Lab2 系统调用

徐圣斌

kingxu@smail.nju.edu.cn

2019-3-19

实验内容

- 内核:基于中断建立完整的系统调用机制
- 库:基于系统调用实现库函数 `scanf` 和 `printf`
- 用户:实现一个调用 `scanf` 和 `printf` 的测试程序

实验内容

- Bootloader 从实模式进入保护模式，加载内核至内存，并跳转执行
- 内核初始化 IDT(中断描述符表)，初始化 GDT(全局描述符表)，初始化 TSS(任务状态段)
- 内核加载用户程序至内存，对内核堆栈进行设置，通过 iret 切换至用户空间，执行用户程序
- 用户程序调用库函数 `scanf` 和 `printf` 陷入内核，由内核完成读取键盘输入，在视频映射的显存地址中写入内容完成字符串的打印等内容
- 测试用例在代码中

硬件外设的 I/O

- 内核的一个主要功能是处理硬件外设的 I/O
  - CPU 速度一般比硬件外设快很多
    - 多任务系统中，CPU 可以在外设进行准备时处理其他任务，在外设完成准备时处理 I/O
    - I/O 处理方式包括:轮询、中断、DMA 等
  - 基于中断机制可以解决轮询处理硬件外设 I/O 时效率低下的问题

特权级代码的保护

- 安全性问题
  - x86 平台 CPU 有 0、1、2、3 四个特权级，其中 level0 是最高特权级，可以执行所有指令
  - level3 是最低特权级，只能执行算数逻辑指令，很多特殊操作(例如 CPU 模式转换，I/O 操作指令)都不能在这个级别下进行
  - 现代操作系统往往只使用到 level0 和 level3 两个特权级，操作系统内核运行时，系统处于 level0(即 CS 寄存器的低两位为 `00b`)，而用戶程序运行时系统处于 level3(即 CS 寄存器的低两位为 `11b`)

特权级代码的保护

- 安全性问题
  - x86 平台使用 CPL、DPL、RPL 来对代码、数据的访存进行特权级检测
    - CPL(current privilege level)为 `CS` 寄存器的低两位，表示当前指令的特权级
    - DPL(discriptor privilege level)为描述符中的 DPL 字段，表示访存该内存段的最低特权级(有时表示访存该段的最高特权级，比如 Conforming-Code Segment)
    - RPL(requested privilege level)为 `DS` 、 `ES` 、 `FS` 、 `GS` 、 `SS` 寄存器的低两位，用于对 CPL 表示的特权级进行补充
    - 一般情况下，同时满足 CPL ≤ DPL，RPL ≤ DPL 才能实现对内存段的访存，否则产生 `#GP` 异常
  - 基于中断机制可以实现对特权级代码的保护

IA-32 中断机制

- 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
- 在跳转执行相应的中断处理程序前，IA-32 硬件会在堆栈中 对当前执行的程序状态进行保存(即将 `EFLAGS`， `CS`， `EIP` 等压栈)，以保证中断处理程序执行结束后能正确返回执行当前程序
- 在保护模式下，对于不同的中断类型，不同的特权等级下 (即当前程序的 CPL 与中断处理程序所在段的 DPL)，IA-32 保存的寄存器内容也不尽相同，保存寄存器内容的堆栈位置也不尽相同

