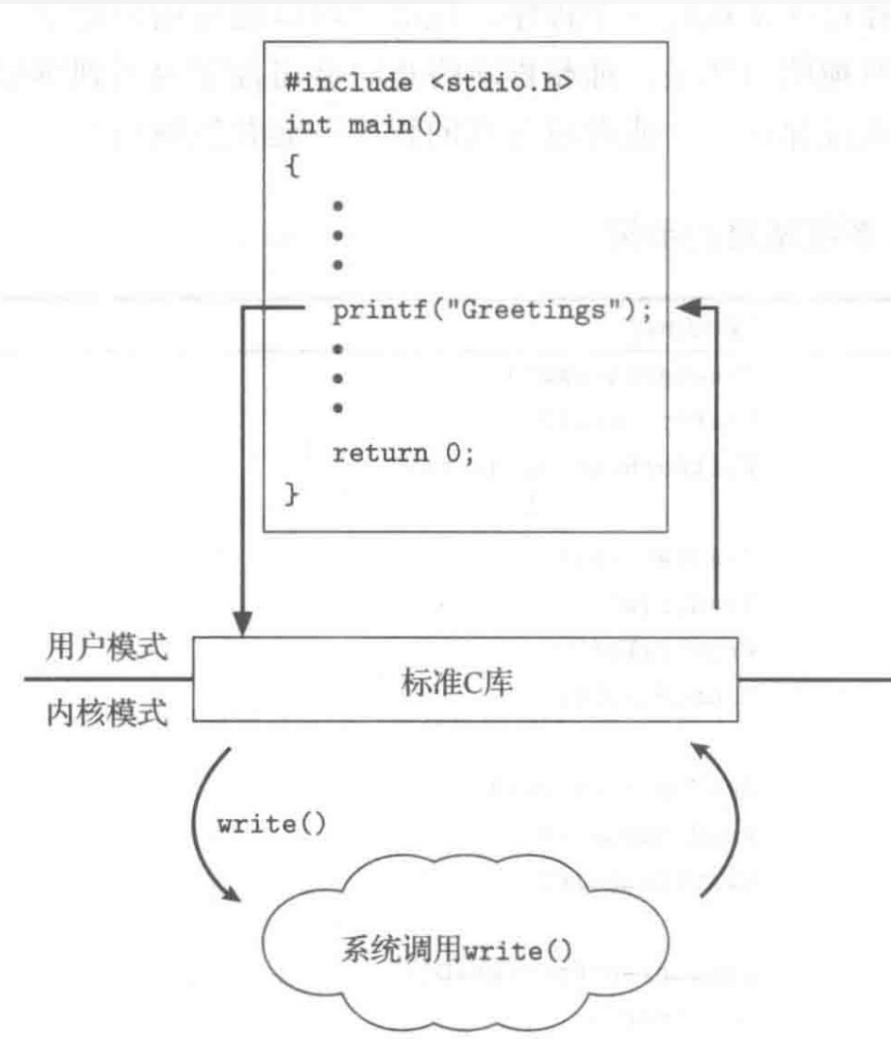




# 标准C程序库的实例



- C语言调用printf(), 程序库劫持这个调用，并调用操作系统的必要系统调用



为什么要有系统调用?

- 1.程序的可移植性
- 2.编程难度





# API与系统调用之间的关系



- API(Application Programming Interface)应用程序接口，是一些预定义的函数。跟内核没有必然的联系。
- 不是所有的API函数都对应一个系统调用，
  - 有时，一个API函数会需要几个系统调用来共同完成函数的功能
  - **也有一些API函数不需要调用系统调用（因此它所完成的不是内核提供的服务，例如：abs(); sqrt()等函数）**
- 程序员只能使用API与系统交互，不能直接使用系统调用
- 系统调用不与程序员进行交互，它根据API函数，**通过一个软中断机制向内核提交请求**，以获取内核服务的接口

