操作系统实验四报告

智能科学与技术　　2024 年 4 月 25 日

实验任务

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：

一、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；2）实现 sleep 内核函数，按整型参数以秒为单位延迟；3）以系统调用的形式向外提供 write 和 sleep 两个系统服务；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。

二、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；2）用 nasm 汇编语言实现对应的 API 库。

三、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；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）。

1　实验环境

1. 电脑操作系统：Windows 11 + 阿里云 Alibaba Cloud Linux（基于 centos）
2. 编程语言：nasm 汇编语言 +C 语言混合编程
3. 辅助工具：

（1）使用 VScode 用于编辑各种文件。

（2）使用本机 Windows11 上的 QEMU 创建一个简单的虚拟机进行实验。

```
qemu-img create -f raw disk.raw 10G
```

（3）使用 Xshell 7 远程连接阿里云服务器，构建 Linux 系统环境，方便使用各种转换与调试工具（makefile，gcc，objcopy，objdump，nasm，ld），使用 Xftp7 进行本机与服务器的文件传输。

（4）使用 makefile 组织和管理整个实验项目的编译、部署和运行

（5）GDB 调试工具

注意：本实验使用 makefile 以及相关依赖文件生成的 app.bin 必须在 Linux 操作系统平台上生成，由于不可知原因，使用本机 Windows 的 gcc 会增加额外的段。并且本实验的 QEMU 部分：也就是把 app.bin 与 kernel.bin 载入 QEMU 磁盘并启动均在 Windows11 上的 QEMU 完成，并尝试过在 wsl 中也能成功实现，但是其他操作系统与平台未尝试过，最好能够在 Windows 的 QEMU 上实现，关于 Windows 上使用的 dd 工具 dd.exe 我也一并放在 app 文件夹。或者，您可以直接在 Windows 的 QEMU 运行我放在 kernel 文件夹的 disk.raw。此外，本实验也配备了实验结果的视频，在实验结果部分，无需下载跳转查看。

2 实验原理

在开始实验之前，我们首先需要明确三大任务的关键点与难点。

2.1 任务一:OS 内核

对于任务一：采用 nasm 汇编语言实现一个实验性质的 OS 内核并部署到 MBR，这是基于前三次实验的前提下完成的，利用前面所学知识，跳过引导过程，相当于在 0x7c00 实现一个简单的内核。对于该任务，我们首先需要理清楚该 MBR（内核）需要实现哪些功能，从而得到可从前面实验迁移的部分与需要新编写的部分：

1. **填写中断向量表**：假定使用 int 80h 以及 int 81h 分别作为 write 与 sleep 内核函数的中断号，我们需要完成 int 8h（时钟中断处理例程），int 80h（write），int 81h（sleep）中断号对应中断处理函数的地址的填写，这一部分属于第三次实验的内容，不再赘述。

2. **设定外部定时器**：这一部分属于第三次实验内容，沿用 20ms 发出一次时钟中断的设置。

3. **加载应用程序:** 根据实验要求，在我们得到应用程序的可执行文件后，需要把它加载进硬盘的第二个扇区（在 CHS 寻址方式下是 2 号扇区，在 LBA 寻址下是 1 号扇区），我们在 MBR 中需要实现的是将它从硬盘加载到内存（本次实验加载至 0x8000）并跳转执行，根据第二次实验的经验，我们可以轻松迁移实现，不再赘述。

4. **提供应用服务程序 write sleep**：实现 write 内核函数，向屏幕输出指定长度的字符串；实现 sleep 内核函数，按整型参数以秒为单位延迟。该部分需要与下面两大任务统筹实现，实现的关键点在于参数的传递。

5.**sleep 函数执行时进程的切换**：该部分需要借助定时器与计数器，编写进程切换程序切换 app 进程与 idle 进程。由于是简单的进程切换，只需要考虑栈的分配与现场保护，在内核中为 APP 应用程序分配地址为 APPSTACK 的栈，为 sleep 期间进行的 idle 函数分配地址为 CORESTACK 的栈。

