实验一：系统软件启动过程

1.实验目的：

操作系统是一个软件，也需要通过某种机制加载并运行它。在这里我们将通过另外一个更加简单的软件-bootloader 来完成这些工作。为此，我们需要完成一个能够切换到 x86 的保护模式并显示字符的 bootloader，为启动操作系统 ucore 做准备。lab1 提供了一个非常小的 bootloader 和 ucore OS，整个 bootloader 执行代码小于 512 个字节，这样才能放到硬盘的主引导扇区中。通过分析和实现这个 bootloader 和 ucore OS，读者可以了解到：

- 基于分段机制的存储管理
- 设备管理的基本概念
- PC 启动 bootloader 的过程
- bootloader 的文件组成
- 编译运行 bootloader 的过程
- 调试 bootloader 的方法
- ucore OS 的启动过程
- 在汇编级了解栈的结构和处理过程
- 中断处理机制
- 通过串口/并口/CGA 输出字符的方法

2.实验内容：

lab1 中包含一个 bootloader 和一个 OS。这个 bootloader 可以切换到 X86 保护模式，能够读磁盘并加载 ELF 执行文件格式，并显示字符。而这 lab1 中的 OS 只是一个可以处理时钟中断和显示字符的幼儿园级别 OS。

本次实验需要提交实验报告，回答下述练习中所所提出的的问题，请将实验报告以纯文本的形式提交到 lab1 的代码目录下并上传到 git 服务器。

2.1 练习

练习 1：理解通过 make 生成执行文件的过程。（要求在报告中写出对下述问题的回答）

在此练习中，大家需要通过阅读代码来了解：

1. 操作系统镜像文件 ucore.img 是如何一步一步生成的？(需要比较详细地解释 Makefile 中每一条相关命令和命令参数的含义，以及说明命令导致的结果)
2. 一个被系统认为是符合规范的硬盘主引导扇区的特征是什么？

补充材料：

如何调试 Makefile

当执行 make 时，一般只会显示输出，不会显示 make 到底执行了哪些命令。

如想了解 make 执行了哪些命令，可以执行：

$ make "V="

要获取更多有关 make 的信息，可上网查询，或者执行

$ man make

练习 2：使用 qemu 执行并调试 lab1 中的软件。（要求在报告中简要写出练习过程）

为了熟悉使用 qemu 和 gdb 进行的调试工作，我们进行如下的小练习：

1. 从 CPU 加电后执行的第一条指令开始，单步跟踪 BIOS 的执行。

2. 在初始化位置 0x7c00 设置实地址断点,测试断点正常。
3. 从 0x7c00 开始跟踪代码运行,将单步跟踪反汇编得到的代码与 bootasm.S 和 bootblock.asm 进行比较。
4. 自己找一个 bootloader 或内核中的代码位置，设置断点并进行测试。

提示：参考附录“启动后第一条执行的指令”

补充材料：

我们主要通过硬件模拟器 qemu 来进行各种实验。在实验的过程中我们可能会遇上各种各样的问题，调试是必要的。qemu 支持使用 gdb 进行的强大而方便的调试。所以用好 qemu 和 gdb 是完成各种实验的基本要素。

默认的 gdb 需要进行一些额外的配置才进行 qemu 的调试任务。qemu 和 gdb 之间使用网络端口 1234 进行通讯。在打开 qemu 进行模拟之后，执行 gdb 并输入

target remote localhost:1234

即可连接 qemu，此时 qemu 会进入停止状态，等待 gdb 的命令。

另外，我们可能需要 qemu 在一开始便进入等待模式，则我们不再使用 make qemu 开始系统的运行，而使用 make debug（实质是在运行 qemu 时增加了选项-S）来完成这项工作。这样 qemu 就在启动之初停下来，等待 gdb 连接。

BIOS 首先运行在 16 位实模式下，第一条指令是 ljmp，执行这条指令后会跳到另外一个地方。gdb 默认是 32bit 线性地址模式，调试 BIOS 的 16bit 代码（段地址）你需要手动计算地址，计算公式如下：

Linear Addr = (cs <<4) + ip

如果 CS=0xf000，EIP=0xe05b，则 Linear Address=0xfe05b

另外，为了正确反汇编 16bit 指令

gdb 中执行

```
(gdb) set architecture i8086
(gdb) x/16i 0xfe05b
0xfe05b: cmpl $0x0,%cs:-0x2f2c
0xfe062: jne 0xfc792
……
```

gdb 的地址断点

在 gdb 命令行中，使用 b *[地址]便可以在指定内存地址设置断点，当 qemu 中的 cpu 执行到指定地址时，便会将控制权交给 gdb。

关于代码的反汇编

有可能 gdb 无法正确获取当前 qemu 执行的汇编指令，通过如下配置可以在每次 gdb 命令行前强制反汇编当前的指令，在 gdb 命令行或配置文件中添加：

```
define hook-stop
x/i$pc
end
```

即可

gdb 的单步命令

在 gdb 中，有 next, nexti, step, stepi 等指令来单步调试程序，他们功能各不相同，区别在于单步的“跨度”上。

next 单步到程序源代码的下一行，不进入函数。

nexti 单步一条机器指令，不进入函数。

step 单步到下一个不同的源代码行（包括进入函数）。

stepi 单步一条机器指令。

练习 3：分析 bootloader 进入保护模式的过程。（要求在报告中写出分析）

BIOS 将通过读取硬盘主引导扇区到内存，并转跳到对应内存中的位置执行 bootloader。请分析 bootloader 是如何完成从实模式进入保护模式的。

提示：需要阅读 3.2.1 小节“保护模式和分段机制”和 lab1/boot/bootasm.S 源码，了解如何从实模式切换到保护模式。

练习 4：分析 bootloader 加载 ELF 格式的 OS 的过程。（要求在报告中写出分析）

通过阅读 bootmain.c，了解 bootloader 如何加载 ELF 文件。通过分析源代码和通过 qemu 来运行并调试 bootloader&OS，

1. bootloader 如何读取硬盘扇区的？
2. bootloader 是如何加载 ELF 格式的 OS？

提示：可阅读 3.2.3“硬盘访问概述”，3.2.4“ELF 执行文件格式概述”。

练习 5：实现函数调用堆栈跟踪函数 （需要编程）

我们需要在 lab1 中完成 kdebug.c 中函数 print_stackframe 的实现，可以通过函数 print_stackframe 来跟踪函数调用堆栈中记录的返回地址。在如果能够正确实现此函数，可在 lab1 中执行 “make qemu”后，在 qemu 模拟器中得到类似如下的输出：

```
……
ebp:0x00007b28 eip:0x00100992 args:0x00010094 0x00010094 0x00007b58 0x00100096
    kern/debug/kdebug.c:305: print_stackframe+22
ebp:0x00007b38 eip:0x00100c79 args:0x00000000 0x00000000 0x00000000 0x00007ba8
    kern/debug/kmonitor.c:125: mon_backtrace+10
ebp:0x00007b58 eip:0x00100096 args:0x00000000 0x00007b80 0xffff0000 0x00007b84
    kern/init/init.c:48: grade_backtrace2+33
ebp:0x00007b78 eip:0x001000bf args:0x00000000 0xffff0000 0x00007ba4 0x00000029
    kern/init/init.c:53: grade_backtrace1+38
ebp:0x00007b98 eip:0x001000dd args:0x00000000 0x00100000 0xffff0000 0x0000001d
    kern/init/init.c:58: grade_backtrace0+23
ebp:0x00007bb8 eip:0x00100102 args:0x0010353c 0x00103520 0x00001308 0x00000000
    kern/init/init.c:63: grade_backtrace+34
ebp:0x00007be8 eip:0x00100059 args:0x00000000 0x00000000 0x00000000 0x00007c53
    kern/init/init.c:28: kern_init+88
ebp:0x00007bf8 eip:0x00007d73 args:0xc031fcfa 0xc08ed88e 0x64e4d08e 0xfa7502a8
    <unknow>: -- 0x00007d72 --
……
```

请完成实验，看看输出是否与上述显示大致一致，并解释最后一行各个数值的含义。

提示：可阅读 3.3.1 小节“函数堆栈”，了解编译器如何建立函数调用关系的。在完成 lab1 编译后，查看 lab1/obj/bootblock.asm，了解 bootloader 源码与机器码的语句和地址等的对应关系；查看 lab1/obj/kernel.asm，了解 ucore OS 源码与机器码的语句和地址等的对应关系。

要求完成函数 kern/debug/kdebug.c::print_stackframe 的实现，提交改进后源代码（可以编译执行），并在实验报告中简要说明实现过程，并写出对上述问题的回答。

补充材料：

由于显示完整的栈结构需要解析内核文件中的调试符号，较为复杂和繁琐。代码中有一些辅助函数可以使用。例如可以通过调用 print_debuginfo 函数完成查找对应函数名并打印至屏幕的功能。具体可以参见 kdebug.c 代码中的注释。

练习 6：完善中断初始化和处理 （需要编程）

请完成编码工作和回答如下问题：

1. 中断向量表中一个表项占多少字节？其中哪几位代表中断处理代码的入口？
2. 请编程完善 kern/trap/trap.c 中对中断向量表进行初始化的函数 idt_init。在 idt_init 函数中，依次对所有中断入口进行初始化。使用 mmu.h 中的 SETGATE 宏，填充 idt 数组内容。注意除了系统调用中断(T_SYSCALL)以外，其它中断均使用中断门描述符，权限为内核态权限；而系统调用中断使用异常，权限为陷阱门描述符。每个中断的入口由 tools/vectors.c 生成，使用 trap.c 中声明的 vectors 数组即可。

3. 请编程完善 trap.c 中的中断处理函数 trap，在对时钟中断进行处理的部分填写 trap 函数中处理时钟中断的部分，使操作系统每遇到 100 次时钟中断后，调用 print_ticks 子程序，向屏幕上打印一行文字”100 ticks”。

要求完成问题 2 和问题 3 提出的相关函数实现，提交改进后的源代码（可以编译执行），并在实验报告中简要说明实现过程，并写出对问题 1 的回答。完成这问题 2 和 3 要求的部分代码后，运行整个系统，可以看到大约每 1 秒会输出一次”100 ticks”，而按下的键也会在屏幕上显示。

提示：可阅读 3.3.2 小节“中断与异常”。

扩展练习 Challenge（需要编程）

- 扩展 proj4,增加 syscall 功能，即增加一用户态函数（可执行一特定系统调用：获得时钟计数值），当内核初始完毕后，可从内核态返回到用户态的函数，而用户态的函数又通过系统调用得到内核态的服务（通过网络查询所需信息，可找老师咨询。如果完成，且有兴趣做代替考试的实验，可找老师商量）。需写出详细的设计和分析报告。完成出色的可获得适当加分。