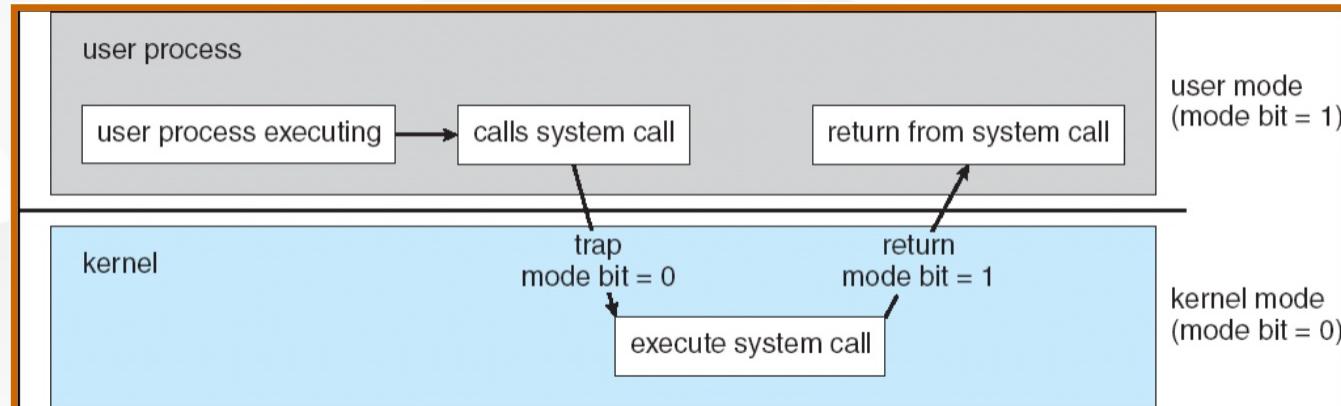




# 用户态与内核态

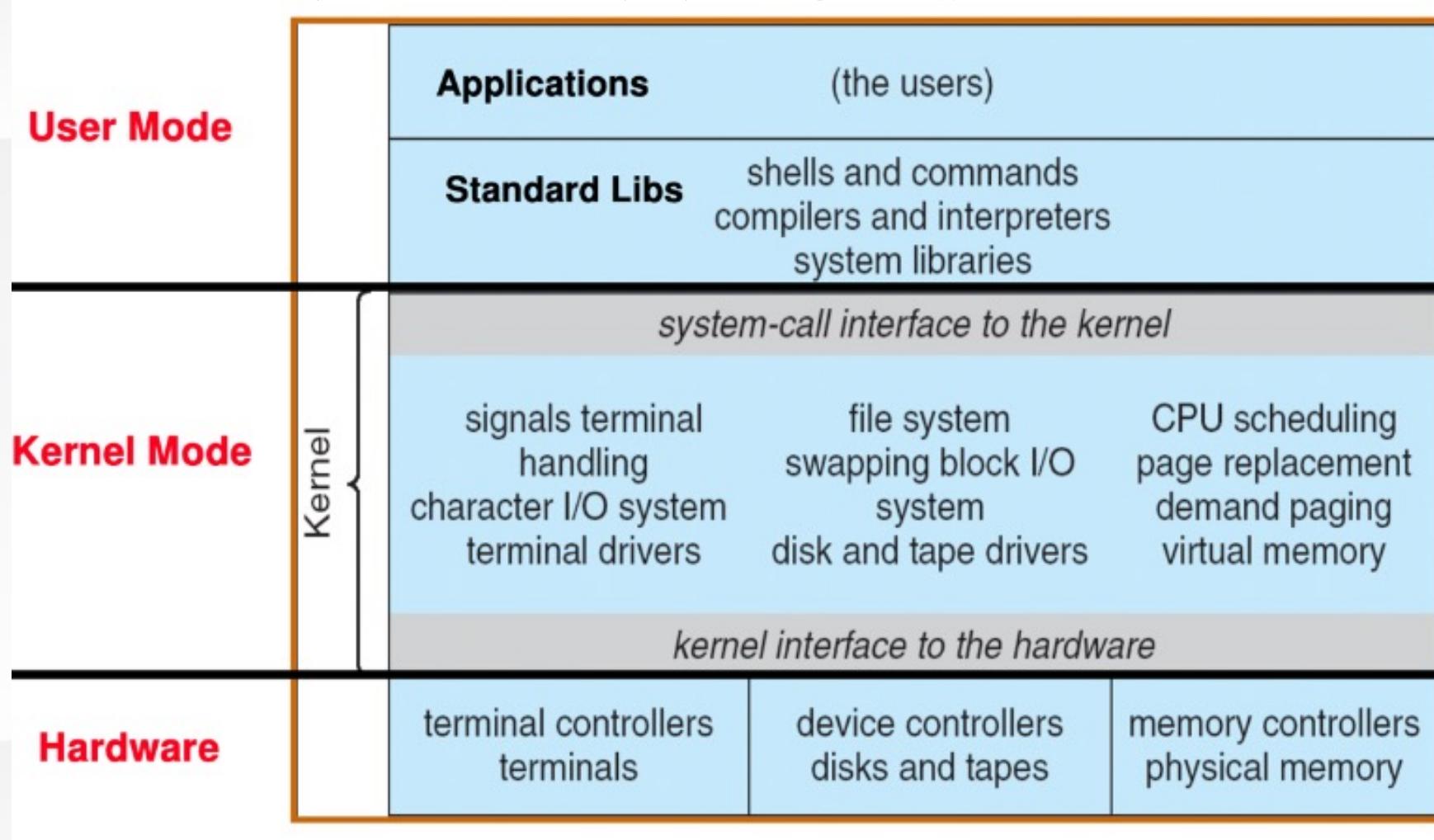


- 计算机系统中，通常运行着两类程序：
  - 系统程序和应用程序
- 计算机设置了两种状态：
  - 系统态(也称为核心态)，操作系统在系统态运行——运行操作系统程序
  - 用户态，应用程序只能在用户态运行——用户程序在实际运行过程中，处理器会在系统态和用户态间切换。





## 为什么要有用户态和内核态？





# 硬件的工作模式



## 硬件至少要支持两种工作模式

1. Kernel Mode (or “supervisor” mode)
2. User Mode

## 在user mode下禁止某些操作

修改页表指针 Changing the page table pointer,

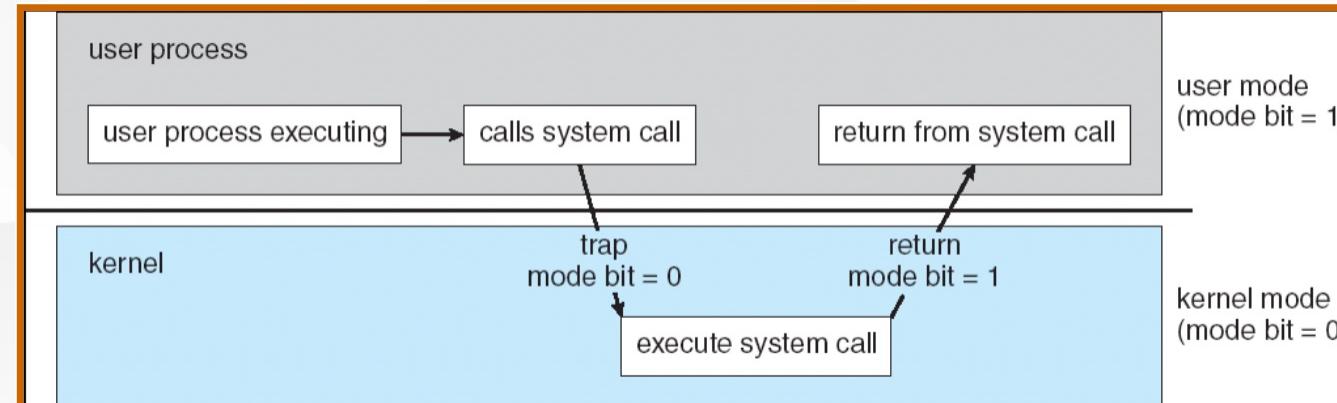
禁止中断 disabling interrupts,

直接操作硬件 interacting directly w/ hardware

修改内核内存 writing to kernel memory

user mode 到 kernel mode 切换的条件

System calls (系统调用) , interrupts(中断) , exceptions (异常)





# 用户态切换到内核态的三种方式

## 1) 系统调用

用户态进程主动要求切换到内核态的一种方式。例如Linux的int 80h中断。

## 2) 异常

当cpu在执行运行在用户态下的程序时，发生了一些没有预知的异常，这时会触发由当前运行进程切换到处理此异常的内核相关进程中，如缺页异常

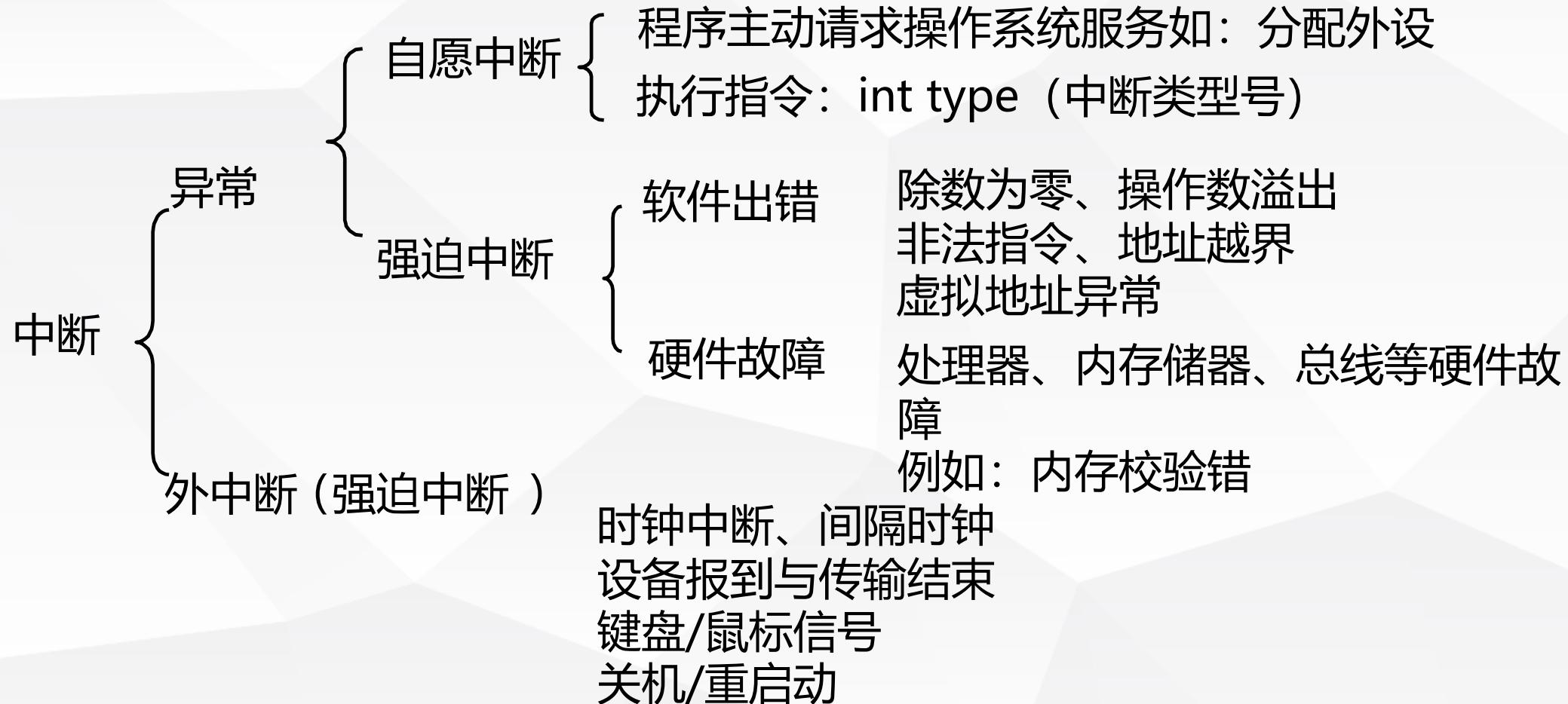
## 3) 外围设备的中断

当外围设备完成用户请求的操作后，会向CPU发出相应的中断信号，这时CPU会暂停执行下一条即将要执行的指令而转到与中断信号对应的处理程序去执行，如果前面执行的指令时用户态下的程序，那么转换的过程自然就会是由用户态到内核态的切换。