2.2 任务二：API 库

任务二分两部分，采用 nasm 汇编与 C 语言混合编程，实现一个 API 库：首先使用 C 语言编写一个包含 write 和 sleep 函数声明的头文件；然后使用 nasm 汇编语言实现函数的定义（因为该步骤需要与硬件关联：使用 int 80h，int 81h 等机器指令才能实现）。

由任务一可知，该任务的关键点在于实现内核函数（任务一），API（任务二），以及 C 语言 main 函数中调用的 write 与 sleep（任务三）三个地方的参数传递方式，其中还涉及了使用 gcc 将 API 库与 main 函数连接成为一个可执行文件的相关知识。在该任务中，难点在于 C 语言是 32 位汇编，而我们最终在 MBR 中加载 app 到内存后，是在 16 位实模式下执行的，需要解决这一冲突，根据后续查询资料，可以在 gcc 过程中添加-m16 参数解决。

2.3 任务三：示范应用

采用 C 语言编写示范应用，用于测试 OS 内核功能，编写部分并无挑战，该任务难点仍在于与任务一以及任务二的统筹联合，关键点在 gcc 阶段。

2.4 总结

根据对三个任务的理解，我们得到了此次任务的比较关键的内容：实现任务一，任务二与任务三关于 write，sleep 的不同实现函数之间的关联与整合，我们可以参照 Figure 1 的 Linux 系统调用执行流程。

根据 Figure 1，由于我们考虑采用了单独的 80h 以及 81h 作为两个函数的中断号，不再需要异常处理例程对 80h 进行分流，那么参考 C 库封装例程，我们需要在 nasm 编写的库函数定义文件 syscall.asm 中实现的主要任务是向内核函数传递参数以及发出中断陷入内核态。对 C 库封装函数与对应内核函数约定使用特定的寄存器进行参数传递。本实验中，write 约定使用 bx，cx 分别传递字符串地址与长度，sleep 约定使用 ax 寄存器传递秒数。

下一个关键点在于 C 示范应用 *main_function.c* 与 C 库封装 *syscall.asm* 之间的参数传递。通过查阅资料可以知道 C 语言的参数传递是通过堆栈实现的，函数如果有多个参数的话，那么最右边的参数先压入堆栈，通过反编译 main.o 文件也可以发现这一点（Figure2）。那么根据该知识，在编写 write 的 C 库封装时，在内部先建立临时栈并保存要用到的寄存器数据，然后依次从栈中取出的参数放到约定好的参数传递寄存器中并发出中断。

最后一个关键点在于 sleep 函数的实现，通过实现 sleep 函数，理解进程切换的本质也是本次实验最重要的目标之一。借助实验三实现的定时器与计数器，我们可以轻松达到计时“睡眠”的效果。但在实际情况中，为了提高 CPU 的利用率，往往会在 sleep 时间内进行另一项任务，这就涉及了进程切换的知识。本次实验要求实现简单的进程切换，具体实现为保存现场与返回地址，切换栈，使用 iret 跳转。尽管 i8086 不区分内核态与用户态，但是出于安全考虑，在进行内核与应用程序（包括 app 与 idle）的切换时，都要求使用 iret 进行控制权转移（因为 iret 可以使 CPU 的状态翻转）。而 iret 相当于 pop IP , pop CS ,popf(标志寄存器状态)，基于此知识，可以实现进程切换。

