操作系统原理实验 （Lab 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：

# 计算机原理

# CPU的编址与寻址: 基于分段机制的内存管理

# CPU的中断机制

# 外设：串口/并口/CGA，时钟，硬盘

# Bootloader软件

# 编译运行bootloader的过程

# 调试bootloader的方法

# PC启动bootloader的过程

# ELF执行文件的格式和加载

# 外设访问：读硬盘，在CGA上显示字符串

# ucore OS软件

# 编译运行ucore OS的过程

# ucore OS的启动过程

# 调试ucore OS的方法

# 函数调用关系：在汇编级了解函数调用栈的结构和处理过程

# 中断管理：与软件相关的中断处理

# 外设管理：时钟

# 实验内容：

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）

列出本实验各练习中对应的OS原理的知识点，并说明本实验中的实现部分如何对应和体现了原理中的基本概念和关键知识点。

在此练习中，大家需要通过静态分析代码来了解：

1. 操作系统镜像文件ucore.img是如何一步一步生成的？(**选做**：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，并请执行

$ 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”，可了解更详细的解释，以及如何单步调试和

查看BIOS代码。

提示：查看 labcodes\_answer/lab1\_result/tools/lab1init 文件，用如下命令试试如何调试

bootloader第一条指令：

$ cd labcodes\_answer/lab1\_result/

$ make lab1-mon

labcodes\_answer这个文件夹我们的虚拟机里面没有提供，具体可以在网络上搜索

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，听从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来完成这项工作。这样qemu便不会在gdb尚未连接的时候擅自运行了。

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，需要了解：

为何开启A20，以及如何开启A20

如何初始化GDT表

如何使能和进入保护模式

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，

bootloader如何读取硬盘扇区的？

bootloader是如何加载ELF格式的OS？

提示：可阅读“硬盘访问概述”，“ELF执行文件格式概述”这两小节。

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，提交改进后源代码包（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. 中断描述符表（也可简称为保护模式下的中断向量表）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。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500次时钟中断后，调用print\_ticks子程序，向屏幕上打印一行文字”500 ticks”。

【注意】除了系统调用中断(T\_SYSCALL)使用陷阱门描述符且权限为用户态权限以外，

其它中断均使用特权级(DPL)为０的中断门描述符，权限为内核态权限；而ucore的应用

程序处于特权级３，需要采用｀int 0x80`指令操作（这种方式称为软中断，软件中断，

Tra中断，在lab5会碰到）来发出系统调用请求，并要能实现从特权级３到特权级０的转

换，所以系统调用中断(T\_SYSCALL)所对应的中断门描述符中的特权级（DPL）需要设

置为３。

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包（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5秒会输出一次”500 ticks”，而按下的键也会在屏幕上显示。

提示：可阅读小节“中断与异常”。

作业提交要求：

1. 提交至<https://easyhpc.net/course/110>
2. 提交报告和修改后的kernel源代码压缩文件 .zip，源代码要求编译后在能在虚拟机上运行
3. 截止日期为2021年4月2日