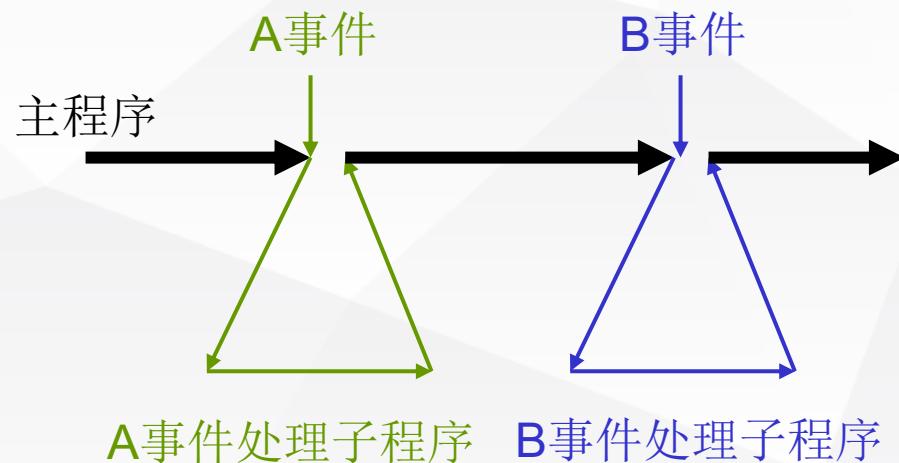
# 异常/中断的类型





# 如何支持中断与异常处理?

- 中断与异常处理：指令系统必须考虑如何支持
- 中断/异常处理的过程
  - 程序执行过程中遇到需处理的事件
  - 暂时中止现行程序的运行
  - 转去执行相应的事件处理程序
  - 待处理完成后再返回原程序被中断处、或调度其他程序执行的过程。