提示：

Challenge 的流程为：

kern_init 调用 switch_test，该函数如下：

```
static void
switch_test(void) {
    print_cur_status();             // print 当前 cs/ss/ds 等寄存器状态
    cprintf("+++ switch to  user  mode +++\n");
    switch_to_user();               // switch to user mode
    print_cur_status();
    cprintf("+++ switch to kernel mode +++\n");
    switch_to_kernel();             // switch to kernel mode
    print_cur_status();
}
```

switch_to_* 函数建议通过中断处理的方式实现。主要要完成的代码是在 trap 里面处理 T_SWITCH_TO* 中断，并设置好返回的状态。

在 lab1 里面完成代码以后，执行 make grade 应该能够评测结果是否正确。

2.2 项目组成

lab1 的整体目录结构如下所示：

```
.
├── boot
│   ├── asm.h
│   ├── bootasm.S
│   └── bootmain.c
├── kern
│   ├── debug
│   │   ├── assert.h
│   │   ├── kdebug.c
│   │   ├── kdebug.h
│   │   ├── kmonitor.c
```

```
    ├── kmonitor.h
    ├── panic.c
    └── stab.h
  ├── driver
    ├── clock.c
    ├── clock.h
    ├── console.c
    ├── console.h
    ├── intr.c
    ├── intr.h
    ├── kbdreg.h
    ├── picirq.c
    └── picirq.h
  ├── init
    └── init.c
  ├── libs
    ├── readline.c
    └── stdio.c
  ├── mm
    ├── memlayout.h
    ├── mmu.h
    ├── pmm.c
    └── pmm.h
  └── trap
      ├── trap.c
      ├── trapentry.S
      ├── trap.h
      └── vectors.S
├── libs
  ├── defs.h
  ├── elf.h
  ├── error.h
  ├── printfmt.c
  ├── stdarg.h
  ├── stdio.h
  ├── string.c
  ├── string.h
  └── x86.h
├── Makefile
└── tools
    ├── function.mk
    ├── gdbinit
    ├── grade.sh
    ├── kernel.ld
    ├── sign.c
    └── vector.c

10 directories, 48 files
```

其中一些比较重要的文件说明如下：

bootloader 部分

- boot/bootasm.S ：定义并实现了 bootloader 最先执行的函数 start，此函数进行了一定的初始化，完成了从实模式到保护模式的转换，并调用 bootmain.c 中的 bootmain 函数。
- boot/bootmain.c：定义并实现了 bootmain 函数实现了通过屏幕、串口和并口显示字符串。bootmain 函数加载 ucore 操作系统到内存，然后跳转到 ucore 的入口处执行。
- boot/asm.h：是 bootasm.S 汇编文件所需要的头文件，主要是一些与 X86 保护模式的段访问方式相关的宏定义。

ucore 操作系统部分：

系统初始化部分：

- kern/init/init.c：ucore 操作系统的初始化启动代码

内存管理部分：

- kern/mm/memlayout.h：ucore 操作系统有关段管理（段描述符编号、段号等）的一些宏定义
- kern/mm/mmu.h：ucore 操作系统有关 X86 MMU 等硬件相关的定义，包括 EFLAGS 寄存器中各位的含义，应用/系统段类型，中断门描述符定义，段描述符定义，任务状态段定义，NULL 段声明的宏 SEG_NULL, 特定段声明的宏 SEG，设置中断门描述符的宏 SETGATE（在练习 6 中会用到）
- kern/mm/pmm.[ch]：设定了 ucore 操作系统在段机制中要用到的全局变量：任务状态段 ts，全局描述符表 gdt[]，加载全局描述符表寄存器的函数 lgdt，临时的内核栈 stack0；以及对全局描述符表和任务状态段的初始化函数 gdt_init

外设驱动部分

- kern/driver/intr.[ch]：实现了通过设置 CPU 的 eflags 来屏蔽和使能中断的函数；
- kern/driver/picirq.[ch]：实现了对中断控制器 8259A 的初始化和使能操作；
- kern/driver/clock.[ch]：实现了对时钟控制器 8253 的初始化操作；
- kern/driver/console.[ch]：实现了对串口和键盘的中断方式的处理操作；

中断处理部分：

- kern/trap/vectors.S：包括 256 个中断服务例程的入口地址和第一步初步处理实现。注意，此文件是由 tools/vector.c 在编译 ucore 期间动态生成的；
- kern/trap/trapentry.S：紧接着第一步初步处理后，进一步完成第二步初步处理；并且有恢复中断上下文的处理，即中断处理完毕后的返回准备工作；
- kern/trap/trap.[ch]：紧接着第二步初步处理后，继续完成具体的各种中断处理操作；

内核调试部分：

- kern/debug/kdebug.[ch]：提供源码和二进制对应关系的查询功能，用于显示调用栈关系。其中补全 print_stackframe 函数是需要完成的练习。其他实现部分不必深究。
- kern/debug/kmonitor.[ch]：实现提供动态分析命令的 kernel monitor，便于在 ucore 出现 bug 或问题后，能够进入 kernel monitor 中，查看当前调用关系。实现部分不必深究。
- kern/debug/panic.c | assert.h：提供了 panic 函数和 assert 宏，便于在发现错误后，调用 kernel monitor。大家可在编程实验中充分利用 assert 宏和 panic 函数，提高查找错误的效率。

公共库部分

- libs/defs.h：包含一些无符号整型的缩写定义。
- Libs/x86.h：一些用 GNU C 嵌入式汇编实现的 C 函数（由于使用了 inline 关键字，所以可以理解为宏）。

工具部分

- Makefile 和 function.mk：指导 make 完成整个软件项目的编译，清除等工作。
- sign.c：一个 C 语言小程序，是辅助工具，用于生成一个符合规范的硬盘主引导扇区。
- tools/vector.c：生成 vectors.S，此文件包含了中断向量处理的统一实现。

编译方法

按照lab0的要求进行环境配置并下载源代码后，在lab1目录下执行make，可以生成ucore.img（生成于bin目录下）。ucore.img是一个包含了bootloader或OS的硬盘镜像，通过执行如下命令可在硬件虚拟环境 qemu中运行bootloader或OS：

```
$ make qemu
```

3 从机器启动到操作系统运行的过程

3.1 BIOS启动过程

当计算机加电后，一般不直接执行操作系统，而是执行系统初始化软件完成基本IO初始化和引导加载功能。简单地说，系统初始化软件就是在操作系统内核运行之前运行的一段小软件。通过这段小软件，我们可以初始化硬件设备、建立系统的内存空间映射图，从而将系统的软硬件环境带到一个合适的状态，以便为最终调用操作系统内核准备好正确的环境。最终引导加载程序把操作系统内核映像加载到RAM中，并将系统控制权传递给它。

对于绝大多数计算机系统而言，操作系统和应用软件是存放在磁盘（硬盘/软盘）、光盘、EPROM、ROM、Flash等可在掉电后继续保存数据的存储介质上。计算机启动后，CPU一开始会到一个特定的地址开始执行指令，这个特定的地址存放了系统初始化软件，负责完成计算机基本的IO初始化，这是系统加电后运行的第一段软件代码。对于Intel 80386的体系结构而言，PC机中的系统初始化软件由BIOS (Basic Input Output System，即基本输入/输出系统，其本质是一个固化在主板Flash/CMOS上的软件)和位于软盘/硬盘引导扇区中的OS Boot Loader（在ucore中的bootasm.S和bootmain.c）一起组成。BIOS实际上是被固化在计算机ROM（只读存储器）芯片上的一个特殊的软件，为上层软件提供最底层的、最直接的硬件控制与支持。更形象地说，BIOS就是PC计算机硬件与上层软件程序之间的一个"桥梁"，负责访问和控制硬件。

以Intel 80386为例，计算机加电后，CPU从物理地址0xFFFFFFF0（详细过程请参考附录“启动后第一条执行的指令”）开始执行。在0xFFFFFFF0这里只是存放了一条跳转指令，通过跳转指令跳到BIOS例行程序起始点。BIOS做完计算机硬件自检和初始化后，会选择一个启动设备（例如软盘、硬盘、光盘等），并且读取该设备的第一扇区(即主引导扇区或启动扇区)到内存一个特定的地址0x7c00处，然后CPU控制权会转移到那个地址继续执行。至此BIOS的初始化工作做完了，进一步的工作交给了ucore的bootloader。

3.2 bootloader启动过程

BIOS将通过读取硬盘主引导扇区到内存，并转跳到对应内存中的位置执行bootloader。bootloader完成的工作包括：

- 切换到保护模式，启用分段机制
- 读磁盘中ELF执行文件格式的ucore操作系统到内存
- 显示字符串信息
- 把控制权交给ucore操作系统

对应其工作的实现文件在lab1中的boot目录下的三个文件asm.h、bootasm.S和bootmain.c。下面从原理上介绍完成上述工作的计算机系统硬件和软件背景知识。

3.2.1 保护模式和分段机制

为何要了解Intel 80386的保护模式和分段机制？首先，我们知道Intel 80386只有在进入保护模式后，才能充分发挥其强大的功能，提供更好的保护机制和更大的寻址空间，否则仅仅是一个快速的8086而已。没有一定的保护机制，任何一个应用软件都可以任意访问所有的计算机资源，这样也

就无从谈起操作系统设计了。且 Intel 80386 的分段机制一直存在，无法屏蔽或避免。其次，在我们的 bootloader 设计中，涉及到了从实模式到保护模式的处理，我们的操作系统功能（比如分页机制）是建立在 Intel 80386 的保护模式上来设计的。如果我们不了解保护模式和分段机制，则我们面向 Intel 80386 体系结构的操作系统设计实际上是建立在一个空中楼阁之上。

【注意】虽然大家学习过 X86 汇编，对 X86 硬件架构有一定了解，但对 X86 保护模式和 X86 系统编程可能了解不够。为了能够清楚了解各个实验中汇编代码的含义，我们建议大家阅读如下参考资料：

- 可先回顾一下 lab0-manual 中的“了解处理器硬件”一节的内容。
- 《Intel 80386 Reference Programmers Manual-i386》：第四、六、九、十章。在后续实验中，还可以进一步阅读第五、七、八等章节。

(1) 实模式

在 bootloader 接手 BIOS 的工作后，当前的 PC 系统处于实模式（16 位模式）运行状态，在这种状态下软件可访问的物理内存空间不能超过 1MB，且无法发挥 Intel 80386 以上级别的 32 位 CPU 的 4GB 内存管理能力。

实模式将整个物理内存看成分段的区域，程序代码和数据位于不同区域，操作系统和用户程序并没有区别对待，而且每一个指针都是指向实际的物理地址。这样，用户程序的一个指针如果指向了操作系统区域或其他用户程序区域，并修改了内容，那么其后果就很可能是灾难性的。通过修改寄存器 CR0 的最低位可以完成从实模式到保护模式的转换，在此之前需要使能高位（20 位以上）地址线。有关 A20 的进一步信息可参考附录“关于 A20 Gate”。

(2) 保护模式

只有在保护模式下，80386 的全部 32 根地址线有效，可寻址高达 4G 字节的线性地址空间和物理地址空间，可访问 64TB（有 2^14 个段，每个段最大空间为 2^32 字节）的逻辑地址空间，可采用分段存储管理机制和分页存储管理机制。这不仅为存储共享和保护提供了硬件支持，而且为实现虚拟存储提供了硬件支持。通过提供 4 个特权级和完善的特权检查机制，既能实现资源共享又能保证代码数据的安全及任务的隔离。

(3) 分段存储管理机制

只有在保护模式下才能使用分段存储管理机制。分段机制将内存划分成以起始地址和长度限制这两个二维参数表示的内存块，这些内存块就称之为段（Segment）。编译器把源程序编译成执行程序时用到的代码段、数据段、堆和栈等概念在这里可以与段联系起来，二者在含义上是一致的。

分段机涉及 4 个关键内容：逻辑地址、段描述符（描述段的属性）、段描述符表（包含多个段描述符的“数组”）、段选择子（段寄存器，用于定位段描述符表中表项的索引）。转换逻辑地址（Logical Address,应用程序员看到的地址）到物理地址（Physical Address, 实际的物理内存地址）分以下两步：

