

操作系统实验报告 PA4

221900180 田永铭

2024 年 4 月 24 日

目录

一、 实验要求	3
二、 实验环境	3
三、 实验原理	4
四、 实验结果展示	4
五、 最终成功的实验过程	4
5.1 配置环境	5
5.2 打通内核、调用接口、app，简单实现 write	5
5.2.1 确定代码框架	5
5.2.2 编写 myos.h 和 main.c	6
5.2.3 编写 api	7
5.2.4 编写 kernel	7
5.2.5 编写 makefile	8
5.3 实现 sleep	10
5.3.1 sleep 时长的控制	10
5.3.2 进程的切换	11
5.4 写入磁盘，查看运行效果	12
5.5 进行一些优化	12
5.5.1 实现换行	12
5.5.2 优化 makefile 文件	13

六、 各种尝试、失败、debug 过程	13
6.1 makefile 失败！	13
6.2 链接失败！	13
6.3 关于打印换行的问题	13
6.4 关于打印字符的问题	14
6.5 时钟中断停止了！	14
6.6 打印到 16 个左右 hello 就出 bug！	15
6.7 中断使用 1ch 不行 8h 可以	15
6.8 其他尝试：利用 jmp 切换进程	15
6.9 其他问题	16
七、 总结与反思	17
八、 其它参考文献和致谢	17

一、实验要求

本次实验分为 OS 内核、API 库、示范应用三个部分，为综合性实验项目（建议合理构造代码目录结构，采用 makefile 组织和管理整个实验项目的编译、部署和运行）：

1. 采用 nasm 汇编语言实现一个实验性质的 OS 内核并部署到 MBR，主要功能如下：
 - 1) 实现 write 内核函数，向屏幕输出指定长度的字符串（尽量保证每次输出另起一行）；
 - 2) 实现 sleep 内核函数，按整型参数以秒为单位延迟（以 PA3 为基础实现，包括设置定时器和时钟中断处理例程）；
 - 3) 以系统调用的形式向外提供 write 和 sleep 两个系统服务（建议使用 int 80h，也可分别以不同的用户自定义中断实现不同的系统服务，提供系统调用中断处理例程）；
 - 4) 内核完成初始化后，从 1 号逻辑扇区开始加载示范性应用程序并运行（期间当应用进程 sleep 时，需切换至内核的无限循环处执行，当应用进程 sleep 结束时，需切换回应用进程执行）。
2. 采用 nasm 汇编与 C 语言混合编程，实现一个 API 库
 - 1) 提供一个包含 write 和 sleep 函数原型的 C 语言头文件（建议名称 myos.h）；
 - 2) 用 nasm 汇编语言实现对应的 API 库（封装 OS 内核提供的两个系统调用服务，注意在此处妥善处理示范应用 32 位指令与 OS 内核 16 位指令的对接!!!）
3. 采用 C 语言实现一个示范应用，并部署到磁盘
 - 1) 采用 C 语言编写示范应用，用于测试 OS 内核功能，示范应用大致内容如下图所示（仅供参考!!!）；
 - 2) 采用 GCC 编译生成 ELF32 格式代码，并用 objcopy 等工具提取出相应的代码段和数据段，最后装入 1 号逻辑扇区开始的连续磁盘空间（由示范应用的大小确定）。

提示：

 - 1) MBR 中的代码采用的是 i8086 体系架构上的 16 位指令（另需要注意代码尺寸，不能超越分区表!!!）
 - 2) 合理利用 BIOS 提供的基本输入输出功能：int10（屏幕输出），int 13（磁盘访问），int 16（键盘输入）

二、实验环境

- 操作系统：wsl2
- 编程语言：汇编语言 + C 语言
- 使用工具：makefile;qemu;objdump;dd;nasm 等

- 虚拟系统版本:windows-7;linux-20-desktop

三、实验原理

- **总体原理**: 在内核实现需要利用特权指令的功能, 将其整理成接口暴露给上层的应用程序, 应用程序通过调用接口, 实现对内核函数的使用。
- **加载应用程序的原理**: 在计算机启动时, 应用程序被加载并执行, 部分原理已经在前几次作业完全搞清楚了, 注意这次实验中要把应用程序加载到第一扇区。
- **write 和 sleep 的原理**: write 只需要调用 BIOS 功能 int 10 即可, 相对简单。而 sleep 执行时, 我的实现是: 内核里面有一个 idle 进程 (死循环), 这个进程具有栈地址。在调用 sleep 的时候, cpu 执行会先保存 app 下一条语句的栈地址, 再从 app 的执行流程中, 切换到 idle 进程执行, 相当于是一次调度和切换。当 sleep 结束的时候, 再从 idle 进程利用栈地址切换到 app 的进程中继续执行。
- **时钟中断原理**: 已经在 PA3 中阐述清楚。
- **makefile 的原理**: makefile 是一种用于构建和管理源代码的工具, 它主要用于自动化软件编译过程。其原理基于一种称为“依赖关系”的概念, 即文件之间的相互依赖关系。它通过这种依赖关系依次编译文件。

