实验三：C 与汇编开发独立批处理的内核

实验目的:

1、加深理解操作系统内核概念

2、了解操作系统开发方法

3、掌握汇编语言与高级语言混合编程的方法

4、掌握独立内核的设计与加载方法

5、加强磁盘空间管理工作

实验要求:

1、知道独立内核设计的需求

2、掌握一种 x86 汇编语言与一种 C 高级语言混合编程的规定和要求

3、设计一个程序，以汇编程序为主入口模块，调用一个 C 语言编写的函数处理汇编模块定义的数据，然后再由汇编模块完成屏幕输出数据，将程序生成 COM 格式程序，在 DOS 或虚拟环境运行。

4、汇编语言与高级语言混合编程的方法，重写和扩展实验二的的监控程序，从引导程序分离独立，生成一个 COM 格式程序的独立内核。

5、再设计新的引导程序，实现独立内核的加载引导，确保内核功能不比实验二的监控程序弱，展示原有功能或加强功能可以工作。

6、编写实验报告，描述实验工作的过程和必要的细节，如截屏或录屏，以证实实验工作的真实性

实验内容:

(1) 寻找或认识一套匹配的汇编与 c 编译器组合。利用 c 编译器，将一个样板 C 程序进行编译，获得符号列表文档，分析全局变量、局部

变量、变量初始化、函数调用、参数传递情况，确定一种匹配的汇编语言工具，在实验报告中描述这些工作。

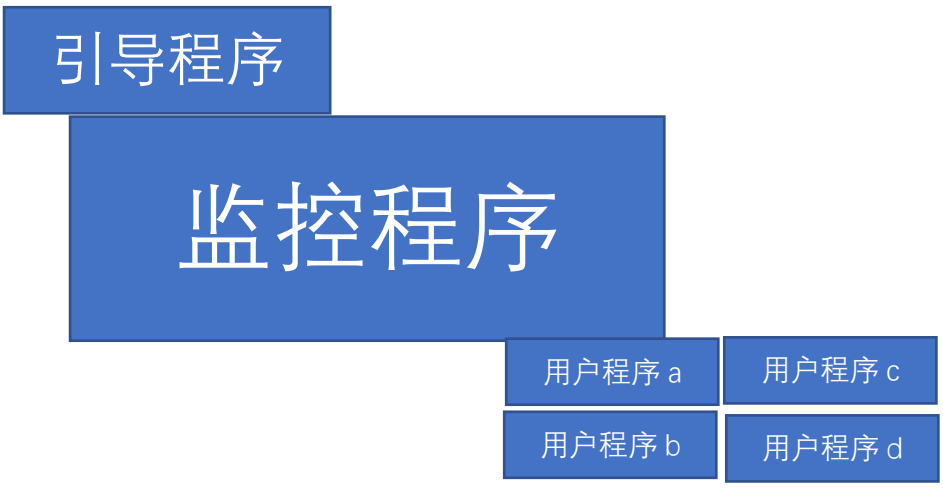
(2)写一个汇编程和 c 程序混合编程实例，展示你所用的这套组合环境的使用。汇编模块中定义一个字符串，调用 C 语言的函数，统计其中某个字符出现的次数（函数返回），汇编模块显示统计结果。执行程序可以在 DOS 中运行。

(3) 重写实验二程序，实验二的的监控程序从引导程序分离独立，生成一个 COM 格式程序的独立内核，在 1.44MB 软盘映像中，保存到特定的几个扇区。利用汇编程和 c 程序混合编程监控程序命令保留原有程序功能，如可以按操作选择，执行一个或几个用户程序、加载用户程序和返回监控程序；执行完一个用户程序后，可以执行下一个。

(4) 利用汇编程和 c 程序混合编程的优势，多用 c 语言扩展监控程序命令处理能力。

(5) 重写引导程序，加载 COM 格式程序的独立内核。

(6)拓展自己的软件项目管理目录，管理实验项目相关文档



实验环境：

- Windows 10-64bit
- Vmware WorkStation 15 pro 15.5.1 build-15018445：虚拟机软件
- NASM version 2.13.02：汇编程序的编译器，在 linux 下通过 sudo apt-get install nasm 下载
- Ubuntu-18.04.4:安装在 Vmware 的虚拟机上
- 代码编辑器：Visual Studio Code 1.44.2
- GNU ld 2.30：链接器