1. 分段地址转换：CPU 把逻辑地址（由段选择子 selector 和段偏移 offset 组成）中的段选择子的内容作为段描述符表的索引，找到表中对应的段描述符，然后把段描述符中保存的段基址加上段偏移值，形成线性地址（Linear Address）。如果不启动分页存储管理机制，则线性地址等于物理地址。
2. 分页地址转换，这一步中把线性地址转换为物理地址。（注意：这一步是可选的，由操作系统决定是否需要。在后续试验中会涉及。

上述转换过程对于应用程序员来说是不可见的。**线性地址空间**由一维的线性地址构成，线性地址空间和物理地址空间对等。线性地址 32 位长，线性地址空间容量为 4G 字节。分段地址转换的基本过程如下图所示。

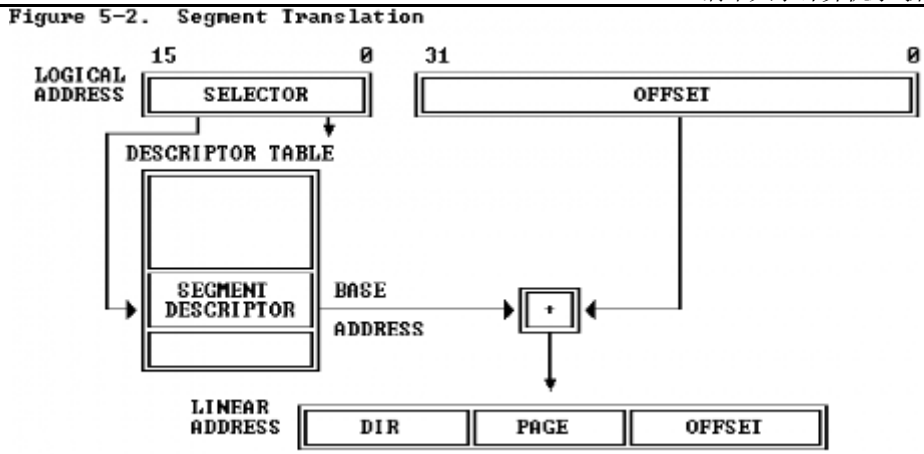

图 1 分段地址转换基本过程

分段存储管理机制需要在启动保护模式的前提下建立。从上图可以看出，为了使得分段存储管理机制正常运行，需要建立好段描述符和段描述符表（参看 bootasm.S，mmu.h，pmm.c）。

段描述符

在分段存储管理机制的保护模式下，每个段由如下三个参数进行定义：段基地址(Base Address)、段界限(Limit)和段属性(Attributes)。在 ucore 中的 kern/mm/mmu.h 中的 struct segdesc 数据结构中有具体的定义。

- 段基地址：规定线性地址空间中段的起始地址。在 80386 保护模式下，段基地址长 32 位。因为基地址长度与寻址地址的长度相同，所以任何一个段都可以从 32 位线性地址空间中的任何一个字节开始，而不象实方式下规定的边界必须被 16 整除。
- 段界限：规定段的大小。在 80386 保护模式下，段界限用 20 位表示，而且段界限可以是以字节为单位或以 4K 字节为单位。
- 段属性：确定段的各种性质。
  - 段属性中的粒度位（Granularity），用符号 G 标记。G=0 表示段界限以字节位位单位，20 位的界限可表示的范围是 1 字节至 1M 字节，增量为 1 字节；G=1 表示段界限以 4K 字节为单位，于是 20 位的界限可表示的范围是 4K 字节至 4G 字节，增量为 4K 字节。
  - 类型（TYPE）：用于区别不同类型的描述符。可表示所描述的段是代码段还是数据段，所描述的段是否可读/写/执行，段的扩展方向等。
  - 描述符特权级（Descriptor Privilege Level）（DPL）：用来实现保护机制。
  - 段存在位（Segment-Present bit）：如果这一位为 0，则此描述符为非法的，不能被用来实现地址转换。如果一个非法描述符被加载进一个段寄存器，处理器会立即产生异常。图 5-4 显示了当存在位为 0 时，描述符的格式。操作系统可以任意的使用被标识为可用（AVAILABLE）的位。
  - 已访问位（Accessed bit）：当处理器访问该段（当一个指向该段描述符的选择子被加载进一个段寄存器）时，将自动设置访问位。操作系统可清除该位。

上述参数通过段描述符来表示，段描述符的结构如下图所示：

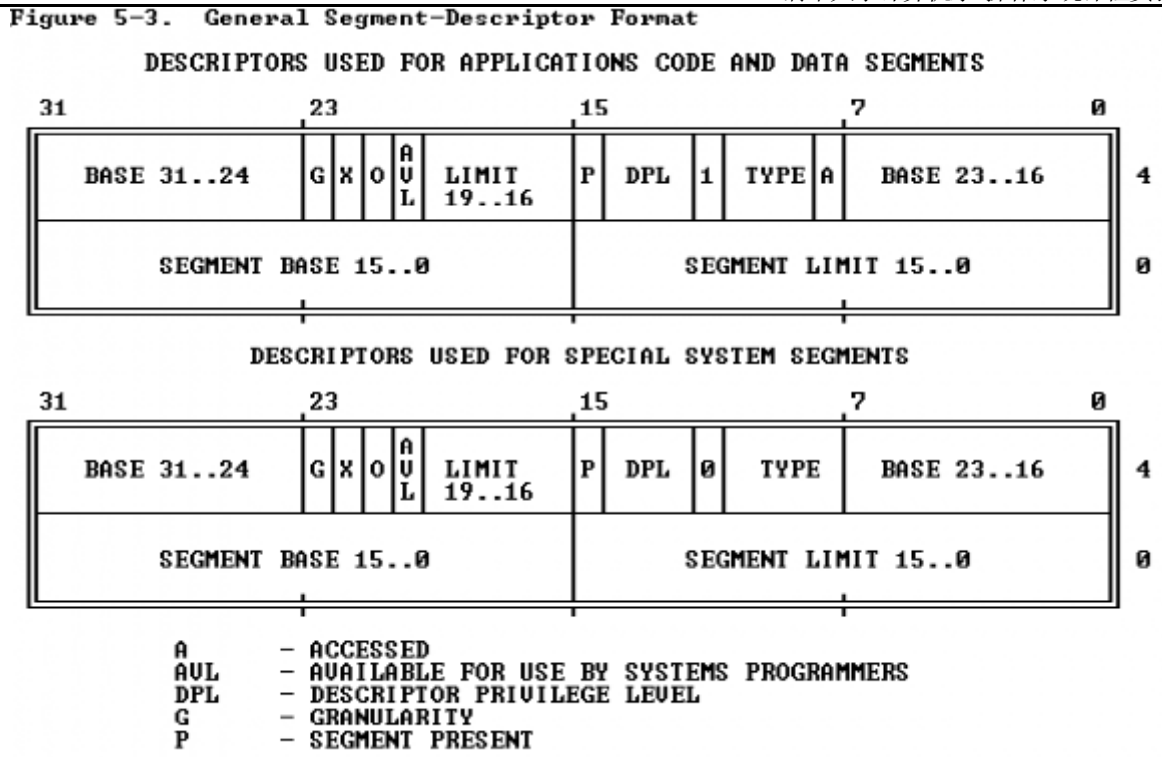

图 2 段描述符结构

全局描述符表

全局描述符表的是一个保存多个段描述符的“数组”，其起始地址保存在全局描述符表寄存器 GDTR 中。GDTR 长 48 位，其中高 32 位为基地址，低 16 位为段界限。由于 GDT 不能有 GDT 本身之内的描述符进行描述定义，所以处理器采用 GDTR 为 GDT 这一特殊的系统段。注意，全部描述符表中第一个段描述符设定为空段描述符。GDTR 中的段界限以字节为单位。对于含有 N 个描述符的描述符表的段界限通常可设为 8*N-1。在 ucore 中的 boot/bootasm.S 中的 gdt 地址处和 kem/mm/pmm.c 中的全局变量数组 gdt[]分别有基于汇编语言和 C 语言的全局描述符表的具体实现。

选择子

线性地址部分的**选择子**是用来选择哪个描述符表和在该表中索引一个描述符的。选择子可以做为指针变量的一部分，从而对应用程序员是可见的，但是一般是由连接加载器来设置的。选择子的格式如下图所示：

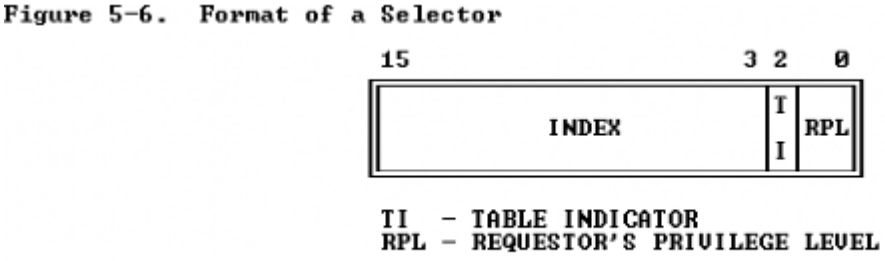

图 3 段选择子结构

- **索引（Index）**：在描述符表中从 8192 个描述符中选择一个描述符。处理器自动将这个索引值乘以 8（描述符的长度），再加上描述符表的基址来索引描述符表，从而选出一个合适的描述符。
- **表指示位（Table Indicator，TI）**：选择应该访问哪一个描述符表。0 代表应该访问全局描述符表（GDT），1 代表应该访问局部描述符表（LDT）。

- **请求特权级（Requested Privilege Level，RPL）**：保护机制，在后续试验中会进一步讲解。

全局描述符表的第一项是不能被 CPU 使用，所以当一个段选择子的索引（Index）部分和表指示位（Table Indicator）都为 0 的时（即段选择子指向全局描述符表的第一项时），可以当做一个空的选择子（见 mmu.h 中的 SEG_NULL）。当一个段寄存器被加载一个空选择子时，处理器并不会产生一个异常。但是，当用一个空选择子去访问内存时，则会产生异常。

(4) 保护模式下的特权级

在保护模式下，特权级总共有 4 个，编号从 0（最高特权）到 3（最低特权）。有 3 种主要的资源受到保护：内存，I/O 端口以及执行特殊机器指令的能力。在任一时刻，x86 CPU 都是在一个特定的特权级下运行的，从而决定了代码可以做什么，不可以做什么。这些特权级经常被称为为保护环（protection ring），最内的环（ring 0）对应于最高特权 0，最外面的环（ring 3）一般给应用程序使用，对应最低特权 3。在 ucore 中，CPU 只用到其中的 2 个特权级：0（内核态）和 3（用户态）。

有大约 15 条机器指令被 CPU 限制只能在内核态执行，这些机器指令如果被用户模式的程序所使用，就会颠覆保护模式的保护机制并引起混乱，所以它们被保留给操作系统内核使用。如果企图在 ring 0 以外运行这些指令，就会导致一个一般保护异常（general-protection exception）。对内存和 I/O 端口的访问也受类似的特权级限制。