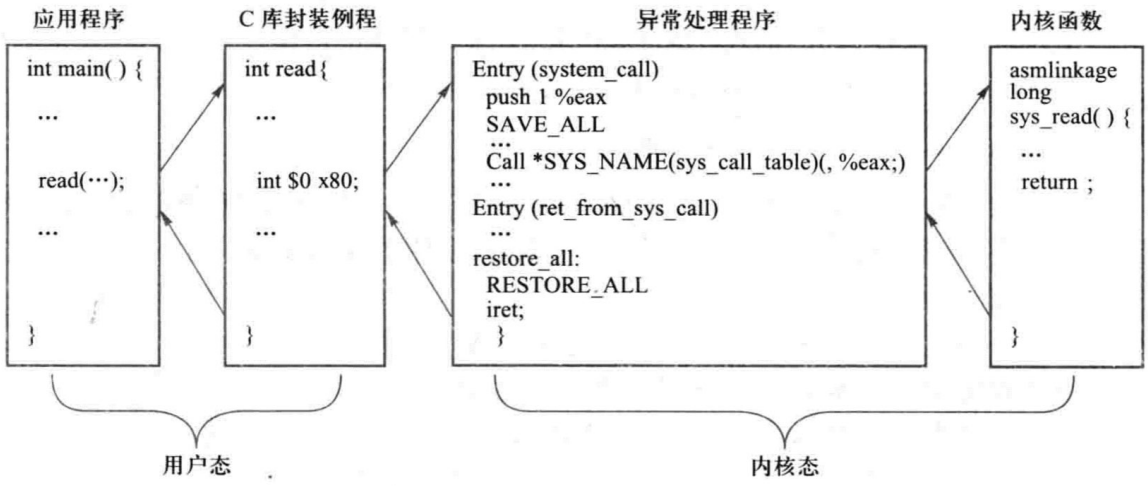

图 1-11　Linux 系统调用执行流程

Figure 1: Linux 系统调用执行流程

```
8063:   66 6a 05                pushl  $0x5
8066:   66 68 fa 81 00 00       pushl  $0x81fa
806c:   66 e8 4e 00 00 00       calll  80c0 <write>
```

Figure 2: main 函数调用 write 函数

在理清楚三个任务之间的关联后，该实验总体上其实只需要分成两大部分编写，借助 QEMU 作为桥梁连接：

1. 实现 MBR（kernel.asm），使用 nasm 编译后放到硬盘的第一个扇区

2. 实现 API 库 ($myos.h, syscall.asm$) 与示范应用 ($main\_function.c$), 借助 gcc，ld 链接三个文件使之成为可执行文件 app.elf，但此时 app.elf 仍旧存在程序头，多余的段等多余的信息，通过 objdump 反编译 app.elf 可以得到所有的段名与内容，我们只需要.text（代码段）以及.rodata（数据段）的内容，使用 objcopy 截取出这两部分的内容，最终得到 app.bin（注意过程中我们需要自定义 ld（my.ld）链接器编排地址并使 app.bin 为 512 字节）。在这个过程中我们编写好所有文件，借助 makefile 得到 app.bin。最终，我们把 app.bin 放到硬盘的第二个扇区，这时候启动 QEMU，实验就完成了。

3 实验过程

根据实验原理的分析，我们以总结部分最后分成的两部分（编写 MBR，得到 app.bin）进行编排。由于实验的复杂性，可以考虑先实现 write 内核函数，把整个流程跑通，在此基础上再进行 sleep 函数的实现。