实验思路：

首先，本次实验我没有使用 tcc+tasm+tlink 的方案，而是采用的是 gcc+nasm+ld 的方案。

前提：使用 gcc+nasm 将 C 语言和汇编语言进行混合编译，运行实模式下的操作系统内核并不是不可以。只是需要添加较多的参数。

为了生成兼容 16 位的代码，我们需要在所有的 asm 文件的头部上加上下面的说明符，否则会因为不兼容而出错。

```
BITS 16
```

要实现 C 和汇编的混合编译，首先我们需要了解汇编和 C 互相调用是如何实现的，下面给出模板：

汇编调用 C：

```
extern function_name                                ;汇编程序头部声明
```

C 调用汇编：

```
global function_name                                //汇编程序头部声明
extern void function_name()                         //C 程序头部声明
```

参数的传递：

```
uint16_t function_name(uint16_t x,uint16_t y)  ;C 函数头部声明
```

通过以上的模板，我们便可以实现不同的函数在汇编和 C 之间的互相调用了。

程序的编写：

准备好以上的工作之后，我们便可以编写我们要用到的各个程序了。

引导程序（bootloader.asm）：

引导程序主要完成三项工作：1、显示提示信息；2、加载用户程序的信息表到内存中；3、加载并且跳转到操作系统的内核。

该程序我已经在实验 2 的时候完成了，因此直接照搬过来即可。

用户程序信息表（userproginfo.asm）：

由名字就可以知道该程序实际上是一个表，存放着各种各样的数据，并且将会被引导程序（bootloader.asm）装入内存，并且由操作系统内核逐步读取数据。其核心代码如下：

```
%macro UserProgInfoBlock 7    ; 参数：(PID,程序名,字节数,柱面,磁头,扇区,内存地址)
    pid%1 db %1               ; 程序编号 PID；相对偏移 0
    name%1 db %2              ; 程序名（至多 32 字节）；相对偏移 1
    times 16-($-name%1) db 0 ; 程序名占 6 字节
    size%1 dw %3              ; 程序大小；相对偏移 17
    cylinder%1 db %4          ; 柱面；相对偏移 19
    head%1 db %5              ; 磁头；相对偏移 20
    sector%1 db %6            ; 扇区；相对偏移 21
    addr%1 dw %7              ; 内存中的地址；相对偏移 22
%endmacro                     ; 共 24 个字节
```

定义了 7 个数据，包含程序编号、程序名等等。

```
UserProgInfo:
    UserProgInfoBlock 1, '     b', 1024, 0, 1, 1, offset_usrprog1
    UserProgInfoBlock 2, '     a', 1024, 0, 1, 3, offset_usrprog2
    UserProgInfoBlock 3, '     c', 1024, 0, 1, 5, offset_usrprog3
```

```
    UserProgInfoBlock 4, '     d', 1024, 0, 1, 7, offset_usrprog4
```

定义了 4 个用户程序（a、b、c、d），并为以后的将其加载至操作系统屏幕上做准备。

用户程序信息表和引导程序一样，都不参与和 C 的混合编译，故不需要编译再用 ld 连接。

操作系统内核：

操作系统内核包含了如下程序:

文件名	功能
oskernel.asm	监控程序，接收用户命令，执行相应的用户程序
myos.asm	包含 n 个汇编编写的函数
myos_c.c	包含 n 个 C 编写的函数
stringio.h	myos_c.c 的头文件，实现了输入输出等功能

oskernel.asm:

与实验 2 的代码相差不大，只是把具体的命令行给放置到 shell 里面了，监控程序只是起到调用 shell 的功能。

myos.asm:

定义实现了大量的函数，在此只介绍读取并运行用户程序的函数，其他函数具体看源代码:

```
loadAndRun:                 ; 函数：从软盘中读取扇区到内存并运行用户程序
    pusha
    mov bp, sp
    add bp, 16+4            ; 参数地址
    mov ax,cs               ; 段地址；存放数据的内存基地址
```

```
    mov es,ax               ; 设置段地址（不能直接 mov es,段地址）
    mov bx, [bp+16]         ; 偏移地址; 存放数据的内存偏移地址
    mov ah,2                ; 功能号
    mov al,[bp+12]          ; 扇区数
    mov dl,0                ; 驱动器号; 软盘为 0，硬盘和 U 盘为 80H
    mov dh,[bp+4]           ; 磁头号; 起始编号为 0
    mov ch,[bp]             ; 柱面号; 起始编号为 0
    mov cl,[bp+8]           ; 起始扇区号 ; 起始编号为 1
    int 13H                 ; 调用读磁盘 BIOS 的 13h 功能
    call dword pushCsIp     ; 手动压栈 CS、IP
    pushCsIp:
    mov si, sp              ; si 指向栈顶
    mov word[si], afterrun ; 修改栈中 IP 的值，这样用户程序返回后就可以继续执
行了
    jmp [bp+16]
    afterrun:
    popa
```