数据段选择子的整个内容可由程序直接加载到各个段寄存器（如 SS 或 DS 等）当中。这些内容里包含了请求特权级（Requested Privilege Level，简称 RPL）字段。然而，代码段寄存器（CS）的内容不能由装载指令（如 MOV）直接设置，而只能被那些会改变程序执行顺序的指令（如 JMP、INT、CALL）间接地设置。而且 CS 拥有一个由 CPU 维护的**当前特权级**字段（Current Privilege Level，简称 CPL）。二者结构如下图所示：

图 4 DS 和 CS 的结构图

代码段寄存器中的 CPL 字段（2 位）的值**总是等于** CPU 的当前特权级，所以只要看一眼 CS 中的 CPL，你就可以知道此刻的特权级了。

CPU 会在两个关键点上保护内存：当一个段选择符被加载时，以及，当通过线性地址访问一个内存页时。因此，保护也反映在内存地址转换的过程之中，既包括分段又包括分页。当一个数据段选择符被加载时，就会发生下述的检测过程：

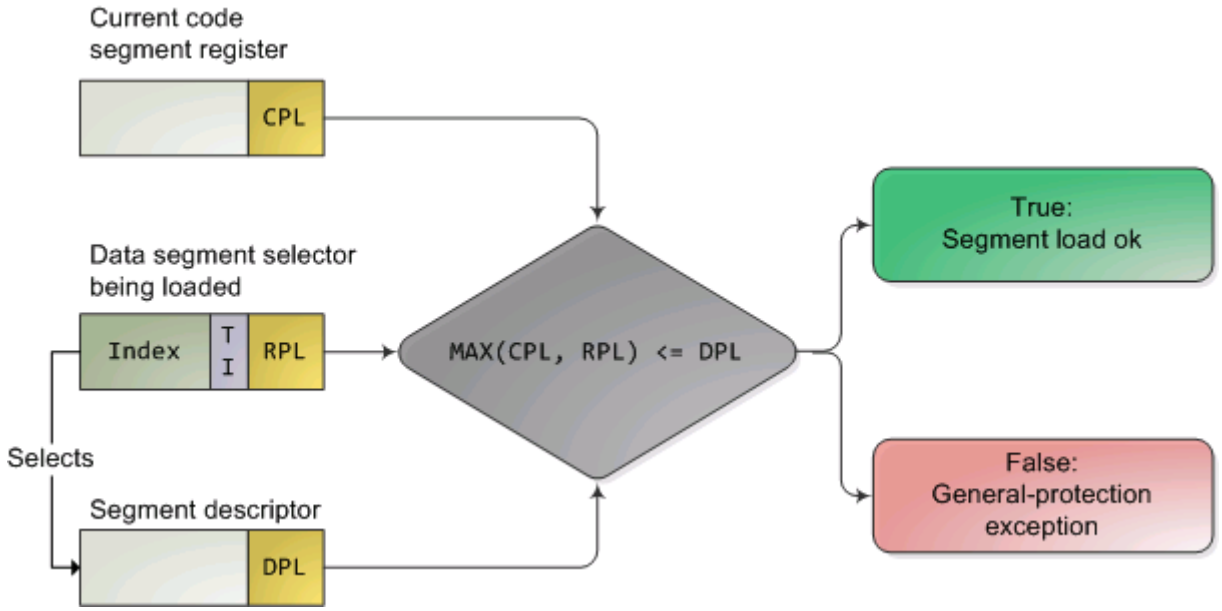

图 5 内存访问特权级检查过程

因为越高的数值代表越低的特权，上图中的 MAX()用于选择 CPL 和 RPL 中特权最低的一个，并与描述符特权级（Descriptor Privilege Level，简称 DPL）比较。如果 DPL 的值大于等于它，那么这个访问可正常进行了。RPL 背后的设计思想是：允许内核代码加载特权较低的段。比如，你可以使用 RPL=3 的段描述符来确保给定的操作所使用的段可以在用户模式中访问。但堆栈段寄存器是个例外，它要求 CPL，RPL 和 DPL 这 3 个值必须完全一致，才可以被加载。下面再总结一下 CPL、RPL 和 DPL：

- CPL：当前特权级（Current Privilege Level）保存在 CS 段寄存器（选择子）的最低两位，CPL 就是当前活动代码段的特权级，并且它定义了当前所执行程序的特权级别）
- DPL：描述符特权（Descriptor Privilege Level） 存储在段描述符中的权限位，用于描述对应段所属的特权等级，也就是段本身真正的特权级。
- RPL：请求特权级 RPL(Request Privilege Level) RPL 保存在选择子的最低两位。RPL 说明的是进程对段访问的请求权限，意思是当前进程想要的请求权限。RPL 的值由程序员自己来自由的设置，并不一定 RPL>=CPL，但是当 RPL<CPL 时，实际起作用的就是 CPL 了，因为访问时的特权检查是判断：**max(RPL,CPL)<=DPL** 是否成立，所以 RPL 可以看成是每次访问时的附加限制，RPL=0 时附加限制最小，RPL=3 时附加限制最大。

3.2.2 地址空间

分段机制涉及 4 个关键内容：逻辑地址（Logical Address,应用程序员看到的地址，在操作系统原理上称为虚拟地址，以后提到虚拟地址就是指逻辑地址）、物理地址（Physical Address, 实际的物理内存地址）、段描述符表（包含多个段描述符的“数组”）、段描述符（描述段的属性，及段描述符表这个“数组”中的“数组元素”）、段选择子（即段寄存器中的值，用于定位段描述符表中段描述符表项的索引）

(1) 逻辑地址空间

从应用程序的角度看，逻辑地址空间就是应用程序员编程所用到的地址空间，比如下面的程序片段：

```
int val=100;
int * point=&val;
```

其中指针变量 point 中存储的即是一个逻辑地址。在基于 80386 的计算机系统中，逻辑地址有一个 16 位的段寄存器（也称段选择子，段选择子）和一个 32 位的偏移量构成。

(2) 物理地址空间

从操作系统的角度看，CPU、内存硬件（通常说的“内存条”）和各种外设是它主要管理的硬件资源而内存硬件和外设分布在物理地址空间中。物理地址空间就是一个“大数组”，CPU 通过索引（物理地址）来访问这个“大数组”中的内容。物理地址是指 CPU 提交到内存总线上用于访问计算机内存和外设的最终地址。

物理地址空间的大小取决于 CPU 实现的物理地址位数，在基于 80386 的计算机系统中，CPU 的物理地址空间为 4GB，如果计算机系统实际上有 1GB 物理内存（即我们通常说的内存条），而其他硬件设备的 IO 寄存器映射到起始物理地址为 3GB 的 256MB 大小的地址空间，则该计算机系统的物理地址空间如下所示：

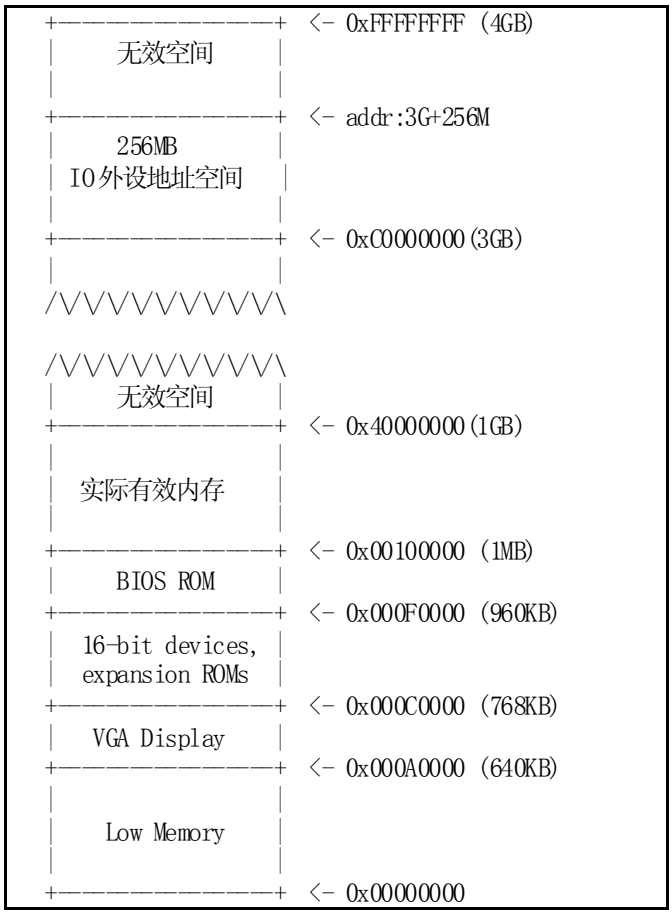

图 6 X86 计算机系统的物理地址空间

(3) 线性地址空间

一台计算机只有一个物理地址空间，但在操作系统的管理下，每个程序都认为自己独占整个计算机的物理地址空间。为了让多个程序能够有效地相互隔离和使用物理地址空间，引入线性地址空间（也称虚拟地址空间）的概念。线性地址空间的大小取决于 CPU 实现的线性地址位数，在基于 80386 的计算机系统中，CPU 的线性地址空间为 4GB。线性地址空间会被映射到某一部分或整个物理地址空间，并通过索引（线性地址）来访问其中的内容。线性地址又称虚拟地址，是进行逻辑地址转换后形成的地址索引，用于寻址线性地址空间。但 CPU 未启动分页机制时，线性地址等于物理地址；当 CPU 启动分页机制时，线性地址还需经过分页地址转换形成物理地址后，CPU 才能访问内存硬件和外设。三种地址的关系如下所示：

- **启动分段机制，未启动分页机制：逻辑地址--> (分段地址转换) -->线性地址==物理地址**
- **启动分段和分页机制：逻辑地址--> (分段地址转换) -->线性地址-->(分页地址转换) -->物理地址**

在操作系统的管理下，采用灵活的内存管理机制，在只有一个物理地址空间的情况下，可以存在多个线性地址空间。

3.2.3 硬盘访问概述

bootloader让CPU进入保护模式后，下一步的工作就是从硬盘上加载并运行OS。考虑到实现的简单性，bootloader访问硬盘都是基于逻辑块地址（LBA）的PIO（Programmed IO）方式，即所有的IO操作是通过CPU访问硬盘的IO地址寄存器完成，每个逻辑块对应硬盘上的一个扇区。

一般主板有2个IDE通道，每个通道可以接2个IDE硬盘。访问第一个硬盘的扇区可设置IO地址寄存器0x1f0-0x1f7实现的，具体参数见下表。一般第一个IDE通道通过访问IO地址0x1f0-0x1f7来实现，第二个IDE通道通过访问0x170-0x17f实现。每个通道的主从盘的选择通过第6个IO偏移地址寄存器来设置。

表一 磁盘IO地址和对应功能

IO地址	功能
0x1f0	读数据，当0x1f7不为忙状态时，可以读。
0x1f2	要读写的扇区数，每次读写前，你需要表明你要读写几个扇区。最小是1个扇区
0x1f3	如果是LBA模式，就是LBA参数的0-7位
0x1f4	如果是LBA模式，就是LBA参数的8-15位
0x1f5	如果是LBA模式，就是LBA参数的16-23位
0x1f6	第0~3位：如果是LBA模式就是24-27位　　第4位：为0主盘；为1从盘 第6位：为1=LBA模式；0 =CHS模式　　第7位和第5位必须为1
0x1f7	状态和命令寄存器。操作时先给命令，再读取，如果不是忙状态就从0x1f0端口读数据