3.1 编写 MBR

根据实验原理的分析，MBR 主线程需要实现 5 大部分，对于可从前面实验迁移的部分不再赘述，给出以下汇编代码以及注释，需要注意的是在数据段预先分配好给 app（APPSTACK）与 idle（CORESTACK）的栈：

```
BITS 16
SECTION MBR vstart=0x7c00
_start:
    ;清屏
    mov ax, 0600h
    mov bx, 0700h
    mov cx, 0
    mov dx, 184fh
    int 10h

    ;初始化栈：给应用程序app用的
    xor ax,ax
    mov sp,APPSTACK
    mov ss,ax

    ;填写中断向量表

        ;时钟中断
    xor ax,ax
    mov ds,ax
    mov bx,32
    mov word [bx],clock_interrupt_handler-$$
    mov word [bx+2],07c0h
```

```
24
25          ;write :int 80h
26      mov ds,ax
27      mov bx,0x80*4
28      mov word [bx],write_function-$$
29      mov word [bx+2],07c0h
30
31          ;sleep :int 81h
32      mov ds,ax
33      mov bx,0x81*4
34      mov word [bx],sleep_function-$$
35      mov word [bx+2],07c0h
36
37      ;设定外部定时器，规定每20ms发出一次时钟中断
38
39          ;设置时钟频率
40          ; 设置8253的控制字，选择通道0并设置为方式3
41      mov dx, 0x43
42      mov al, 0x36
43      out dx, al
44
45          ;设置计数初值为23863
46      mov dx, 0x40
47      mov ax, 23863
48      out dx, al
49      mov al, ah
50      out dx, al
51      sti
52
53      ;加载应用程序，使用CHS地址法，将位于2号扇区的app.bin拉取到0x8000并跳转执行
54      xor ax,ax
55      mov es,ax
56      mov bx,0x8000
57
58      mov ah, 2 ; BIOS读取扇区功能号
59      mov al, 1 ; 读取扇区的数量
60      mov ch, 0 ; 柱面号清零
61      mov cl, 2 ; 从第二个扇区开始
62      mov dh, 0 ; 磁头号清零
63      mov dl, 0x80 ; 使用硬盘的驱动器号
64
65      int 0x13 ; 调用BIOS中断13h读取磁盘扇区
66
67      pushf
68      push word 0 ;cs
69      push word 0x8000 ;ip
70      iret ;从内核到应用程序执行
```

```
71
72      ...
73  ;数据段
74  APPSTACK equ 0200h ;应用程序栈
75  CORESTACK equ 0x6000 ;idle栈
```

接下来，实现此次 MBR 最重要的部分，实现三个中断服务处理例程：时钟中断处理例程，write 内核函数，sleep 内核函数。其中，时钟中断处理例程是为 sleep 内核函数服务的，并且要加入“进程切换”机制，这其中涉及栈的切换与现场保存，使得在 sleep 的时间内 CPU 也不空闲，而是执行内核初始化后的 idle 函数（进程）。

3.1.1 write 内核函数

根据实验原理，我们只需要知道与 C 库封装系统调用函数约定好的参数传递方式是：通过 bx 寄存器传递字符串的地址，通过 cx 寄存器存储字符串长度（这样设置是因为 int 10h 也是通过 cx 来传递字符串长度），关于每次输出换行，我们可以设定一个计数器 counter2，每一次输出便加一，并且以 counter2 的数值作为下一次输出的行号，这样便可以做到换行输出。

```
1   ;int 80h
2   write_function:
3       xor ax,ax
4       mov ds,ax
5       mov es,ax
6
7       ;cx = 串长度
8       mov bp,bx ; es:bp = 串地址
9       mov ax,01301h ; ah = 13, al = 01h
10      mov bx,000ch ;页号为 0(bh = 0) 黑底红字(bl = 0Ch,高亮)
11      mov dh,[counter2] ;显示的行号
12      mov dl,39 ;显示的列号
13      int 10h
14      inc byte [counter2]
15      cmp dh,24 ;当到达页面最后一行，那么返回第一行输出
16      jz zero
17      iret
18  zero:
19      mov word [counter2],0
20      iret
```

3.1.2 sleep 内核函数

根据实验原理，我们知道与 C 库封装系统调用函数约定好的参数传递方式是：通过 ax 寄存器传递 sleep 的时间（1s 为单位），我们在进入 sleep 内核函数初始化一个计数器 counter（初始化 counter 为秒数 *50+1，因为时钟中断每 20ms 发出一次中断，50 次后是 1s，加 1 是方便时钟中断处理函数区分“睡眠时间是否结束”与未处在 sleep 的状态），并跳转至进程切换程序 *wait_section*：利用应用程序栈 APPSTACK 进行保护现场，压进 app 进程发出 sleep 中断前的地址，然后切换给 idle 函数分配的