四、实验结果展示

注意: 我的 main 函数是“写一个 hello! + sleep 一秒 + 写一个 goodbye! + sleep 一秒”的循环。

- **视频展示**: [221900180_田永铭_实验4_视频演示.mp4](#)
- **助教本地运行检验**: 只需要解压代码压缩文件, 在 wsl2 环境下打开终端, 只需运行“make”。如果没有配置好 qemu 路径, 需要将 qemu 路径在代码对应位置填写完整。如果再运行不了, 仍可通过利用 wsl 来 make, 再利用 windows 上的 qemu 来运行 qemu 下各个指令。本人确保已经能正常运行成功。

五、最终成功的实验过程

注意: 此部分着重介绍最终成功的实验过程, 而尝试过程、失败过程和各种 debug 过程见后面部分。

实验主要分为以下几个步骤:

1. 配置环境

2. 打通内核、调用接口、app, 简单实现 write
3. 实现 sleep
4. 写入磁盘, 查看运行效果
5. 进行一些优化

5.1配置环境

由于 wsl2 环境下更利于本次实验 makefile 的使用, 能够更好的链接好可执行文件, 所以我首先配置了 windows-11 下的 wsl2 环境。

参考[wsl2 的安装教程](#), 首先安装好 wsl2。再参考[vscode 远程连接 wsl 教程](#), 配置 vscode, 添加 WSL 插件, 在 vscode 中连接 wsl 环境。

工作区配置完成后, 如下图所示, 可以在 vscode 中方便地编写代码。

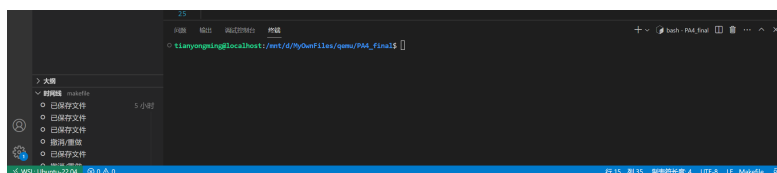


图 1: wsl2 工作区

5.2打通内核、调用接口、app, 简单实现 write

首先搭建起整套代码框架, 打通内核、调用接口、app, 简单实现 write, 这是比较关键的一步, 实现了这一步, 后面才调试得起来, 才有写好 sleep 的可能。

5.2.1 确定代码框架

我使用了和助教一样的代码框架, 框架如下 (link.ld 是我为了更好的链接而写的文件):

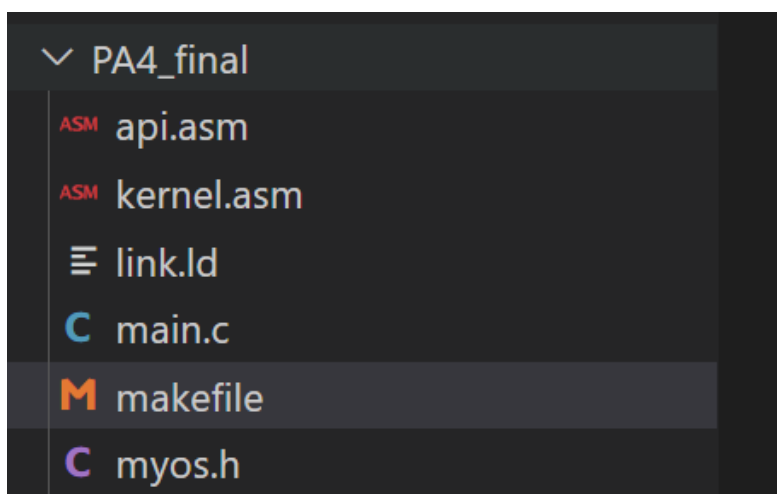


图 2: 代码框架

5.2.2 编写 myos.h 和 main.c

这部分最为简单，头文件定义函数的原型，上层应用文件就是简单的一直调用。针对我来说，我希望每次上层应用文件是“写一个 hello! + sleep 一秒 + 写一个 goodbye! + sleep 一秒”的循环。代码如下：

```
; myos.h
#ifndef MYOS_H
#define MYOS_H

void write(char* str, int len);
void sleep(int seconds);

#endif
```

```
; main.c
int main(void)
{
    while(1)
    {
        write("hello!", 6);
        sleep(1);
        write("goodbye!", 8);
    }
}
```

```

        sleep(1);
    }
}