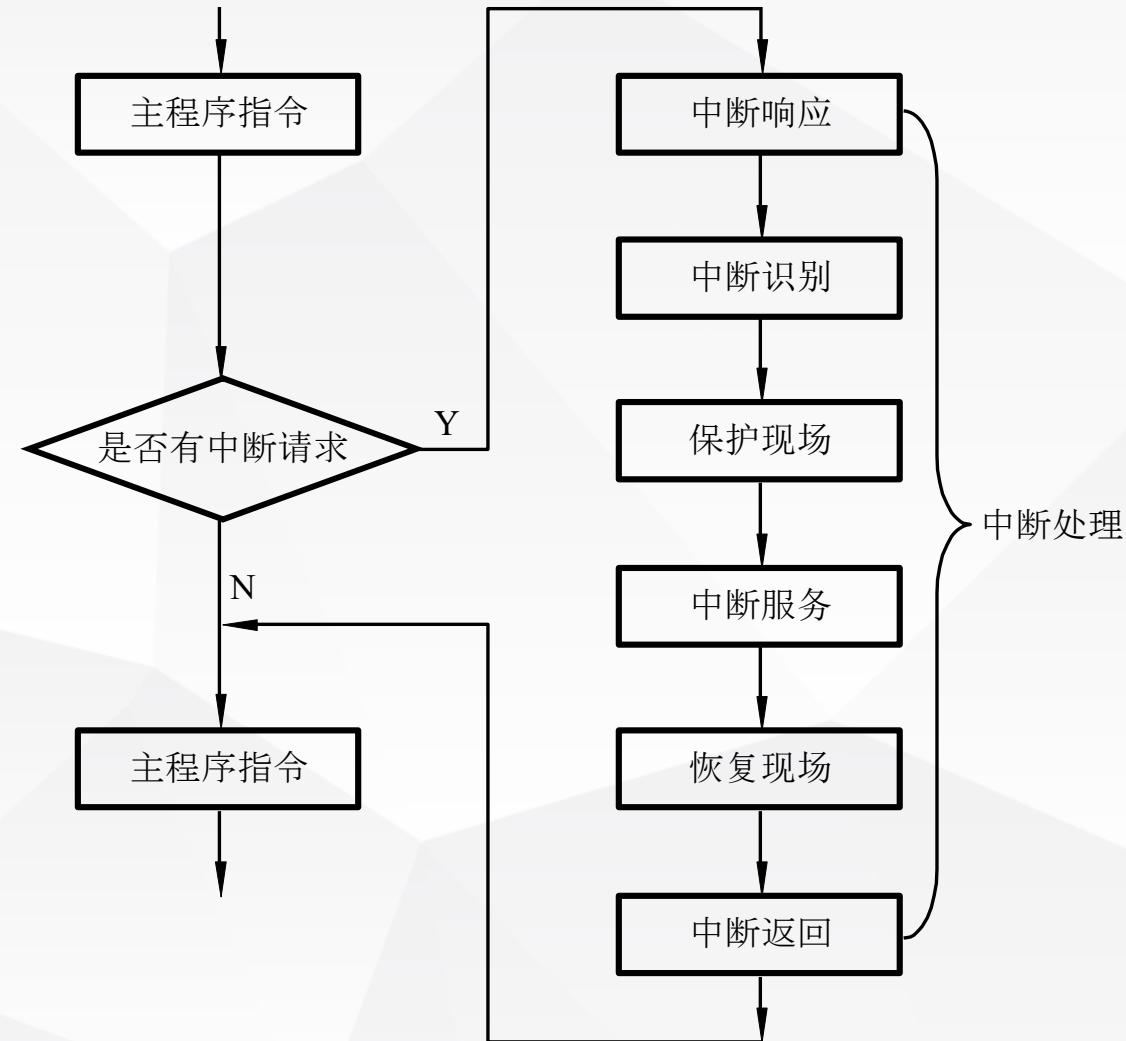




# 如何支持中断与异常处理?

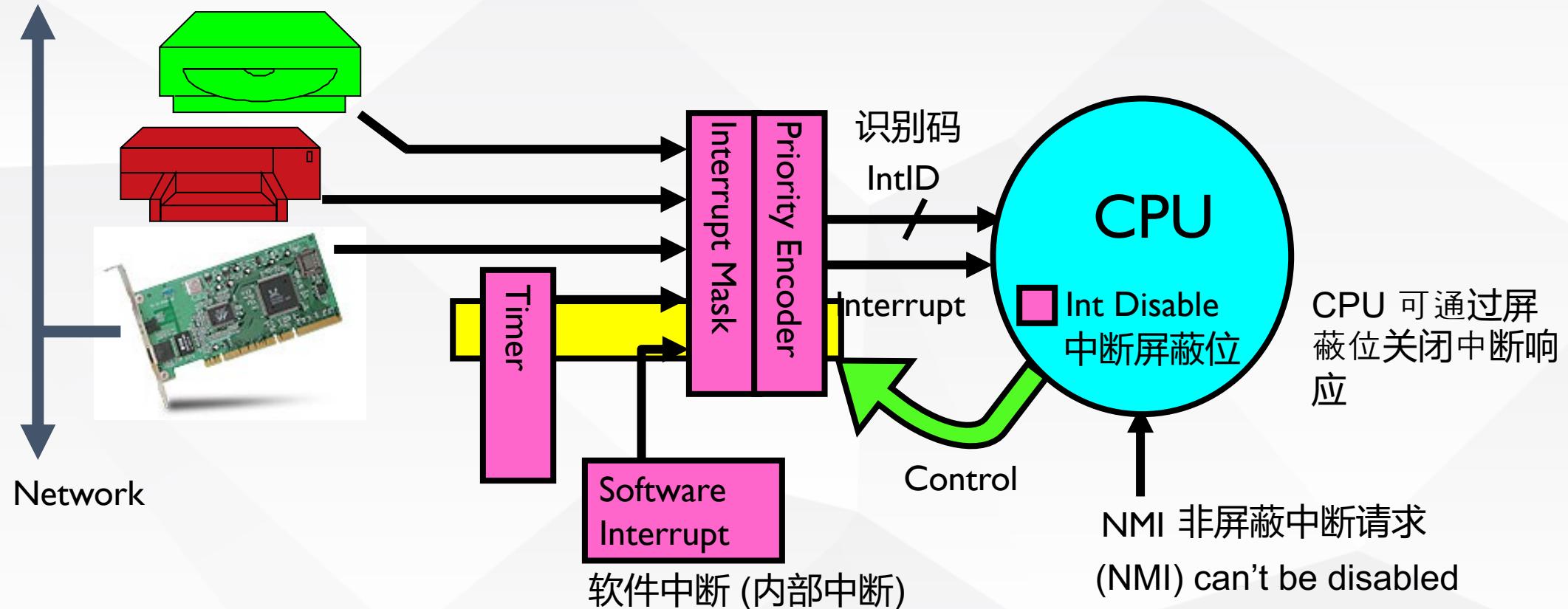


- 指令系统应该作出的规定:
  - 异常和中断类型的定义
  - 自陷指令
  - 中断允许位
  - 开中断、关中断
  - 异常/中断原因的识别和记录
  - 断点信息的保存
  - 中断处理时软硬件之间的协同
  - ...





# 中断控制器 Interrupt Controller



- Interrupt controller chooses interrupt request to honor
  - Interrupt identity specified with ID line
  - Mask enables/disables interrupts
  - Priority encoder picks highest enabled interrupt
  - Software Interrupt Set/Cleared by Software



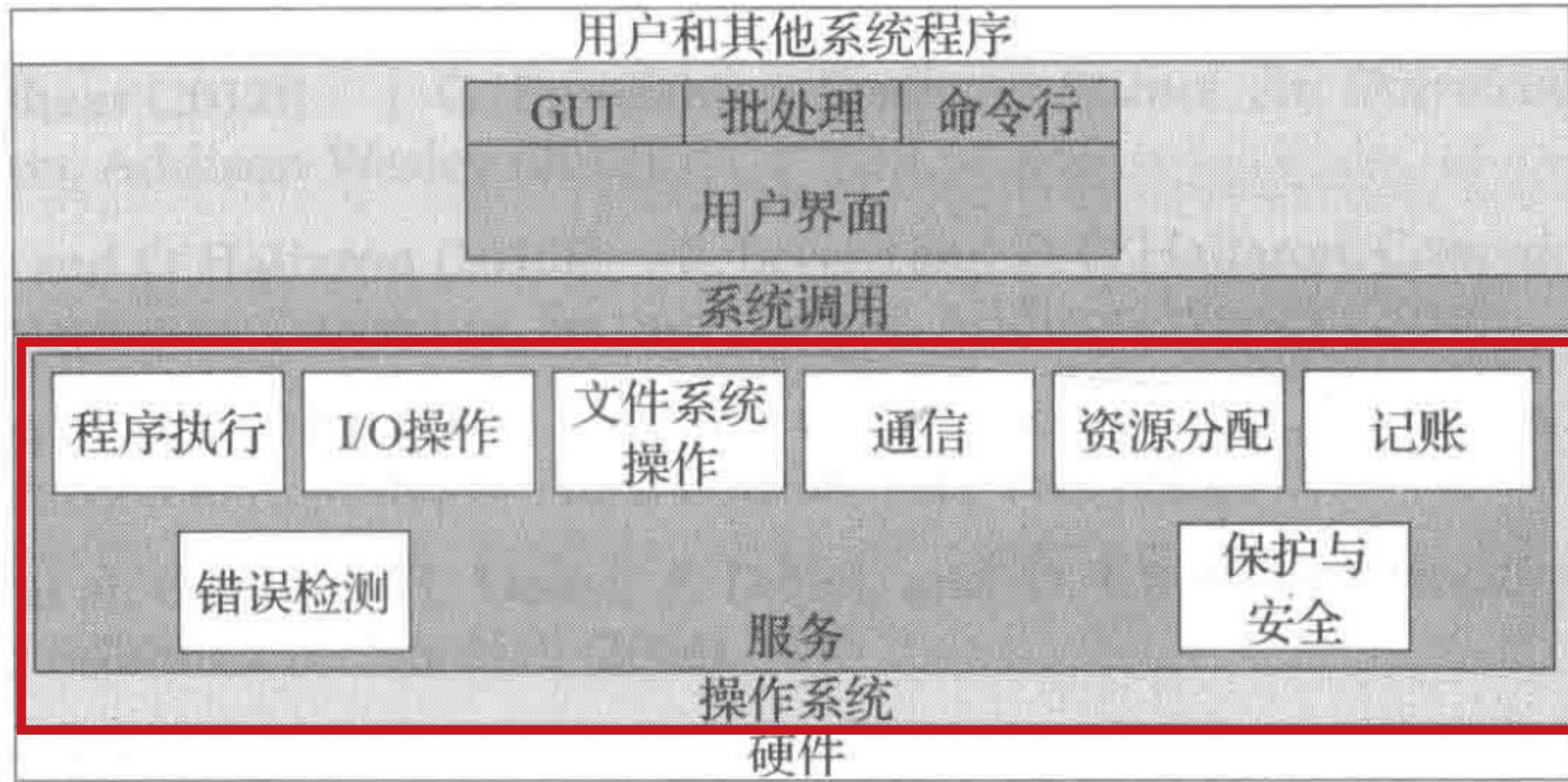


下列选项中，会导致用户进程从用户态切换到内核态的操作是

- I. 设置系统时钟
  - II. `sin()` 函数调用
  - III. `read` 系统调用
- 
- A. 仅 I、II
  - B. 仅 I、III
  - C. 仅 II、III
  - D. I、II 和 III



# 操作系统服务





# 操作系统的概念与设计-设计目标

- 定义目标和规范
  - 批处理、分时、单用户、多用户、分布式、实时、通用
- 确定需求
  - 用户目标
    - 优良的性能、便于使用、可靠、安全
  - 系统目标
    - 易于设计、灵活、高性能、可靠





# 操作系统的概念与实现-实现

- 可以通过汇编、C、C++编写实现，根据场景决定
- 采用高级语言（C、C++）可以实现操作系统
  - 编写难度低
  - 更易移植到其他硬件
- 如MS-DOS用Intel 8088汇编语言编写，可以直接用于Intel X86类型的CPU

