操作系统A

Principles of Operating System

北京大学计算机科学技术系 陈向群

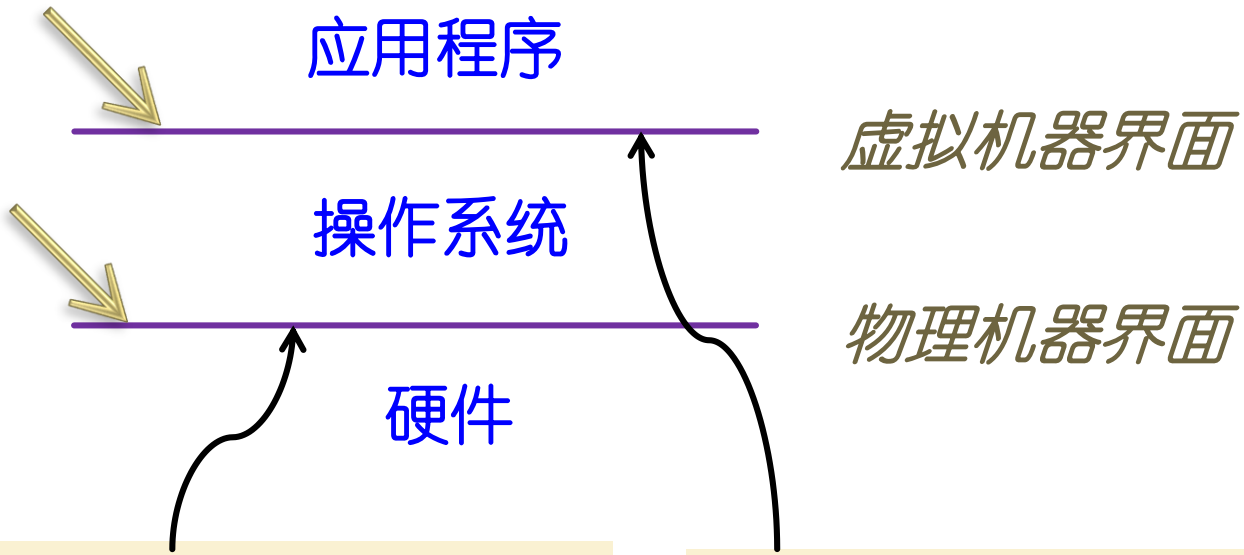
Department of computer science and Technology, Peking University

2020 Autumn

本章要求掌握的概念

CPU状态(模式)	特权指令	非特权指令
内核态/用户态	管态/目态	R0，R3
中断与异常	中断向量表	中断描述符
系统调用	机制与策略	……

本讲主要内容



- 操作系统运行环境
  - CPU状态
  - 中断/异常机制

- 操作系统运行机制
  - 系统调用

大纲

- 运行环境
  - 处理器及寄存器
  - 中断机制
  - 存储系统
  - I/O系统
  - 时钟
- 运行机制
  - 系统调用
- 机制与策略

讨论操作系统对运行硬件环境的要求

即操作系统设计者考虑的硬件问题

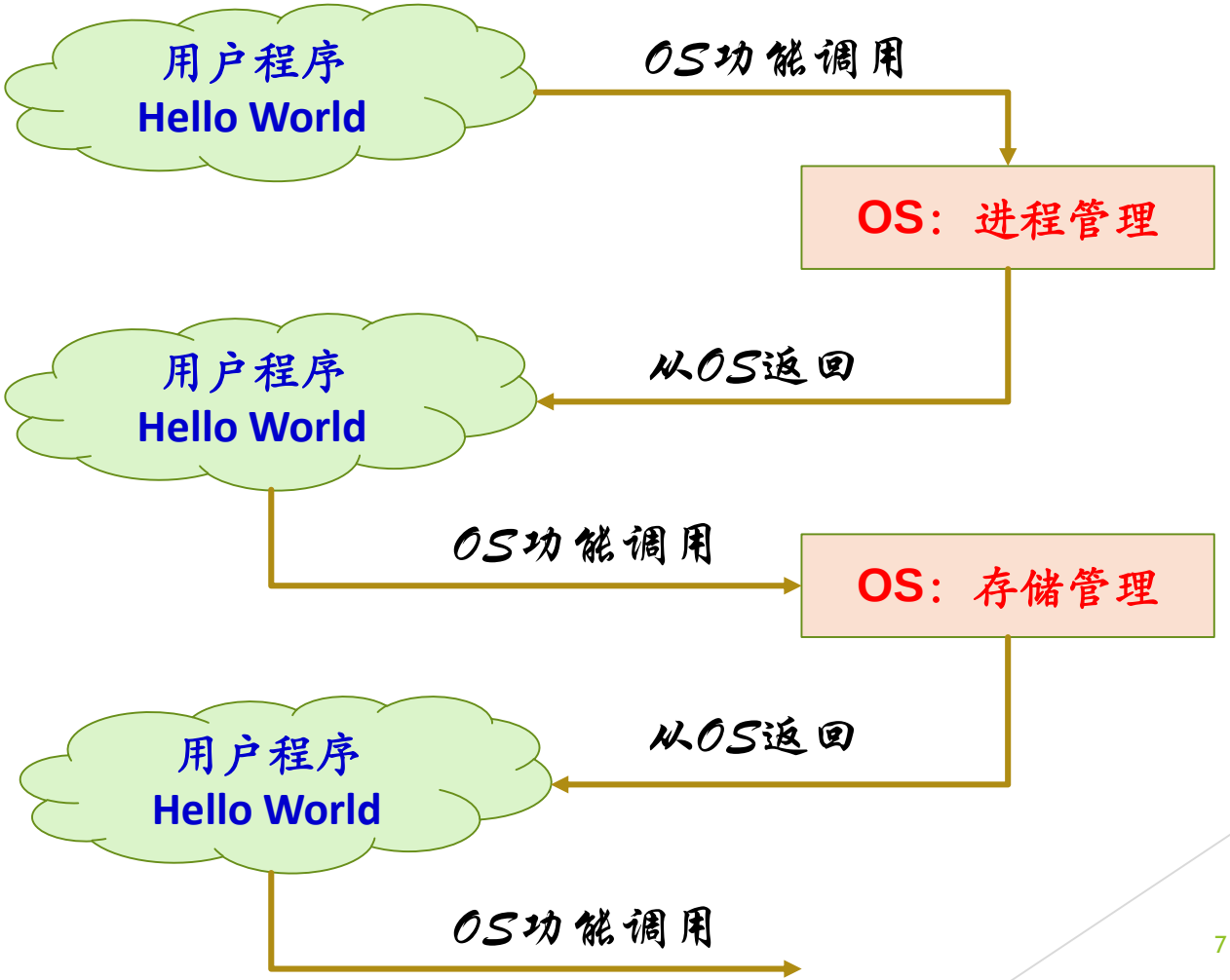
这一部分很重要

回顾

ICS课程之异常控制流ECF

- 理解ECF 将 帮助你 理解重要的系统概念
- 理解ECF 将 帮助你
  理解应用程序是如何与操作系统交互的
- 理解ECF 将 帮助你 编写有趣的新应用程序
- 理解ECF 将 帮助你 理解并发
- 理解ECF 将 帮助你 理解软件异常如何工作

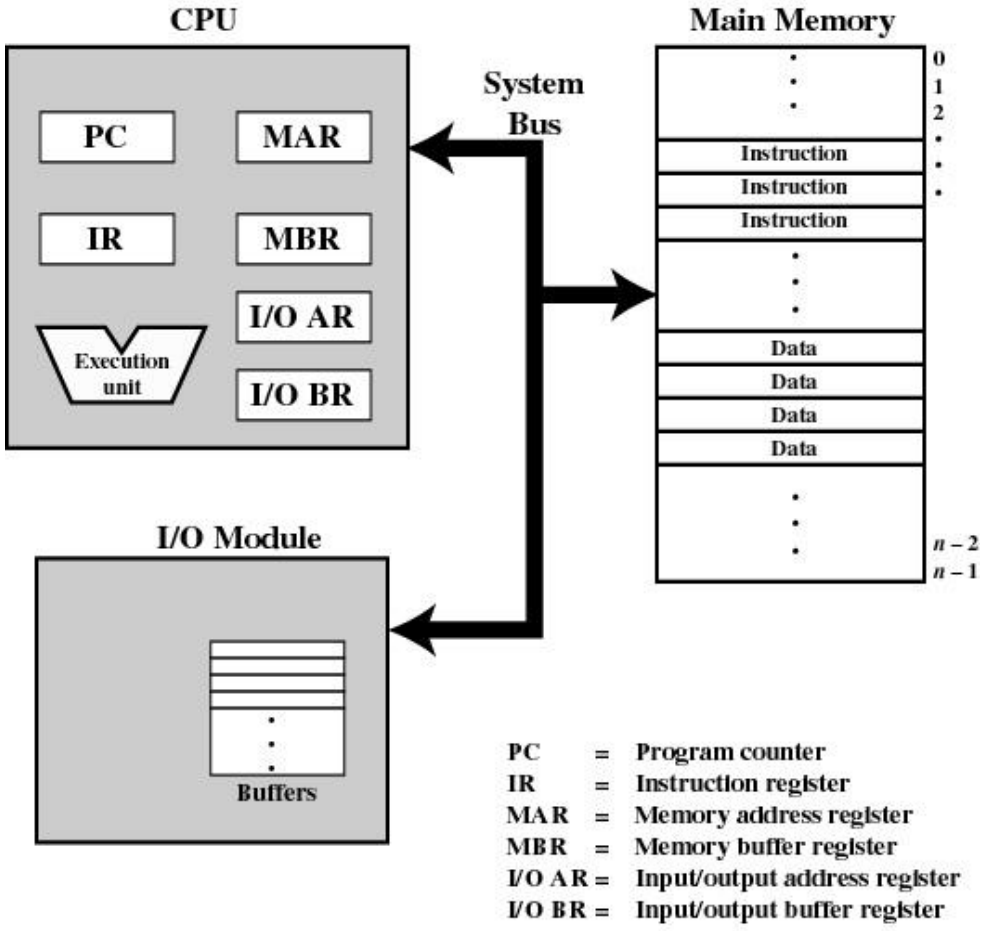
回顾：从用户与操作系统的关系看问题



操作系统设计者考虑的硬件问题……

操作系统运行环境

一、中央处理器（CPU）



- 处理器由运算器、控制器、一系列的寄存器以及高速缓存构成
- 两类寄存器
  - 用户可见寄存器：高级语言编译器通过优化算法分配并使用之，以减少程序访问内存次数
  - 控制和状态寄存器：用于控制处理器的操作

通常由操作系统代码使用

用户可见寄存器

- 包括数据寄存器、地址寄存器以及条件码寄存器
  机器语言可直接引用

  - 数据寄存器（data register）又称通用寄存器
    主要用于各种算术逻辑指令和访存指令
  - 地址寄存器（address register）用于存储数据及指令的物理地址、线性地址或者有效地址，用于某种特定方式的寻址。如index register、segment pointer、stack pointer
  - 条件码寄存器：保存CPU操作结果的各种标记位
    如算术运算产生的溢出、符号等等