栈，最后利用 iret 将控制权交给 idle 进程（由于 idle 作为用户进程之一，所以从内核到 idle 需要使用 iret，而不能直接使用 jmp，不符合安全规范）。

在时钟中断处理例程中，每次时钟中断发出时检测 counter 的值，如果为 0 则什么也不干，如果为 1，则说明 sleep 时间到，否则 counter 每次减一，以标识 sleep 时间的流逝。睡眠时间到后，由于发出时钟中断时压入栈的地址是 idle 进程的地址，所以我们把 idle 进程的地址出栈，再压入进程切换程序的地址，使用 iret 指令转至 *wait_section_back*，跟上面的进程处理程序相似，该进程切换主要是从 idle 进程切换到 app 进程（利用 iret）。这样就实现了在 sleep 的时间内执行 idle 进程。

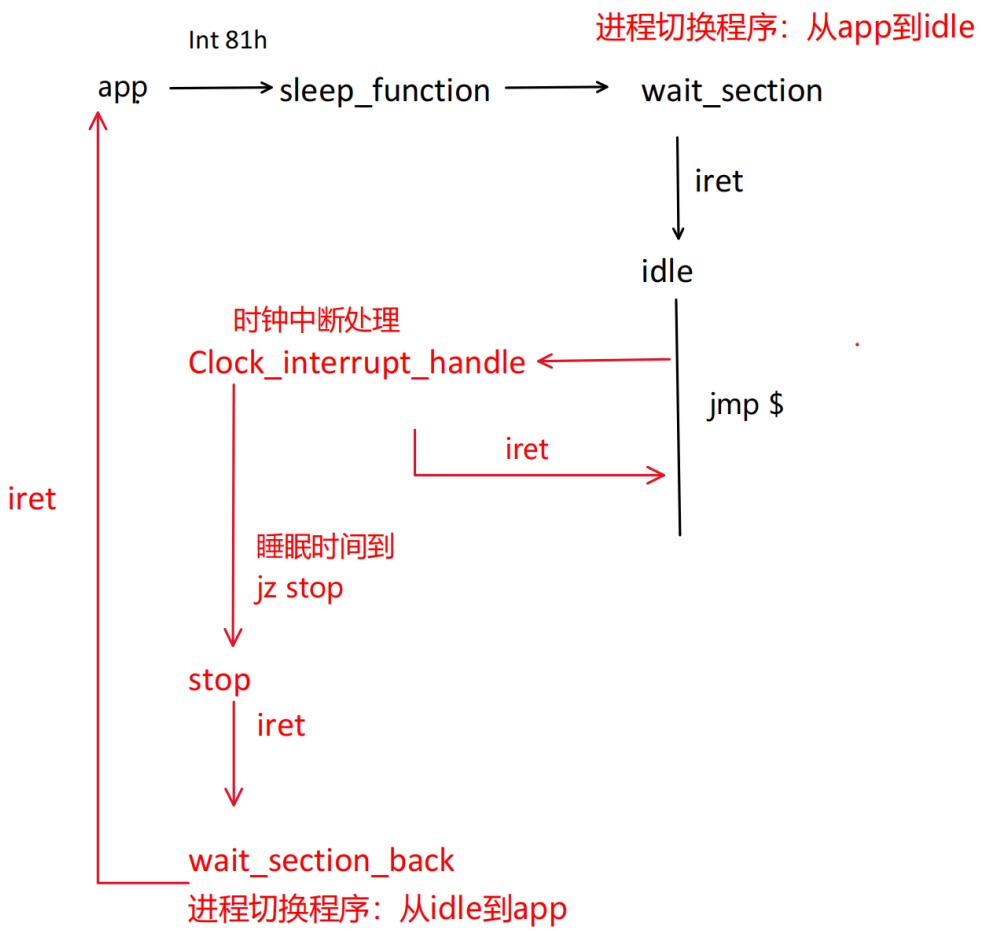

Figure 3: sleep 流程图

```
; int 81h
sleep_function:
    ; ax里装着秒数
    mov cx, 50 ; 将50存入cx寄存器
    imul ax, cx ; 将ax和cx的值相乘，结果存入ax
    mov [counter], ax ; 将ax的值存入counter
    inc byte [counter] ;加一方便时钟中断处理程序判断睡眠时间是否结束
    jmp wait_section ;切换进程

;进程切换程序：从app到idle
wait_section:
    ;保存调取中断前app的指令地址：由于在调用int 81h时会在应用程序栈压进中断前的地址，只
```

```
        要使用iret就能回去，故这一块不再重复保存调取中断前app的指令地址的操作，考虑在睡
        眠时间结束后直接切换回应用程序的栈，然后使用iret指令（sleep_back标签）
    ;保护现场：在idle进程只可能用到dx
    push dx
    ;换栈
    push bp
    mov bp,sp
    mov sp,CORESTACK ;切换栈

    pushf ;保存标志寄存器状态
    push word 0 ;cs
    push word idle ;ip
    iret ;使用iret从内核到进程函数idle


idle:
    sti
    jmp $

;时钟中断处理例程
clock_interrupt_handler:
    mov dx, [counter]
    cmp dx, 0
    jz skip_display ;如果为0则跳转，不执行

    cmp dx,1 ;如果为1，说明sleep已经结束了
    jz stop

    ;否则减1
    dec byte [counter]
    mov al,20h
    out 20h,al
    iret ;仍旧是返回idle