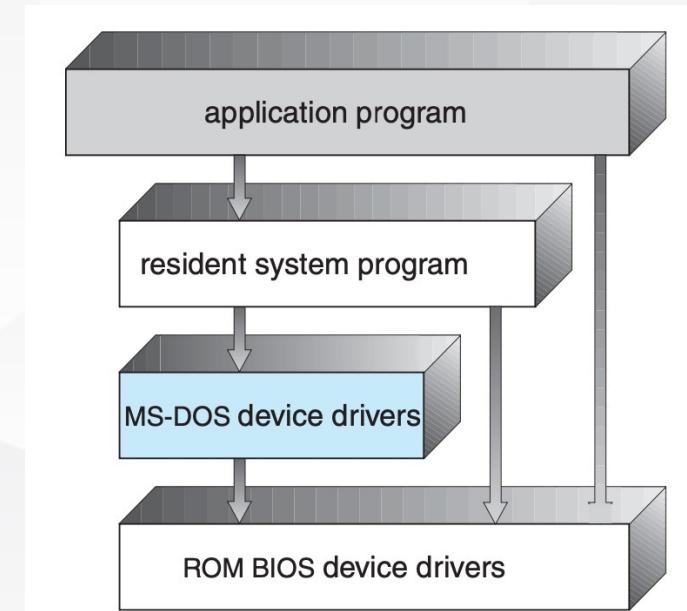




# 操作系统的结构



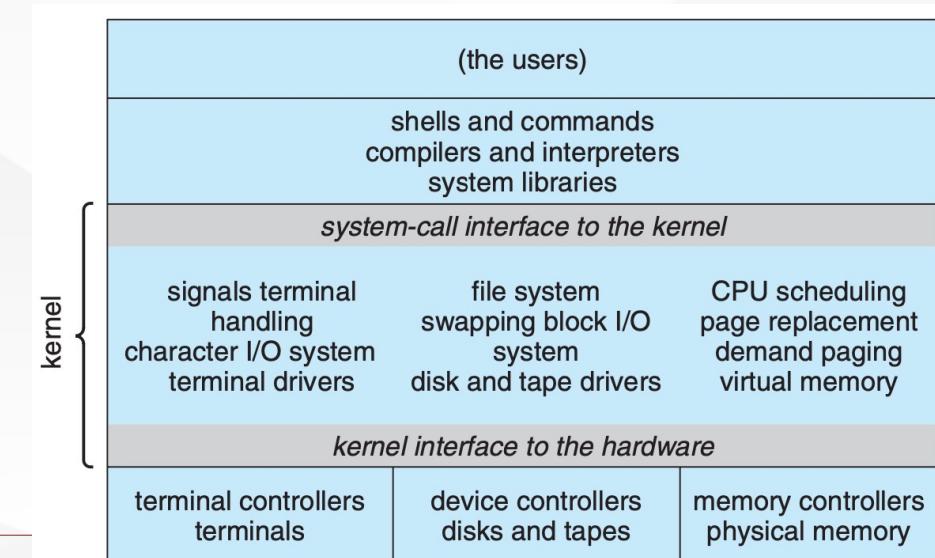
- 现代操作系统通常被分成子系统/模块，模块之间合作以构成内核
- 简单结构
  - MS-DOS：没有很好地区分功能的借口和层次。如应用程序可以访问基本的I/O程序，并直接操作显示器；容易受到错误影响，系统崩溃





# 操作系统的结构

- 现代操作系统通常被分成子系统/模块，模块之间合作以构成内核
- 简单结构
  - MS-DOS：没有很好地区分功能的接口和层次。如应用程序可以访问基本的I/O程序，并直接操作显示器；容易受到错误影响，系统崩溃
- 最早期的UNIX，采用有限结构，由内核和系统程序组成
  - 内核中功能过多，难以实现
  - 优势：系统调用接口与内核通信开销小

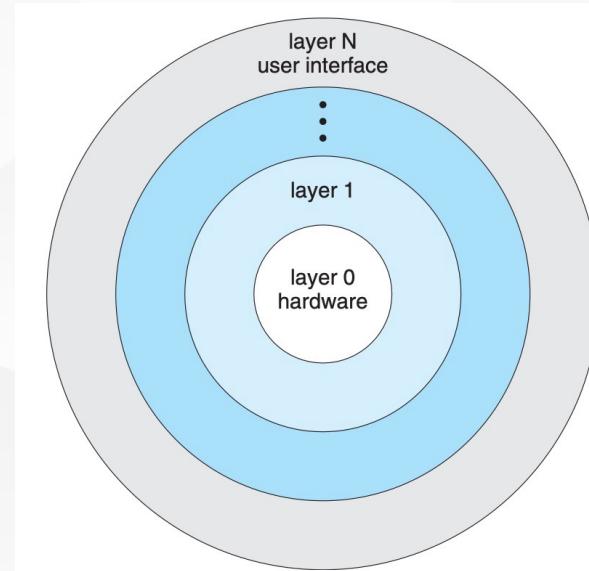




# 操作系统的结构



- 分层方法
  - 最底层为硬件，最高层为用户接口，每层只能调用更低一层的功能和服务
  - 简化了构造和调试
  - 难点在于合理定义各层，如备份存储驱动程序（需要被CPU调度，同时需要等待I/O完成）





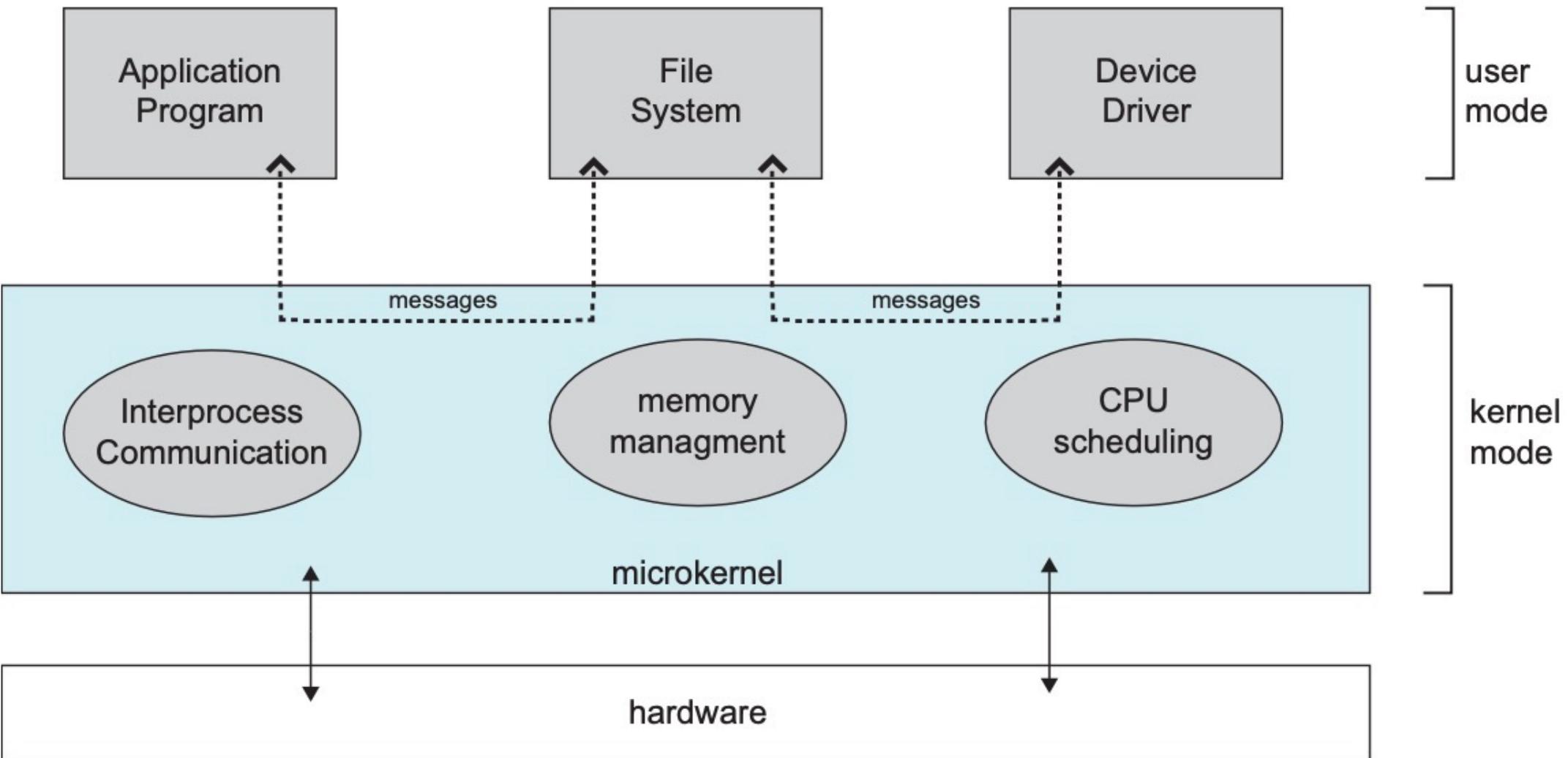
# 操作系统的结构

- 微内核
  - 采用微内核技术将内核模块化，从内核中删除不必要部件，使内核较小
  - 分为内核模式与用户模式，便于管理、扩展系统、移植系统
  - 通过消息传递进行通信，如一个客户程序要访问文件，不直接与文件系统交互，通过微内核
- 优势
  - 便于扩展操作系统，新服务在用户空间增加，不需要修改内核
  - 较好的安全性和可靠性，因为大多数服务不必作为内核进程来运行，即使出错也不会影响微内核的其他部分