控制和状态寄存器

- 用于控制处理器的操作
- 须在某种特权级别下可以访问、修改

- 常见的控制和状态寄存器
  - 程序计数器（PC：Program Counter），记录将要取出的指令的地址
  - 指令寄存器（IR：Instruction Register），包含最近取出的指令
  - 程序状态字（PSW：Program Status Word），记录处理器的运行状态如条件码、模式、控制位等

1. 操作系统的需求——保护

- 从操作系统的特征考虑

  并发、共享

  提出要求 → 实现保护与控制

需要硬件提供基本运行机制：

- 处理器具有特权级别，能在不同的特权级运行的不同指令集合
- 硬件机制可将OS与用户程序隔离

2. 处理器的状态（模式mode）

- 现代处理器通常将CPU状态设计划分为两种、三种或四种
- 在程序状态字寄存器PSW中专门设置一位，根据运行程序对资源和指令的使用权限而设置不同的CPU状态

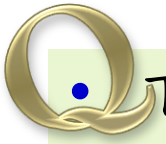
例:X86架构中的EFLAGS寄存器

描述符中也设置了权限级别

3. 特权指令和非特权指令

- 操作系统需要 2 种CPU状态
  - 内核态(Kernel Mode)：运行操作系统程序
  - 用户态(User Mode)：运行用户程序

- 特权(privilege)指令：只能由操作系统使用、用户程序不能使用的指令
- 非特权指令：用户程序可以使用的指令



- 下列哪些是特权指令？哪些是非特权指令？
  - 启动I/O　控制转移　内存清零　修改程序状态字
    设置时钟　算术运算　允许/禁止中断　访管指令
    取数指令　停机

4. 实例：x86系列处理器

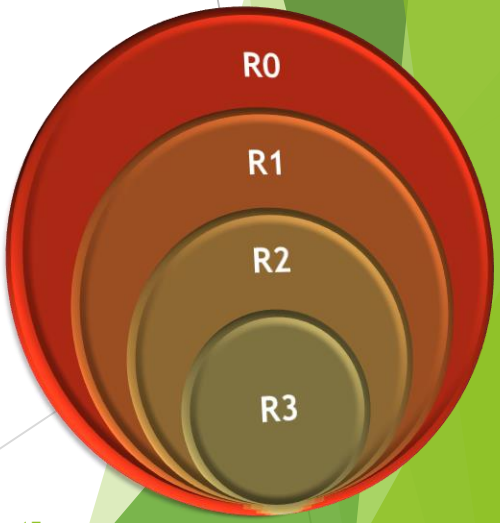
- X86支持4个处理器特权级别

  特权环：R0、R1、R2和R3

  - 从R0到R3，特权能力由高到低
  - R0相当于内核态；R3相当于用户态；R1和R2则介于两者之间
  - 不同级别能够运行的指令集合不同

- 目前大多数基于x86处理器的操作系统

  只用了R0和R3两个特权级别



5. CPU状态之间的转换

- 用户态 → 内核态

  唯一途径 → 中断/异常/陷入机制

- 内核态 → 用户态

  设置程序状态字PSW

管态、特权态、内核态、核心态、系统态

目态、普通态、用户态

一条特殊的指令：陷入指令（又称访管指令）

提供给用户程序的接口，用于调用操作系统的功能（服务）

例如：int，trap，syscall，sysenter/sysexit

16 因为内核态也被称为supervisor mode

二、中断机制

- 中断对于操作系统的重要性

  就如 汽车发动机、飞机引擎 的作用

  →→ 操作系统是由“中断驱动”或“事件驱动”的

主要作用

- 及时处理设备发来的中断请求
- 可使OS可以捕获用户程序提出的服务请求
- 防止用户程序执行过程中的破坏性活动 ……等等

中断与异常的引入原因

术语演化
的历史背景

- 中断的引入：为了支持CPU和设备之间的并行操作
  - 当CPU启动设备进行输入/输出后，设备便可以独立工作，CPU转去处理与此次输入/输出不相关的事情；当设备完成输入/输出后，通过向CPU发中断报告此次输入/输出的结果，让CPU决定如何处理以后的事情
- 异常的引入：表示CPU执行指令时本身出现的问题
  - 如算术溢出、除零、取数时的奇偶错，访存地址时越界或执行了“陷入指令” 等，这时硬件改变了CPU当前的执行流程，转到相应的错误处理程序或异常处理程序或执行系统调用

1. 中断/异常的概念

何时？
何处？

硬件完成这一过程

在以后某个时刻继续

- CPU对系统发生的某个事件作出的一种反应
- CPU暂停正在执行的程序，保留现场后自动转去执行相应事件的处理程序，处理完成后返回断点，继续执行被打断的程序

- 事件的发生改变了处理器的控制流

特点：
- 是随机发生的
- 是自动处理的
- 是可恢复的

2. 事件

中断：外部事件，正在运行的程序所不期望的
异常：由正在执行的指令引发

事件
- 中断(外中断)
  - I/O中断
  - 时钟中断
  - 硬件故障
- 异常(内中断)
  例外
  - 系统调用
  - 页错误/页故障
  - 保护性异常
  - 断点指令
  - 其他程序性异常(如算术溢出等)

举例

	Unexpected	Deliberate
Exceptions (sync)	fault	syscall trap
Interrupts (async)	interrupt	software interrupt

ICS：中断与异常的小结

类别	原因	异步/同步	返回行为
中断 Interrupt	来自I/O设备、其他硬件部件	异步	总是返回到下一条指令
陷入Trap	有意识安排的	同步	返回到下一条指令
故障Fault	可恢复的错误	同步	返回到当前指令
终止Abort	不可恢复的错误	同步	不会返回

3. 中断/异常机制工作原理

- 中断/异常机制是现代计算机系统的核心机制之一

  通过**硬件和软件相互配合**

  而使计算机系统得以充分发挥能力

- **硬件该做什么事？** —— 中断/异常响应

  捕获中断源发出的中断/异常**请求**，以一定方式**响应**，将处理器**控制权交给**特定的处理程序

- **软件要做什么事？** —— 中断/异常处理程序

  识别中断/异常类型并完成**相应的处理**

4. 中断响应

中断响应：

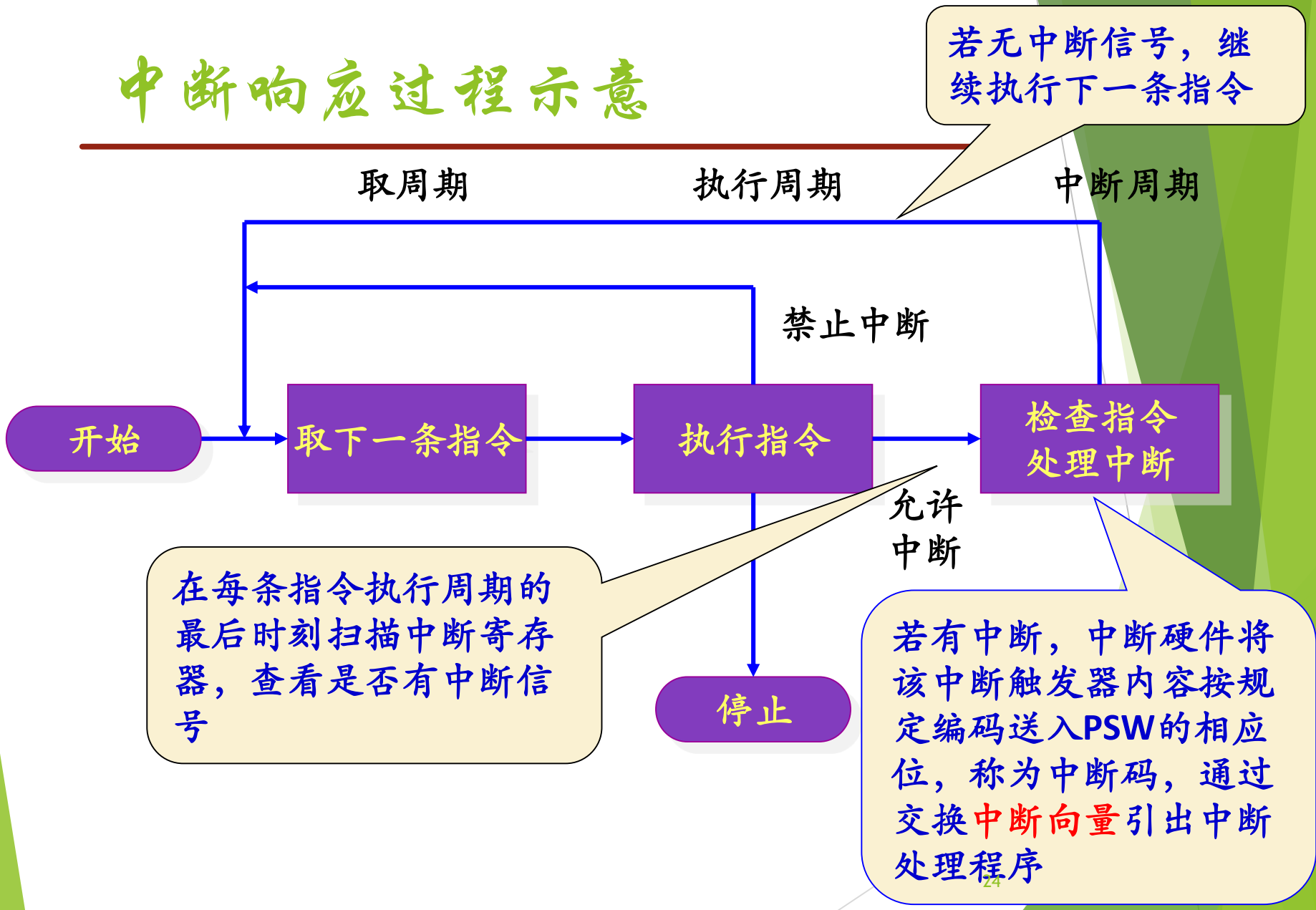
发现中断、接收中断的过程

由中断硬件部件完成

处理器控制部件中　设有　中断寄存器

CPU何时响应中断？

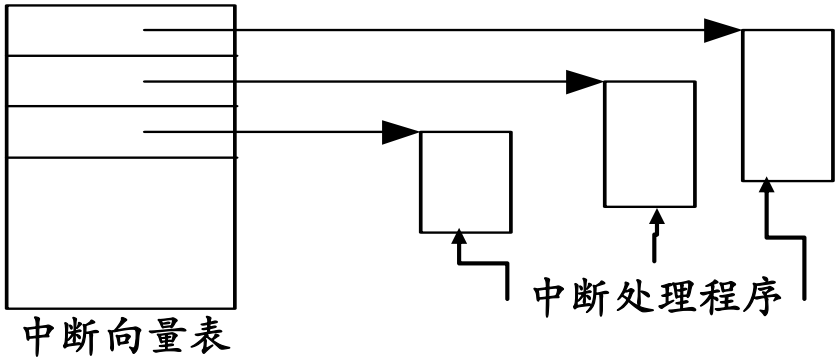
中断响应过程示意


若无中断信号，继续执行下一条指令
取周期
执行周期
中断周期
禁止中断
开始
取下一条指令
执行指令
检查指令处理中断
允许中断
在每条指令执行周期的最后时刻扫描中断寄存器，查看是否有中断信号
停止
若有中断，中断硬件将该中断触发器内容按规定编码送入PSW的相应位，称为中断码，通过交换中断向量引出中断处理程序