各个语句的功能已经在注释中表明，该函数主要就是读取扇区到内存并运行用户程序。

myos_c.c：

同样实现了大量的函数，其中最重要的就是 shell()函数（也就是监控程序不断调用的部分）。在本程序中，shell()实现了以下几个功能：

命令	功能
help	显示所有可执行的命令和相应功能
list	显示可以运行的用户程序（即上面提及的用户程序信息表的一部分）
run	运行程序的命令
clear	清屏操作

sh	执行已经初始化好的脚本（本程序中仅初始化好了 init.cmd）
poweroff	关机操作

其中最重要的，无疑是 run 命令，故在此对其进行详细阐述：

run 命令主要由 myos_c.c 中的 loadAndRun 函数进行实现，其定义如下：

```
extern void loadAndRun(uint8_t cylinder, uint8_t head, uint8_t sector,
uint16_t len, uint16_t addr);
```

由此可知，run 命令可接受多个参数，且在调用该 c 函数时，c 函数同时调用了 myos.asm 中的多个汇编函数，并将这些函数的返回值当作参数传到 loadAndRun 进行调用。其运用例子如下：

```
loadAndRun(getUsrProgCylinder(pid_to_run), getUsrProgHead(pid_to_run),
getUsrProgSector(pid_to_run), getUsrProgSize(pid_to_run)/512, getUsrPro
gAddr(pid_to_run));
```

同时，设计的 run 命令是可以进行有效批处理的，例子如下：

```
run 1 2 3 4
```

执行如上命令，操作系统会依次执行用户程序 1、2、3、4。其采用的方法是通过 while 循环进行实现的，具体的操作是：获取 run 后面的参数，通过循环对参数逐个遍历，遍历到有效参数（如上述命令的 1）时运行相应的用户程序。当通过 Esc 退出该用户程序时，循环会自动遍历到下一个参数（如上述命令的 2），并执行相应的用户程序，直到将所有的用户程序执行完毕。详情请看源代码。

（注意：run 后面的参数必须全部都是有效的，不能出现无效参数夹杂在有效参数之中的情况，如 run 1 a 2 3 这里面的 a 是无效参数，所以该命令也是无效的）。

stringio.h：

实现了 myos_c.c 所需要的功能，主要是对字符串进行了各种各样的操作，这不是操作系统实验的重点，在此便不在详细叙述了，详情可见源代码。

用户程序：

与实验 2 一样，本实验的四个用户程序和实验 2 基本相同，不同点在于多了压栈与出栈的指令，具体如下：

```
start:
    pusha                   ;压栈
    call ClearScreen        ; 清屏
    mov ax,cs
    mov es,ax               ; ES = CS
    mov ds,ax               ; DS = CS
    mov es,ax               ; ES = CS
    mov ax,0B800h
    mov gs,ax               ; GS = B800h，指向文本模式的显示缓冲区
    mov byte[char],'B'

    PRINT_IN_POS hint1, hint1len, 15, 1
QuitUsrProg:

    popa                    ;出栈
    retf
```

pusha 和 popa 分别将寄存器压栈出栈，用户程序执行完毕之后通过 retf 取得 ip 和 cs 的值，从而返回操作系统内核，进而回到 shell。

至此，所有需要完成程序都已经完成，下面对其进行混合编译链接。

混合编译链接：

同实验 2 一样，采用 shell script 在 Linux 下进行相同的操作，不同的是使用了 dd 命令将 asm 文件写入软盘当中，myos.sh 详细如下：