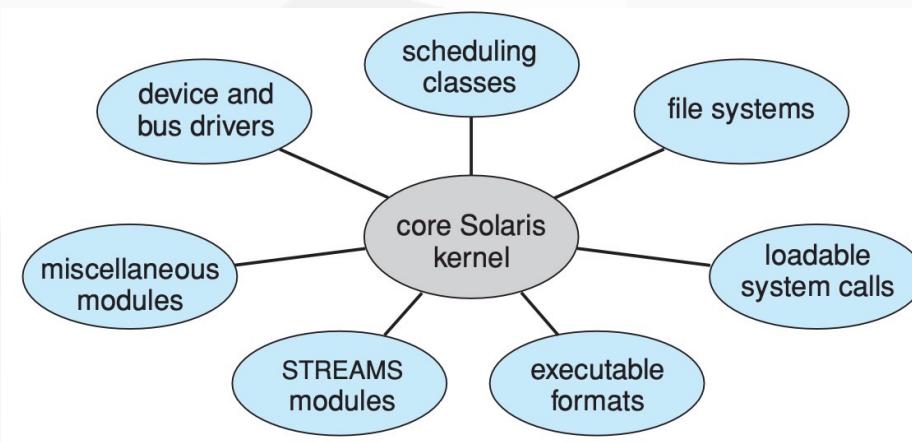


# 操作系统的结构

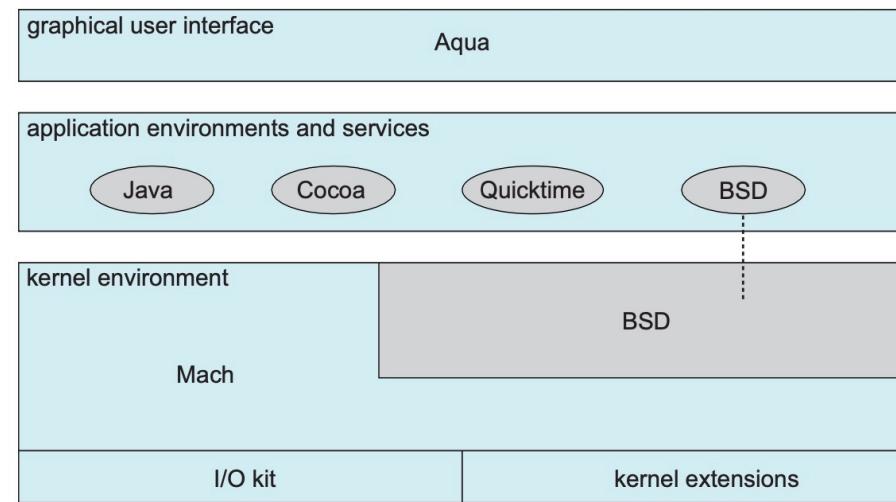




- 模块
  - 采用可加载的内核模块构建操作系统，通过模块链入额外服务（Windows、Solaris、Linux）
  - 内核提供核心服务，其他服务（进程调度）在内核运行时动态实现
  - 动态链接服务优于直接添加新功能到内核，不需重新编译内核



- 混合系统
  - 结合多种结构，形成混合系统，解决性能、安全性、灵活性等问题
- Mac OS X
  - 分层系统：顶层为用户界面与应用程序服务，底层为内核环境（Mach微内核和BSD UNIX内核）





# 操作系统的结构

- iOS
  - Cocoa Touch: 用于Objective-C的API，支持移动设备独有的“触摸”操作
  - 媒体服务: 提供图像、音频等服务
  - 核心服务: 数据库、文件系统
  - 核心OS: 操作系统内核 (与Mac OS X类似)

Cocoa Touch

Media Services

Core Services

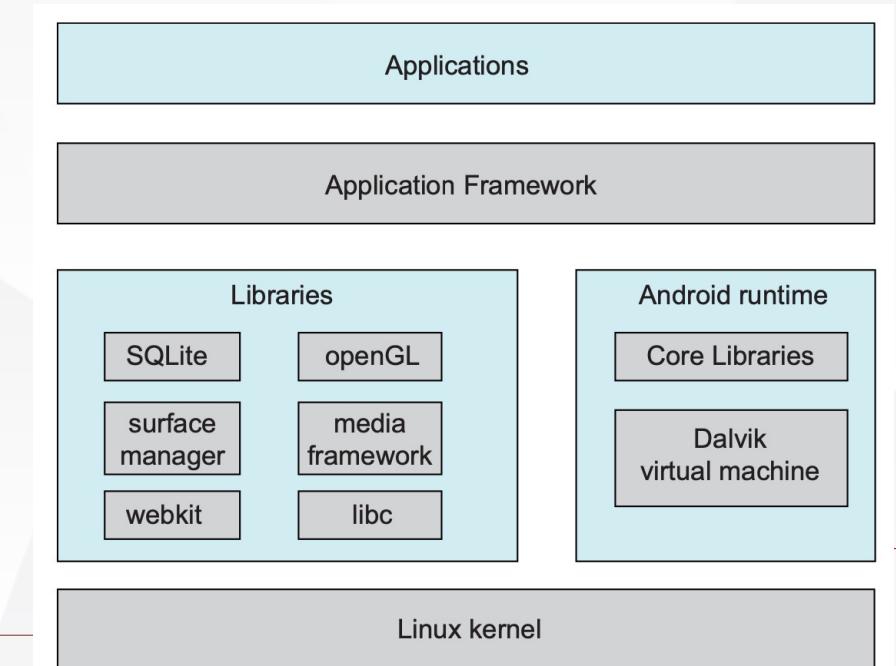
Core OS



# 操作系统的结构



- Android
  - Google主导设计，用于移动设备及电脑，开源项目
  - 分层方法，底层为Linux内核，上层包括一套核心库与Dalvik虚拟机
  - 用于Android应用程序（Java开发）的库包含：用于开发Web的框架、数据库支持、多媒体





# 操作系统的调试

- 调试：解决系统错误、解决瓶颈提高性能
  - 故障分析
    - 当进程发生故障，故障信息将写入日志文件（log file）
    - 应用程序故障，生成core dump文件，捕获当前进程的内存信息
    - 内核故障，生成crash dump文件，捕获内核信息
    - 根据日志信息，定位错误





# 操作系统的调试

- 调试：解决系统错误、解决瓶颈提高性能
  - 性能优化
    - 利用跟踪列表（trace listing），记录相关事件发生的时间与重要参数，显示系统性能
    - 使用交互工具
      - Unix命令top---显示系统使用的资源
      - Windows任务管理器