当前硬盘数据是储存到硬盘扇区中，一个扇区大小为512字节。读一个扇区的流程（可参看boot/bootmain.c中的readsect函数实现）大致如下：

1. 等待磁盘准备好
2. 发出读取扇区的命令
3. 等待磁盘准备好
4. 把磁盘扇区数据读到指定内存

3.2.4 ELF文件格式概述

ELF(Executable and linking format)文件格式是Linux系统下的一种常用目标文件(object file)格式，有三种主要类型:

- 用于执行的可执行文件(executable file)，用于提供程序的进程映像，加载的内存执行。 这也是本实验的OS文件类型。
- 用于连接的可重定位文件(relocatable file)，可与其它目标文件一起创建可执行文件和共享目标文件。
- 共享目标文件(shared object file),连接器可将它与其它可重定位文件和共享目标文件连接成其它的目标文件，动态连接器又可将它与可执行文件和其它共享目标文件结合起来创建一个进程映像。

这里只分析与本实验相关的ELF可执行文件类型。ELF header在文件开始处描述了整个文件的组织。ELF的文件头包含整个执行文件的控制结构，其定义在elf.h中：

```
struct elfhdr {
  uint magic;  // must equal ELF_MAGIC
  uchar elf[12];
  ushort type;
  ushort machine;
  uint version;
  uint entry;  // 程序入口的虚拟地址
  uint phoff;  // program header 表的位置偏移
  uint shoff;
```

```
    uint flags;
    ushort ehsize;
    ushort phentsize;
    ushort phnum; //program header 表中的入口数目
    ushort shentsize;
    ushort shnum;
    ushort shstrndx;
};
```

program header 描述与程序执行直接相关的目标文件结构信息，用来在文件中定位各个段的映像，同时包含其他一些用来为程序创建进程映像所必需的信息。可执行文件的程序头部是一个 program header 结构的数组， 每个结构描述了一个段或者系统准备程序执行所必需的其它信息。目标文件的“段”包含一个或者多个“节区”（section），也就是“段内容（Segment Contents）”。程序头部仅对于可执行文件和共享目标文件有意义。可执行目标文件在 ELF 头部的 e_phentsize 和 e_phnum 成员中给出其自身程序头部的大小。程序头部的数据结构如下表所示：

```
struct proghdr {
    uint type;     // 段类型
    uint offset;   // 段相对文件头的偏移值
    uint va;       // 段的第一个字节将被放到内存中的虚拟地址
    uint pa;
    uint filesz;
    uint memsz;    // 段在内存映像中占用的字节数
    uint flags;
    uint align;
};
```

根据 elfhdr 和 proghdr 的结构描述，bootloader 就可以完成对 ELF 格式的 ucore 操作系统的加载过程（参见 boot/bootmain.c 中的 bootmain 函数）。

[补充材料]

Link addr& Load addr

Link Address 是指编译器指定代码和数据所需要放置的内存地址，由链接器配置。Load Address 是指程序被实际加载到内存的位置（由程序加载器 ld 配置）。一般由可执行文件结构信息和加载器可保证这两个地址相同。Link Addr 和 LoadAddr 不同会导致：

- 直接跳转位置错误
- 直接内存访问(只读数据区或 bss 等直接地址访问)错误
- 堆和栈等的使用不受影响，但是可能会覆盖程序、数据区域

注意：也存在 Link 地址和 Load 地址不一样的情况（例如：动态链接库）。

3.3 操作系统启动过程

当 bootloader 通过读取硬盘扇区把 ucore 在系统加载到内存后，就转跳到 ucore 操作系统在内存中的入口位置（kern/init.c 中的 kern_init 函数的起始地址），这样 ucore 就接管了整个控制权。当前的 ucore 功能很简单，只完成基本的内存管理和外设中断管理。ucore 主要完成的工作包括：

- 初始化终端；

- 显示字符串；
- 显示堆栈中的多层函数调用关系；
- 切换到保护模式，启用分段机制；
- 初始化中断控制器，设置中断描述符表，初始化时钟中断，使能整个系统的中断机制；
- 执行 while（1）死循环。

以后的实验中会大量涉及各个函数直接的调用关系，以及由于中断处理导致的异步现象，可能对大家实现操作系统和改正其中的错误有很大影响。而理解好函数调用关系的建立机制和中断处理机制，对后续实验会有很大帮助。下面就练习 5 涉及的函数栈调用关系和练习 6 中的中断机制的建立进行阐述。

3.3.1 函数堆栈

栈是一个很重要的编程概念（编译课和程序设计课都讲过相关内容），与编译器和编程语言有紧密的联系。理解调用栈最重要的两点是：栈的结构，以及 EBP 寄存器的作用。一个函数调用动作可分解为零到多个 PUSH 指令（用于参数入栈）和一个 CALL 指令。CALL 指令内部其实还暗含了一个将返回地址（即 CALL 指令下一条指令的地址）压栈的动作（由硬件完成）。几乎所有本地编译器都会在每个函数体之前插入类似如下的汇编指令：

```
pushl    %ebp
movl     %esp , %ebp
```

这样在程序执行到一个函数的实际指令前，已经有以下数据顺序入栈：参数、返回地址、ebp 寄存器。由此得到类似如下的栈结构（参数入栈顺序跟调用方式有关，这里以 C 语言默认的 CDECL 为例）：

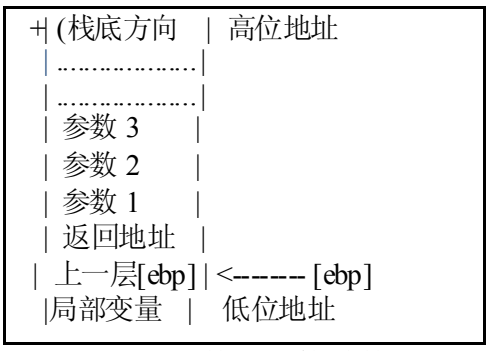

图 7 函数调用栈结构

这两条汇编指令的含义是：首先将 ebp 寄存器入栈，然后将栈顶指针 esp 赋值给 ebp。“movl %esp %ebp”这条指令表面上看是用 esp 覆盖 ebp 原来的值，其实不然。因为给 ebp 赋值之前，原 ebp 值已经被压栈（位于栈顶），而新的 ebp 又恰恰指向栈顶。此时 ebp 寄存器就已经处于一个非常重要的地位，该寄存器中存储着栈中的一个地址（原 ebp 入栈后的栈顶），从该地址为基准，向上（栈底方向）能获取返回地址、参数值，向下（栈顶方向）能获取函数局部变量值，而该地址处又存储着上一层函数调用时的 ebp 值。

一般而言，ss:[ebp+4]处为返回地址，ss:[ebp+8]处为第一个参数值（最后一个入栈的参数值，此处假设其占用 4 字节内存），ss:[ebp-4]处为第一个局部变量，ss:[ebp]处为上一层 ebp 值。由于 ebp 中的地址处总是“上一层函数调用时的 ebp 值”，而在每一层函数调用中，都能通过当时的 ebp 值“向上（栈底方向）”能获取返回地址、参数值，“向下（栈顶方向）”能获取函数局部变量值。如此形成递归，直至到达栈底。这就是函数调用栈。

提示：练习 5 的正确实现取决于对这一小节的正确理解和掌握。

3.3.2 中断与异常

操作系统需要对计算机系统中的各种外设进行管理，这就需要 CPU 和外设能够相互通信才行。一般外设的速度远慢于 CPU 的速度。如果让操作系统采用通常的轮询(polling)机制，使用 CPU“主动关心”外设的事件，就太浪费 CPU 资源了。所以需要操作系统和 CPU 能够一起提供某种机制，让外设在需要操作系统处理外设相关事件的时候，能够“主动通知”操作系统，即打断操作系统和应用的正常执行，让操作系统完成外设的相关处理，然后在恢复操作系统和应用的正常执行。在操作系统中，这种机制称为中断机制。中断机制给操作系统提供了处理意外情况的能力，同时它也是实现进程/线程抢占式调度的一个重要基石。但中断的引入导致了对操作系统的理解更加困难。

在操作系统中，有三种特殊的中断事件。由 CPU 外部设备引起的外部事件如 I/O 中断、时钟中断、控制台中断等是异步产生的（即产生的时刻不确定），与 CPU 的执行无关，可称之为异步中断(asynchronous interrupt)也称外部中断,简称中断(interrupt)。而把在 CPU 执行指令期间检测到不正常的或非法的条件(如除零错、地址访问越界)所引起的内部事件称作同步中断(synchronous interrupt)，也称内部中断，简称异常(exception)。把在程序中使用请求系统服务的系统调用而引发的事件，称作陷入中断(trap interrupt)，简称 trap，也可称为软中断，系统调用即为陷入中断，在后续试验中会进一步讲解系统调用。

本实验只描述保护模式下的处理过程。当 CPU 收到中断（通过 8259A 完成，有关 8259A 的信息请看附录 A）或者异常的事件时，它会暂停执行当前的程序或任务，通过一定的机制跳转到负责处理这个信号的相关处理例程中，在完成对这个事件的处理后再跳回到刚才被打断的程序或任务中。中断向量和中断服务例程的对应关系主要是由 IDT（中断描述符表）负责。操作系统在 IDT 中设置好各种中断向量对应的中断描述符，留待 CPU 在产生中断后查询对应中断服务例程的起始地址。而 IDT 本身的起始地址保存在 idtr 寄存器中。

(1) 中断描述符表（Interrupt Descriptor Table）

中断描述符表把每个中断或异常编号和一个指向中断服务例程的描述符联系起来。同 GDT 一样，IDT 是一个 8 字节的描述符数组，但 IDT 的第一项可以包含一个描述符。CPU 把中断（异常）号乘以 8 做为 IDT 的索引。IDT 可以位于内存的任意位置，CPU 通过 IDT 寄存器（IDTR）的内容来寻址 IDT 的起始地址。指令 LIDT 和 SIDT 用来操作 IDTR。两条指令都有一个显示的操作数：一个 6 字节表示的内存地址。指令的含义如下：

- **LIDT（Load IDT Register）指令：**使用一个包含线性地址基址和界限的内存操作数来加载 IDT。操作系统创建 IDT 时需要执行它来设定 IDT 的起始地址。这条指令只能在特权级 0 执行。（可参见 libs/x86.h 中的 lidt 函数实现，其实就是一条汇编指令）
- **SIDT（Store IDT Register）指令：**拷贝 IDTR 的基址和界限部分到一个内存地址。这条指令可以在任意特权级执行。