skip_display:
    mov al,20h
    out 20h,al
    ; 结束中断处理例程
    iret
;睡眠时间结束
stop:
    mov word [counter],0
    add sp,4 ;把返回idle的地址出栈
    push word 0 ;cs
    push word wait_section_back ;IP
    mov al,20h
    out 20h,al
```

```
58      iret ;sleep已经结束了,那么跳到进程切换处理程序
59
60  ;进程切换程序: 从idle到app
61  wait_section_back:
62      mov sp,bp ;切换回app的栈
63      pop bp
64      pop dx ;恢复现场
65      jmp sleep_back ;切换进程,回到app
66  sleep_back:
67      iret ;由于已经切换回app的栈，该栈的栈顶已经保存了app发出中断前的地址，此时iret就可
            以回到app
68
69  counter dw 0 ; 计数器变量，用于记录时钟中断触发次数
70  counter2 dw 0 ; write换行
71  APPSTACK equ 0200h ;应用程序栈
72  CORESTACK equ 0x6000 ;idle栈
73  times 510-($-$$) db 0 ;填充剩余空间使程序大小为512字节
74  dw 0xAA55 ; MBR标志
```

3.2 生成应用程序可执行文件 app.bin

该部分需要实现多个文件，总的来说主要是 API 库与示例应用。示例应用 *main_function.c* 就是简单的 C 语言 main 函数，循环调用 write 与 sleep 函数：

main_function.c

```
1  #include "myos.h"
2  int main(void)
3  { while(1)
4     {
5       write("hello",5);
6       sleep(1);
7     }
8  }
```

接下来，使用 c 语言实现 myos.h，用于声明两个函数，然后使用 nasm 汇编语言实现两个函数的定义，根据实验原理中的关于参数传递寄存器的约定以及 C 语言函数参数压栈的知识，在函数内部先建立临时栈并保存要用到的寄存器数据，然后依次从栈中取出参数放到约定好的参数传递寄存器中并发出中断。

myos.h

```
1  #ifndef MYOS_H
2  #define MYOS_H
3  // 声明write函数原型
4  void write(char *msg, short len);
5  // 声明sleep函数原型
6  void sleep(short seconds);
```

```
#endif /* MYOS_H */
```