# 系统引导

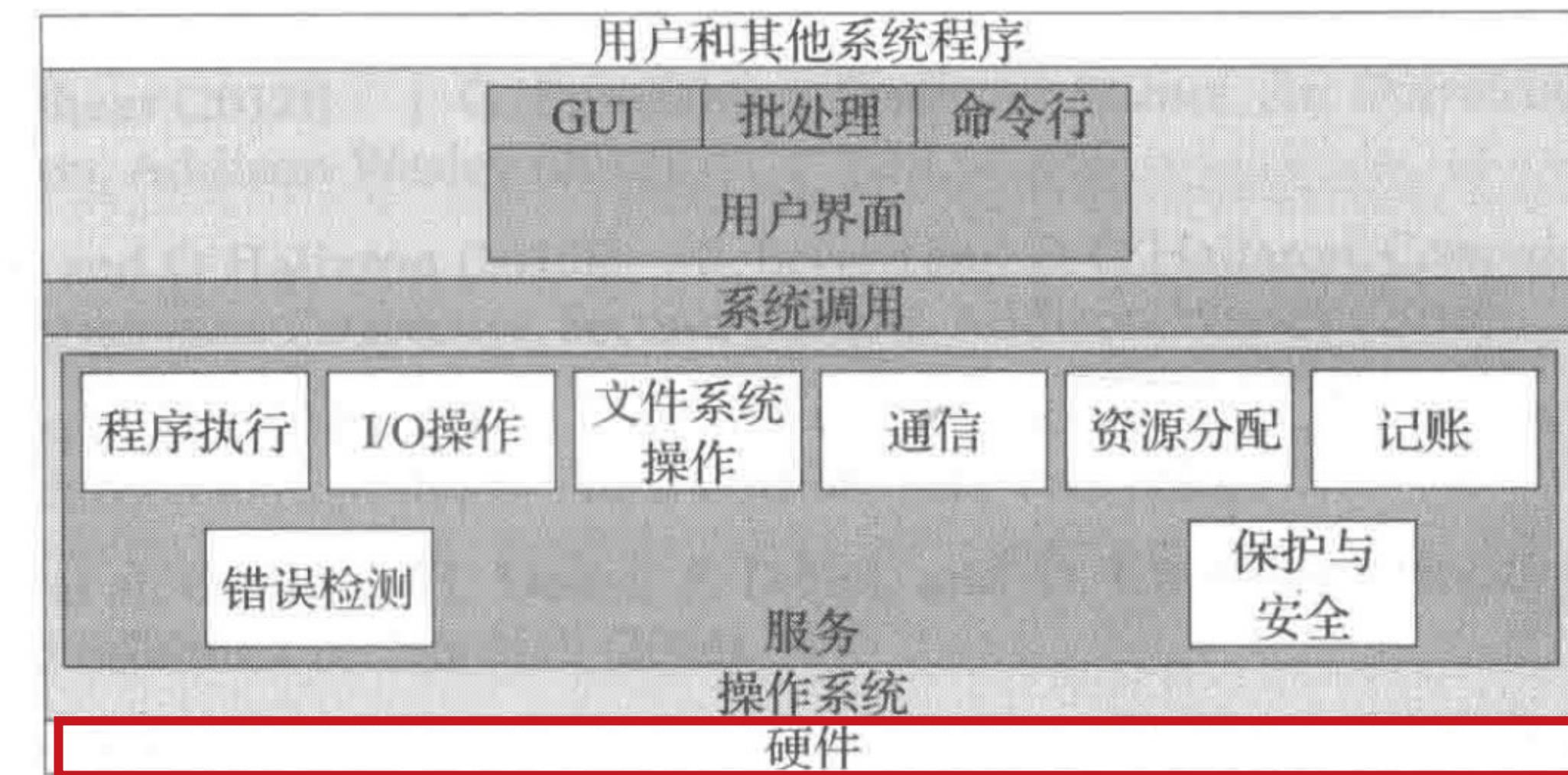


- 如何让硬件知道内核在哪里？加载内核以启动计算机的过程，称为系统引导 (booting)
- 计算机系统有一小块代码，称为引导程序，用于定位内核，加载到内存以便开始执行
- 当CPU收到重置事件（开机、重启），指令寄存器加载引导程序，以加载内核
  - 初始化系统
  - **引导程序一般存放于只读存储器 (Read-only memory, ROM) , 保证准确性**



