南京大学
NANJING UNIVERSITY

Lab2 系统调用

助教：郑贺

实验简介

- 实验内容
  - 在保护模式的基础上进一步引入了内核与特权代码
  - 区分了用户态与内核态
  - 实现系统调用
- 大家的任务
  - 程序加载：用 bootloader 加载 kernel，kernel 加载用户程序
  - 完善中断机制：在 kernel 部分完善中断机制，为用户系统调用提供服务
  - 系统调用：在用户程序实现系统调用的实例测试

知识点概览

- 系统调用
  - 加载程序
  - 中断机制
    - 特权级代码保护
    - 保护机制
    - 中断机制的作用
    - 中断的分类
    - 实现结构
    - 中断处理流程
  - 系统调用
    - 系统调用的作用
    - 实现结构

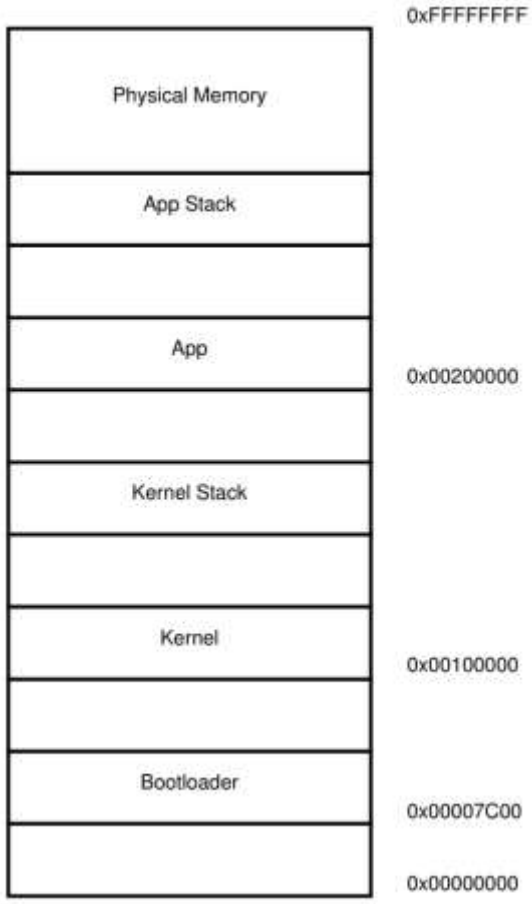
加载程序

- 磁盘分布
  - 0 号为 bootloader，1-200 号为内核部分，201 号以后则是用户程序部分

Sector 0	Sector 1	Sector 2		.....		Sector 201	.....	
bootloader	kernel					App		

加载程序

- 内存分布
  - 编译时内核 .text 段的起始地址设为 0x100000
  - GDT 中内核数据段的基地址设置为 0x0
  - 内核加载至物理内存 0x100000 开始的位置
  - 编译时用户程序 .text 段的起始地址设为 0x200000
  - GDT 中用户程序数据段的基地址设置为 0x0
  - 用户程序加载至物理内存 0x200000 开始的位置



加载程序

- 加载方式
  - 由 bootloader 加载 kernel：与 lab1 中从硬盘加载到用户程序方式一致
  - 由 kernel 加载用户程序：可以模仿 bootloader 的加载方式

IA-32中断机制-中断功能

- 处理硬件外设 I/O
  - CPU 的处理速度大于大部分外部设备，因此为了提高 CPU 效率，需要对 I/O 操作进行额外处理。常见的处理方式有：轮询、中断、DMA 等。其中中断机制是一种比较灵活高效的方式
- 保护特权代码
  - x86 平台的 CPU 有 0、1、2、3 四个特权级，数字越小等级越高。ring0 是最高特权级，可以执行所有命令；ring3 是最低特权级，只能执行简单的算术逻辑指令。
  - 现代操作系统往往只用到这两个特权等级，其中 ring0 就是我们常说的内核态，ring3 则是指用户态