```

5.2.3 编写 api

参考文献[汇编实现过程体的调用教程](#), 首先编写一个 api 暴露 write 的接口, 通过 esp 来访问参数, 同时注意保护栈以及 32 位下 ret 指令是 retq。代码如下:

```

BITS 16
section .text
global write
global sleep

write:
; 保护栈
push ebp
mov  ebp, esp
push ebx
push ecx

mov  ebx, [ebp+8] ; 地址
mov  ecx, [ebp+12] ; 长度
int  0x80        ; 调用中断

pop  ecx
pop  ebx
pop  ebp
retq             ; 回跳要用 32 位的指令, ret 是 16 位的

```

参数是通过 ebp 来获取的, 其中 write 有两个参数, 需要两个寄存器来存储。然后再调用 80 号中断, 来自动跳转到内核进行处理。

5.2.4 编写 kernel

仿照 PA3, 我们需要将 80 号用户自定义中断写入中断向量表, 代码如下:

```

mov word [0x80*4], write_handler

```

```
mov word [0x80*4+2],0
```

至于 write_handler 的编写，可以先完全仿照 PA3。仍有一部分关键的步骤是对 app 的加载，我们需要在内核函数运行结束之前，将其拉到第一扇区。根据老师所言，通过切换栈指针 iret 的方式来进行，我最终采用。代码如下：

```
; 加载应用程序
load_application:
xor ax,ax
mov es,ax
mov bx,0x8000; link.ld 中定义 text 在 0x8000
mov ah, 2
mov al, 1
mov ch, 0
mov cl, 2
mov dh, 0
mov dl, 0x80
int 0x13
mov sp,0x8000 ; 切换到应用栈
; 通过 iret 跳转应用进程执行
pushf
push word 0 ; cs
push word 0x8000;
iret
```

5.2.5 编写 makefile

我在助教发 makefile 样例之前完成了对这个 makefile 的编写，中途出了非常多的问题，这将在后一部分讨论，此处给出成功的实现和注释：

```
# 运行命令：make即可
# 反汇编指令：objdump -D -Mi8086, suffix app.elf
all:app.bin

app.bin:app.elf
objcopy -R.eh_frame \
-R \
```



```
.comment \  
-O \  
binary \  
app.elf app.bin  
make qemu  
  
# 为了让被执行的第一个函数是 main, 用 -e main  
app.elf:main.o api.o  
ld -m \  
elf_i386 \  
-T link.ld \  
-e main \  
main.o api.o \  
-o app.elf  
  
# 要用 -m16  
main.o:main.c myos.h  
gcc -m16 \  
-ffreestanding \  
-fno-pic \  
-I. \  
-c main.c \  
-o main.o  
  
api.o:api.asm  
nasm -f elf32 api.asm -o api.o  
  
qemu:  
qemu-img create -f raw PA4_disk.raw 5G  
nasm -f bin -o kernel.bin kernel.asm  
make dd  
  
dd:  
dd if=kernel.bin of=PA4_disk.raw bs=512 count=1  
dd if=app.bin of=PA4_disk.raw bs=512 seek=1 count=1  
# dd 命令中的 seek=1 用于将写入光标定位到磁盘映像文件 (PA4_disk.raw)
```

```
# 的第1个扇区，然后再将来自 app.bin 的数据写入。
# 实现从第二个扇区开始写入 app.bin 的数据，而不是覆盖现有数据。
make run

run:
qemu-system-x86_64 -drive file=PA4_disk.raw,format=raw
```

至此，我已经完成了简单的 **write** 的打通。在 wsl2 下运行 **make**，发现能够非常迅速地打印 **hello!**(此时我的应用程序还没写 **sleep**) 和 **goodbye!** 但是打印速度飞快，为了更好地看清，我先将 **main** 函数的 **while(1)** 改成了 **for** 循环语句，发现运行效果符合预期。

