# 实验一、最小的操作系统

很多编程书籍给出的第一个例子往往是在终端里输出一个字符串“Hello world!”,那么要写操作系

统的第一步给出的例子自然就是如何在屏幕上打印出一个字符串喽。所以，我们首先看《自己动手写操

作系统》一书中给出的第一个示例代码，在屏幕上打印“Hello OS world!”

## 实验目的：

*通过编写最小的操作系统，学习掌握make和Makefile的运作流程，以及操作系统的启动机制。*

## 知识准备:

1. **整体介绍操作系统的运行原理**
   1. 操作系统简单运行原理，内核中重要的服务功能简要介绍（对比或者举例说明）；
   2. 硬件准备；
   3. 配置编译环境和调试环境，熟悉使用语言；
   4. 编译链接调试得到操作系统；
2. **QEMU介绍：**
   1. 有两种主要运作模式：
      1. User mode模拟模式，亦即是使用者模式。QEMU能启动那些为不同中央处理器编译的Linux程序。而Wine及Dosemu是其主要目标。
      2. System mode模拟模式，亦即是系统模式。QEMU能模拟整个电脑系统，包括中央处理器及其他周边设备。它使得为跨平台编写的程序进行测试及排错工作变得容易。其亦能用来在一部主机上虚拟多部不同虚拟电脑。
   2. QEMU使用实例:
      1. 以下的指令可以建立一个500MB大小的"qcow"格式的硬盘映像。

qemu-img create -f qcowc.img 500M

* + 1. 而以下的指令会使虚拟机器在128MB内存，使用c.img这个用以上介绍的指令所建立的硬盘映像及以linux.iso为光盘映像的情况下运行。注意，如果使用物理光驱，请将-cdrom linux.iso这个参数变为 -cdrom /dev/cdrom或您所用的光驱。

qemu -clock dynticks -rtc-td-hack -localtime -hda c.img -cdrom linux.iso -boot d -m 128 -enable-audio –localtime

1. **Makefile与make介绍：（例子）**

举例来说明Makefile的结构和make如何运作。

editor: main.otext.o

gcc -o editor main.o text.o

main.o: main.c def.h

gcc -c main.c

text.o: text.c com.h

gcc -c text.c

install:editor

mv editor /usr/local

当我们输入：

make

或者

make editor

当editor这个target文件不存在，或者main.o、text.o这两个依赖文件被修改，都会导致make调用其下的命令“gcc -o editor main.o text.o”；接下来，由于引用到main.o和text.o，make会检查main.o的依赖main.c、def.h有无更新，如果有，则执行其下的命令“gcc -c main.c”；同样的道理，也适用于text.o。于是，可有几种不同的输出：

* 1. 第一次运行：

gcc -c main.c

gcc -c text.c

gcc -o editor main.o text.o

* 1. main.c或/和def.h有修改：

gcc -c main.c

gcc -o editor main.o text.o

* 1. text.c或/和com.h有修改：

gcc -c text.c

gcc -o editor main.o text.o

* 1. main.c和text.c均有修改：

gcc -c main.c

gcc -c text.c

gcc -o editor main.o text.o

* 1. 当我们输入：

make install

make会检查install的依赖editor是否是最新，如果是，则执行其下的命令“mv editor /usr/local”。由于这个过程并没有产生名为“install”的文件，所以，install是一个假目标。

1. **汇编相关知识：**
   1. 查阅显卡或内存地址介绍；
   2. 汇编指令简单回顾复习；

## 实例代码:

/\*

Author:**Chen Peng <chenpeng@blcu.edu.cn>**

Bootasm -old.s

\*/

.code16 #16位模式

cli #关闭中断

cld #关字符串操作自动增加

# 设置重要数据段寄存器

xorw %ax, %ax #将ax清零

movw %ax, %ds #初始化数据段寄存器

movw %ax, %es #初始化附加段寄存器

movw %ax, %ss #初始化堆栈段寄存器

# 通过向显存中写字节流在屏幕中打印"hello world"

movw $0xb800, %ax

//0xb800: 为了性能更好，显示函数直接向显示内存区中写数据。在VGA显示器中，显示内存从绝对地址0x000B8000(或用段、偏移量表示则为B800:0000)

movw %ax, %es

movw $msg1, %si

movw $0xc82, %di //0xc82是？

movw $8, %cx //8是？

repmovsb

movw $str, %si

movw $0xc94, %di

movw $26, %cx

repmovsb

msg1:

.byte 'r', 0xc,'e',0xc,'l',0xc

str:

.byte ':',0xc,' ',0xc,'h',0xc,'e',0xc,'l',0xc,'l',0xc,'o',0xc,',',0xc,'w',0xc,'o',0xc,'r',0xc,'l',0xc,'d',0xc

.org 510 #填充到"510"字节处

.word 0xaa55 #结束标志

## 备注：

1. **AT&T汇编语法**
   1. 1、前缀

在AT&T中，寄存器前冠以“％”，而立即数前冠以“$”。

* 1. 2、操作数的方向

在AT&T中，第一个数是源操作数，第二个数是目的操作数。

* 1. 操作码的后缀

在AT&T的操作码后面有一个后缀，其含义就是指出操作码的大小。“l”表示长整数（32位），“w”表示字（16位），“b”表示字节（8位）。

1. **Intel 80386寄存器**

80386的寄存器可以分为8组：通用寄存器，段寄存器，指令指针寄存器，标志寄存器，系统地址寄存器，控制寄存器，调试寄存器，测试寄存器，它们的宽度都是32位。一般程序员看到的寄存器包括通用寄存器，段寄存器，指令指针寄存器，标志寄存器。

1. General Register(通用寄存器)：EAX/EBX/ECX/EDX/ESI/EDI/ESP/EBP这些寄存器的低16位就是8086的 AX/BX/CX/DX/SI/DI/SP/BP，对于AX,BX,CX,DX这四个寄存器来讲,可以单独存取它们的高8位和低8位 (AH,AL,BH,BL,CH,CL,DH,DL)。它们的含义如下:

EAX：累加器

EBX：基址寄存器

ECX：计数器

EDX：数据寄存器

ESI：源地址指针寄存器

EDI：目的地址指针寄存器

EBP：基址指针寄存器

ESP：堆栈指针寄存器

2. Segment Register(段寄存器，也称 Segment Selector，段选择符，段选择子)：除了8086的4个段外(CS,DS,ES,SS)，80386还增加了两个段FS，GS,这些段寄存器都是16位的，它们的含义如下：

CS：代码段(Code Segment)

DS：数据段(Data Segment)

ES：附加数据段(Extra Segment)

SS：堆栈段(Stack Segment)

FS：附加段

GS 附加段

3. Instruction Pointer(指令指针寄存器)：EIP的低16位就是8086的IP，它存储的是下一条要执行指令的内存地址，在分段地址转换中，表示指令的段内偏移地址。

4. Flag Register(标志寄存器)：EFLAGS,和8086的16位标志寄存器相比，增加了4个控制位，这20位控制/标志位的位置如下图所示：

相关的控制/标志位含义是：

CF(Carry Flag)：进位标志位；

PF(Parity Flag)：奇偶标志位；

AF(Assistant Flag)：辅助进位标志位；

ZF(Zero Flag)：零标志位；

SF(Singal Flag)：符号标志位；

IF(Interrupt Flag)：中断允许标志位,由CLI，STI两条指令来控制；设置 IF 使CPU可识别外部（可屏蔽）中断请求。复位 IF 则禁止中断。 IF 对不可屏蔽外部中断和故障中断的识别没有任何作用。

DF(Direction Flag)：向量标志位，由CLD，STD两条指令来控制；

OF(Overflow Flag)：溢出标志位；

IOPL(I/O Privilege Level)：I/O特权级字段，它的宽度为2位,它指定了I/O指令的特权级。如果当前的特权级别在数值上小于或等于IOPL，那么I/O指令可执行。否则，将发生一个保护性故障中断。

NT(Nested Task)：控制中断返回指令IRET，它宽度为1位。若NT=0，则用堆栈中保存的值恢复EFLAGS，CS和EIP从而实现中断返回；若NT=1，则通过任务切换实现中断返回。