IDT 和 IDTR 寄存器的结构和关系如下图所示：

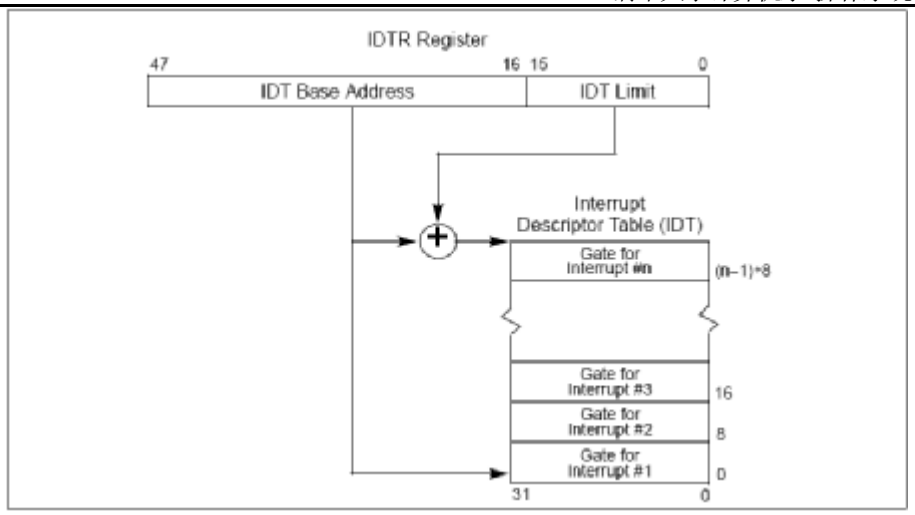

图 8 IDT 和 IDTR 寄存器的结构和关系图

在保护模式下，最多会存在 256 个 Interrupt/Exception Vectors。范围[0，31]内的 32 个向量被异常 Exception 和 NMI 使用，但当前并非所有这 32 个向量都已经被使用，有几个当前没有被使用的，请不要擅自使用它们，它们被保留，以备将来可能增加新的 Exception。范围[32，255]内的向量被保留给用户定义的 Interrupts。Intel 没有定义，也没有保留这些 Interrupts。用户可以将它们用作外部 I/O 设备中断（8259A IRQ），或者系统调用（System Call 、Software Interrupts）等。

(2) IDT gate descriptors

Interrupts/Exceptions 应该使用 Interrupt Gate 和 Trap Gate，它们之间的唯一区别就是：当调用 Interrupt Gate 时，Interrupt 会被 CPU 自动禁止；而调用 Trap Gate 时，CPU 则不会去禁止或打开中断，而是保留它原来的样子。在 IDT 中，可以包含如下 3 种类型的 Descriptor:

- Task-gate descriptor （这里没有使用）
- Interrupt-gate descriptor （中断方式用到）
- Trap-gate descriptor（系统调用用到）

下图图显示了 80386 的任务门描述符、中断门描述符、陷阱门描述符的格式：

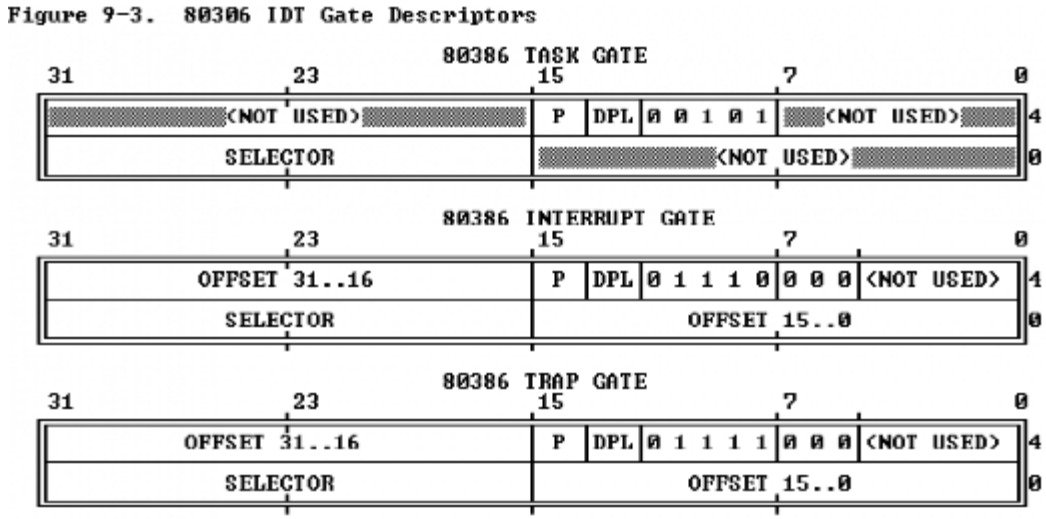

图 9 X86 的各种门的格式

可参见 kern/mm/mmu.h 中的 struct gatedesc 数据结构对中断描述符的具体定义。

(3) 中断处理中硬件负责完成的工作

中断服务例程包括具体负责处理中断（异常）的代码是操作系统的重要组成部分。需要注意区别的是，有两个过程由硬件完成：

- 硬件中断处理过程1（起始）：从CPU收到中断事件后，打断当前程序或任务的执行，根据某种机制跳转到中断服务例程去执行的过程。其具体流程如下：
  1) CPU在执行完当前程序的每一条指令后，都会去确认在执行刚才的指令过程中中断控制器（如：8259A）是否发送中断请求过来。如果有，那么CPU就会在相应的时钟脉冲到来时从总线上读取中断请求对应的中断向量；
  2) CPU根据得到的中断向量（以此为索引）到IDT中找到该向量对应的中断描述符，中断描述符里保存着中断服务例程的段选择子；
  3) CPU使用IDT查到的中断服务例程的段选择子从GDT中取得相应的段描述符，段描述符里保存了中断服务例程的段基址和属性信息，此时CPU就得到了中断服务例程的起始地址，并跳转到该地址；
  4) CPU会根据CPL和中断服务例程的段描述符的DPL信息确认是否发生了特权级的转换。比如当前程序正运行在用户态，而中断程序是运行在内核态的，则意味着发生了特权级的转换，这时CPU会从当前程序的TSS信息（该信息在内存中的起始地址存在TR寄存器中）里取得该程序的内核栈地址，即包括内核态的ss和esp的值，并立即将系统当前使用的栈切换成新的内核栈。这个栈就是即将运行的中断服务程序要使用的栈。紧接着就将当前程序使用的用户态的ss和esp压到新的内核栈中保存起来；
  5) CPU需要开始保存当前被打断的程序的现场（即一些寄存器的值），以便于将来恢复被打断的程序继续执行。这需要利用内核栈来保存相关现场信息，即依次压入当前被打断程序使用的eflags，cs，eip，errorCode（如果是有错误码的异常）信息；
  6) CPU利用中断服务例程的段描述符将其第一条指令的地址加载到cs和eip寄存器中，开始执行中断服务例程。这意味着先前的程序被暂停执行，中断服务程序正式开始工作。
- 硬件中断处理过程2（结束）：每个中断服务例程在有中断处理工作完成后需要通过iret（或iretd）指令恢复被打断的程序的执行。CPU执行IRET指令的具体过程如下：
  1) 程序执行这条iret指令时，首先会从内核栈里弹出先前保存的被打断的程序的现场信息，即eflags，cs，eip重新开始执行；
  2) 如果存在特权级转换（从内核态转换到用户态），则还需要从内核栈中弹出用户态栈的ss和esp，这样也意味着栈也被切换回原先使用的用户态的栈了；
  3) 如果此次处理的是带有错误码（errorCode）的异常，CPU在恢复先前程序的现场时，并不会弹出errorCode。这一步需要通过软件完成，即要求相关的中断服务例程在调用iret返回之前添加出栈代码主动弹出errorCode。

下图显示了从中断向量到GDT中相应中断服务程序起始位置的定位方式:

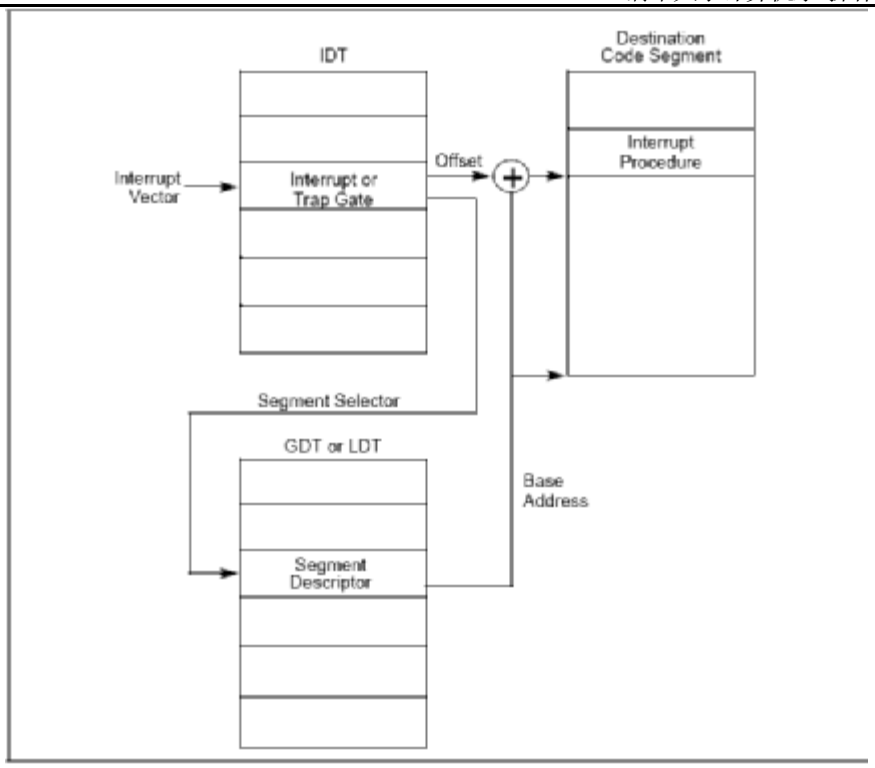

图 10 中断向量与中断服务例程起始地址的关系

(4) 中断产生后的堆栈栈变化

下图显示了给出相同特权级和不同特权级情况下中断产生后的堆栈栈变化示意图：

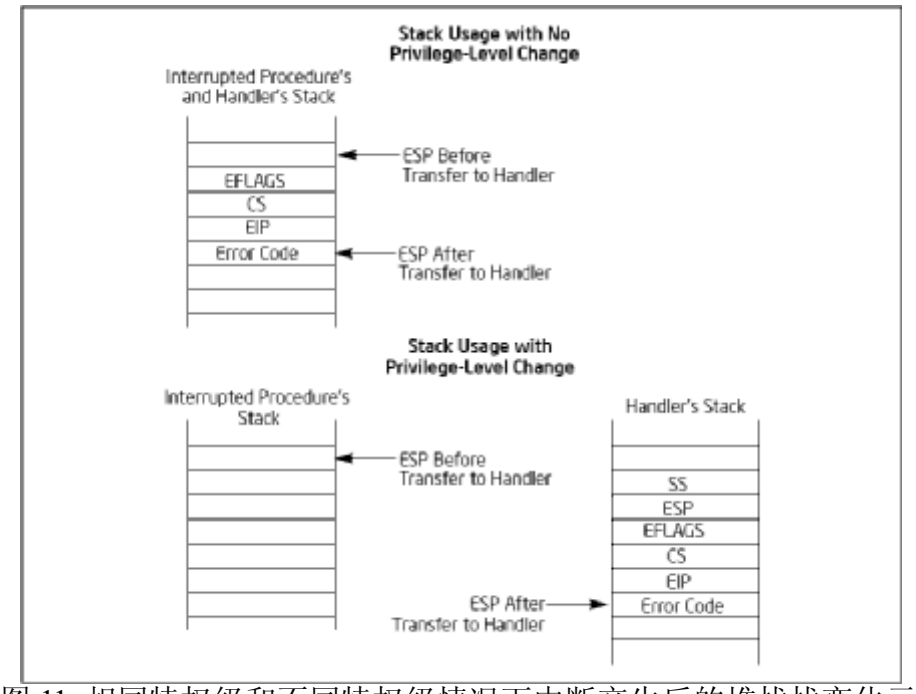

图 11 相同特权级和不同特权级情况下中断产生后的堆栈栈变化示意图

(5) 中断处理的特权级转换

中断处理得特权级转换是通过**门描述符**（gate descriptor）和**相关**指令来完成的。一个门描述符就是一个系统类型的段描述符，一共有 4 个子类型：调用门描述符（call-gate descriptor），中断门描述符（interrupt-gate descriptor），陷阱门描述符（trap-gate descriptor）和任务门描述符（task-gate

descriptor）。与中断处理相关的是中断门描述符和陷阱门描述符。这些门描述符被存储在中断描述符表（Interrupt Descriptor Table，简称 IDT）当中。CPU 把中断向量作为 IDT 表项的索引，用来指出当中断发生时使用哪一个门描述符来处理中断。中断门描述符和陷阱门描述符几乎是一样的。中断发生时实施特权检查的过程如下图所示：