IA-32中断机制

- 中断的作用：
  - 中断会改变 CPU 执行指令的顺序，由当前指令跳转执行相应中断的处理程序，由用户态切换到内核态，使得代码的执行环境区分开来，以此保护特级代码
- 权限机制：x86 平台使用 CPL、DPL、RPL 来对代码、数据的访问进行特权级检测
  - CPL（current privilege level）表示当前指令的特权级，它由 CS 寄存器的低两位表示
  - DPL（descriptor privilege level）表示访问该内存段需要的最低特权级，它由描述符中的 DPL 字段表示
  - RPL（requested privilege level）用于对 CPL 的特权级进行补充，它由 DS、ES、FS、GS、SS 寄存器的低两位表示
  - 一般情况下，同时满足 CPL ≤ DPL，RPL ≤ DPL 才能实现对内存段的访存

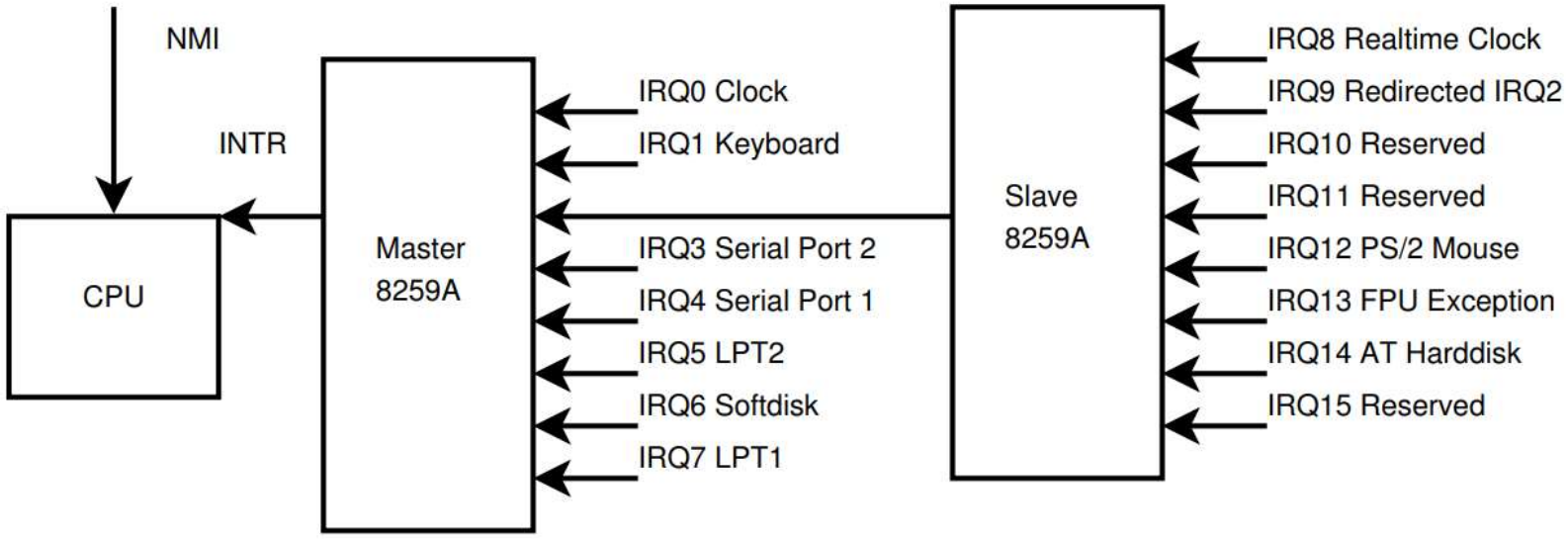
IA-32中断机制-中断分类

- 中断（广义）的分类：
  - Interrupt：外部硬件产生的中断（时钟、硬盘、网卡等）
  - Exception：CPU执行指令过程中产生的异常
  - Software Interrupt：由 Int 等指令产生的软中断