IA-32 中断机制

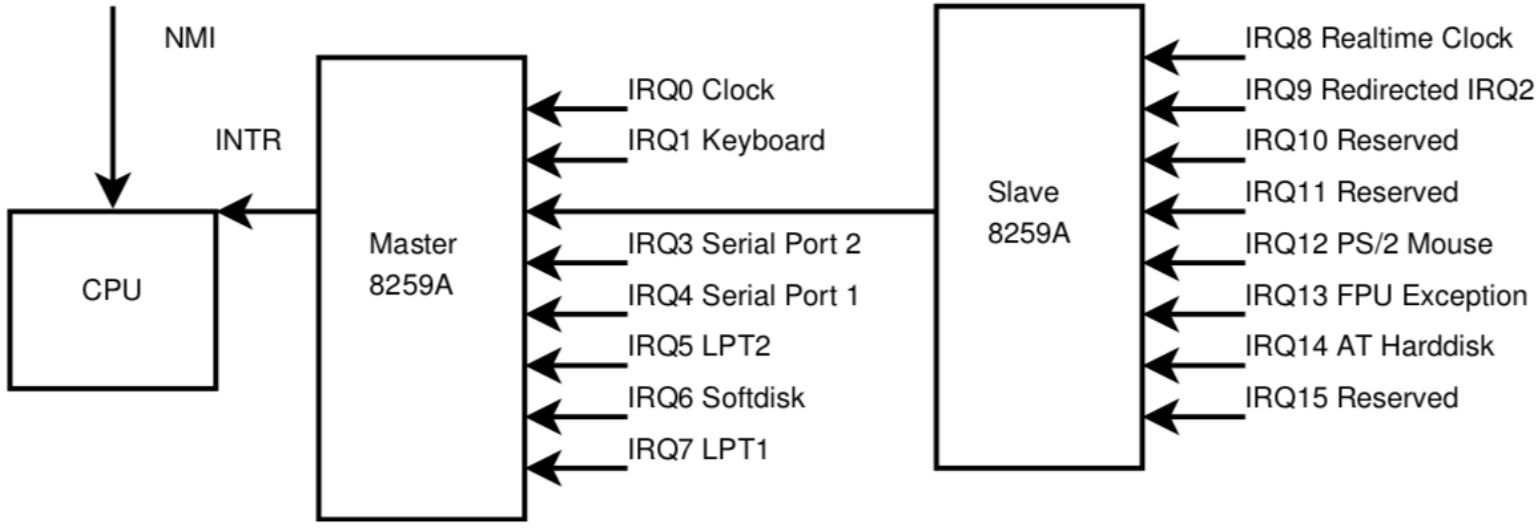
- 保护模式下广义的中断源包括 外部硬件产生的中断(Interrupt)
  - 例如时钟、磁盘、键盘等外部硬件
- CPU 执行指令过程中产生的异常(Exception)
  - 例如除法错(`#DE`)，页错误(`#PF`)，常规保护错误(`#GP`)
- 由 `int` 等指令产生的软中断(Software Interrupt)
  - 例如系统调用使用的 `int $0x80`

IA-32 中断机制

- 外部硬件产生的中断
  - 可屏蔽中断(Maskable Interrupt)
    - I/O 设备发出的所有中断请求信号(IRQ)都是可屏蔽的
    - I/O 设备发出的 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 CPU
    - 通过 `sti`，`cli` 指令设置 CPU 的 `EFLAGS` 寄存器中的 `IF` 位，可以控制对这些中断进行屏蔽与否
    - 通过设置 8259A 芯片，可以对每个 IRQ 分别进行屏蔽
  - 非屏蔽中断(Nonmaskable Interrupt)
    - 仅有几个特定的事件才能引起非屏蔽中断，例如硬件故障以及或是掉电，其由 NMI 引脚输入 CPU

IA-32 中断机制

- 8259A 可编程中断控制器



IA-32 中断机制

- CPU 执行指令过程中产生的异常
  - 根据硬件在堆栈中保存的 `EIP` 取值的不同，异常可进一步分为
    - Fault: `EIP` 取值为引起异常的指令的地址
    - Trap:大多数情况下， `EIP` 取值为引起异常的指令的下一条指令的地址
    - Abort: `EIP` 取值无效，严重错误，需要强制终止受影响的进程
  - 对于部分异常，除却 `EFLAGS` ， `CS` ， `EIP` 这些寄存器，硬件会在堆栈内再压入一个 `Error Code`

