

Lab2 实验文档

0. 前言

在 lab1 中，我们配置好了实验环境，并用汇编实现了一个可在 qemu 上运行的最小 OS 内核。本次实验中，我们将在源代码层面实现从汇编语言到 C 语言的衔接，这样，通过汇编实现 OS 的启动以及必要的初始化之后，便可以使用 C 语言来编写和实现 vga 输出等其他功能。

1. 代码框架说明

MultibootHeader

此部分内容对应 `multibootheader/multibootheader.S`，是 lab1 中我们已经实现过的 multibootHeader。不同的是，在代码段中，我们不再用汇编指令进行输出，而是通过 `call` 指令，调用 `_start` 过程。`_start` 是 `myOS/start32.S` 中代码起始处的标号，从此处开始，进行必要的初始化，以支持 C 程序运行。

因此，multibootHeader 的作用，是构建一个可被 qemu 识别的启动头。系统启动后，便将控制转入初始化过程 `myOS/start32.S` 中。

```
.globl start

# multiboot specification version 0.6.96
MULTIBOOT_HEADER_MAGIC = 0x1BADB002
MULTIBOOT_HEADER_FLAGS = 0x00000000
MULTIBOOT_HEADER_CHECKSUM = -(MULTIBOOT_HEADER_MAGIC + MULTIBOOT_HEADER_FLAGS)

.section ".multiboot_header"
.align 4
    # multiboot header
    .long MULTIBOOT_HEADER_MAGIC
    .long MULTIBOOT_HEADER_FLAGS
    .long MULTIBOOT_HEADER_CHECKSUM

.text
.code32
start:
    call _start
    hlt
```

myOS

此部分用于实现 OS 的初始化和具体功能模块。

start32.S

上面已经提到，这个文件是对 OS 进行必要的初始化，以支持 C 程序运行。我们来具体介绍一下代码内容：

- `_start`

```
.globl _start      # GNU default entry point
.text
.code32
_start:
    jmp establish_stack

dead: jmp dead      # Never here
```

程序从 `_start` 处开始运行，跳转到 `establish_stack` 处，构建栈。

`dead` 部分是应对程序出错而设置的死循环，正常情况下，程序不会到达这个位置。

- `establish_stack`

```
establish_stack:
    movl $????????, %eax      # 填入栈底地址

    movl %eax, %esp           # set stack pointer
    movl %eax, %ebp           # set base pointer
```

这部分代码是在构建程序运行所需的栈。`esp` 寄存器中存储的是栈顶指针（栈顶地址），`ebp` 寄存器中存储的是栈底指针（栈底地址），初始时，它们位于同一位置。我们所要做的，是在 `movl $????????, %eax` 一行填入一个地址，其将作为栈底地址，被赋给 `esp` 和 `ebp`。

需要注意，栈是从高向低增长的，因此，栈底地址是一个高地址。

此外，通过此种方式填入的地址，是一个硬编码的固定地址。如果你不喜欢硬编码，可以采取以下方案：

```
establish_stack:
    movl $????????, %eax      # 填入正确的内容
    addl $STACK_SIZE, %eax    # make room for stack
    andl $0xfffffffffe0, %eax # align

    movl %eax, %esp           # set stack pointer
    movl %eax, %ebp           # set base pointer
```

此方案是利用操作系统在内存中结束处的地址（即 `????????`，是一个变量，需要你来填入），与参数 `STACK_SIZE` 相加，来确定栈底地址。

- `zero_bss`

```
# Zero out the BSS segment
zero_bss:
    cld                        # make direction flag count up
    movl $_end, %ecx           # find end of .bss
    movl $_bss_start, %edi     # edi = beginning of .bss
    subl %edi, %ecx            # ecx = size of .bss in bytes
    shr1 %ecx                  # size of .bss in longs
    shr1 %ecx

    xorl %eax, %eax            # value to clear out memory
    repne                                # while ecx != 0
    stosl                      # clear a long in the bss
```

这部分是将 `bss` 段清零。`bss` 段的定义课堂上已经讲过，这里不再重复。上面每行代码的具体含义，本实验不做要求，感兴趣的同学可以自行了解。

- `to_main:`

```
to_main:
    call osStart
```

这部分是将控制转移到 `osStart`，也就是 C 程序的入口函数。从此处开始，我们终于回到熟悉的 C 语言了。

osStart.c

```
// 用户程序入口
void myMain(void);

void osStart(void) {

    clear_screen();
    myPrintk(0x2, "Starting the OS...\n");
    myMain();
    myPrintk(0x2, "Stop running... shutdown\n");
    while(1);
}
```