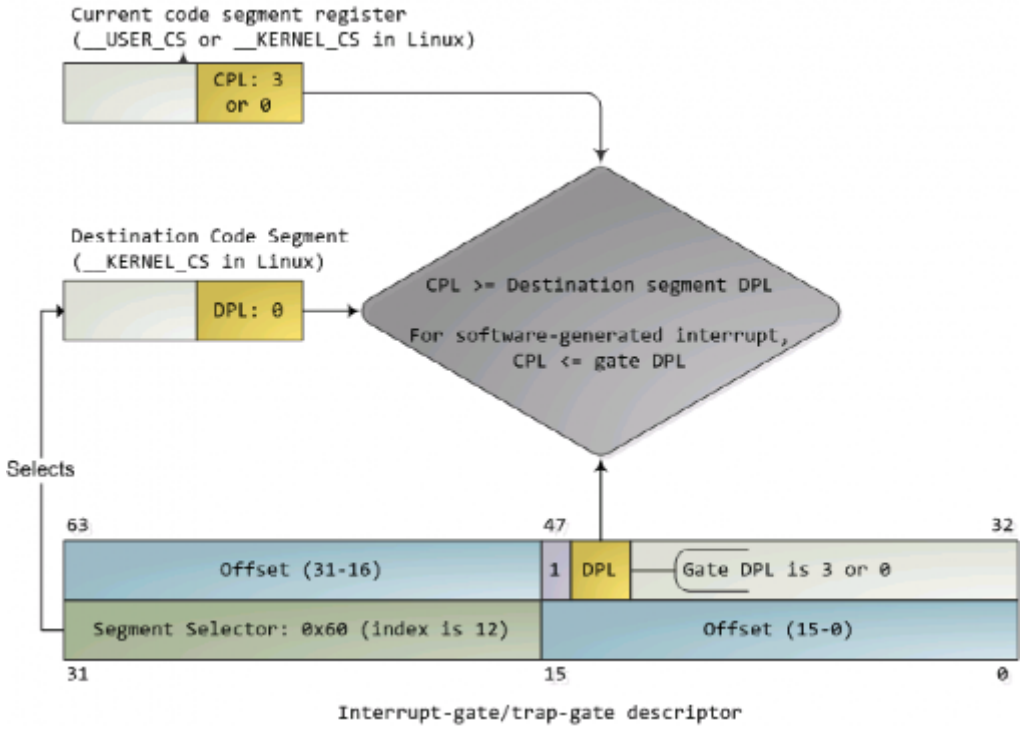

图 12 中断发生时实施特权检查的过程

门中的 DPL 和段选择符一起控制着访问，同时，段选择符结合偏移量（Offset）指出了中断处理例程的入口点。内核一般在门描述符中填入内核代码段的段选择子。产生中断后，CPU 一定不会将运行控制从高特权环转向低特权环，特权级必须要么保持不变（当操作系统内核自己被中断的时候），或被提升（当用户态程序被中断的时候）。无论哪一种情况，作为结果的 CPL 必须等于目的代码段的 DPL。如果 CPL 发生了改变，一个堆栈切换操作（通过 TSS 完成）就会发生。如果中断是被用户态程序中的指令所触发的（比如软件执行 INT n 生产的中断），还会增加一个额外的检查：门的 DPL 必须具有与 CPL 相同或更低的特权。这就防止了用户代码随意触发中断。如果这些检查失败，会产生一个一般保护异常（general-protection exception）。

3.3.3 lab1 中对中断的处理实现

(1) 外设基本初始化设置

Lab1 实现了中断初始化和对键盘、串口、时钟外设进行中断处理。串口的初始化函数 serial_init（位于/kern/driver/console.c）中涉及中断初始化工作的很简单：

```
    ……
// 使能串口1接收字符后产生中断
    outb(COM1 + COM_IER, COM_IER_RDI);
    ……
// 通过中断控制器使能串口1中断
    pic_enable(IRQ_COM1);
```

键盘的初始化函数 kbd_init（位于 kern/driver/console.c 中）完成了对键盘的中断初始化工作，具体操作更加简单：

```
    ……
// 通过中断控制器使能键盘输入中断
    pic_enable(IRQ_KBD);
```

时钟是一种有着特殊作用的外设，其作用并不仅仅是计时。在后续章节中将讲到，正是由于有

了规律的时钟中断，才使得无论当前 CPU 运行在哪里，操作系统都可以在预先确定的时间点上获得 CPU 控制权。这样当一个应用程序运行了一定时间后，操作系统会通过时钟中断获得 CPU 控制权，并可把 CPU 资源让给更需要 CPU 的其他应用程序。时钟的初始化函数 clock_init（位于 kern/driver/clock.c 中）完成了对时钟控制器 8253 的初始化：

```
    ……
//设置时钟每秒中断100次
    outb(IO_TIMER1, TIMER_DIV(100) % 256);
    outb(IO_TIMER1, TIMER_DIV(100) / 256);
// 通过中断控制器使能时钟中断
    pic_enable(IRQ_TIMER);
```

(2) 中断初始化设置

操作系统如果要正确处理各种不同的中断事件，就需要安排应该由哪个中断服务例程负责处理特定的中断事件。系统将所有的中断事件统一进行了编号（0～255），这个编号称为中断向量。以 ucore 为例，操作系统内核启动以后，会通过 idt_init 函数初始化 idt 表 (参见 trap.c)，而其中 vectors 中存储了中断处理程序的入口地址。vectors 定义在 vector.S 文件中，通过一个工具程序 vector.c 生成。其中仅有 System call 中断的权限为用户权限 (DPL_USER)，即仅能够使用 int 0x30 指令。此外还有对 tickslock 的初始化，该锁用于处理时钟中断。

vector.S 文件通过 vectors.c 自动生成，其中定义了每个中断的入口程序和入口地址 (保存在 vectors 数组中)。其中，中断可以分成两类：一类是压入错误编码的 (error code)，另一类不压入错误编码。对于第二类， vector.S 自动压入一个 0。此外，还会压入相应中断的中断号。在压入两个必要的参数之后，中断处理函数跳转到统一的入口 alltraps 处。

(3) 中断的处理过程

trap 函数（定义在 trap.c 中）是对中断进行处理的过程，所有的中断在经过中断入口函数 __alltraps 预处理后 (定义在 trapasm.S 中)，都会跳转到这里。在处理过程中，根据不同的中断类型，进行相应的处理。在相应的处理过程结束以后，trap 将会返回，被中断的程序会继续运行。整个中断处理流程大致如下：

<table>
<tr><td>trapasm.S
1)产生中断后，CPU 跳转到相应的中断处理入口 (vectors)，并在栈中压入相应的 error_code（是否存在与异常号相关） 以及 trap_no，然后跳转到 alltraps 函数入口：
注意：此处的跳转是 jmp 过程
<table><tr><td>(high)</td><td>...</td></tr><tr><td>产生中断时的 eip→</td><td>eip</td></tr><tr><td></td><td>error_code</td></tr><tr><td>esp→</td><td>trap_no</td></tr><tr><td>(low)</td><td>...</td></tr></table>在栈中保存当前被打断程序的 trapframe 结构(参见过程 trapasm.S)。设置 kernel (内核) 的数据段寄存器，最后压入 esp，作为 trap 函数参数(struct trapframe * tf) 并跳转到中断处理函数 trap 处：
注意：此时的跳转是 call 调用，会压入返回地址 eip，注意区分此处 eip 与 trapframe 中 eip：
trapframe 的结构为：</td><td>trap.c</td></tr>
</table>

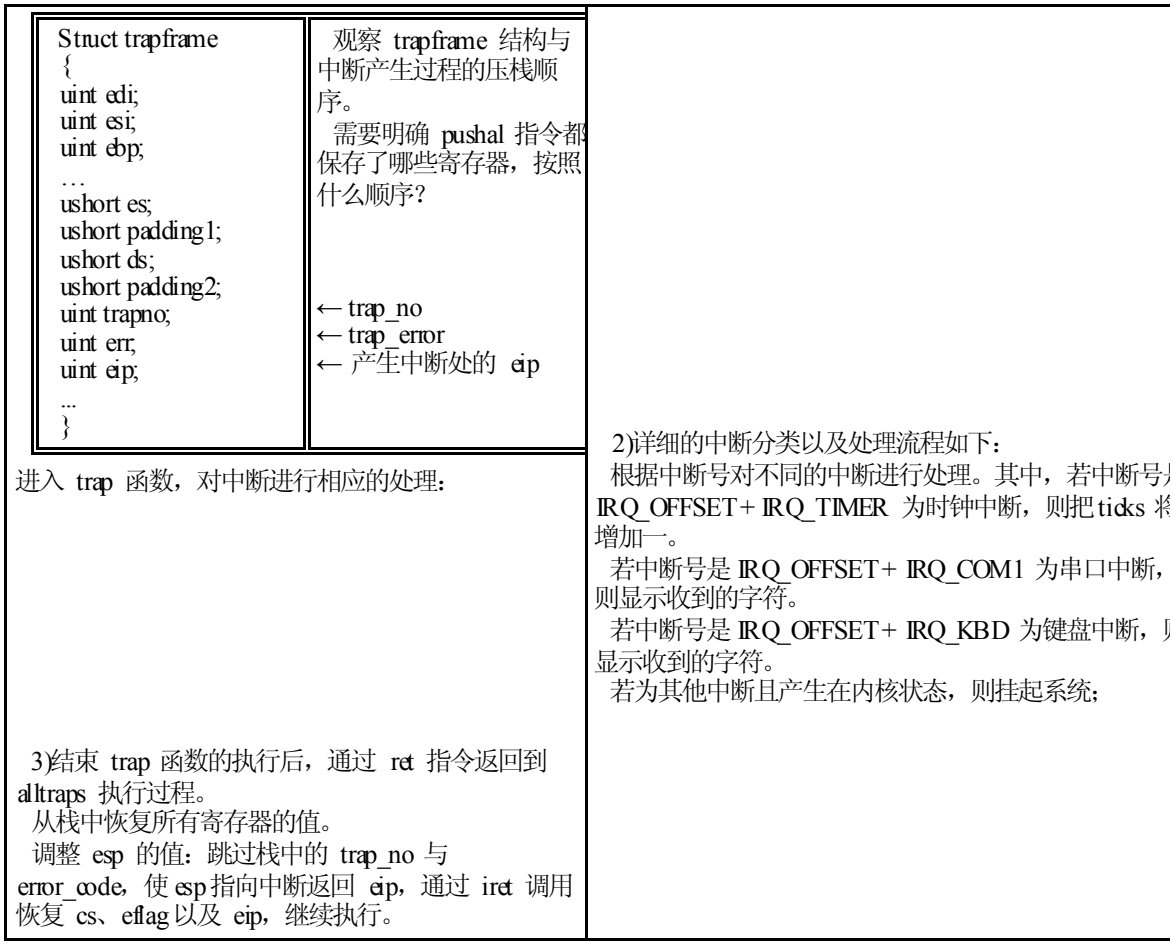

图 13 ucore 中断处理流程

至此，对整个 lab1 中的主要部分的背景知识和实现进行了阐述。请大家能够根据前面的练习要求完成所有的练习。

4 实验报告要求

使用 git 账号下载实验代码后，在目录 lab1 下创建实验报告 report.txt，回答所有练习中提出的问题，同时在该目录的代码中完成本实验的编程练习。完成实验后，用 git add、git commit 和 git push 提交、上传你的结果（详细过程请参考实验零文档的 2.2.1 节）。