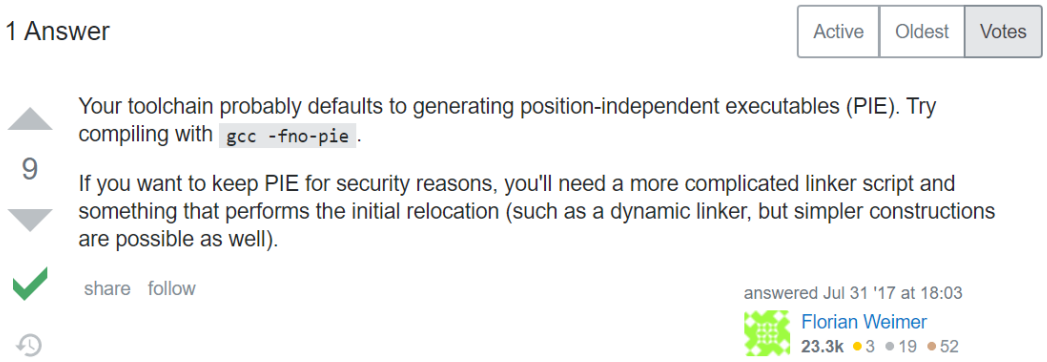
```
#!/bin/bash
rm -rf temp
```

```
mkdir temp
rm *.img

nasm bootloader.asm -o ./temp/bootloader.bin
nasm userproginfo.asm -o ./temp/userproginfo.bin

cd userprog
nasm b.asm -o ../temp/b.bin
nasm a.asm -o ../temp/a.bin
nasm c.asm -o ../temp/c.bin
nasm d.asm -o ../temp/d.bin
cd ..

nasm -f elf32 oskernel.asm -o ./temp/oskernel.o
nasm -f elf32 myos.asm -o ./temp/myos.o
gcc -c -m16 -march=i386 -masm=intel -nostdlib -ffreestanding -
mpreferred-stack-boundary=2 -lgcc -shared myos_c.c -o ./temp/myos_c.o
ld -m elf_i386 -N -Ttext 0x8000 --
oformat binary ./temp/oskernel.o ./temp/myos.o ./temp/myos_c.o -
o ./temp/kernel.bin
rm ./temp/*.o

dd if=./temp/bootloader.bin of=Condor_OS.img bs=512 count=1 2>/dev/null
dd if=./temp/userproginfo.bin of=Condor_OS.img bs=512 seek=1 count=1 2>
/dev/null
dd if=./temp/kernel.bin of=Condor_OS.img bs=512 seek=2 count=16 2>/dev/
null
dd if=./temp/b.bin of=Condor_OS.img bs=512 seek=18 count=2 2>/dev/null
dd if=./temp/a.bin of=Condor_OS.img bs=512 seek=20 count=2 2>/dev/null
dd if=./temp/c.bin of=Condor_OS.img bs=512 seek=22 count=2 2>/dev/null
dd if=./temp/d.bin of=Condor_OS.img bs=512 seek=24 count=2 2>/dev/null

echo "Finished."
```

不得不说, gcc+nasm+ld 需要的参数是真的多, 将其保存在 myos.sh, 放到对应的目录下，同实验 2 将其设置为可执行程序文件并运行，得到的结果如下:

```
condor@condor-virtual-machine:~/oslab/实验3/myos$ ./merge.sh
./temp/libc.o：在函数‘strlen’中：
libc.c:(.text+0x11)：对‘_GLOBAL_OFFSET_TABLE_’未定义的引用
./temp/libc.o：在函数‘strcmp’中：
libc.c:(.text+0x5c)：对‘_GLOBAL_OFFSET_TABLE_’未定义的引用
./temp/libc.o：在函数‘print’中：
libc.c:(.text+0xfc)：对‘_GLOBAL_OFFSET_TABLE_’未定义的引用
./temp/libc.o：在函数‘readToBuf’中：
libc.c:(.text+0x172)：对‘_GLOBAL_OFFSET_TABLE_’未定义的引用
./temp/libc.o：在函数‘itoa’中：
libc.c:(.text+0x316)：对‘_GLOBAL_OFFSET_TABLE_’未定义的引用
./temp/libc.o:libc.c:(.text+0x3b7)：跟着更多未定义的参考到 _GLOBAL_OFFSET_TABLE
_
[+] Done.
```

这个错误在群里已经有同学问了两次了，根据大佬的解答是需要在 gcc 后面再附带上参数-fno-pie 即可。StackOverflow 的解答如下：



具体的原因暂且不明，疑似是生成了与位置无关的信息，且在低版本的 gcc 貌似不会出现这种错误。

将参数加上去之后 gcc 的命令如下：

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

再次执行该文件，还是出错了：

```
In file included from /usr/include/features.h:448:0,
                 from /usr/include/x86_64-linux-gnu/bits/libc-header-start.h:33,
                 from /usr/include/string.h:26,
                 from libc.c:7:
/usr/include/x86_64-linux-gnu/gnu/stubs.h:7:11: fatal error: gnu/stubs-32.h: 没有那个文件或目录
 # include <gnu/stubs-32.h>
           ^~~~~~~~~~~~~~~~
compilation terminated.
```