5. 中断向量表

- **中断向量**

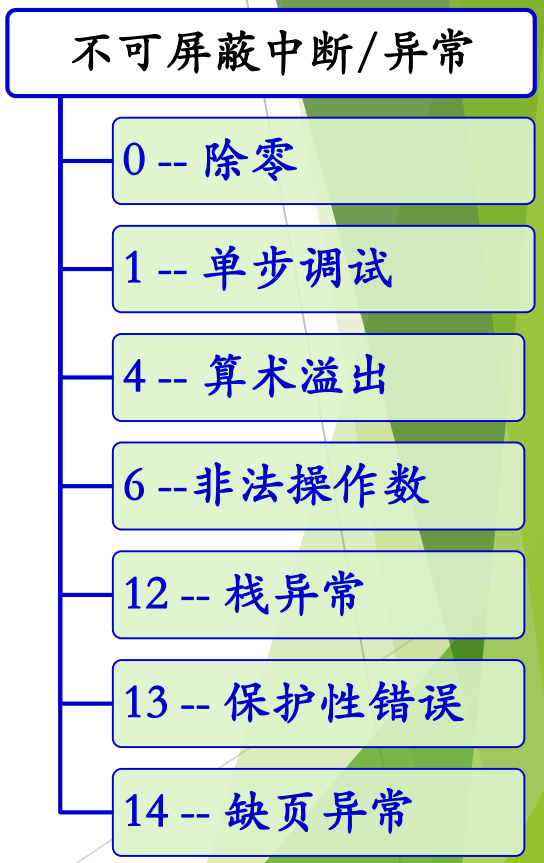
一个内存单元，存放中断处理程序入口地址和程序运行所需的处理机状态字



硬件执行流程按中断号/异常类型的不同，通过中断向量表转移控制权给中断处理程序

Linux中的中断向量

向量范围	用途
0～19	不可屏蔽中断和异常
20～31	Intel保留
32～127	外部中断（IRQ）
128（0x80）	用于系统调用的可编程异常
129～238	外部中断
239	本地APIC时钟中断
240	本地APIC高温中断
241～250	Linux保留
251～253	处理器间中断
254	本地APIC错误中断
255	本地APIC伪中断



Identifier	Description
0	Divide error
1	Debug exception
2	Non-maskable interrupt
3	Breakpoint
4	Overflow
5	Bounds check
6	Invalid opcode
7	Coprocessor not available
8	Double fault
9	(reserved)
10	Invalid TSS
11	Segment not present
12	Stack exception
13	General protection fault
14	Page fault
15	(reserved)
16	Coprocessor error
17	alignment error (80486)
18-31	(reserved)
32-255	External (HW) interrupts

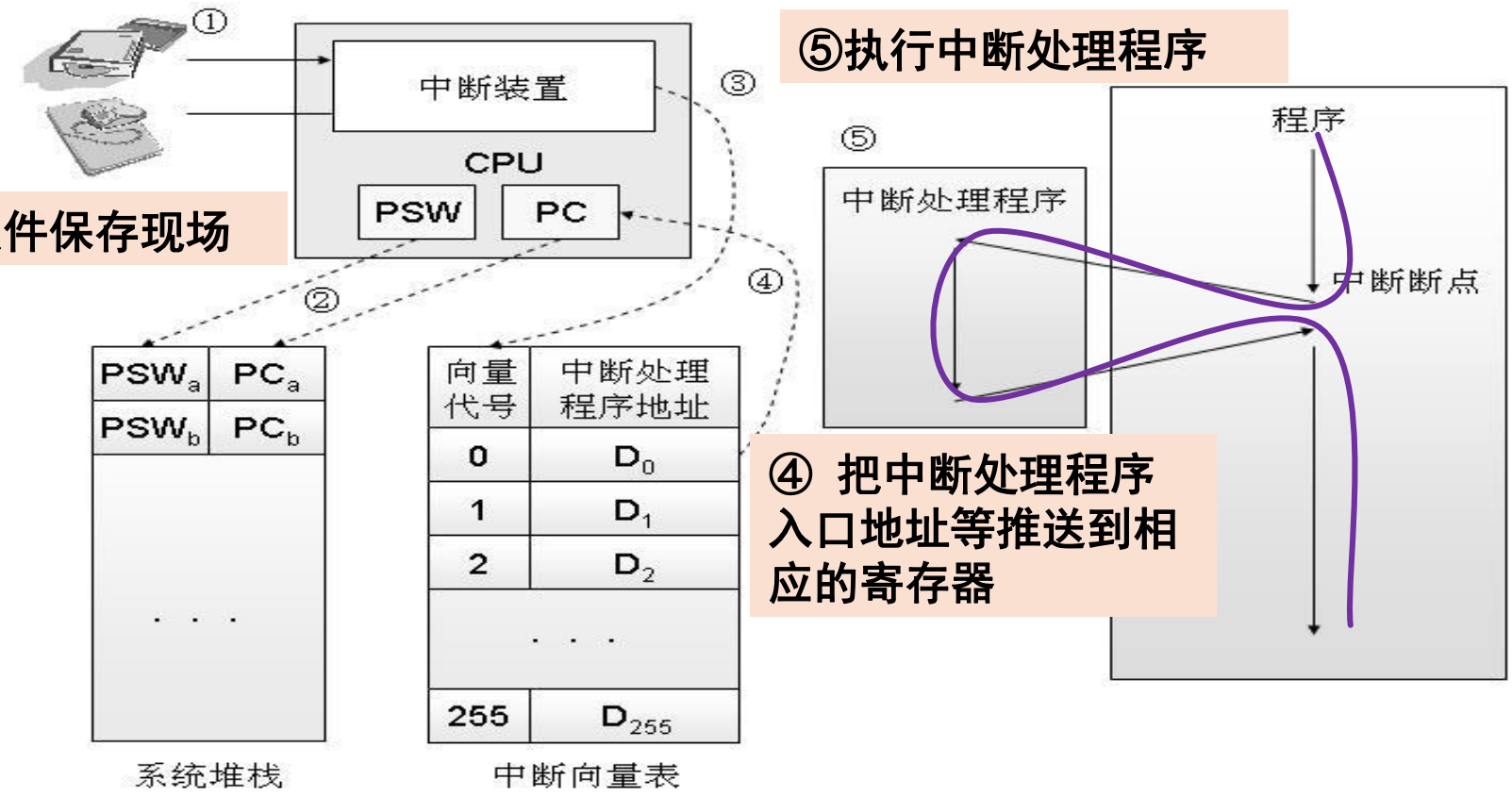
思考题：对5-12、16-17给出描述

中断响应示意图

① 设备发中断信号

③ 根据中断码查表

⑤执行中断处理程序



② 硬件保存现场

④ 把中断处理程序入口地址等推送到相应的寄存器

6. 中断处理程序

- 设计操作系统时，为每一类中断/异常事件编好相应的处理程序，并设置好中断向量表

- 系统运行时若响应中断，中断硬件部件将CPU控制权转给**中断处理程序**：
  - 保存相关寄存器信息
  - 分析中断/异常的具体原因
  - 执行对应的处理功能
  - 恢复现场，返回被事件打断的程序

软件提前设置好
硬件部件来执行

中断/异常机制小结（1/2）

以设备输入输出中断为例：

硬件

- 打印机给CPU发中断信号
- CPU处理完当前指令后检测到中断，判断出中断来源并向相关设备发确认信号

硬件

- CPU开始为软件处理中断做准备：
  - 处理器状态被切换到内核态
  - 在系统栈中保存被中断程序的重要上下文环境，主要是程序计数器PC、程序状态字PSW

中断/异常机制小结（2/2）

硬件

- CPU根据中断码查中断向量表，获得与该中断相关的处理程序的入口地址，并将PC设置成该地址，新的指令周期开始时，CPU控制转移到中断处理程序
- 中断处理程序开始工作
  - 在系统栈中保存现场信息
  - 检查I/O设备的状态信息，操纵I/O设备或者在设备和内存之间传送数据等等

软件

- 中断处理结束时，CPU检测到中断返回指令，从系统栈中恢复被中断程序的上下文环境，CPU状态恢复成原来的状态，PSW和PC恢复成中断前的值，CPU开始一个新的指令周期

硬件

举例：I/O中断处理程序

通常分为两类处理：

- I/O操作正常结束
  - 若有程序正等待此次I/O的结果，则应将其唤醒
  - 若要继续I/O操作，需要准备好数据重新启动I/O
- I/O操作出现错误
  - 需要重新执行失败的I/O操作
  - 重试次数有上限，达到时系统将判定硬件故障

举例：时钟中断

- 系统推动力：时钟中断处理程序通常做与系统运转、管理和维护相关的工作
- 维护软件时钟：系统有若干个软件时钟，控制定时任务以及进程的处理器时间配额，时钟中断需要维护、定时更新这些软件时钟
- 处理器时间调度：维护当前进程时间片软件时钟，并在当前进程时间片到时以后运行调度程序选择下一个被调度的进程
- 控制系统定时任务：通过软件时钟和调度程序定时激活一些系统任务，如监测死锁、系统记帐、系统审计等
- 实时处理

中断/异常机制实例

IA32体系结构对中断的支持

基本概念——X86处理器

- 中断
  - 由硬件信号引发的，分为可屏蔽和不可屏蔽中断
- 异常
  - 由指令执行引发的，比如除零异常
  - 80x86处理器发布了大约20种不同的异常
  - *对于某些异常，CPU会在执行异常处理程序之前产生硬件出错码，并压入内核态堆栈*
- 系统调用
  - 异常的一种，用户态到系统态的唯一入口