此文件中是 OS C 程序部分的起始。这段程序中，将继续进行系统初始化（本实验中仅包含清屏），然后转入用户程序中。

i386

此目录下是与 i386 底层硬件密切相关的代码文件。本实验中，只有一个 `io.c`。

- `io.c`

此文件中，通过 C 语言内嵌汇编，实现了 C 接口的 `inb` 和 `outb` 函数。这两个函数是我们进行串口输出和读写 vga 光标位置的基础。

```
unsigned char inb(unsigned short int port_from) {
    unsigned char value;
    __asm__ __volatile__ ("inb %w1, %b0": "=a"(value): "Nd"(port_from));
    return value;
}
```

代码简单介绍：

`__asm__`：修饰符，表明接下来的代码是内嵌汇编。

`__volatile__`：修饰符，确保这条指令代码不会被编译器优化。

`inb`：汇编指令名。

`%w1`：占位符。`w` 表示数据宽度为 16 bits (`w = word`)，`1` 表示此处使用第 1 个参数 (`port_from`)。

`%b0`：占位符。`b` 表示数据宽度为 8 bits (`b = byte`)，`0` 表示此处使用第 0 个参数 (`value`)。

`"=a"(value)`：`a` 表示寄存器 `eax/ax/al`（分别对应 `eax` 寄存器的 32 bits、低 16 bits、低 8 bits），`=` 表示只读。组合起来，表示指令会将执行结果存入 `eax/ax/al` 中，然后 C 再从中读数据，存入 `value` 变量中。

`"Nd"(port_from)`：`d` 表示 `edx/dx/di` 寄存器，此参数作为立即数存入 `edx/dx/di` 中，然后参与指令的执行。

`N` 代表范围为 0-255（1 字节）的立即数，用于对 `out/in` 指令进行位宽约束。

此外，可以注意到这条代码被两个冒号分隔。第一个冒号之前为汇编指令内容，第一、二个冒号之间是从这条汇编指令的执行所得到的参数，第二个冒号之后是 C 语言提供给汇编指令的参数。

```
void outb(unsigned short int port_to, unsigned char value) {
    __asm__ __volatile__ ("outb %b0, %w1":: "a"(value), "Nd"(port_to));
}
```

`outb` 的内嵌汇编与 `inb` 类似，这里只介绍不同之处：

`"a"(value)`，`a` 仍然代表寄存器 `eax/ax/al`，但这个参数是出现在第二个冒号之后。所以，`a` 的意义是：将 `value` 值存入寄存器 `eax/ax/al` 中，然后由这个寄存器参与指令的执行。

dev

此目录下是与 qemu 模拟的硬件设备（device）密切相关的代码，也是本实验中，同学们需要完成的主要内容。

- `vga.c`

请同学们在此文件中实现 vga 输出的功能接口，具体要求在文件的注释中。

vga 输出规则和显存地址与 lab1 相同。注意：向 vga 输出，是通过修改显存实现的，显存也属于内存，因此**建议使用 C 语言指针进行显存读写**。

此外，vga 输出需要进行光标位置的设置，它应该位于你最近输出过的字符的下一位（如果是换行符 `\n`，它应该位于下一行的首位置）。光标位置的读写，请参考老师的 ppt，使用 `outb` 和 `inb` 函数。

注意：光标的列号、行号不是 x、y 坐标值，而是相对于 0（屏幕第一个字符处）的一个 16 bits 偏移量：想象一下将所有行连接起来，形成一个类似一维数组的结构，这种情况下，第一行起始位置为 0，第二行起始位置为 80，第三行起始位置为 160，以此类推。行号是这个偏移量的高 8 bits，列号则是低 8 bits。

- `uart.c`

请同学们在此文件中实现串口输出的功能接口，具体要求在文件的注释中。

串口的输出规则与 lab1 相同。

串口是硬件端口，向其输出内容，请使用 `outb` 函数。

lib 和 printk

- `lib/vsprintf.c`

需要在此文件中实现一个函数 `vsprintf`，它是 `printf` 类函数的格式串处理函数。

例如，我们输入格式串：

```
"%d is a digit", 1
```

通过 `vsprintf` 函数处理，会得到一个正常文本串：

```
"1 is a digit"
```

此函数推荐同学们从 C 语言库函数中移植，以支持后续实验（这也是将其放入 `lib` 目录的原因）。当然，你也可以选择自编实现，这种情况下，你需要能至少支持 `%d`。

