NASM与GCC联合编译的批处理内核

学院名称：	数据科学与计算机学院
专业（班级）：	18计科教学3班
学生姓名:	夏一溥
学号:	18340178
时间:	2020 年 5 月 15 日
实验三：	NASM与GCC联合编译的批处理内核

实验内容

- [x] 使用一套汇编与C的编译器组合，编译样板c程序，获得符号列表文档并分析。
- [x] 实现一个c语言与asm汇编联合编程实例，完成对汇编模块中字符串中字符出现次数的统计并用汇编模块显示统计结果。
- [x] 重写实验二，将监控程序与引导程序分离独立。且在实验二的基础上完成功能的加强与完善。
- [x] 重写引导程序，使其能加载独立内核。

实验过程

Part 1： 样版c语言的反汇编与符号列表

我选择的样板c语言程序为要求二中自己独立完成的c语言记数部分，编译与反汇编生成表格指令如下：

```
nasm -f elf32 <input>.asm
g++ -o print -m32 <inputc>.cpp <inputasm>.o
objdump -t <output>
```

```
xyp@xypsysutt:~/Desktop/LAB/OS_LAB/obj$ nasm -f elf32 string.asm
xyp@xypsysutt:~/Desktop/LAB/OS_LAB/obj$ g++ -o print -m32 print.cpp string.o
xyp@xypsysutt:~/Desktop/LAB/OS_LAB/obj$ objdump -t print

print:     file format elf32-i386
```

所生成的符号列表文件与源代码在压缩包中可见。

Part 2 : NASM与GCC联合编译实例

实验环境:

- 操作系统： Ubuntu 18.04.4 LTS
- 汇编编译器： NASM
- c语言编译器： GCC

设计思路:

所要求实现的功能是记数在汇编代码中定义的某字符串的各字母出现次数。

由于高级语言更适合来处理比如记数，标记等逻辑操作，所以解决思路是将汇编中该字符串的地址传递给c程序代码，通过高级语言完成对字符串中字符的统计，再交给汇编输出打印。

为了能在Linux上兼容运行，我选择使用32位文件格式。于是寄存器会相应使用%eax等代替ax。

代码实现与说明：（完整代码附于压缩包中）

返回字符串地址：

```
my_str:
    mov eax,msg
    ret
```

由于函数调用的参数传递在本质上来说，其实就是参数压栈与出栈，于是我们想要得到对应的返回值，只需要将返回值赋予对应寄存器，在32位程序中表现位eax（16位中为ax）。同时[sp+4]对应于传入第一个参数。

打印字符串（结果输出）：

```
my_print:
    mov ecx,[esp+4]
    mov edx,4
    mov ebx,1
    mov eax,4
    int 80h
    ret
```

调用 int 80h中断，输出字符串。

运行结果：

汇编内字符串为"hello world"

```
xyp@xypsysutt:~/Desktop/LAB/OS_LAB/obj$ ./print
hello world
d:1
e:1
h:1
l:3
o:2
r:1
w:1
```

Part 3 : NASM与GCC 重写实验二

重写思路：

首先明确的是高级语言相比于汇编更适合处理字符串，所以可以考虑在高级语言中处理指令的输入，回显并提升程序鲁棒性，而汇编部分则负责装载与跳转这类底层的操作。

代码实现与说明：

引导程序BOOT：

引导程序基本功能与实验二没有区别，需要实现对用户程序表的加载和监控程序的加载：

```
read_secotr PTB,1,0,0,0,2        ;占1个扇区
read_secotr mon,16,0,0,0,3       ;占16个扇区
```

其中read_sector 使用宏实现对13H中断的使用：

```
%macro read_secotr 6  ;(offset,扇区号,驱动器号，磁头号，柱面号，起始扇区号)
    mov ax,cs
    mov es,ax
    mov bx,%1
    mov ah,2
    mov al,%2           ;扇区数
    mov dl,%3           ;驱动器号
    mov dh,%4           ;磁头号
    mov ch,%5           ;柱面号
    mov cl,%6           ;起始扇区号
    int 13H
%endmacro
```

监控程序：

汇编部分：

这里与实验二有很大不同，监控程序（汇编部分）主要完成的工作仅仅只是加载由C语言完成的OS函数，但值得注意的是：

1. 在16位汇编中为了使编译器能够明确编译位数，需要使用伪指令

```
BITS 16
```

```
2. 在汇编语言联合编译中需要指明函数入口，使用_start ,且声明为全局态：
```

```
global _start
_start:
```