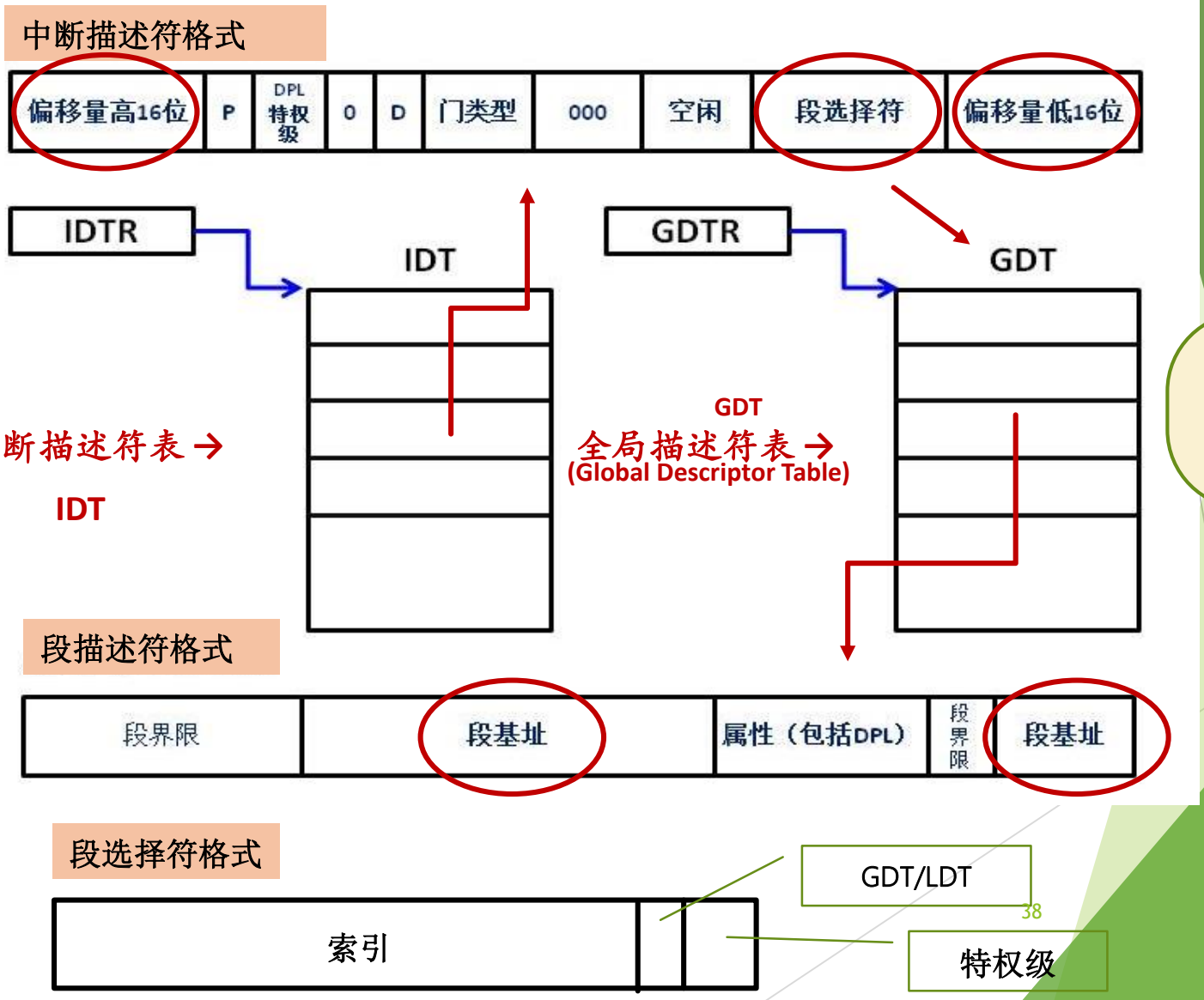
IA32体系结构对中断的支持(1/7)

- 中断控制器（PIC或APIC）
  - 负责将硬件的中断信号转换为中断向量，引发CPU中断
- 实模式：中断向量表（Interrupt Vector）
  - 存放**中断服务程序的入口地址**
    - 入口地址＝段地址左移4位＋偏移地址
    - 不支持CPU运行状态切换
    - 中断处理与一般的过程调用相似
- 保护模式：中断描述符表（Interrupt Descriptor table）
  - 采用门(gate)描述符数据结构描述中断向量

IA32体系结构对中断的支持(2/7)

- 中断向量表/中断描述符表
  - 表项包含四种类型门描述符
    - 任务门(Task Gate)
      - 中断发生时，必须取代当前进程的那个进程的TSS选择符存放在任务门中（Linux没有使用任务门）
    - 中断门(Interrupt Gate)
      - 给出段选择符 (Segment Selector)、中断/异常程序的段内偏移量 (Offset)
      - 通过中断门后系统会自动禁止中断
    - 陷阱门(Trap Gate)
      - 与中断门类似，但通过陷阱门后系统不会自动关中断
    - 调用门(Call Gate)

IA32体系结构对中断的支持(3/7)



中断服务程序入口地址 = 段基址＋偏移

X86处理器对中断的支持(4/7)

中断/异常的**硬件处理过程**:

- 确定与中断或异常关联的**向量i**
- 通过IDTR寄存器找到IDT表，获得**中断描述符**（表中的第i项）
- 从GDTR寄存器获得GDT的地址；结合中断描述符中的**段选择符**，在GDT表获取对应的段描述符；从**该段描述符**中得到中断或异常处理程序所在的**段基址**
- 特权级检查

IA32体系结构对中断的支持(5/7)

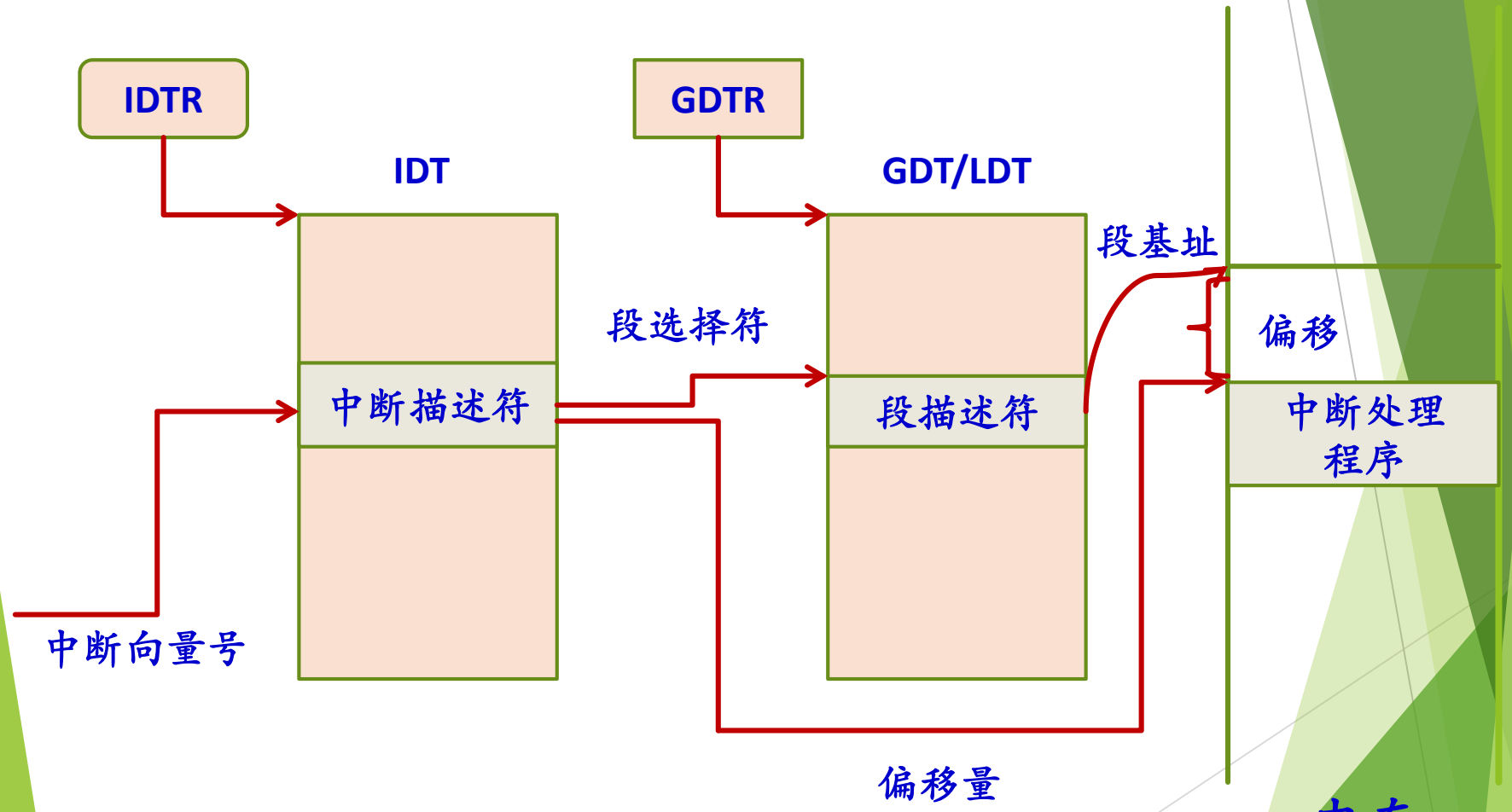
- 特权级检查
  - 确保CPL（CS寄存器中）≤ 门描述符DPL
    - 避免应用程序访问特殊的陷阱门或中断门
  - 确保CPL ＞ 段描述符DPL
    - 当前特权级不低于中断处理程序的特权级

- CPL（当前程序运行的权限级别）与门描述符DPL作比较，有效权限级别高于DPL，则通过，否则不允许访问
- 再与段描述符的DPL作比较，有效权限级别高于DPL，则通过，否则不允许访问
- 例： CPL=2，门描述符DPL=3，段描述符DPL=1，能否访问？（注：数值越小，权限越高）

X86处理器对中断的支持(6/7)

- 检查是否发生了特权级的变化，如果是，则进行堆栈切换(必须使用与新的特权级相关的栈)
- 硬件压栈，保存上下文环境；如果异常产生了硬件出错码，也将它保存在栈中
- 如果是中断，清IF位
- 通过中断描述符中的段内偏移量和段描述符中的基地址，找到中断/异常处理程序的入口地址，执行其第一条指令

IA32体系结构对中断的支持(7/7)



三、存储系统

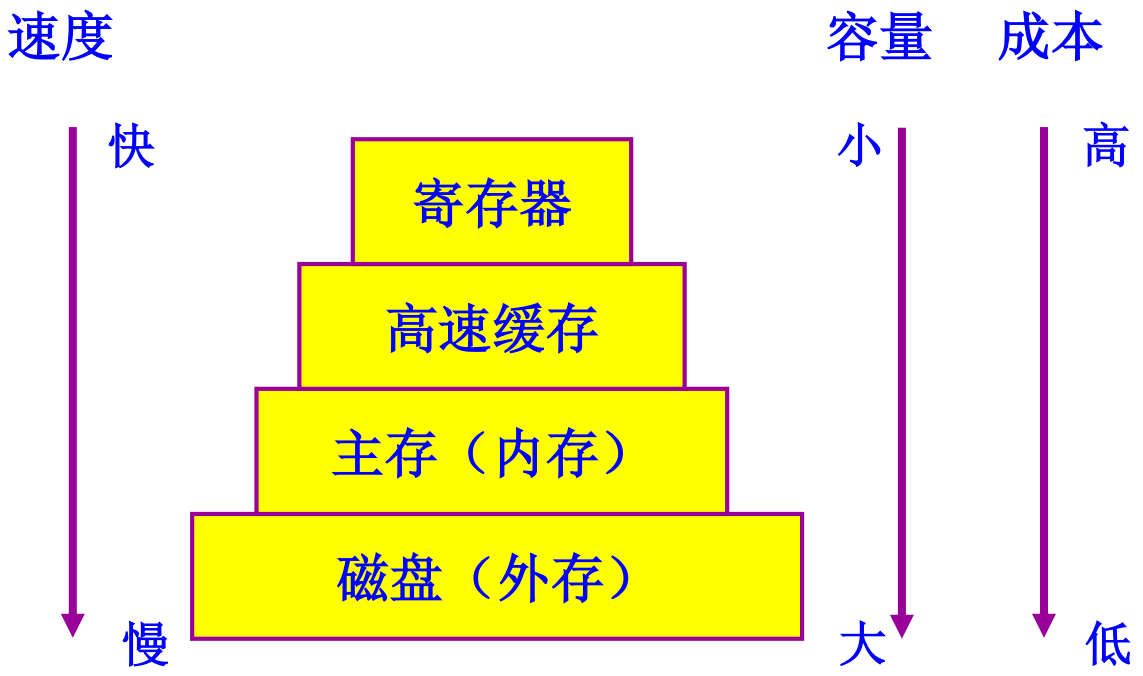
支持OS运行硬件环境的一个重要方面：

- 进程必须将其程序和数据放在内存中才能运行
- 操作系统本身也要放在内存中运行
- 多道程系统中，若干个程序和相关的数据要放入内存

→→ 操作系统要管理、保护程序和数据，使它们不至于受到破坏

1. 存储器的层次结构

- 存储系统设计三个问题：容量、速度和成本



2. 存储访问局部性原理

- 提高存储系统性能的关键：存储访问局部性原理

- 程序执行时，有很多循环和子程序调用，一旦进入这样的程序段，就会重复存取相同的指令集合
- 对数据存取也有局部性，在较短的时间内，稳定保持在一个存储器的局部区域