IA-32 中断机制

- 保护模式下的中断向量(Interrupt Number)
  - 每个中断(Exception，Interrupt，Software Interrupt)都由一个 8-Bits 的向量来标识，Intel 称其为中断向量
    - `0x00` 至 `0x1F` 号向量为 Intel 保留，其映射至非屏蔽中断(中断向量为 `0x02`)，以及其他保护模式下 CPU 执行指令过程中产生的异常
    - `0x20` 至 `0xFF` 号向量供用户定义，其中 `0x20` 至 `0x2F` 一般映射至 16 个可屏蔽中断(可屏蔽中断对应的中断向量 可以通过对 8059A 的设置来改变)

IA-32 中断机制

- 中断描述符表(IDT)
  - 与 256 个中断向量对应，IDT 中存有 256 个表项，表项称为门描述符(Gate Descriptor)，每个描述符占 8 个字节，在开启外部硬件中断前，内核需对 IDT 完成初始化，并使用 `lidt` 指令设置 `IDTR` 寄存器
    - Trap Gate:当中断向量对应的门描述符为 Trap Gate，跳转执行该中断对应的处理程序时，`EFLAGS` 中的 `IF` 位不会置为 0
    - Interrupt Gate:当中断向量对应的门描述符为 Interrupt Gate，跳转执行该中断对应的处理程序时，`EFLAGS` 中的 `IF` 位会被置为 0
    - Task Gate:Intel 设计用于任务切换，现代操作系统中一般不使用

IA-32 中断机制

- 任务状态段(TSS)
  - TSS 原由 Intel 设计用于实现基于硬件的任务切换，现代操作系统中一般不使用
  - TSS 中记录着 `ring0`，`ring1`，`ring2` 环的 `SS` 与 `ESP`
    程序在运行时产生中断后
    - 若当前程序的 CPL 大于中断处理程序所在段的 DPL，则硬件依据 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` 至当前程序的堆栈
    - 对于特定中断，还需压入 `Error Code`

IA-32 中断机制

- 任务状态段(TSS)
  - 内核开启外部硬件中断进入用户空间前，需要对 TSS 进行初始化，并使用 `ltr` 指令设置 `TR` 寄存器，即 GDT 中对应 TSS 的描述符的选择子
- 保护模式下的中断的硬件处理流程
  - 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
  - 若为软中断，需比较 CPL 与门描述符中的 DPL，若 CPL>DPL，则产生 `#GP` 异常

IA-32 中断机制

- 保护模式下的中断的硬件处理流程
  - 比较 `CPL` 与段描述符中的 `DPL`，若 `CPL>DPL`，则发生特权级变化
    - 读取 `TR` 寄存器，访问 TSS
    - 选取 TSS 中记录的与 `DPL` 一致的 `SS` 与 `ESP` 切换堆栈
    - 在切换后的堆栈中保存之前堆栈的 `SS` 与 `ESP`
  - 在堆栈中保存 `EFLAGS`
    - 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

IA-32 中断机制

- 使用 `iret` 指令从高特权级返回低特权级
  - 对于 `iret` 指令，硬件会依次从当前栈顶 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` 小于当前程序的 `CPL`，`iret` 还会继续 pop 出 `ESP` 以及 `SS`，即切换堆栈
- **`ring0 与 ring3`**
  - i386 基于 IDT 与 TSS，实现了基于中断的程序运行流程转换，同时实现了对特权级代码的保护，即所谓的内核态(`ring0`)与用戶态(`ring3`)

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，用以确定通过中断陷入内核后，该用哪个函数进行处理
- `Lab2` 不要求一定使用何种方式完成库函数的实现