C语言部分：

为了能反复执行用户命令，我选择使用循环来反复识别用户指令，而指令识别则交予if等选择判断语句完成，整体没有构思难度。

用户程序表：

还是生成的静态用户程序表，用字符串形式存储，通过引导程序加载进内存。

汇编函数库（lib_asm.asm）：

其实就是实验二中内核的功能分割，实现了5个函数：

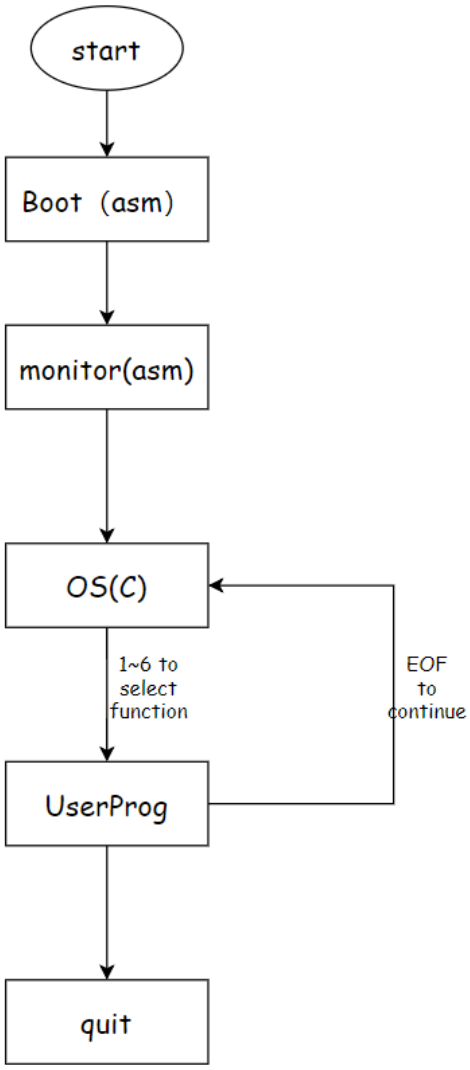
1. _CLS()： 清屏
2. _putchar(char)： 打印一个字符
3. _printLine(char *msg,short len,short X,int Y)： 指定位置打印一行字符串
4. _getchar()： 阻塞读一个字符
5. _execute(short sec,short addr)： 运行指定扇区与加载地址的用户程序

C语言函数库(lib_c.c)：

主要完成对字符串的操作，完成了以下函数功能：

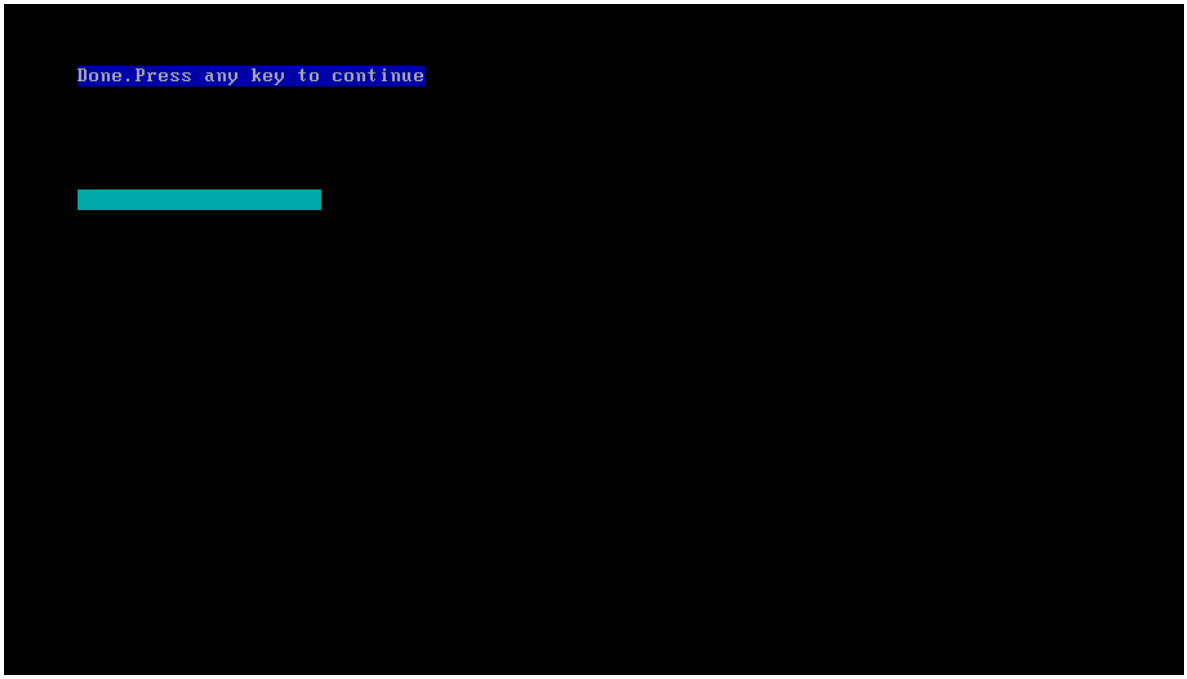
1. print() : 光标处打印一行字符串，遇到'\0'停止打印
2. help() : 打印帮助
3. OS() : 辅助实现监控程序