- 处理器主要和存储器的局部打交道
- 经过一段时间以后，使用的代码和数据集合会改变

3. 存储分块

- 存储最小单位：“二进位”，包含信息为0或1
- 编址单位：字节　或　字
- 为简化分配和管理，存储器被划分为块，称一个物理页（Page Frame），亦称**页框、页帧**
- 块（页）的大小

512B、1KB、4KB、8KB、16KB、64KB、256KB、1MB、4MB、16MB

4. 高速缓存

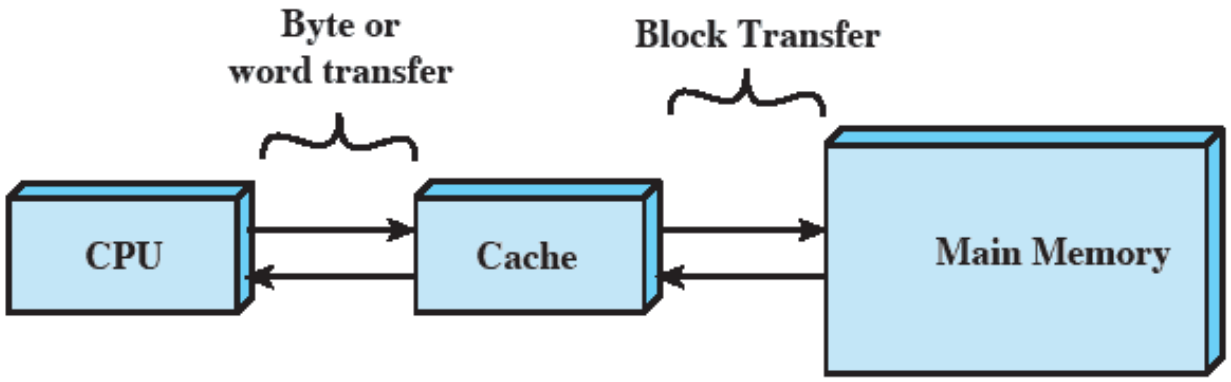
- 高速缓存对操作系统不可见，但了解其原理非常重要

- 问题的提出：
  - CPU取指令时至少访问一次存储器
  - CPU速度远远快于内存访问速度

- 解决方案：

  在CPU和内存之间设置一个容量小、速度快的存储器——Cache(SRAM)

高速缓存和内存



- 包含一部分内存数据的副本
- 处理器读取（一个字节或字）时，先检查Cache
  - ✓ 在Cache中 或 不在Cache中
- 一次从内存到Cache的数据读取(以Block为单位)，很可能满足多次读取需求(访问局部性原理)

四、I/O访问技术

I/O控制使用下面几种技术:

- 程序控制方式
- 中断驱动方式
- 直接存储器存取（DMA）方式

1.程序控制I/O技术

- 由CPU提供I/O相关指令来实现
- I/O处理单元处理请求并设置I/O状态寄存器相关位
- 不中断处理器，也不给处理器警告信息
- CPU定期轮询I/O单元的状态，直到处理完毕

- I/O软件包含直接操纵I/O的指令
  - 控制指令: 用于激活外设，并告诉它做什么
  - 状态指令: 用于测试I/O控制中的各种状态和条件
  - 数据传送指令: 用于在设备和主存之间来回传送数据
- 主要问题：CPU必须关注I/O处理单元的状态，因而耗费大量时间轮询信息，降低了系统性能

2. 中断驱动I/O技术

- 为解决程序控制I/O方式的问题
  让CPU从轮询任务中解放出来
  → 使I/O操作和指令执行并行起来

- 具体做法：
  当I/O处理单元准备好与设备交互的时候
  通过物理信号通知CPU，即中断CPU的执行

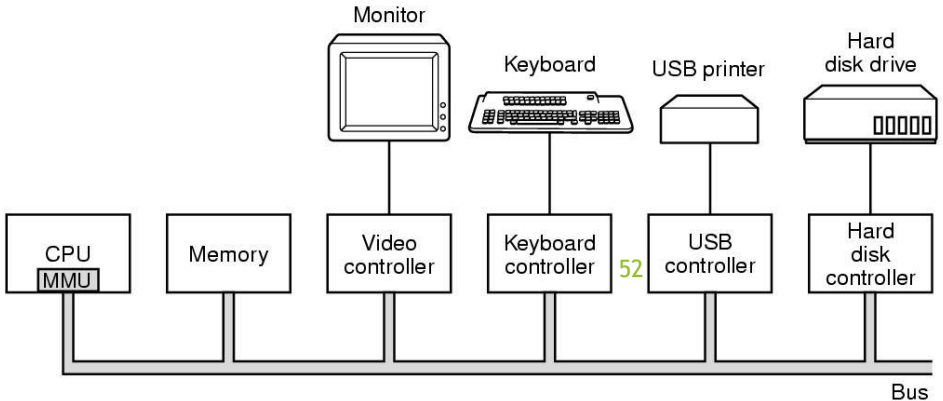
3.DMA技术（1/3）

问题

- 引入中断 提高了CPU处理I/O的效率
- 当CPU和I/O间传送数据时，效率仍旧不高

解决方法：

- 直接存储器访问（DMA：Direct Memory Access）
- 通过系统总线中一独立控制单元——DMA控制器，自动控制成块数据在内存和I/O单元间的传送
- 提高了处理I/O的性能



DMA技术（2/3）

- 当CPU需要读写一整块数据时

  给DMA控制单元发送一条命令

  包含：是否请求一次读或写，I/O设备的编址，开始读或写的内存编址，需要传送的数据长度等信息

- CPU发送完命令后就可处理其他事情

  DMA控制器将自动完成数据的传送

  当传送过程完成后， DMA控制器给CPU发一个中断，CPU只在开始传送和传送结束时进行关注

DMA技术（3/3）

- CPU和DMA传送不完全并行

  因为　会有总线竞争的情况发生

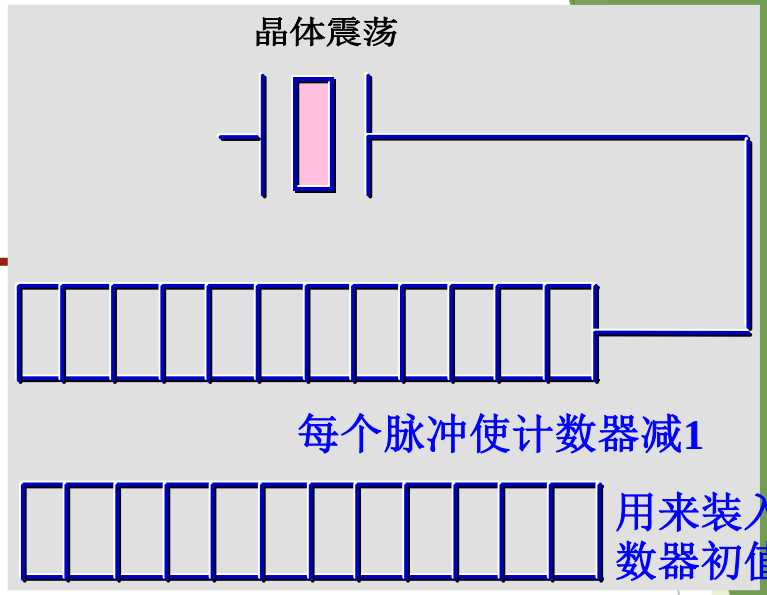
  - CPU用总线时可能稍作等待
  - 不会引起中断
  - 不引起程序上下文的保存
- 通常过程只有一个总线周期
- 在DMA传送时，CPU访问总线速度会变慢
- 对于大量数据I/O传送，DMA技术是很有价值

五、时钟

时钟为计算机完成以下必不可少的工作：

- 在多道程序运行环境中，为系统发现陷入死循环（编程错误）的作业，防止机时的浪费
- 在交互式系统中，实现进程间按时间片轮转
- 在实时系统中，可完成延时与超时控制等功能
- 定时唤醒要求延迟执行的各外部事件（如定时为各进程计算优先数，银行中定时运行某类结账程序等）
- 记录用户使用设备的时间和某外部事件发生时间
- 维持日历时间

时钟



- 时钟

  一种周期性信号源

  处理延时、超时、定时等与时间有关的事件

- 硬件时钟寄存器

  按时钟电路所产生的脉冲数对时钟寄存器进行加1或减1的工作

时钟分类(1/2)

- 绝对时钟：记录当前时间

  一般来说，绝对时钟准确，当停机时，绝对时钟值仍然自动修改

- 相对时钟(间隔时钟)：用时钟寄存器实现

  设置时间间隔初值，每经过一个单位时间，时钟值减1，直到该值为负时，则触发时钟中断，并进行相应中断处理

时钟分类(2/2)

硬件时钟:

某个寄存器来模拟

(根据脉冲频率定时加1，减1)

软件时钟:

用作相对时钟，用内存单元来模拟时钟

应用例子:

CPU保护：防止进程得到CPU后不放弃控制权

解决：分配给每个进程一段时间（时间片），时间片到，发时钟中断，控制权交给操作系统

例子：x86体系结构的定时硬件(1/3)

- x86体系结构上有4种定时硬件
  - 实时时钟（RTC）
  - 时间戳计数器（TSC）
  - 可编程间隔定时器（PIT）
  - SMP系统中的本地APIC定时器
- 内核使用RTC和TSC跟踪当前时间
- 内核对PIT和本地APIC编程，使其以固定的、预先定义的频率发出中断

x86体系结构的定时硬件(2/3)