5.3 实现 **sleep**

首先基于 **write** 的实现简单地模仿，更新 **api** 文件、**main** 文件、**myos** 头文件，在 **kernel** 文件中先将 **PA3** 中的时钟中断复用，再模仿 **write** 定义好 **sleep** 的 81 号中断。此部分代码很简单，略去不展示，详见附件中的代码。难点在于 **sleep** 时长的控制和进程的切换，实现如下：

5.3.1 **sleep** 时长的控制

由于 **PA3** 中我的时钟中断是 20ms 一次，所以 50 次才是 1s，所以需要注意参数的归一化。由 **api** 传来的秒数我存放在了 **ax** 中，我通过一个 **loop** 循环和几个段地址的计数器实现了 **sleep** 时长的控制，代码如下：

```
clock_handler:
inc word [counter]; 增加一次 counter 的数值

mov dx, [num]
mov cx, [counter]
cmp cx, dx
; 到点就跳转回去
je back_to_main
iret

loop:      ; 归一化 1s
mov dx, [normalize]
add ax, dx
dec cx
```

```

cmp cx, 0
je continue
jmp loop

sleep_handler:
cli
mov cx, 50 ;20ms*50=1s, cx 存入 50
mov [normalize], ax ;ax 参数存入了 normalize
jmp loop

continue:
mov [num], ax; loop 后, 次数 ax 已经是 sleep 需要的时钟中断的个数
sti ; 一定要开中断, 否则时钟中断被屏蔽, 无法产生

```

5.3.2 进程的切换

我最终是按照老师的要求, **全部**利用 **iret** 进行切换。这有一个重要的理由: 实际上 OS 都是有保护模式的, 不能通过 **jmp** 在内核和应用程序间随意切换。

简单来讲, 当 sleep 开始时, 调用了 81 号中断, 操作系统从用户态陷入内核态, 此时在内核态通过 **iret** 方式切换到 idle 进程, 此后此 idle 进程会被时钟中断打断, 统计到 sleep 时间达到后, 需要切换栈指针, 模仿 **iret** 的标准实现, 从内核态安全地返回应用程序继续执行。核心代码如下:

```

; 此处是跳转 idle 的代码
; 保护现场 cx, dx
push cx
push dx
push bp
mov bp, sp; bp 此时就是回去的栈指针
mov sp, idle_stack ; 切换到 idle 栈
; 通过 iret 跳转 idle 进程执行
pushf
push word 0 ; cs
push word idle;
iret

```

; idle 进程，除非要跳转回 main 函数，否则一直死循环

idle:

 jmp idle

; 此处是返回 main 的代码

mov sp, bp ; 将堆栈指针设置为基址指针 (Base Pointer, BP) 的

 ; 实际上是将堆栈指针重新指向了应用程序的栈。

pop bp ; 弹出堆栈中的值，将其存入基址指针 (BP) 寄存器

 ; 这样恢复了应用程序的堆栈帧。

pop dx ; 恢复现场

pop cx ; 恢复现场

iret ; 通过 iret 回到 app

至此，我实现了 sleep。当然这样的代码还会有问题，这将在后面讨论。

5.4 写入磁盘，查看运行效果

写入磁盘，运行效果符合预期。屏幕上每隔 1 秒依次输出一个 hello! 或者 goodbye! 不过，此时我还没有换行，它们是连在一起输出的。这样的运行效果完全符合预期。