搜索了一下出现该错误的原因，却貌似得到两种截然不同的结果：（详情见下图）

解决在linux下编译32程序出现"/usr/include/gnu/stubs.h:7:27: 致命错误：gnu/stubs-32.h：没有那个文件或目录问题"

centos64位编译32位代码，出现/usr/include/gnu/stubs.h:7:27: 致命错误：gnu/stubs-32.h：没有那个文件或目录，需要安装32位的glibc库文件。

安装32位glibc库文件命令：

sudo yum install glibc-devel.i686(安装C库文件)

sudo dnf install glibc-devel.i686(fedora命令)

安装32位glibc++库文件命令

sudo yum install libstdc++-devel.i686

sudo dnf install libstdc++-devel.i686（fedora命令）

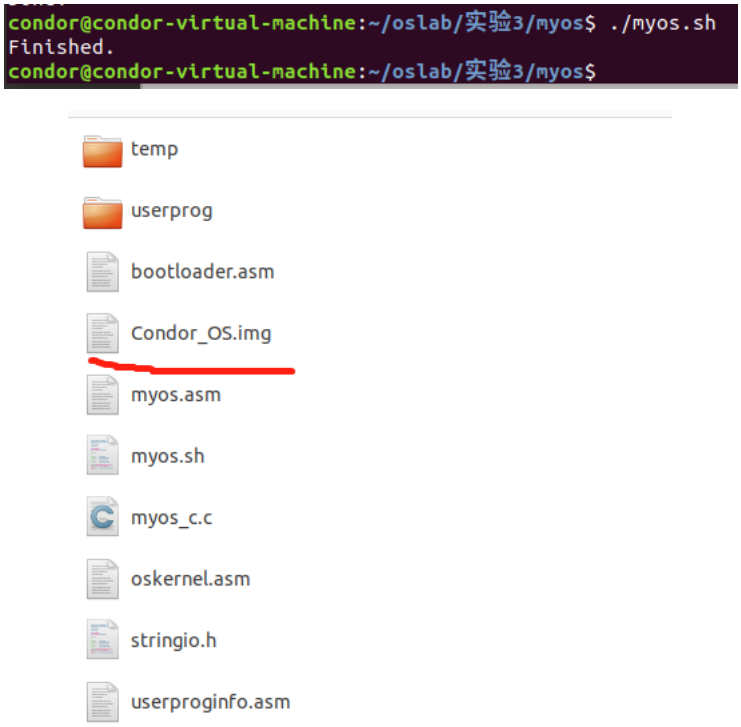
Ubuntu解决命令：

sudo apt-get install g++-multilib

```
4、/usr/include/features.h:324:26: fatal error: bits/predefs.h: No such file or directoryIn file
included from /home/hudan/android/source/android4.1.1/prebuilts/gcc/linux-x86/host/i686-linux-
glibc2.7-4.6/bin/../sysroot/usr/include/sys/types.h:27:0,
                 from frameworks/native/include/utils/Errors.h:20,
                 from frameworks/native/include/utils/String8.h:20,
                 from cts/suite/audio_quality/lib/include/Settings.h:21,
                 from cts/suite/audio_quality/lib/src/Settings.cpp:17:
/usr/include/features.h:324:26: fatal error: bits/predefs.h: No such file or directoryIn file
included from /usr/include/stdlib.h:25:0,
                 from cts/suite/audio_quality/lib/src/Adb.cpp:16:
/usr/include/features.h:324:26: fatal error: bits/predefs.h: No such file or directory

compilation terminated.
compilation terminated.

compilation terminated.
make: *** [out/host/linux-
x86/obj/STATIC_LIBRARIES/libcts_audio_quality_intermediates/src/SimpleScriptExec.o] Error 1
make: *** Waiting for unfinished jobs....
make: *** [out/host/linux-x86/obj/STATIC_LIBRARIES/libcts_audio_quality_intermediates/src/Adb.o]
Error 1
make: *** [out/host/linux-
x86/obj/STATIC_LIBRARIES/libcts_audio_quality_intermediates/src/Settings.o] Error 1
In file included from /usr/include/semaphore.h:22:0,
                 from cts/suite/audio_quality/lib/include/Semaphore.h:21,
                 from cts/suite/audio_quality/lib/src/Semaphore.cpp:17:
/usr/include/features.h:324:26: fatal error: bits/predefs.h: No such file or directory
compilation terminated.
make: *** [out/host/linux-
x86/obj/STATIC_LIBRARIES/libcts_audio_quality_intermediates/src/Semaphore.o] Error
解决方法：sudo apt-get install gcc-multilib
```