IA-32中断机制-Interrupt

- 由硬件随机产生的，在程序执行的任何时候都可能出现（异步）
  - Maskable Interrupt（可屏蔽中断）
    - 可以被 CPU 选择屏蔽掉请求的中断类型（可以被屏蔽也可以不被屏蔽）
    - I/O 设备发出的所有中断请求（IRQ）都可以产生可屏蔽中断
    - IRQ 由 8259A 这个可编程中断控制器（PIC）统一处理，并转化为 8-Bits 中断向量由 INTR 引脚输入到 CPU（通过设置 8259A 芯片，可以对每个 IRQ 分别进行屏蔽）
  - Non-maskable Interrupt（非屏蔽中断）
    - 不可以被 CPU 屏蔽的中断类型
    - 只有少数几个特定的危机事件才会产生非屏蔽中断，比如硬件故障、掉电等
    - 由 NMI 引脚输入 CPU
- CPU屏蔽机制
  - Eflags 中的 IF 标志：0-关中断，1-开中断
  - 关中断时，CPU不响应中断控制器发布的任何中断请求
  - 内核中使用 cli 和 sti 指令来清除（关）和设置（开）该标志

IA-32中断机制-Interrupt



IA-32中断机制-Exception

- 在（特殊的或出错的）指令执行时由 CPU 控制单元产生（同步）
  - Fault：故障
    - EIP 取值为引起异常的指令的地址
    - 通常可以纠正，处理完异常时，该指令被重新执行。例如缺页异常（page fault）等
  - Trap：陷阱
    - 大多数情况下，EIP 取值为引起异常指令的下一条指令地址
    - 例如单步调试等
  - Aborts：终止
    - EIP 取值无效，严重错误，需要强制终止受影响的进程
    - 例如段越界等
- 对于部分异常，除了 EFLAGS，CS，EIP 这些寄存器以外，硬件会在堆栈内再压入一个 Error Code

IA-32中断机制-Software Interrupt

- 也可叫做编程异常，由编程者发出的特定请求产生，通常由 int 类指令触发（同步）
  - EIP 取值为引起异常指令的下一条指令地址
  - 例如系统调用（系统调用 **int 0x80** 是一个软中断，是应用程序与 Linux 内核交互的接口）

IA-32中断机制-实现结构

- 中断向量（Interrupt Number）
- 中断描述符表（Interrupt Descriptor Table）
- 任务状态段（Task Statement Segment）
- Iret 指令

IA-32中断机制-中断向量

- 由以 0～255 之间的数（8 位）来标识，Intel 称其为中断向量
  - Interrupt
    - 可屏蔽中断的向量可以通过对中断控制器的编程来改变（0x20 至 0x2F 一般映射至 16 个可屏蔽中断）
    - 非屏蔽中断的向量是固定的（0x02）
  - Exception
    - Exception 的中断向量一般也是固定的（除 0x02 外）
  - Software Interrupt
    - 0x30~0xFF 号向量供用户定义给软中断

IA-32中断机制-中断描述符表

- 与 256 个中断向量对应，有 256 个表项，表项称为门描述符（Gate Descriptor）
  - 每个描述符占 8 个字节，记录了对应的向量相应的处理程序的入口地址
- 在开启外部硬件中断之前，内核需要对 IDT 完成初始化
  - CPU 的 IDTR 寄存器指向 IDT 表的物理基地址，使用 lidt 指令加载