syscall.asm

```
BITS 16
section .text
    global write
    global sleep

write:
    push ebp
    mov ebp, esp ;建临时栈
    push ebx ;保存要用到的寄存器
    push ecx

    ;注意此处容易出错：在跳转write之前在栈中还会压入跳转前下一条指令的地址，所以第一个参数应该是+8
    mov ebx, [ebp+8] ;第一个参数，字符串地址，char*类型
    mov ecx, [ebp+12] ;第二个参数，字符串长度,在h头文件里是short类型

    int 0x80 ;write内核函数

    pop ecx
    pop ebx
    pop ebp

    retd ;32位回跳

sleep:
    push ebp
    mov ebp,esp
    push eax
    mov eax,[ebp+8]
    int 0x81
    pop eax
    pop ebp
    retd
```

最后，使用 gcc，ld 等工具，将 *main_function.c*，syscall.asm 分别转成 main.o 和 syscall.o 目标文件，再使用 ld 链接两个目标文件（由于需要使用自定义 my.ld 编排 app.elf 的地址与大小），最终使用 objcopy 得到只有.text 与.rodata 两个段的 app.bin 文件，其中 makefile 文件如下，qemu 部分是用于写入硬盘和启动 QEMU 的命令行：

makefile ：请注意，务必在 Linux 系统上执行该文件生成 app.bin，否则会出现错误，具体看实验报告的第 5 部分

```
app.bin:app.elf
    objcopy -R.eh_frame -R .comment -O binary app.elf app.bin
```

```
3
4  app.elf:main.o syscall.o
5      ld -m elf_i386 -T my.ld -e main main.o syscall.o -o app.elf
6
7  main.o:main_function.c myos.h
8      gcc -m16 -I. -c main_function.c -o main.o
9
10 syscall.o:syscall.asm
11     nasm -f elf32 syscall.asm -o syscall.o
12
13 qemu:
14     qemu-img create -f raw disk.raw 10G
15     nasm -f bin -o kernel.bin kernel.asm
16     dd if=kernel.bin of=disk.raw bs=512 count=1
17     dd if=app.bin of=disk.raw bs=512 seek=1 count=1
18     qemu-system-x86_64 -drive file=disk.raw,format=raw
19
20 clean:
21     rm app.elf syscall.o main.o
```

注意，必须把 main.o 放在 syscall.o 前面，否则在.text 中 main 函数不是 app.elf 的第一个函数（入口）。其中，my.ld 设置了.text 段和.rodata 段的地址，通过查看初始实验使用非自定义 ld 得到的 app.bin 可以得到当字节数为 512 时（借助 VScode 的 Hex Editor），.rodata 的起始地址应该是 0x81FA，从而保证最后生成的 app.bin 为 512 个字节，并且，最后 app.bin 会被拉取到 0x8000，通过设置地址可以保证字符串地址正确编址。注意：此处设置地址为 0x81FA 是为了凑 512 个字节，实际上不需要 512 个字节，通常把 rodata 段放前面一点，这样修改 main 的输入字符串时不会出错。

my.ld

```
1  SECTIONS {
2    . = 0x8000;
3    .text : {*(.text)}
4    . = 0x81FA;
5    .rodata : {*(.data)}
6  }
```

最后，将 MBR 与 app.bin 分别写入硬盘的第一个扇区和第二个扇区并启动，相关命令行：

```
1  nasm -f bin -o kernel.bin kernel.asm
2  dd if=kernel.bin of=disk.raw bs=512 count=1
3  dd if=app.bin of=disk.raw bs=512 seek=1 count=1
4  qemu-system-x86_64 -drive file=disk.raw,format=raw
```

4 实验结果

启动虚拟机，根据 main 函数的参数设置，在启动界面不断换行打印字符串“hello”，且每两次之间间隔 1s。视频演示结果：点击观看。注：为了提供更多样的体验，提交的文件中 sleep 的参数设置