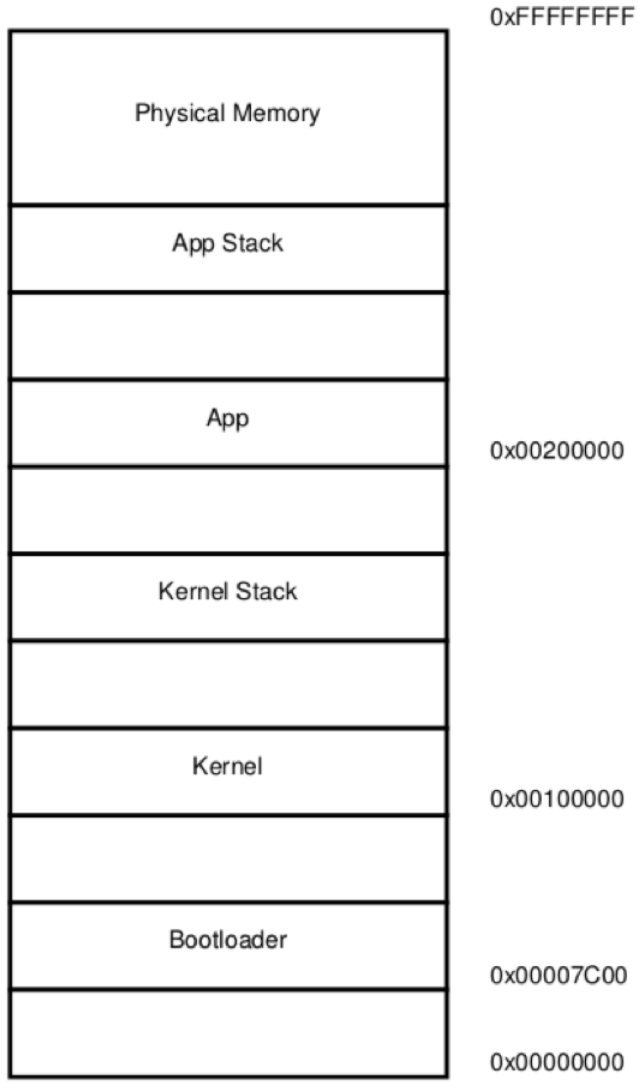
系统调用

- 普通 C 语言的函数的参数传递是通过将参数从右向左依次压入堆栈来实现
- 系统调用涉及到用户堆栈至内核堆栈的切换，不能像普通函数一样直接使用堆栈传递参数
- 可以使用 `EAX`，`EBX` 等等这些通用寄存器从用户态向内核态传递参数
    - 代码框架 `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` 指令压入内核堆栈的

加载内核与用户程序

- 与 `lab1` 不同， `lab2` 提供的 `Makefile` 将内核、用户程序分别编译为 `kMain.elf` 、 `uMain.elf` 这两个 ELF 文件，最终生成的虚拟镜像 `os.img` 包含内容如下
  - Binary 文件 `Bootloader.bin` 扩展至 512 字节，设置最后两个字节为 `0x55` 、 `0xaa` ，放在 0 号扇区
  - ELF 文件 `kMain.elf` 扩展至 200 个扇区，放在 1-200 号扇区
  - ELF 文件 `uMain.elf` 放在 201 号开始的扇区
- Bootloader 加载内核并跳转执行时，首先需要读取其 ELF 文件头以及程序头，设置正确的内存加载地址以及跳转地址
- 内核加载用户程序陷入用户空间并执行时，同样需要首先读取其 ELF 文件头以及程序头，设置正确的加载地址，并正确设置堆栈中的 `CS` 与 `EIP` ，保证通过执行 `iret` 能陷 入 `ring3` 的用户空间，执行用户程序

加载内核与用户程序



- 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

实验攻略

- 库函数
  - lab2 要求完成 `lib/syscall.c` 中的 `scanf` 和 `printf`
- 中断处理
  - lab2 要求完成 `kernel/kernel/irqHandle.c` 中的 `syscallScan`

作业提交

- 本次作业需提交实验2的相关源码与报告
- **截止时间:2019-4-8 23:59:55**
- 如果你无法完成实验, 可以选择不提交, 作为学术诚信的奖励, 你将会获得10%的分数; 但若发现抄袭现象, 抄袭双方(或团体)在本次实验中得0分，后续可能有其他惩罚
- **本实验的最终解释权由助教所有**