实时时钟（RTC）

- 集成在CMOS芯片上，独立于CPU和所有其他芯片
- 电池供电
- 在IRQ8上发出周期性中断
- Linux只用RTC来获取时间和日期

时间戳计数器（TSC）

- 64bits（Pentium之后）
- 读：汇编指令rdtsc
- 每个处理器时钟信号到来时TSC寄存器的值加1

  （例如：对于500Mhz的处理器，每2ns时间戳计数器加1）
- 可以获得ns级的时间精度

x86体系结构的定时硬件(3/3)

可编程间隔定时器(PIT)

- 通过编程设置固定频率周期性的产生时钟中断IRQ0上产生
  → 系统“心跳”
- Linux设置PIT大概每10ms发出一次中断，这个时间间隔称为一个节拍(Tick)
- PIT 中断为全局性中断，系统中的任一CPU都可以对其处理

本地APIC定时器

- 用于SMP系统
- 32bits，基于总线时钟信号
- 作用：只把中断传递给自己的处理器，为局部性中断

系统调用的作用

操作系统运行机制

系统调用(System call)

Linux操作系统提供多少个系统调用？如何在Linux中增加一个新的系统调用？

系统调用是什么？

系统调用的作用

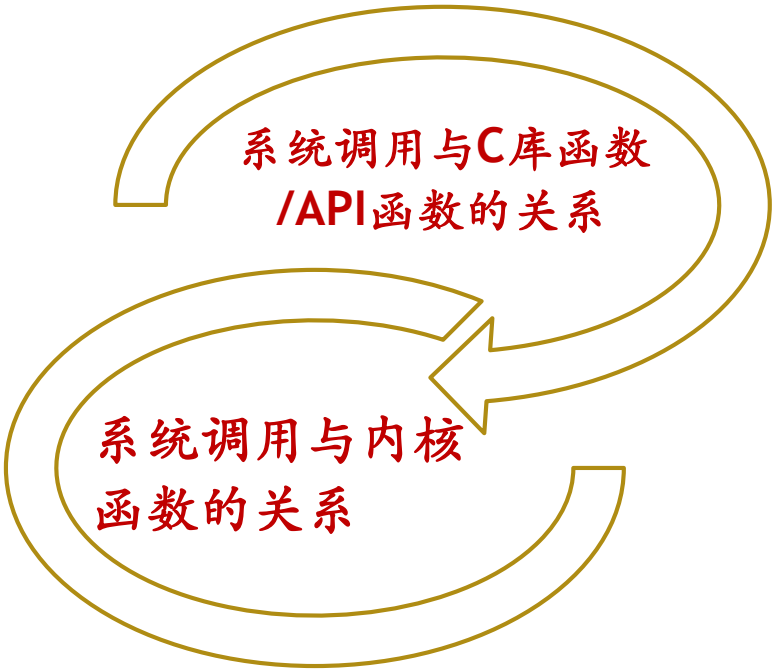
典型系统调用举例

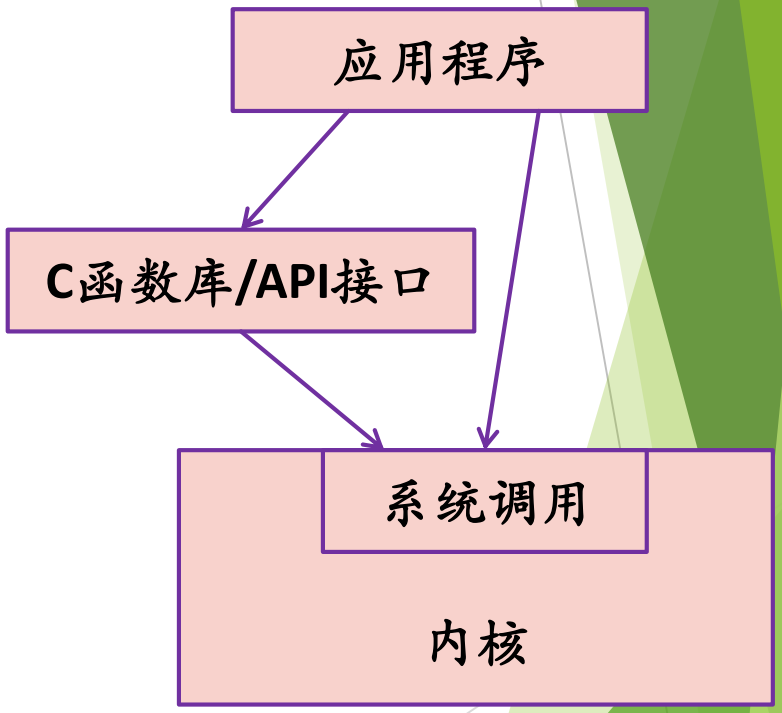
系统调用：用户在编程时可以调用的操作系统功能

- 系统调用是操作系统提供给编程人员的唯一接口
- 使CPU状态从用户态陷入内核态

每个操作系统都提供几百种系统调用（进程控制、进程通信、文件使用、目录操作、设备管理、信息维护等）

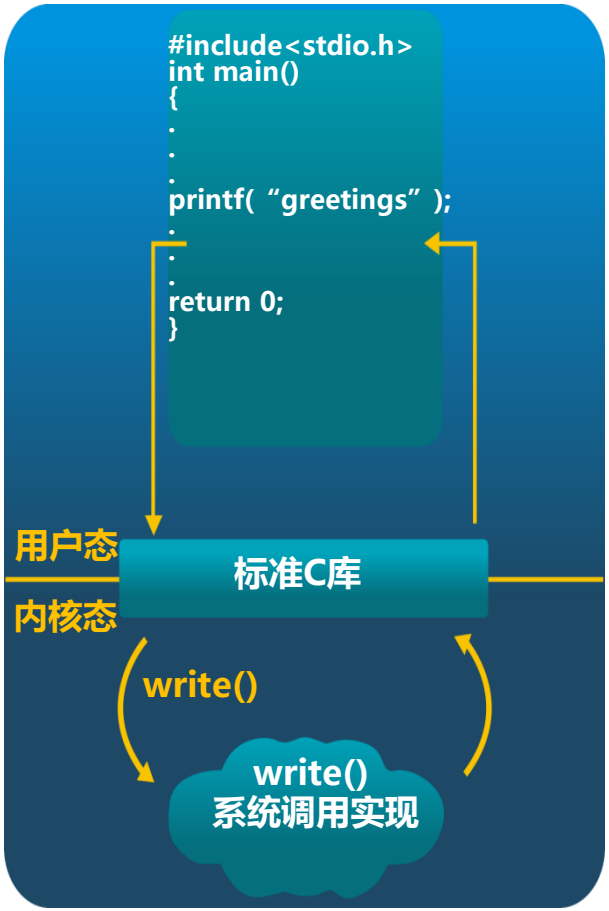
系统调用、库函数、API、内核函数

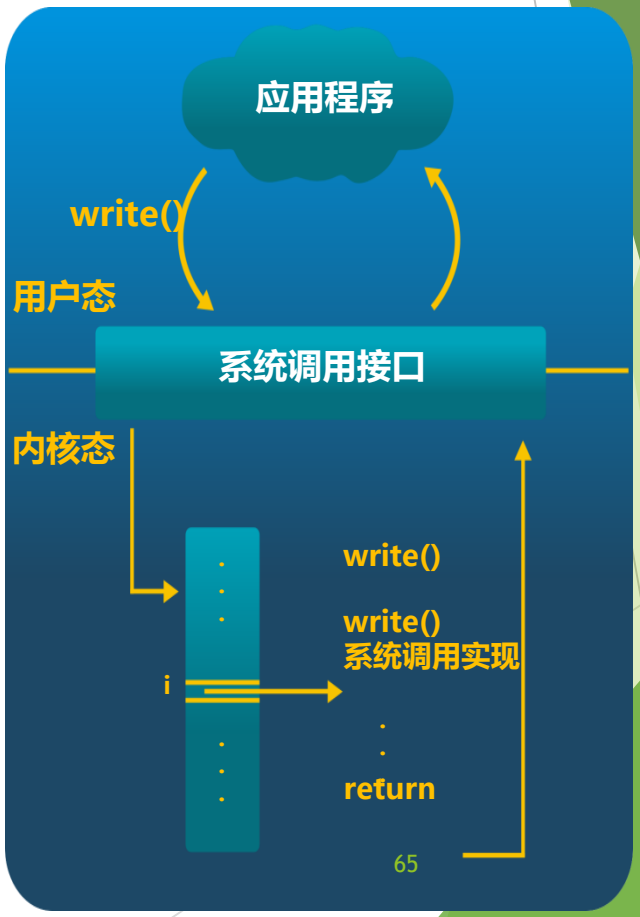




例子

- 应用程序调用printf() 时，会触发系统调用write()





Acknowledgement: 清华大学 向勇、陈渝

系统调用机制设计与执行过程

系统调用机制的设计

中断/异常机制

支持系统调用服务的实现 ①

选择一条特殊指令：陷入指令(亦称访管指令)

引发异常，完成用户态到内核态的切换 ②

系统调用号和参数

每个系统调用都事先给定一个编号(功能号)

③

系统调用表

存放系统调用服务例程的入口地址 ④

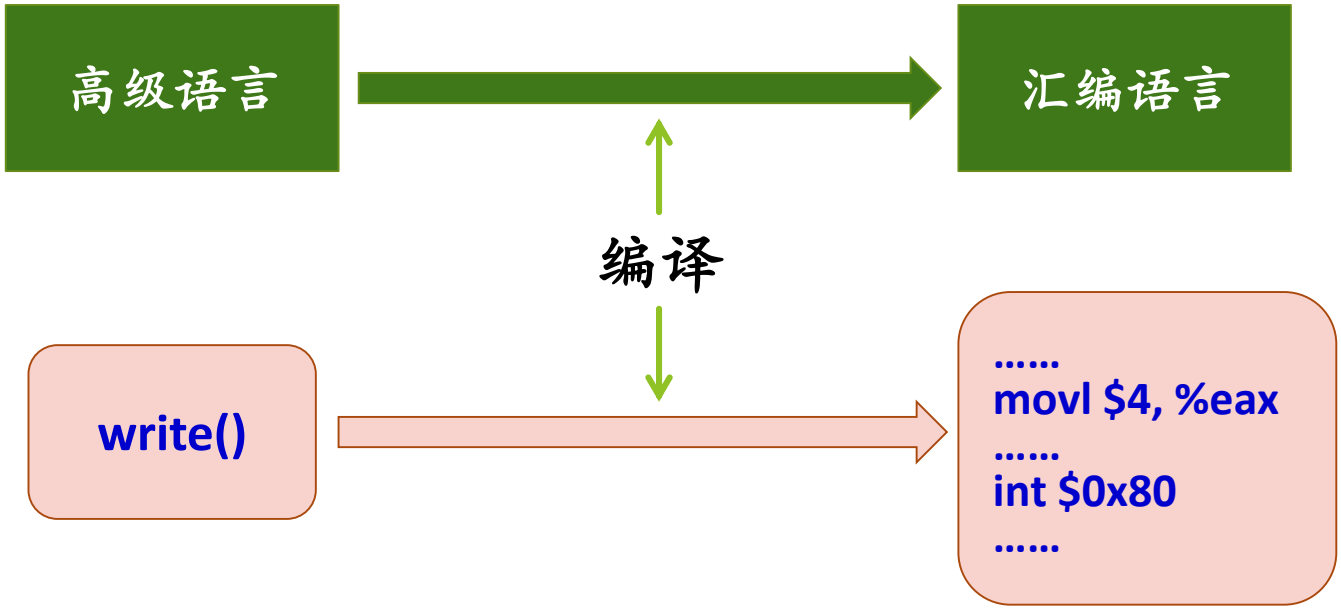
参数传递过程问题

- 怎样实现用户程序的参数传递给内核?