- `printk/myPrintk.c`

请同学们利用已经实现的 vga 和串口功能接口，补全此文件中的 `myPrintk` 和 `myPrintf` 函数。

`myPrintk` 和 `myPrintf` 的区别在于，它们分别是内核和用户的输出函数。

include 和 userInterface.h

`include` 中是各个功能模块的头文件，其中声明了它们对外提供的接口。

`userInterface.h` 中是 OS 内核向用户程序提供的函数接口。

userApp

此部分是用户程序，本实验中仅包含一个简单的测试程序。

请同学们在 `main.c` 中填写自己的学号姓名，以供测试。

2. 编译、运行和测试

本次提供了一键编译和运行的脚本，完成实验后，在命令行中进入实验根目录，输入：

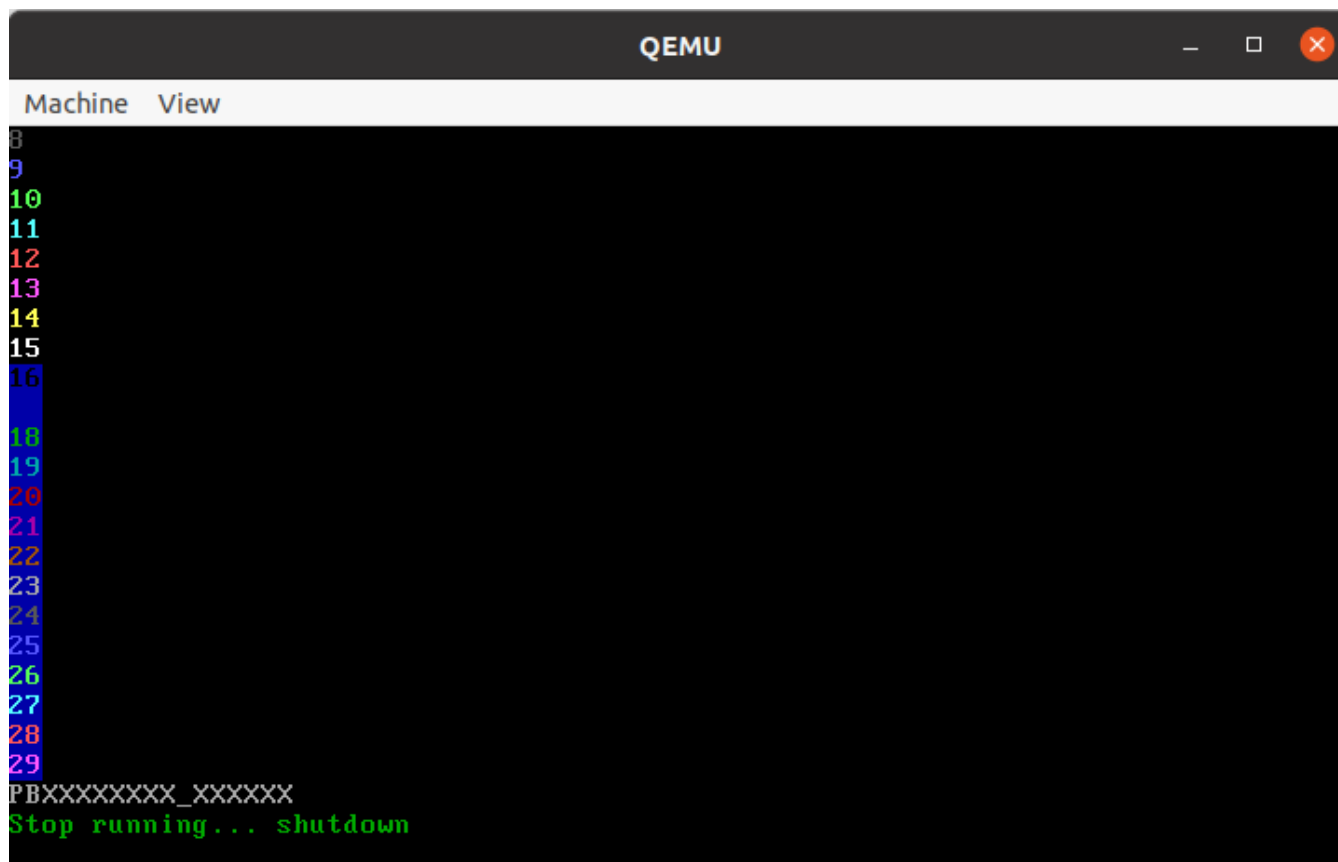
```
./source2img.sh
```

即可一键编译、运行。

你也可以采用 lab1 中的方式来编译、运行：

```
make  
qemu-system-i386 -kernel output/myOS.elf -serial stdio
```

如果一切正常，qemu 会输出如下结果：



```
8  
9  
10  
11  
12  
13  
14  
15  
16  
17  
18  
19  
20  
21  
22  
23  
24  
25  
26  
27  
28  
29  
PBXXXXXXXXX_XXXXXX  
Stop running... shutdown
```