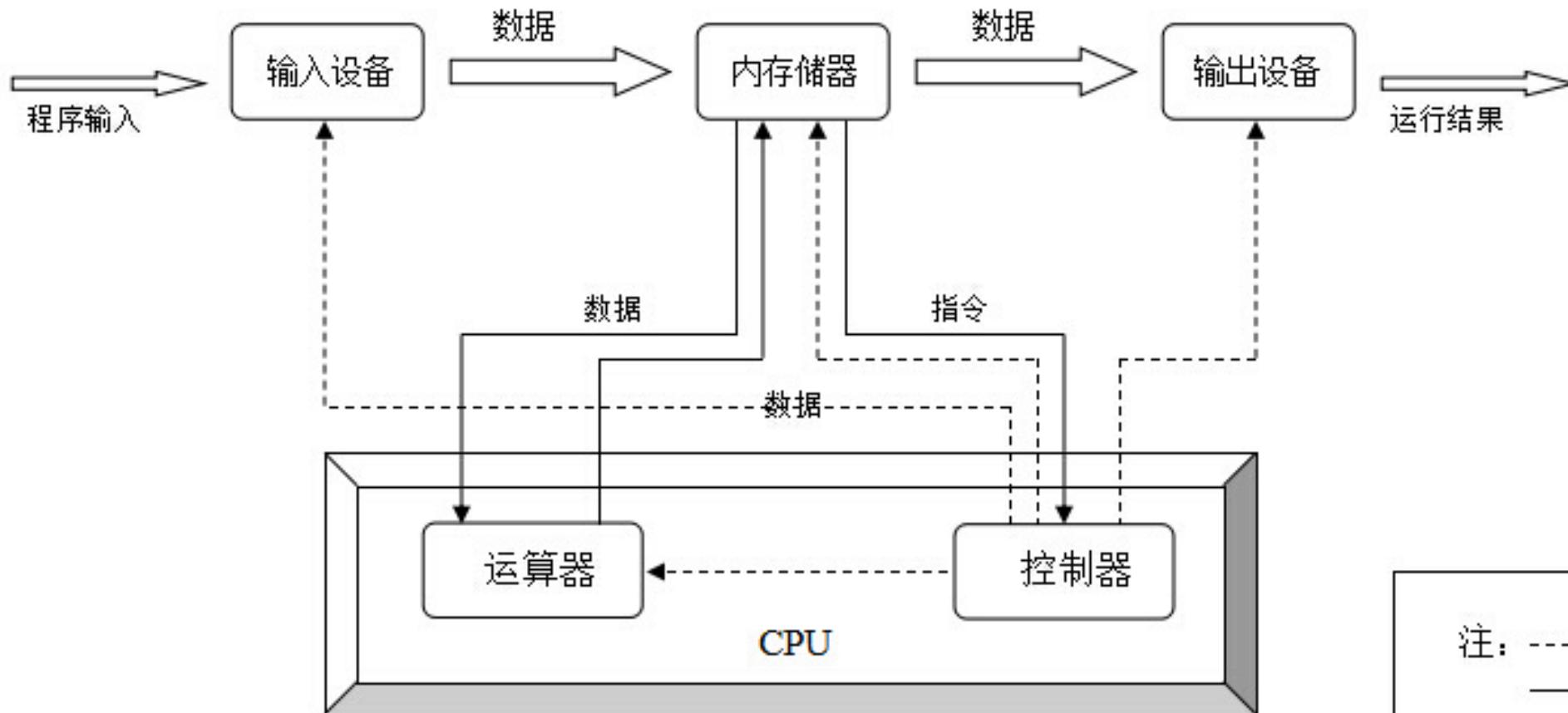


# 操作系统服务





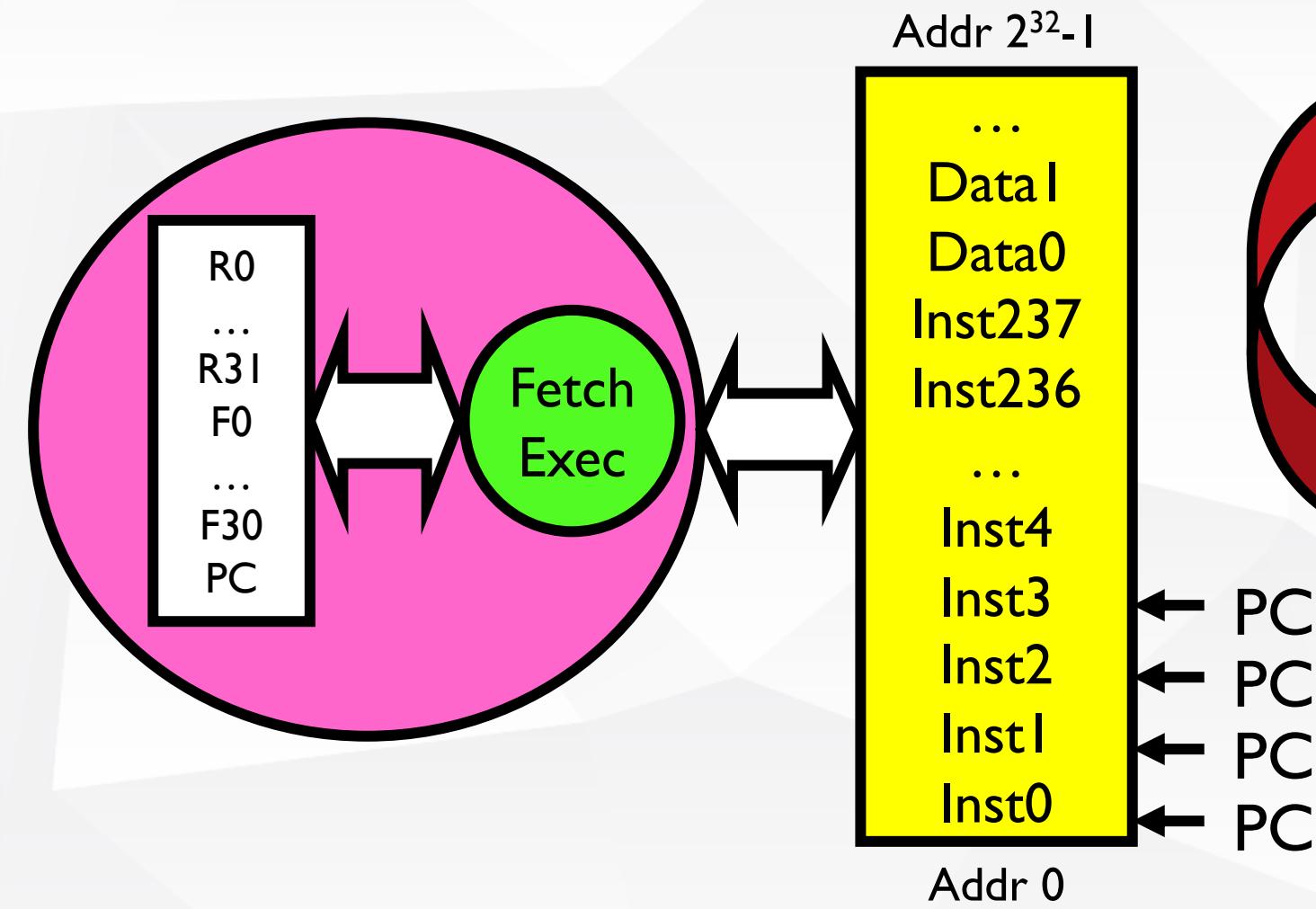
# 计算机的组成部件



计算机的工作原理示意图



# 指令的执行过程



指令执行过程:

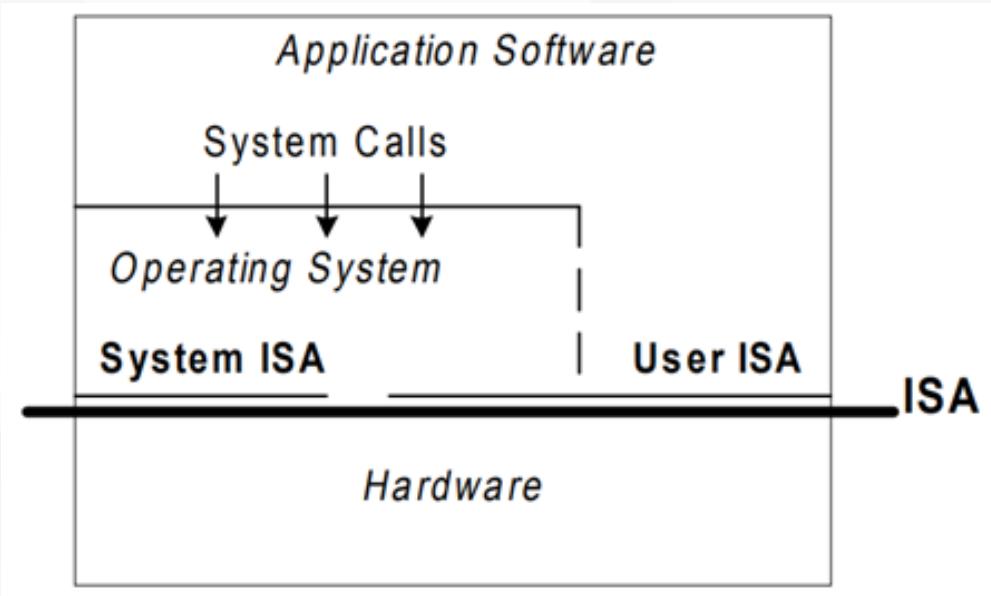
- 取指: 根据PC, 从存储器中取指令到IR (指令寄存器)
- 解码: 解译IR中的指令
- 执行: 控制CPU部件, 执行运算, 产生结果并写回, 同时在标志寄存器里设置运算结果标志; 跳转指令根据跳转地址修改PC, 其他指令递增PC值
- 重复上述过程



# 指令集体系结构 ( ISA )



- 一个ISA包括：用户级指令+特权级指令
  - 特权级ISA 适用于（操作系统）对硬件资源的管理
  - 用户级ISA 适用于应用程序和操作系统





- 应用程序可见的指令集

## Overview of User ISA

Integer	Memory	Control Flow	Floating Point
Add	Load byte	Jump	Add single
Sub	Load word	Jump equal	Mult. double
And	Store multiple	Call	Sqrt double
Compare	Push	Return	...
...	...	...	



# “特权级” / “系统级” 指令架构



- 特权指令：用于管理系统硬件资源
- 系统级软件、如操作系统可见的指令
- “系统级” 指令架构概貌
  - 特权级别 (Privilege levels) : 至少支持两种工作模式
  - 控制寄存器 (Control registers)
  - 管理关键资源的指令
    - Processor (scheduling, time-sharing)
    - Memory (e.g., isolated address spaces)
    - I/O (e.g., isolated disk storage space)





# RISC-V控制寄存器举例



- RISC-V中定义了三种特权模式，机器模式（machine mode）、监管者模式（supervisor mode）和用户模式（user mode）
- RISC-V架构定义了一些控制和状态寄存器（CSR），用于配置或记录运行的状态。
- CSR寄存器的访问采用专用的CSR指令，包括CSRRW、CSRRS、CSRRC、CSRRWI、CSRRSI以及CSRRCI指令
  - CSRRC (atomic read and clear bits in CSR) 读取CSR的值，将其扩展到32位，写入整数寄存器rd中

31	20 19	15 14	12 11	7 6	0
csr	rs1	funct3	rd	opcode	
12	5	3	5	7	
source/dest	source	CSRRW	dest	SYSTEM	
source/dest	source	CSRRS	dest	SYSTEM	
source/dest	source	CSRRC	dest	SYSTEM	
source/dest	uimm[4:0]	CSRRWI	dest	SYSTEM	
source/dest	uimm[4:0]	CSRRSI	dest	SYSTEM	
source/dest	uimm[4:0]	CSRRCI	dest	SYSTEM	