流程图：

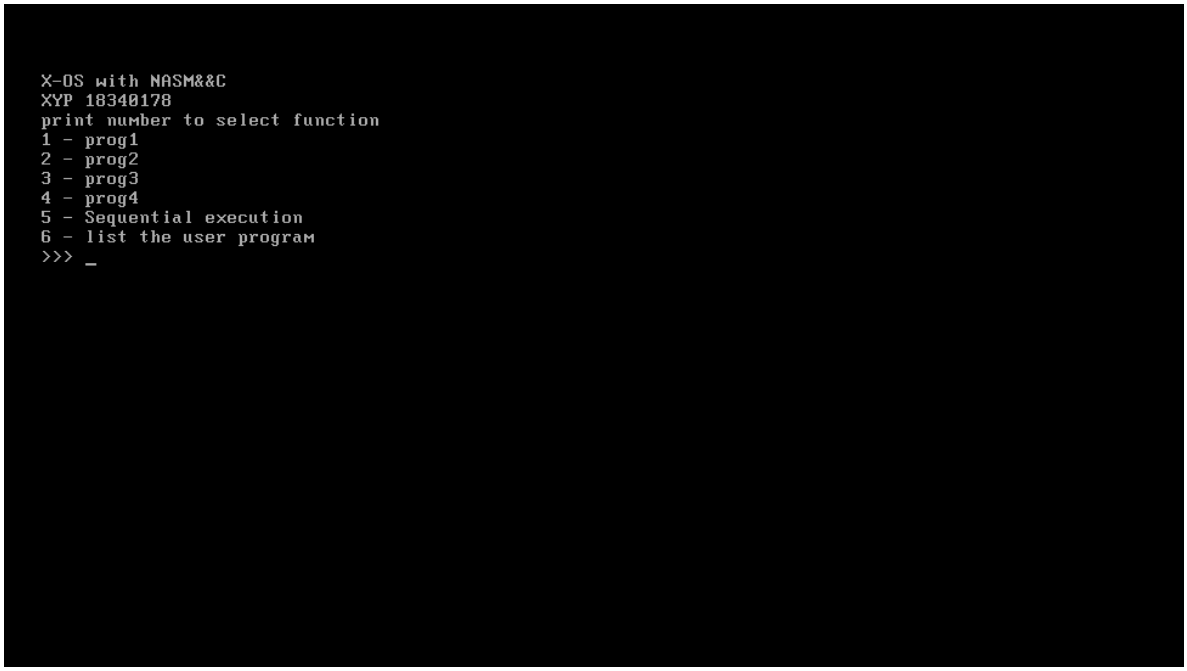


实验结果：

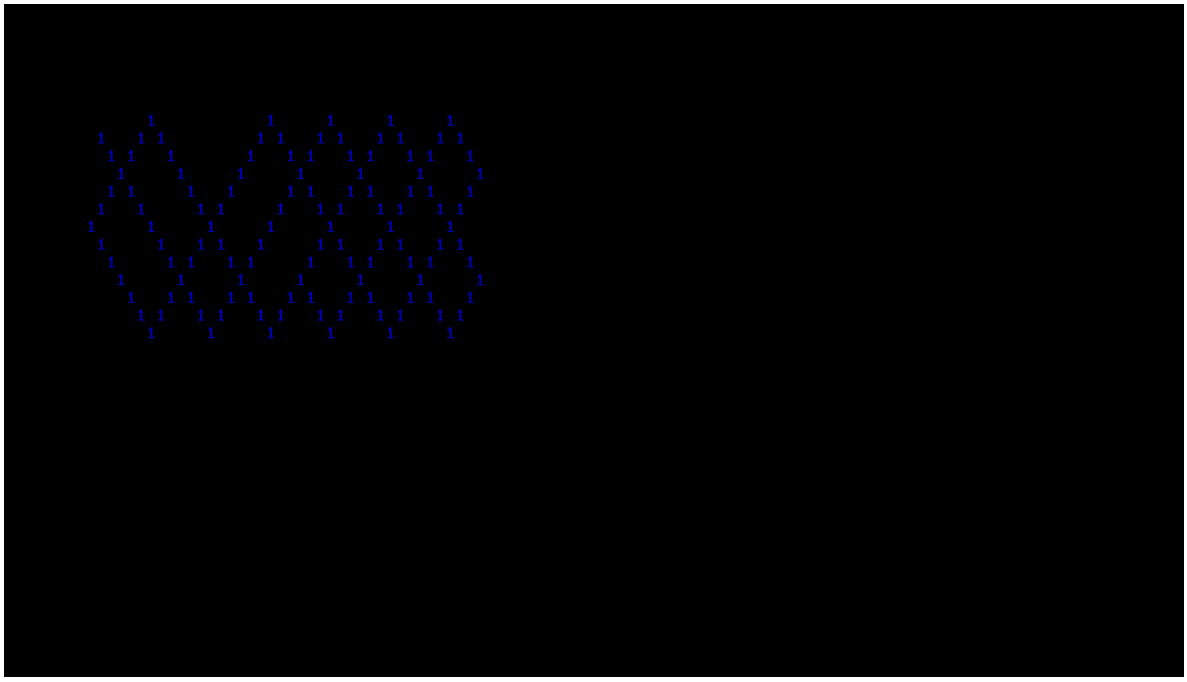
Boot，任意键继续：



监控程序：

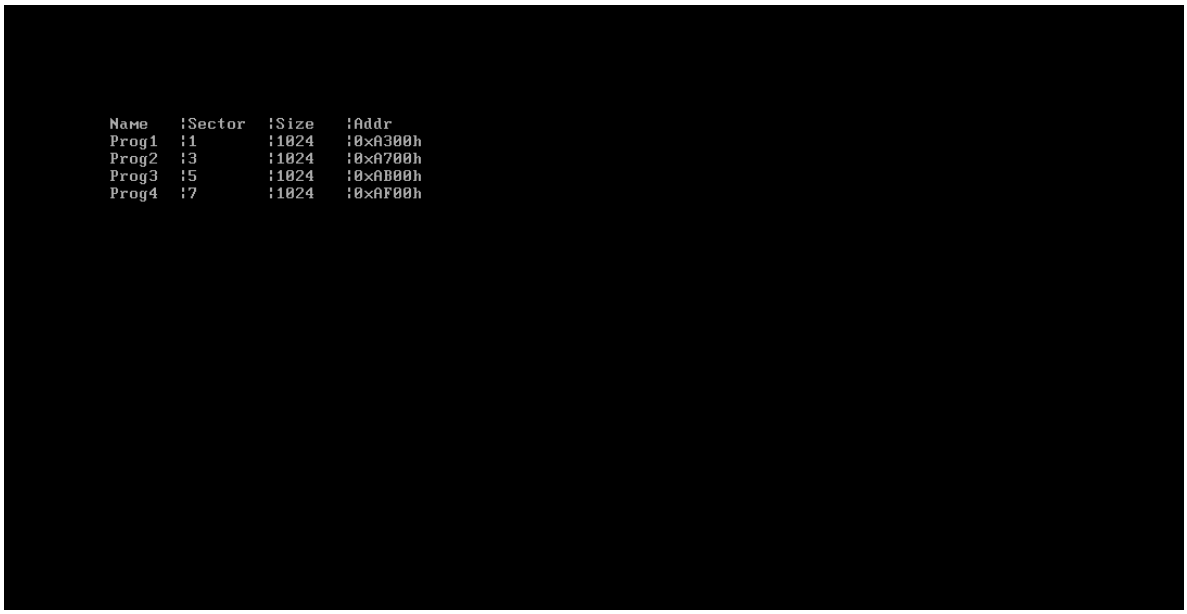


用户程序：EOF to continue （Ctrl + z）



1~5 类似，图略。

PTB：



纠错过程：

1. 尝试把c语言文件编译为16位程序时，提示不能找到 -lstdc++ 和 -lgcc

```
xyp@xypsysutt:~/Desktop/LAB/OS_LAB/obj$ g++ -o print -m16 print.cpp string.o
/usr/bin/ld: skipping incompatible /usr/lib/gcc/x86_64-linux-gnu/7/libstdc++.so
when searching for -lstdc++
/usr/bin/ld: skipping incompatible /usr/lib/gcc/x86_64-linux-gnu/7/libstdc++.a w
hen searching for -lstdc++
/usr/bin/ld: cannot find -lstdc++
/usr/bin/ld: skipping incompatible /usr/lib/gcc/x86_64-linux-gnu/7/../../../x86_
64-linux-gnu/libm.so when searching for -lm
/usr/bin/ld: skipping incompatible /usr/lib/gcc/x86_64-linux-gnu/7/../../../x86_
64-linux-gnu/libm.a when searching for -lm
/usr/bin/ld: skipping incompatible /usr/lib/x86_64-linux-gnu/libm.so when search
ing for -lm
/usr/bin/ld: skipping incompatible /usr/lib/x86_64-linux-gnu/libm.a when searchi
ng for -lm
/usr/bin/ld: skipping incompatible /usr/lib/gcc/x86_64-linux-gnu/7/libgcc_s.so.1