其中图一比较简洁，图二复杂到完全不想看，不过两者的方法都是sudo apt-get install gcc/g++-mutilib，由于不是很懂遂将疑惑上传到群聊咨询，得到了大佬的回复：我引用了库函数头文件，检查了

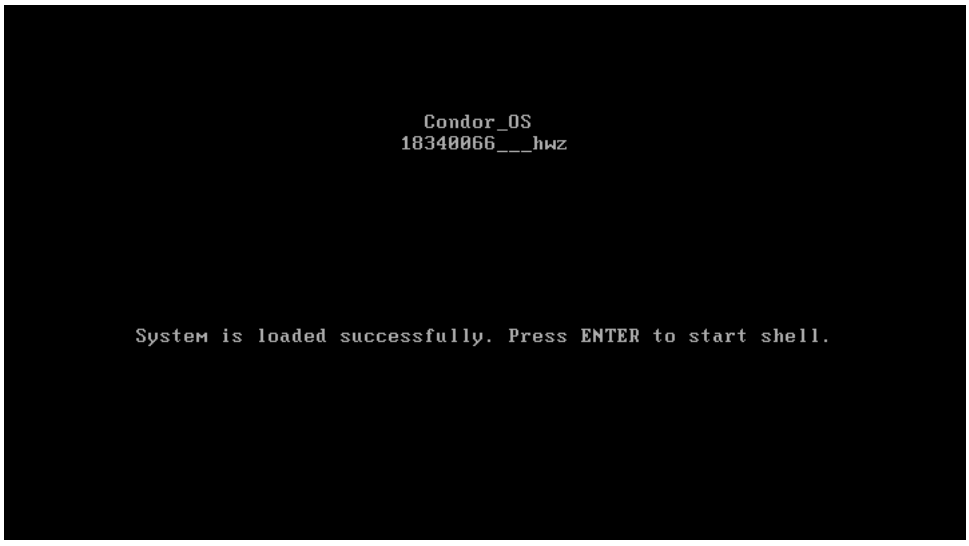
一下 myos_c.c 文件，发现确实如此……将多余的头文件删去（如 string.h），再次运行得到了正确的运行结果，如下图：



成功得到了我们想要的软盘镜像文件。

实验结果：

首界面：



进入 shell：

```
Shell for Condor_OS
This is a shell which is used for Condor_OS. These shell commands are defined in
ternally. Use 'help' to see the list.

    help - show information about builtin commands
    clear - clear the terminal screen
    list - show a list of user programmes and their PIDs
    run <PIDs> - run user programmes in sequence, e.g. 'run 1 2 3 4'
    sh <file> - run a script file, e.g. 'sh init.cmd'
    poweroff - force shutdown the machine
Condor_OS $ _
```

执行命令（如 run 3）：

执行命令（如 list）：

```
Shell for Condor_OS
This is a shell which is used for Condor_OS. These shell commands are defined in
ternally. Use 'help' to see the list.

    help - show information about builtin commands
    clear - clear the terminal screen
    list - show a list of user programmes and their PIDs
    run <PIDs> - run user programmes in sequence, e.g. 'run 1 2 3 4'
    sh <file> - run a script file, e.g. 'sh init.cmd'
    poweroff - force shutdown the machine
Condor_OS $ list
You can use 'run <PID>' to run a user programme.
PID  -  Name  -  Size  -  Addr  -  Cylinder  -  Head  -  Sector
1  -       b  -  1024  -  A300  -  0  -  1  -  1
2  -       a  -  1024  -  A700  -  0  -  1  -  3
3  -       c  -  1024  -  AB00  -  0  -  1  -  5
4  -       d  -  1024  -  AF00  -  0  -  1  -  7
Condor_OS $
```

实验心得：

本次实验的准备工作是真的很繁琐很复杂，从同学们在群里问的问题能看得出来很折磨人。老师提供了两个方案：tcc+tasm+tlink 的方案实在是有点古董了，而 gcc 的方案的编译参数又显得很繁杂，哪条路都不好走。本次采用了 gcc，我也只是仅仅停留在知道它能用的层次而不知道原理。而且说实话，老师提供的参考文件…有点一言难尽的感觉，最后算是勉强完成吧。同时也希望老师以后能指出一条更加明确的路让我们走，而不是让我们在伸手不见五指的环境下越过沼泽。

（最后感谢一下“csdn 等老师”给予的各种帮助。）