5.5 进行一些优化

5.5.1 实现换行

为了实现换行，可以每次输出后打印一个换行，但是这会导致打印的东西错开，很丑，如下图所示：于是，我才用定义一个数据段保存打印的行号，每次打印后行号加 1 的方式来

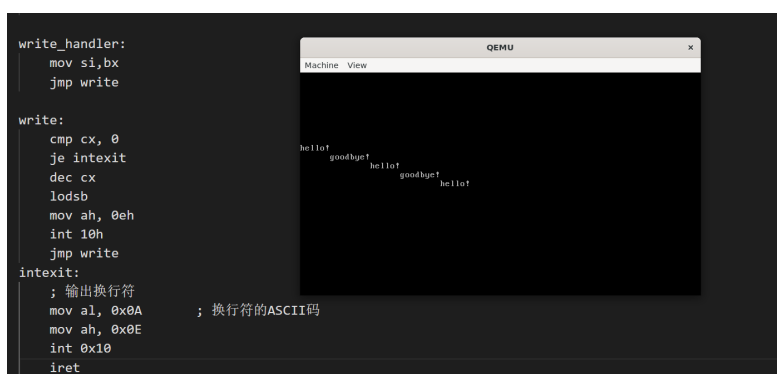



图 3: 很丑的换行

实现，解决了这个问题，代码相对简单，不在此展示。

5.5.2 优化 makefile 文件

此前，我的 makefile 需要两个指令才能运行结束，我做了修改，使得一个 “make” 指令就能直接跑起来，方便助教验收。

六、各种尝试、失败、debug 过程

这部分是体现作业是自己做的核心，这次作业，我遇到了非常非常多的问题 ，我解决了绝大部分，才得以得到上一部分展示的成功，接下来我将详细展示。

6.1 makefile 失败！

在一开始 makefile 中我运行 make 的时候，始终报 “Makefile:2: *** missing separator. Stop.” 我查找了文献得知这是缩进产生的问题。参考文献[Makefile:2: *** missing separator. Stop. 解决方法](#)，我修改了我的缩进配置，但是仍然没有解决问题。偶然间我发现我的 vscode 里面的状态栏显示，我用的是空格进行缩进而不是 tab，如下图所示。修改为采用 tab 缩进后立马解决了问题。



6.2 链接失败！

在编写链接文件的时候，我还遇到了 “arm-linux-ld: warning: cannot find entry symbol __start; defaulting to 00000000” 的问题，这是由于我没用 __start 来作为标签开始写代码，这很好解决。同时，链接失败还因为我代码运行的第一个函数不是 main 了，这通过 objdump 指令看了出来，所以我用了 -e main 指令告知程序入口是 main。在其次，gcc 编译的时候一定要用 -m16，这样才能成功搞好。

6.3 关于打印换行的问题

我后来发现，我的代码打印到屏幕底端就会出现奇怪的显示，如下图所示。因此我又添加了一个计数器，统计一旦快打印到屏幕底端就清屏从头再打，这个代码相对简单，不在这

里展示。最终解决了这个问题。

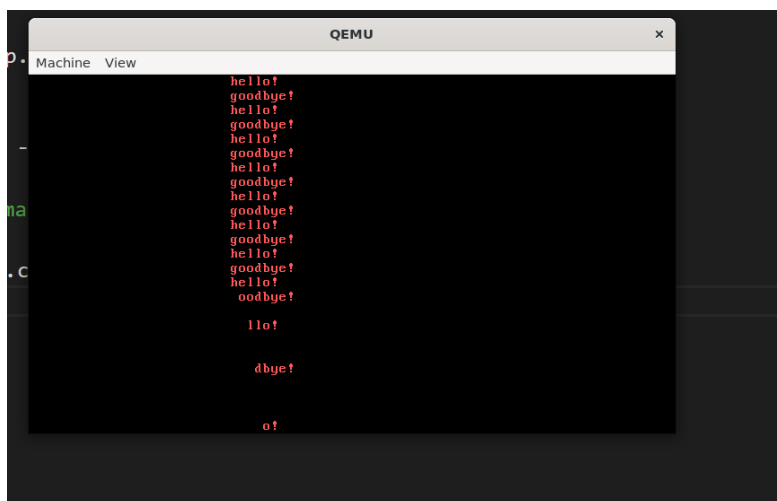


图 4: 奇怪的打印

6.4关于打印字符的问题

由于我的数据段一开始定义的刚刚好写到 512 字节结束，所以控制了打印的字符个数限制，导致我后来想再打印一个 goodbye 的时候出现了问题。我一开始以为是 sleep 实现错了，后来反应过来，将数据段在 ld 文件中的位置提前了一些，成功能够打印，发现还是 sleep 实现的有问题。

6.5时钟中断停止了！

在某过程，我发现，我的代码只能打印两个 hello 和一个 goodbye，表现如下图所示：我编写了测试代码，将 counter 计数器的数值输出，以此来定位问题的位置。（此处我的实现是 counter 到 0 就应该打印一次）我将如下的代码放到时钟中断、sleep 中断等各个位置测试：

```
调试 counter 变化是否正确
mov ah, 0x0E      ; AH = 0x0E 表示打印字符功能
mov bx, 0x0000    ; 颜色
mov cx, [counter]
add cx, '0'       ; 将计数器的值转换为对应的 ASCII 字符
```