 when searching for libgcc_s.so.1
/usr/bin/ld: skipping incompatible /lib/x86_64-linux-gnu/libgcc_s.so.1 when sear
ching for libgcc_s.so.1
/usr/bin/ld: skipping incompatible /usr/lib/gcc/x86_64-linux-gnu/7/libgcc.a when
 searching for -lgcc
/usr/bin/ld: cannot find -lgcc
collect2: error: ld returned 1 exit status
```

其实是因为调用了库函数，而c语言或c++的标准库函数是不能兼容16位程序的，同时在编译操作系统内核的时候，不适用标准库可以添加如下参数：

```
-ffreestanding -nostdlib
```

从而取消链接时候的依赖关系。

2. 链接时提示undefined reference to `*GLOBAL_OFFSET_TABLE*'

找不到截图了，错误信息的内容如下：

```
print.c:(.text+0x16): undefined reference to `_GLOBAL_OFFSET_TABLE_'
xyp@xypsysutt:~/Desktop/LAB/OS_LAB/LAB3$ ld -m elf_i386 -N -Ttext 0x8000 --
oformat binary print.o string.o -o p -fno-pie
ld: -f may not be used without -shared
```

产生原因是由于GCC的一个古老特性，不知道是不是BUG，应该在编译命令末尾加上如下参数：

```
 -fno-pie
```

3. 文本输出乱码，格式不正确，跳转不正确：

其实是因为链接需要指明代码段地址。链接时添加如下参数：

```
-Ttext 对应代码段地址
```

4. 从汇编返回C程序的时候，在IP和CS正确的前提下，retf指令不能实现返回正确地址：

是因为C语言的callret为32为，和汇编中的callret不同步。所以为了返回正确地址，有两种手段：

1. 手动压栈：

```
call dword cs_ip    ;必须手动入栈操作