此外，串口（命令行）也会输出同样的内容。

3. 实验要求

内容要求

- 1. 完成 myOS/dev/vga.c 文件，实现 vga 输出功能
- 2. 完成 myOS/dev/uart.c 文件，实现串口输出功能
- 3. 通过自编或移植，完成 myOS/lib/vsprintf.c 文件
- 4. 完成 myOS/printk/myPrintk.c 文件，实现 myPrintk/f 函数
- 5. 在 userApp/main.c 文件中，填写具有与你相符的特异性的内容（如姓名、学号）
- 6. 完成实验报告

实验报告要求

1. 给出软件的框图，并加以概述（5分）

参考老师 ppt 第 5-6 页

2. 说明主流程及其实现，画出流程图（5分）

说明系统从启动到进入 C 程序运行阶段的流程

3. 说明主要功能模块及其实现（10分）

结合你的源代码，说明你是如何实现预期功能的

4. 源代码组织说明（目录组织、Makefile 组织）（5分）

- 目录组织：以树形结构展示你的源码目录
- Makefile 组织：阅读各个 Makefile 文件，以树形结构，给出所有 Makefile 路径变量（如 OS_OBJS）和 目标文件 .o 的组织方式，例如：

```
.
├── MULTI_BOOT_HEADER
│   └── output/multibootheader/multibootHeader.o
└── OS_OBJS
    ├── MYOS_OBJS
    │   ├── ...
    │   ├── ...
    │   └── ...
    └── USER_APP_OBJS
        └── ...
```

5. 代码布局说明（地址空间）（5分）

阅读 myOS/myOS.ld 文件，给出各个段的起始地址（单位：字节）。如果难以确定地址的具体值，则要给出它是按多少字节对齐的。

例如，lab1 的代码布局：

Section	Offset (Base = 1M)
.multiboot_header	0
.text（代码段）	16（为什么？）

备注：.ld 中的变量可在汇编和 C 代码中使用，例如 `__end`

其表示的地址值可在汇编中作为立即数使用：

```
$_end
```

在 C 语言中得到其表示的地址值：

```
extern unsigned long _end;
unsigned long _end_addr = (unsigned long) &_end;
```

6. 编译过程说明 (5分)

只需说明两点：

- 编译所用指令（以防有同学的实验需要使用特殊的编译方法）
- 编译的大致过程：第一步，编译各个文件，生成相应的 .o 目标文件；第二步，根据链接描述文件，将各 .o 目标文件进行链接，生成 myOS.elf 文件。

7. 运行和运行结果说明 (5分)

- 给出你的程序的运行方式（以防有同学的实验需要使用特殊的运行方法）
- 以截图的形式，展示你程序的运行结果

8. 遇到的问题 and 解决方案说明 (可选)

提交要求

提交内容应为 **实验报告 + 源代码**，整个文件夹目录结构如下（你可以根据需要，自行修改 src/myOS 和 src/userApp 中的目录结构，但请保证可以正常编译及运行）：

```
.
├── doc
│   └── report.pdf
└── src
    ├── Makefile
    ├── multibootheader
    │   └── multibootHeader.S
    ├── myOS
    │   ├── dev
    │   │   ├── Makefile
    │   │   ├── uart.c
    │   │   └── vga.c
    │   ├── i386
    │   │   ├── io.c
    │   │   └── Makefile
    │   ├── include
    │   │   ├── io.h
    │   │   ├── myPrintk.h
    │   │   ├── uart.h
    │   │   ├── vga.h
    │   │   └── vsprintf.h
    │   ├── lib
    │   │   ├── Makefile
    │   │   └── vsprintf.c
    │   ├── Makefile
    │   ├── myOS.ld
    │   ├── osStart.c
    │   ├── printk
    │   │   ├── Makefile
    │   │   └── myPrintk.c
    │   ├── start32.S
    │   └── userInterface.h
    └── source2img.sh
```

```
└─ userApp
   └─ main.c
   └─ Makefile
```

请将此文件夹打包为压缩包（格式不限），命名为 `学号_姓名_lab2`，在 bb 系统作业区提交

你只需要提交 `PB18111888_张三_lab2.zip` 一个文件

提交 DDL：2022 年 3 月 28 日 23:59，UTC/GMT+08:00，逾期不接受补交

评分规则

内容	分值
实现 vga 输出	35
实现串口输出	5
成功移植/自编 vsprintf 函数	10
实现 myPrintk/f 函数，通过 userApp 中的测试	5
实验报告	40
代码风格	5