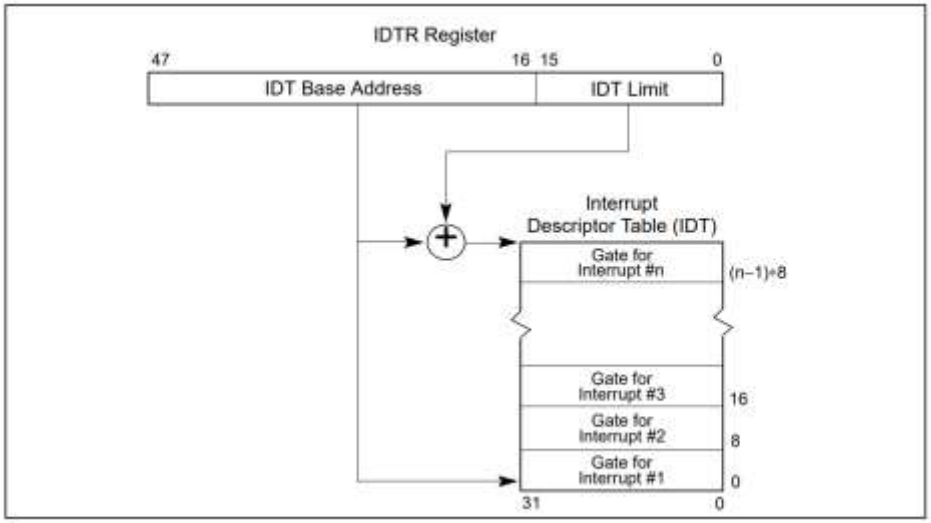
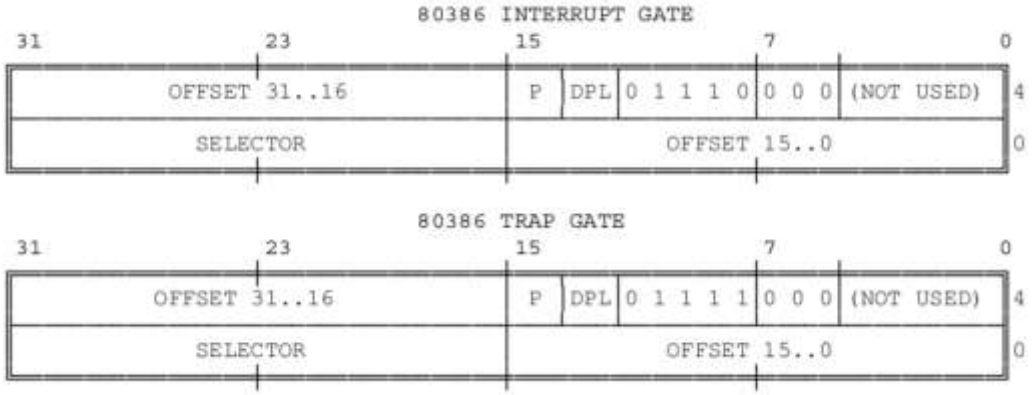


Figure 6-1. Relationship of the IDTR and IDT

IA-32中断机制-门描述符

- Trap Gate：当中断向量对应的门描述符为 Trap Gate 时，跳转执行处理程序前，EFLAGS 中的 IF 位不会置为 0（即系统不会进入关中断状态）
  - 如异常，软中断
- Interrupt Gate：当中断向量对应的门描述符为 Interrupt Gate 时，跳转执行处理程序前，EFLAGS 中的 IF 位会被置为 0（即系统会关中断）
  - 如IRQ
- Task Gate：Intel 为任务切换设计的，但现代操作系统一般不使用它



IA-32中断机制-任务状态段

- 由于需要将中断处理程序的栈段和栈指针保存起来，以便用户程序在运行过程中产生异常或者中断后使用，这就需要用到任务状态段（TSS）
- TSS 用来记录“I/O 权限位图”和 ring0，ring1，ring2 的 SS 与 ESP
  - 思考题：为什么 tss 不需要记录 ring3 的 SS 和 ESP？

I/O Map Base Address		T
	LDT Segment Selector	
	GS	
	FS	
	DS	
	SS	
	CS	
	ES	
EDI		
ESI		
EBP		
ESP		
EBX		
EDX		
ECX		
EAX		
EFLAGS		
EIP		
CR3		
	SS2	
ESP2		
	SS1	
ESP1		
	SS0	
ESP0		
	Previous Task Link	

Reserved

IA-32中断机制-特权级下的栈切换

- 程序在运行时产生中断
  - 若当前程序的优先级小于中断处理程序所在段的优先级，则根据 DPL 选择 TSS 中记录的相应 SS 与 ESP 进行堆栈切换，并将当前用户程序的 SS、ESP、EFLAGS、CS、EIP 压入切换后的堆栈中
  - 否则，无需切换堆栈，依次压入当前程序的 EFLAGS、CS、EIP 至当前程序堆栈中
  - 对于特定中断，还需要压入 Error Code（参考指导文件中的中断表）
- 程序如何知道内核态的栈在何处？使用 TSS 记录
  - 在开启外部中断前，内核需要对 TSS 完成初始化
  - TR 寄存器是 GDT 中对应 TSS 的描述符的选择子，使用 ltr 命令来设置 TR 寄存器