    cs_ip:              ;否则如使用call会因为c语言callret为32位导致不能返回
    mov si, sp
    mov word[si], ip_new
```

2. 压栈 0：

```
push 0
call addr
```

5. 加载monitor部分的时候只加载了部分：

发现是由于int 13H 按扇区加载，需要指明磁头（18扇区），同时需要将起始地址（monitor）变更为512B（1扇区）的整数倍即可。

实验总结

这次实验最大的难点是c语言与汇编语言联合编译的细节非常多，遇到了很多问题。

1. 当我们使用NASM和GCC的时候需要注意编译参数，ld的链接参数：

```
gcc -c -m16 -march=i386 -masm=intel -nostdlib -ffreestanding -mpreferred-stack-boundary=2 -lgcc -shared input.c -o output.o -fno-pie
```

```
ld -m elf_i386 -N -Ttext [addr] --oformat binary o1.o o2.o o3.o -o output.bin
```

```
2.  汇编与c语言调用返回的区别：
```

c语言ret 与call 是弹出32位，在互相调用的时候应该对汇编部分进行处理如push 0等（具体在代码实现部分有说明），否则会导致ip越界等问题而访问到无效内存地址。

3. 联合编译的优势：

借助与高级语言联合编译，汇编语言可以在接近硬件的操作的同时处理好字符串，逻辑判断等较为繁琐的操作。

Reference

1. gcc命令objdump用法----反汇编 https://blog.csdn.net/cwcwj3069/article/details/8273129
2. 使用BIOS中断显示字符串笔记(int 10h 13号中断) https://blog.csdn.net/pdcxs007/article/details/43378229
3. 《x86汇编语言:从实模式到保护模式》

Appendix

文件附录：

/bin	编译生成的二进制文件
/img	X-OS镜像
/userprog	4个用户程序
lib_asm.asm	汇编库
lib_c.c	c库
boot.asm	引导程序
PTB.img	用户程序信息
monitor.asm	监控程序汇编部分
/Part1	反汇编符号表
/Part2	字母记数程序