为 2s，视频中的设置是 1s。

Figure 4: 启动 0s 后

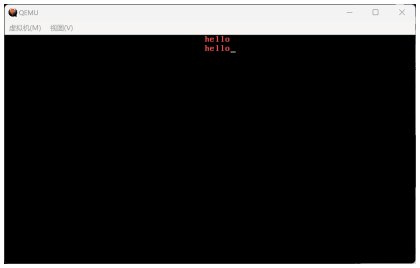

Figure 5: 启动 1s 后

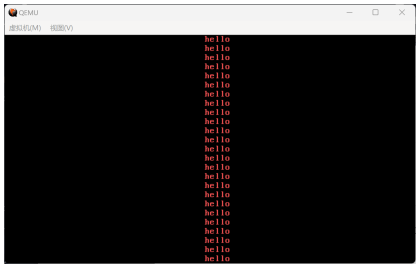

Figure 6: 启动 n s 后

5 实验中的困难与解决

5.1 MBR 16 位与 elf32 位的冲突

由查询到的资料显示，gcc 能生成的二进制文件最低位数是 32 位（elf32），而我们的实验是基于 i8086 的 16 位机器指令进行的，最终生成 app.bin 后加载到内存中执行时需要确保其能在 16 位实模式下运行。根据查阅资料，解决方法是在 gcc 编译 main.c 文件时使用-m16 参数，而在编写 API 库时，注意 syscall.asm 需要在开始注明 BITS 16。但是，这种方法还存在隐藏的问题。通过反汇编 app.elf 文件（Figure 6），我们可以知道，实际上生成的文件仍旧是 32 位的，只不过是做了特殊的处理（如 Figure 6 中显示的每个指令前加了 66），那么我们在编写 syscall.asm 时要特别注意参数的存取以及函数的回跳都需要使用 32 位指令，否则会造成错误。

```
00008000 <main>:
    8000:   67 66 8d 4c 24 04       leal   0x4(%esp),%ecx
    8006:   66 83 e4 f0             andl   $0xfffffff0,%esp
    800a:   67 66 ff 71 fc          pushl  -0x4(%ecx)
    800f:   66 55                   pushl  %ebp
    8011:   66 89 e5                movl   %esp,%ebp
    8014:   66 51                   pushl  %ecx
    8016:   66 83 ec 04             subl   $0x4,%esp
    801a:   66 83 ec 08             subl   $0x8,%esp
    801e:   66 6a 05                pushl  $0x5
    8021:   66 68 fa 81 00 00       pushl  $0x81fa
    8027:   66 e8 23 00 00 00       calll  8050 <write>
```

Figure 7: 反汇编 app.elf 部分内容

5.2 在 Windows 与 Linux 系统上使用 gcc 的不同

在实验刚开始前，使用 Windows 系统进行实验的初次尝试时，研究使用 Windows 的 gcc 进行编译得到的 app.elf 的反汇编结果发现，在链接时会自动加入 C 库的初始化部分（Figure 7），且添加参数无法解决，原因尚且未找到。解决方法是更换生成 app.bin 的实验平台，在 Linux 操作系统平台上进行实验，问题解决。因此，如果需要复现实验，务必注意需要在 Linux 平台上运行 makefile 文件生成 app.bin。

```
app.elf:    file format pei-i386


Disassembly of section .text:

00401000 <___mingw_invalidParameterHandler>:
 401000:55              pushw  %bp
 401001:89 e5               movw   %sp,%bp
 401003:90              nop
 401004:5d              popw   %bp
 401005:c3              retw

00401006 <_pre_c_init>:
 401006:55              pushw  %bp
```

Figure 8: Windows 上 gcc 结果——多了许多 C 库初始化

6 参考文章

1. 使用 C 语言编写运行于 16 位实模式下的代码
2. 从零开始写 OS 内核 - 加载并进入 kernel
3. MyOS(七)：C 语言结合汇编开发操作系统内核
4. 使用 C 语言编写内核
5. ld 链接器
6. 【嵌入式 04】Linux GCC 编译过程详解及 ELF 文件介绍
7. GCC 基本使用
8. Makefile 教程