IA-32中断机制-iret指令

- 用于从中断处理程序中返回
  - 对于 iret 指令，硬件会以此从当前栈顶 pop 出 EIP、CS、EFLAGS，即返回执行产生中断时的程序
  - 若 pop 出的 CS 的 CPL 大于当前程序的 CPL，iret 指令还会继续 pop 出 ESP 以及 SS，即切换堆栈
- 为什么不使用 ret 而是专门的 iret
  - ret 只是把栈中内容(返回地址)弹出至 EIP，而 iret 要考虑的问题更多

IA-32中断机制-中断处理流程

- 确定与中断或异常关联的向量 i(0-255)
- 读取 IDTR 寄存器指向的 IDT 中的第 i 项门描述符
- 从 GDTR 寄存器获得 GDT 的基地址，并在 GDT 中查找， 以读取上述门描述符中的段选择子所标识的段描述符
- 判断中断请求是否合理，进行特权级比较
  - 中断处理程序的权限不能低于引起中断的程序：比较 CS 寄存器的 CPL 和对应 GDT 表项中的 DPL
  - 对于软中断，引起软中断的程序的权限不能低于中断要求的权限：额外还需要比较 CPL 与门描述符中的 DPL，若 CPL>DPL，则产生 #GP 异常
- 若发生了特权级变化，即要由用户态陷入内核态，必须进行堆栈切换
  - 读取 TR 寄存器，访问运行进程的 TSS
  - 选取 TSS 中记录的与新特权级相关的栈段和栈指针装载 SS 与 ESP 寄存器
  - 在切换后的堆栈中保存之前堆栈的 SS 与 ESP

接下页

IA-32中断机制-中断处理流程

- 在堆栈中保存 EFLAGS
  - 将旧 EFLAGS 压入栈，并根据门描述符设置 EFLAGS 的 IF 位
  - 若中断为 Fault，则在堆栈中保存引起中断的 CS 与 EIP
  - 否则，在堆栈中保存下条指令的 CS 与 EIP
- 若中断产生一个 Error Code，则将其保存在堆栈中
- 依据门描述符装载 CS 与 EIP，即执行中断处理程序
- 执行完处理程序后，使用 iret 指令返回

系统调用

- System call 的作用
  - 普通用户需要用到一些不在自己权限内的系统服务，在满足用户需求的同时又要对特权级代码进行保护，因此诞生了系统调用。
- 可以将所有系统调用使用 int $0x80 软中断实现，也可以为不同的系统调用分配不同的中断向量
- 每个系统调用至少需要一个参数，即系统调用号，用于确定通过中断陷入内核后，该用哪个函数进行处理
- 参数传递
  - 普通 C 语言的函数的参数传递时通过将参数从右向左依次压入堆栈来实现的，但是系统调用涉及到堆栈切换，不能直接使用堆栈进行参数传递
  - 可以使用 EAX、EBX 等等这些通用寄存器来从用户态向内核态传递参数
  - 代码框架中 kernel/irqHandle.c 中使用了 TrapFrame 这一数据结构，其中保存了内核堆栈中存储的 7 个寄存器的值，其中的通用寄存器的取值即是通过上述方法从用户态传递至内核态，并通过 pushal 指令压入内核堆栈
- 推荐阅读： Intel manual chapter6：https://cdrdv2.intel.com/v1/dl/getContent/671447

实验注意事项

- 报告提交到课程平台 cslabcms.nju.edu.cn
- 本次实验的工程量很大，请大家早点开始。
- 另外，推荐使用 git 进行版本控制，否则你很快会被各种 bug (无论是最初引入的还是修改时改出来的)搞得心力憔悴
- 最好每次编译前都先 make clean 一下，防止之前的结果的干扰
- 本次实验为期3周，请大家独立完成