注意有“**LAB1**”的注释，代码中所有需要完成的地方（challenge除外）都有“**LAB1**”和“**YOUR CODE**”的注释，请在提交时特别注意保持注释，并将“**YOUR CODE**”替换为自己的学号，并且将所有标有对应注释的部分填上正确的代码。

附录“关于 A20 Gate”

【参考“关于 A20 Gate” http://hengch.blog.163.com/blog/static/107800672009013104623747/ 】

【参考“百度文库 激活 A20 地址线详解”
http://wenku.baidu.com/view/d6efe68fcc22bcd126ff0c00.html】

Intel 早期的 8086 CPU 提供了 20 根地址线,可寻址空间范围即 0~2^20(00000H~FFFFFH)的 1MB 内存空间。但 8086 的数据处理位宽位 16 位，无法直接寻址 1MB 内存空间，所以 8086 提供了段地址加偏移地址的地址转换机制。PC 机的寻址结构是 segment:offset，segment 和 offset 都是 16 位的寄存器，最大值是 0ffffh，换算成物理地址的计算方法是把 segment 左移 4 位，再加上 offset，所以 segment:offset 所能表达的寻址空间最大应为 0ffff0h + 0ffffh = 10ffefh（前面的 0ffffh 是 segment=0ffffh 并向左移动 4 位的结果，后面的 0ffffh 是可能的最大 offset），这个计算出的 10ffefh 是多大呢？大约是 1088KB，就是说，segment:offset 的地址表示能力，超过了 20 位地址线的物理寻址能力。所以当寻址到超过 1MB 的内存时，会发生“回卷”（不会发生异常）。但下一代的基于 Intel 80286 CPU 的 PC AT 计算机系统提供了 24 根地址线，这样 CPU 的寻址范围变为 2^24=16M,同时也提供了保护模式，可以访问到 1MB 以上的内存了，此时如果遇到“寻址超过 1MB”的情况，系统不会再“回卷”了，这就造成了向下不兼容。为了保持完全的向下兼容性，IBM 决定在 PC AT 计算机系统上加个硬件逻辑，来模仿以上的回绕特征，于是出现了 A20 Gate。他们的方法就是把 A20 地址线控制和键盘控制器的一个输出进行 AND 操作，这样来控制 A20 地址线的打开（使能）和关闭（屏蔽\禁止）。一开始时 A20 地址线控制是被屏蔽的（总为 0），直到系统软件通过一定的 IO 操作去打开它（参看 bootasm.S）。很显然，在实模式下要访问高端内存区，这个开关必须打开，在保护模式下，由于使用 32 位地址线，如果 A20 恒等于 0，那么系统只能访问奇数兆的内存，即只能访问 0--1M、2-3M、4-5M.....，这显然是不行的，所以在保护模式下，这个开关也必须打开。

当 A20 地址线控制禁止时，则程序就像在 8086 中运行，1MB 以上的地是不可访问的。在保护模式下 A20 地址线控制是要打开的。为了使能所有地址位的寻址能力,必须向键盘控制器 8042 发送一个命令。键盘控制器 8042 将会将它的的某个输出引脚的输出置高电平，作为 A20 地址线控制的输入。一旦设置成功之后，内存将不会再被绕回(memory wrapping)，这样我们就可以寻址整个 286 的 16M 内存，或者是寻址 80386 级别机器的所有 4G 内存了。

键盘控制器 8042 的逻辑结构图如下所示。从软件的角度来看，如何控制 8042 呢？早期的 PC 机，控制键盘有一个单独的单片机 8042，现如今这个芯片已经给集成到了其它大片子中，但其功能和使用方法还是一样，当 PC 机刚刚出现 A20 Gate 的时候，估计为节省硬件设计成本，工程师使用这个 8042 键盘控制器来控制 A20 Gate，但 A20 Gate 与键盘管理没有一点关系。下面先从软件的角度简单介绍一下 8042 这个芯片。

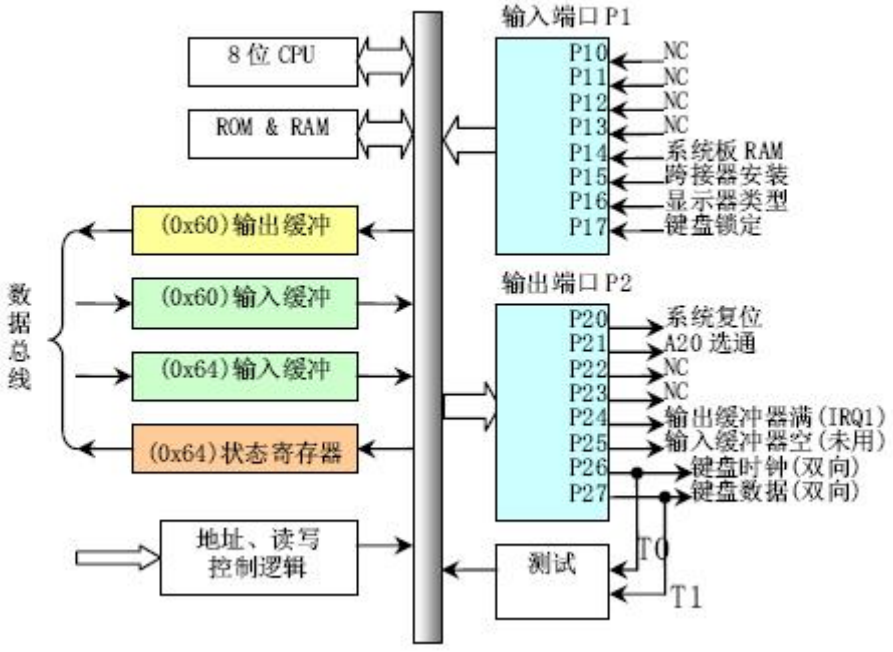

图 键盘控制器 8042 的逻辑结构图

8042 键盘控制器的 IO 端口是 0x60～0x6f，实际上 IBM PC/AT 使用的只有 0x60 和 0x64 两个端口（0x61、0x62 和 0x63 用于与 XT 兼容目的）。8042 通过这些端口给键盘控制器或键盘发送命令或读取状态。输出端口 P2 用于特定目的。位 0（P20 引脚）用于实现 CPU 复位操作，位 1（P21 引脚）用户控制 A20 信号线的开启与否。系统向输入缓冲（端口 0x64）写入一个字节，即发送一个键盘控制器命令。可以带一个参数。参数是通过 0x60 端口发送的。 命令的返回值也从端口 0x60 去读。8042 有 4 个寄存器：

- 1 个 8-bit 长的 Input buffer；Write-Only；
- 1 个 8-bit 长的 Output buffer； Read-Only；
- 1 个 8-bit 长的 Status Register；Read-Only；
- 1 个 8-bit 长的 Control Register；Read/Write。

有两个端口地址：60h 和 64h，有关对它们的读写操作描述如下：

- 读 60h 端口，读 output buffer
- 写 60h 端口，写 input buffer
- 读 64h 端口，读 Status Register
- 操作 Control Register，首先要向 64h 端口写一个命令（20h 为读命令，60h 为写命令），然后根据命令从 60h 端口读出 Control Register 的数据或者向 60h 端口写入 Control Register 的数据（64h 端口还可以接受许多其它的命令）。

Status Register 的定义（要用 bit 0 和 bit 1）：

bit meaning

---

0 output register (60h) 中有数据

1 input register (60h/64h) 有数据

2 系统标志（上电复位后被置为 0）

3 data in input register is command (1) or data (0)

4 1=keyboard enabled, 0=keyboard disabled (via switch)

5 1=transmit timeout (data transmit not complete)

6 1=receive timeout (data transmit not complete)

7 1=even parity rec'd, 0=odd parity rec'd (should be odd)

除了这些资源外，8042 还有 3 个内部端口：Input Port、Outport Port 和 Test Port，这三个端口的操作都是通过向 64h 发送命令，然后在 60h 进行读写的方式完成，其中本文要操作的 A20 Gate 被定

义在 Output Port 的 bit 1 上，所以有必要对 Outport Port 的操作及端口定义做一个说明。

- 读 Output Port：向 64h 发送 0d0h 命令，然后从 60h 读取 Output Port 的内容
- 写 Output Port：向 64h 发送 0d1h 命令，然后向 60h 写入 Output Port 的数据
- 禁止键盘操作命令：向 64h 发送 0adh
- 打开键盘操作命令：向 64h 发送 0aeh

有了这些命令和知识，就可以实现操作 A20 Gate 来从实模式切换到保护模式了。

理论上讲，我们只要操作 8042 芯片的输出端口（64h）的 bit 1，就可以控制 A20 Gate，但实际上，当你准备向 8042 的输入缓冲区里写数据时，可能里面还有其它数据没有处理，所以，我们要首先禁止键盘操作，同时等待数据缓冲区中没有数据以后，才能真正地去操作 8042 打开或者关闭 A20 Gate。打开 A20 Gate 的具体步骤大致如下（参考 bootasm.S）：

1.等待 8042 Input buffer 为空；

2.发送 Write 8042 Output Port （P2）命令到 8042 Input buffer；

3.等待 8042 Input buffer 为空；

4.将 8042 Output Port（P2）得到字节的第 2 位置 1，然后写入 8042 Input buffer；

附录“启动后第一条执行的指令”

【参考 IA-32 Intel Architecture Software Developer’s Manual Volume 3: System Programming Guide Section 9.1.4】

9.1.4 First Instruction Executed

The first instruction that is fetched and executed following a hardware reset is located at physical address FFFFFFF0H. This address is 16 bytes below the processor’s uppermost physical address. The EPROM containing the softwareinitialization code must be located at this address.

The address FFFFFFF0H is beyond the 1-MByte addressable range of the processor while in real-address mode. The processor is initialized to this starting address as follows. The CS register has two parts: the visible segment selector part and the hidden base address part. In real-address mode, the base address is normally formed by shifting the 16-bit segment selector value 4 bits to the left to produce a 20-bit base address. However, during a hardware reset, the segment selector in the CS register is loaded with F000H and the base address is loaded with FFFF0000H. The starting address is thus formed by adding the base address to the value in the EIP register (that is, FFFF0000 + FFF0H = FFFFFFF0H).

The first time the CS register is loaded with a new value after a hardware reset, the processor will follow the normal rule for address translation in real-address mode (that is, [CS base address = CS segment selector * 16]). To insure that the base address in the CS register remains unchanged until the EPROM based softwareinitialization code is completed, the code must not contain a far jump or far call or allow an interrupt to occur (which would cause the CS selector

value to be changed).