常用的3种实现方法:

- **由陷入指令自带参数**: 陷入指令的长度有限，且还要携带系统调用功能号，只能自带有限的参数
- **通过通用寄存器传递参数**: 这些寄存器是操作系统和用户程序都能访问的，但寄存器的个数会限制传递参数的数量
- **在内存中开辟专用堆栈区**来传递参数

系统调用举例(1/3)



系统调用举例(2/3)

```
#include <unistd.h>
int main(){
    char string[7] = {'H', 'e', 'l', 'l', 'o', '!', '\n'};
    write(1, string, 7);
    return 0;
}
```

输出结果：Hello!

系统调用举例(3/3)

```
1.  .section .data
2.  output:
3.      .ascii “Hello!\n”
4.  output_end:
5.      .equ len, output_end - output

6.  .section .text
7.  .globl _start
8.  _start:
9.      movl $4, %eax          #eax存放系统调用号
10.     movl $1, %ebx
11.     movl $output, %ecx
12.     movl $len, %edx
13.     int $0x80              #引发一次系统调用
14. end:
15.     movl $1, %eax          #1这个系统调用的作用?
16.     movl $0, %ebx
17.     int $0x80
```

汇编语言视角

系统调用的执行过程

当CPU执行到特殊的陷入指令时：

- **中断/异常机制**：硬件保护现场；通过查中断向量表把控制权转给系统调用总入口程序
- **系统调用总入口程序**：保存现场；将参数保存在内核堆栈里；通过查系统调用表把控制权转给相应的系统调用处理例程或内核函数
- **执行系统调用例程**
- **恢复现场，返回用户程序**

基于x86处理器

例子：Linux系统调用实现

Linux的系统调用实现
——基于x86体系结构

- 陷入指令选择 int 128

  int $0x80

- 门描述符
  - 系统初始化时：对IDT表中的第128号门初始化
  - 门描述符的2、3两个字节：内核代码段选择符（_KERNEL_CS）

    0、1、6、7四个字节：偏移量（system_call()可执行代码的第一条指令）
  - 门类型：**15，陷阱门，为什么？**
  - DPL：**3，与用户级别相同，允许用户进程使用该门描述符**
  - Include/linux/syscalls.h
  - Include/asm-i386/unistd.h

sched_init()中
set_system_gate(0x80, &system_call)

系统调用号示例(include/asm-i386/unistd.h)

```
# define __NR_exit          1
# define __NR_fork          2
# define __NR_read          3
# define __NR_write         4
# define __NR_open          5
# define __NR_close         6
# define __NR_waitpid       7
# define __NR_creat         8
# define __NR_link          9
# define __NR_unlink        10
# define __NR_execve        11
# define __NR_chdir         12
# define __NR_time          13
...
```

系统执行int $0x80指令（1/2）

- 由于特权级的改变，要切换栈

  用户栈→内核栈

  CPU要从任务状态段TSS中装入新的栈指针（SS：ESP），指向内核栈

- 用户栈的信息（SS：ESP）、EFLAGS、用户态CS、EIP 寄存器的内容压栈（返回用）
- 将EFLAGS压栈后，复位TF，IF位保持不变
- 用128在IDT中找到该门描述符，从中找出段选择符装入代码段寄存器CS
- 代码段描述符中的基地址＋陷阱门描述符中的偏移量 → 定位 system_call()的入口地址

系统执行int $0x80指令（2/2）

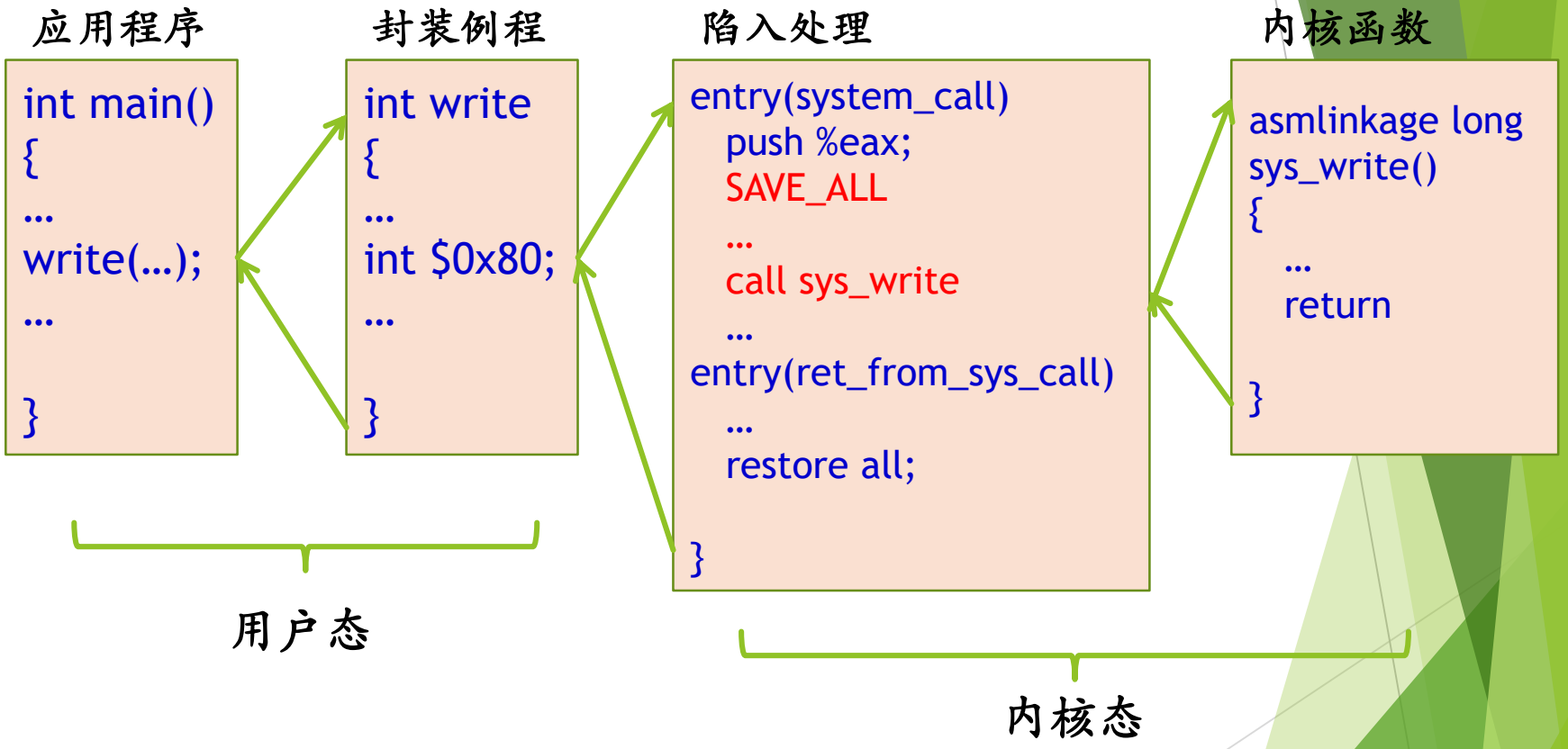
特权级检查

- 规则：代码只能访问相同或较低特权级的数据

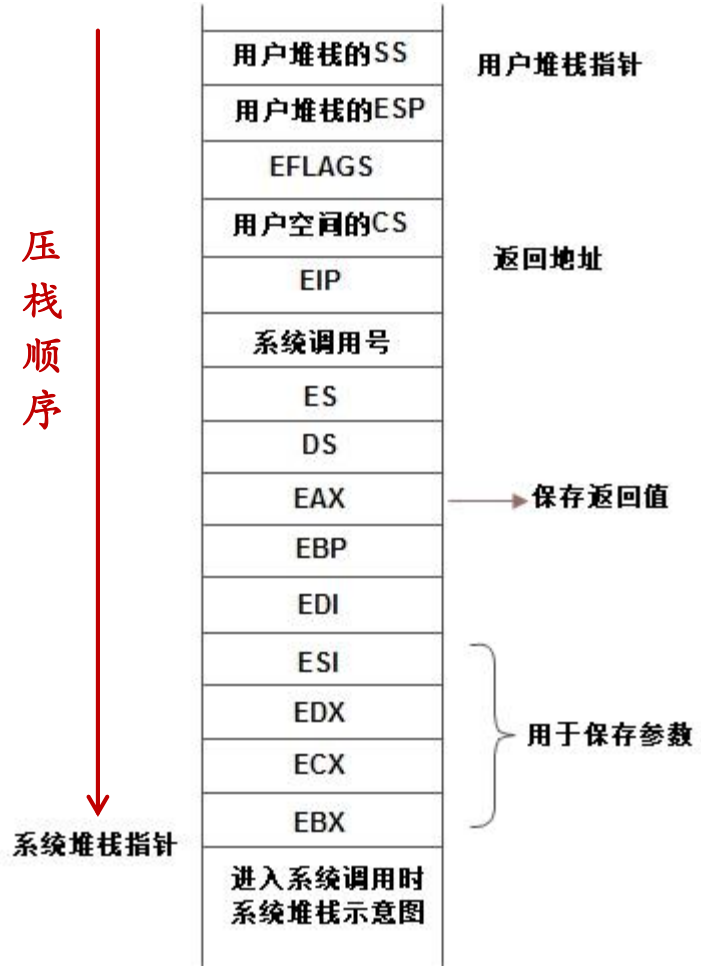
系统调用号、参数

- EAX，EBX、ECX、EDX、ESI、EDI

Linux系统调用执行流程



sys_write()执行完后，经过ret_from_sys_call()例程返回用户程序




```
#define SAVE_ALL \
        cld; \
        pushl %es; \
        pushl %ds; \
        pushl %eax; \
        pushl %ebp; \
        pushl %edi; \
        pushl %esi; \
        pushl %edx; \
        pushl %ecx; \
        pushl %ebx; \
        movl $(__KERNEL_DS), %edx; \
        movl %edx, %ds; \
        movl %edx, %es;
```

系统调用号与处理函数的映射

sys_call_table 描述了系统调用号与内核处理函数之间的对应，位于entry.S文件中：

```
… …
.data
ENTRY(sys_call_table)
.long SYMBOL_NAME(sys_exit)
.long SYMBOL_NAME(sys_fork)
.long SYMBOL_NAME(sys_read)
.long SYMBOL_NAME(sys_write)

… …
```