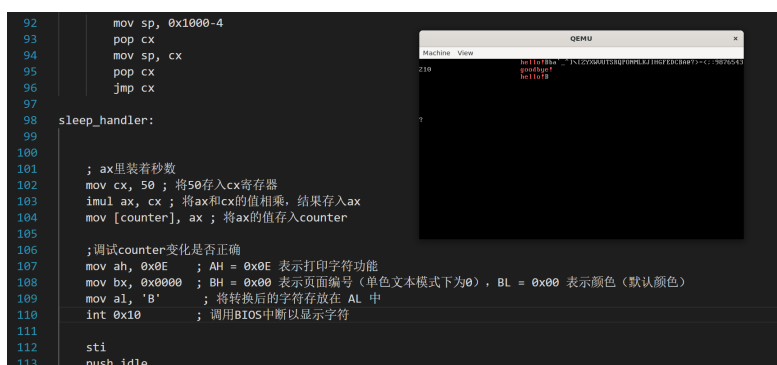


图 5

```

mov al, cl      ; 将转换后的字符存放在 AL 中
int 0x10        ; 调用BIOS中断以显示字符

```

通过测试，我定位到问题在于时钟中断在第二次 sleep 进入内核的时候停止了！通过查阅资料，我得知这是由于我没有在时钟中断 iret 之前告知时钟能继续产生新的中断，于是，我在对应位置加上如下代码，解决了问题：

```

; 告知中断完成，允许产生新中断
mov al, 20h
out 20h, al

```

6.6打印到 16 个左右 hello 就出 bug!

在某过程，我发现，我的代码打印到 16 个 hello/goodbye 就会出问题，我检查代码，发现有个地方没有 iret 直接 jmp 到别的地方去了，这可能会导致栈乱掉或者其他的一些问题。我补上 iret 就重写了一下代码，解决了问题。

6.7中断使用 1ch 不行 8h 可以

在此次作业中，我发现针对我的代码，时钟中断用 1ch 不行，用 8h 可以。这个问题我没有找到本质的答案，但是我猜测，可能是因为 1ch 是用户自定义中断，可以被屏蔽。以及 1ch 的 iret 指令回去的地方可能还需要做一些操作，才能让后续代码继续正常运行。

6.8其他尝试：利用 jmp 切换进程

虽说这是不好的做法，但是我也为此花了不少时间。通过问助教，得知助教推荐使用反汇编语句找出 app 下一条语句位置进行跳转。我实现了一下，代码如下：

```
; api 中代码
; 不够好的实现，通过反汇编找回去的指令跳转
; 与现代操作系统保护模式下内核回 app 不能用 jmp 不符合
not_that_good_sleep:
push ebp
mov ebp, esp
push eax
push ecx

mov eax, [ebp+8] ; 获取参数

push 0x80ab ; 通过反汇编得到，对应的 int 0x81 后的那条语句

mov ecx, esp
mov esp, 0x1000
push ecx
int 0x81
pop ecx
pop eax
pop ebp
ret

; kernel 中跳转回 main 的代码
mov sp, 0x1000-4
pop cx
mov sp, cx
pop cx
jmp cx
```

通过这种方法，也能达到作业的展示效果。同时，我还犯了一个错误，就是修改了 main 文件后忘了修改跳转地址，导致 debug 了一段时间。

6.9 其他问题

在安装 wsl2 以及处理 linux 和 windows 下 make 运行结果不同以及其他小的方面，还出过一些小问题，有过一些小尝试，不过均被我解决。同时，一开始并没有完全理解题意用

iret 方式实现进程切换，现在已经都改为 iret。

七、总结与反思

- 理论好理解不代表实验好做，实验需要非常多的细节，我反思我并没有在课上就完全理解老师的意思，导致我之前用 jmp 来切换进程，实际上这真的不合理，我后来意识到并且全部改为了 iret
- STFW,RTFM 是计算机大类学生必备的素养，知识需要主动检索和学习
- 通过本次实验，我对一个简单的 os 内核有了更深入的理解。我也明白了，像这种大的项目，一定需要有耐心一步一步的拆分、书写、调试。这两句话永远是对的：1. 机器永远是对的 2. 未经检验的代码永远是错的——蒋炎岩

八、其它参考文献和致谢

除了正文给出的参考文献，我参考的文献还有：

实现一个最简单的 HelloOS 内核

ld 链接脚本的编写

汇编语言学习笔记 - 数组和过程体的调用

嵌入式软件之链接脚本.ld

makefile 介绍

同时，衷心感谢与我一起研究讨论 PA4 的好朋友！