Linux 2.6

示例：系统调用的服务例程

用户态下调用C库的库函数，比如func()：

- func()先做好参数传递工作，然后使用int 0x80指令产生一次异常
- CPU通过0x80号在IDT中找到对应的服务例程system_call()，并调用之
- system_call()根据系统调用号索引系统调用表，找到系统调用程序入口，比如sys_func()
- sys_func()执行完后，经过ret_from_sys_call()例程返回用户程序

示例：系统调用的参数传递

- 系统调用使用寄存器传递参数，要传递的参数包括：
  - 系统调用号
  - 系统调用所需的参数
- 用于传递参数的寄存器有：
  - eax用于保存系统调用号和系统调用返回值
  - 系统调用参数保存在ebx,ecx,edx,esi和edi中，参数个数不超过6个
- 进入内核态后，system_call再将这些参数保存在内核堆栈中

示例：系统调用的参数传递

- 假如C库中封装的系统调用号为3的函数原型如下：

  int sys_func(int para1, int para2)

C编译器产生的汇编伪码如下：

```
...
movl 0x8(%esp),%ecx   #将用户态堆栈中的para2放入ecx
movl 0x4(%esp),%ebx   #将用户态堆栈中的para1放入ebx
movl $0x3,%eax        #系统调用号保存在eax中
int   $0x80           #引发系统调用
...
movl %eax,errno       #将结果存入全局变量errno中
movl $-1,%eax         #eax置为-1，表示出错
```

示例： system_call片段

```
system_call:
    pushl %eax          # 将系统调用号压栈
    SAVE_ALL
        ...
    cmpl $(NR_syscalls), %eax
                        # 检查是否是合法的系统调用号
    jb nobadsys
    movl $(-ENOSYS), 24(%esp)
                        # 堆栈中的eax设置为-ENOSYS，作为返回值
    jmp ret_from_sys_call
```

示例： system_call片段（续）

- nobadsys:

```
        …
    call    *SYMBOL_NAME(sys_call_table)(,%eax,4)
            # 调用系统调用表中调用号为eax的系统调用例程
    movl %eax,EAX(%esp)
            # 将返回值存入堆栈中eax中
    jmp ret_from_sys_call
```

示例：

```
ret_from_sys_call:
    cli                         # 关中断
    cmpl $0,need_resched(%ebx)
    jne reschedule
                    # 如果进程描述符中的need_resched位
                    # 不为0，则重新调度
    cmpl $0,sigpending(%ebx)
    jne signal_return
                    # 若有未处理完的信号，则处理

restore_all:
    RESTORE_ALL         # 堆栈弹栈，返回用户态
```

sysenter/sysexit系统调用机制

- X86在PentiumII300之后提供了sysenter/sysexit指令
- 为什么？
- 与 int 0x80/iret的不同？

Solaris的系统调用实现
——基于x86体系结构

三个部分：

- libc中的函数

  用户进程调用libc中的函数以执行系统调用

  因为实现系统调用的内核函数对参数顺序的要求可能和系统调用API不同，所以要在libc中进行必要处理后通过陷入指令进入内核

- 陷入机制

  在Solaris系统中，陷入（trap）机制完成从用户态到内核态的切换。大部分系统调用入口和建立工作依赖于硬件体系结构，例如SPARC和x86

- 功能实现

  主要完成系统调用功能的代码（实际的系统调用），由C语言实现

Solaris的系统调用实现细节（1/3）

- 两种执行系统调用方式

（1）一种是i386芯片提供的int/iret指令

（2）另一种是sysenter/sysexit指令

- 在第一种方式中，用户程序可以执行lcall 0x27或者int 0x91指令运行系统调用

两条指令的不同在于lcall指令是通过调用门进入内核态，而int指令则是通过中断门进入

为统一这两种指令，不论用户调用什么指令进入内核态，Solaris在处理过程中都重新将EFLAGS的值压栈。对中断门，Solaris用当前EFLAGS值覆盖中断门自动保留的值。由于进入中断门以后，关闭EFLAGS中的IF位，所以Solaris会将IF位置为开，然后再压栈，解决了系统调用结束后导致用户进程中中断响应被关闭的问题

Solaris的系统调用实现细节（2/3）

- 通过调用门从用户态切换至内核态时，调用门会自动从用户栈拷贝一个参数到内核栈，而中断门不会这么做，为统一这两种指令，Solaris也不使用这个硬件自动传过来的参数，这个参数在内核栈所占用的空间是为了存放前面提到过的EFLAGS寄存器的值。在iret返回时，该值正好可以被用来恢复EFLAGS寄存器

- 通常来说，lcall指令通过调用门时，处理器会自动将SS、ESP、CS、EIP四个参数压入内核栈，同时在ESP和CS之间保留一个32位的空间，以便将来进入内核态后存储EFLAGS的值。进入内核态后，系统会为r_trapno和r_err两个参数在内核栈顶保留空间，然后依次压入EAX, ECX, EDX, EBX, ESP, EBP, ESI, EDI, DS, ES, FS和GS寄存器的值，在内核栈的顶部形成一个struct regs结构。最后将EFLAGS寄存器当前的值写入之前为它特意保留的空间中

Solaris的系统调用实现细节（3/3）

- sysenter handler主要用于在调用sys_call()函数之前在内核栈顶部构造出和lcall指令效果相同的struct regs结构。这样也保证了{lcall,int}/iret和sysenter/sysexit两种方式的统一，为后面内核函数对栈的正常使用提供保证

- 执行完以上的步骤，系统就变为内核态。接下来就调用sys_call()函数，根据系统调用号找到相应的子程序执行

系统调用小结（1）

- 系统调用：用户在程序中调用操作系统提供的一些子功能

- 一种特殊的过程调用，由特殊的机器指令实现（每种机器的指令集都支持——访管指令）
- 系统调用是操作系统提供给编程人员的唯一接口
- CPU状态从目态转入管态
- 利用系统调用，可以动态请求和释放系统资源
  完成与硬件相关的工作以及控制程序的执行等

- 每个操作系统都提供几百种系统调用（POSIX标准）

系统调用小结（2）

- 系统调用与C函数调用的区别?
- 完成系统调用机制的运行需要什么条件（准备工作）？

  静态 和 动态

  封装内核函数 → 库函数(API)；访管指令与陷入机制；编译器；操作系统(初始化、系统调用编号及参数；系统调用表)

  陷入内核，总入口程序，保存现场（压栈），查表分派，执行返回

中断发生后OS低层工作步骤

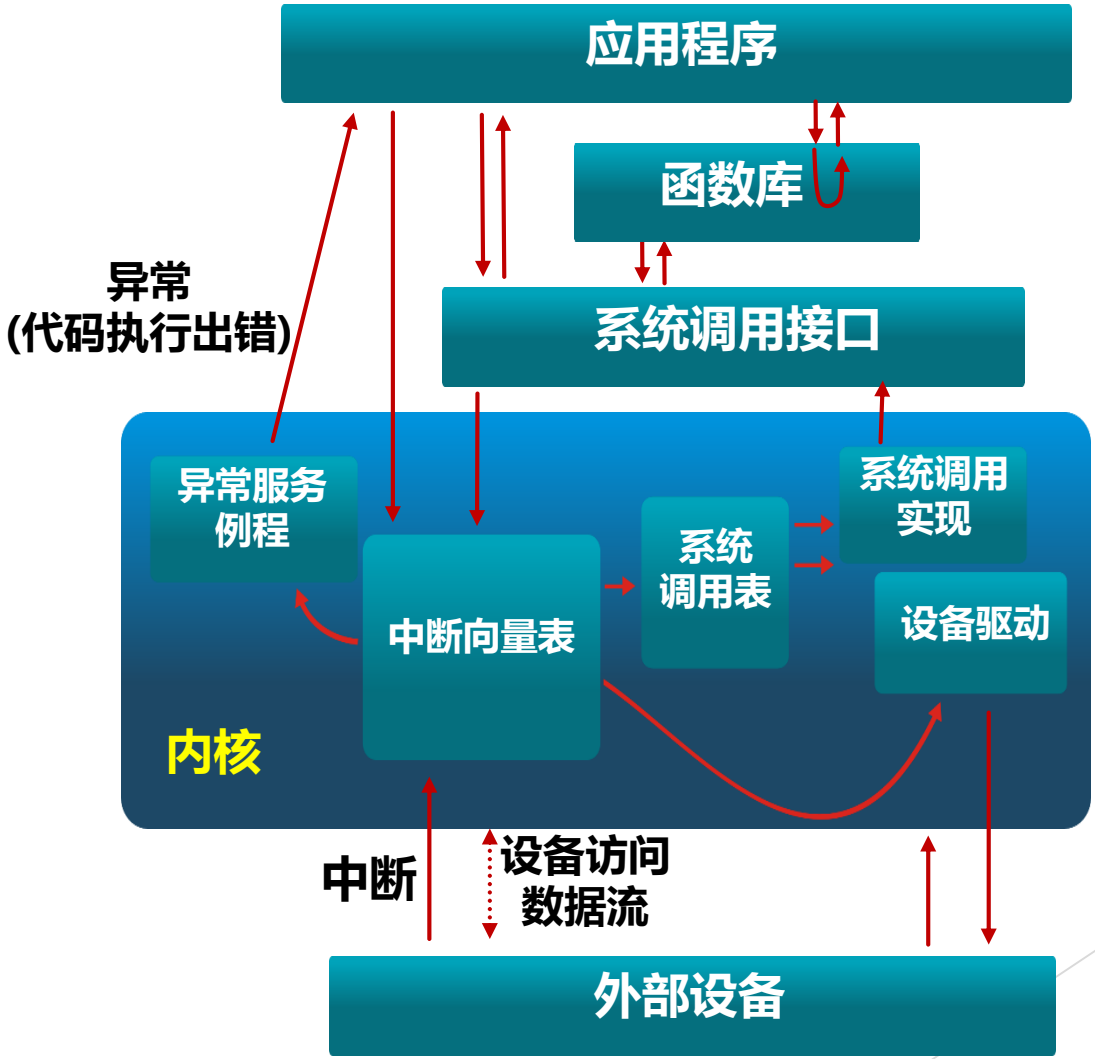
1. 硬件压栈：程序计数器等
2. 硬件从中断向量装入新的程序计数器等
3. 汇编语言过程保存寄存器值
4. 汇编语言过程设置新的堆栈
5. C语言中断服务程序运行（例：读并缓冲输入）
6. 进程调度程序决定下一个将运行的进程
7. C语言过程返回至汇编代码
8. 汇编语言过程开始运行新的当前进程

TANEBAUM 教材第 52 页的图 2-5

操作系统设计者考虑的角度……

机制与策略分离原则

内核的进入与退出



Acknowledgement: 清华大学 向勇、陈渝

作业2

1、了解Linux的中断处理流程，解释为什么引入上半部和下半部处理?

2、以缺页异常为例，调研ARM和x86的中断异常机制，简述并比较两者的处理流程。

3、总结RISC-V体系结构对中断的支持（300-400字）

4、总结中断、异常和系统调用（400-500字）。

5、系统调用与函数/过程调用的区别是什么?

提交时间：2020年10月18日晚23:30

XV6源代码阅读要求

- 阅读XV6源代码之中断/异常部分，撰写阅读报告。

提交时间：2020年10月18日晚23:30

Thanks

The End