实验二讲义

1 实验 2 说明

本实验是系列实验中的第二个，用于在 Multiboot 协议启动后，从汇编语言编程进入 C 语言编程，区分内核代码和用户代码，并从内核启动转入用户代码运行。

1.1 实验 2 基础

本实验在实验 1 的基础上进行。

在实验一提交的截止时间过后，同学们可以就实验一的内容互通有无。

实验二可以在其他同学实验一的基础上进行：

- 无论你使用哪一个（包括自己的），请在实验报告中标注，实验二的基础来自哪个同学（可以是自己）；
- 给你使用的实验一打分。
- 也可以参考助教提供的实验二框架，这个框架已经帮同学们完成了较为关键的部分代码。若直接使用了助教提供的框架，请说明。

2 实验 2 目的

实现一个具有简单输出功能的 OS，该 OS 包含一个 Multiboot 启动头、一套 OS 功能代码和一套用户代码。Multiboot 启动头运行后将转入 OS 功能代码运行。OS 功能代码主要是初始化一个栈、初始化 BSS 段，然后转入 OS 的第一个 C 入口 osStart 运行。在 osStart 中，进行 OS 各个模块的初始化（若有），然后转入用户代码的 main 入口运行。

目前 OS 功能代码主要是提供 myPrint[k/f]，其中，myPrink 只允许 OS 功能代码调用，myPrintf 只允许用户代码调用。

3 实验 2 要求

[1] 【必须】在源代码的语言层面，完成从汇编语言到 C 语言的衔接。

[2] 【必须】在功能上，实现清屏、格式化输入输出，设备包括 VGA 和串口，接口符合要求

[3] 【必须】在软件层次和结构上，完成 multiboot_header、myOS 和 userApp 的划分，体现在文件目录组织和 Makefile 组织上

[4] 【必须】采用自定义测试用例和用户（助教）测试用例相结合的方式进行验收

[5] 【必须】提供脚本完成编译和执行

4 实验准备（预备知识和准备工作）

4.1 如何组织代码的文件和目录

我们使用不同的子目录来将 Multibootheader 和 myOS 的内核代码和用户代码在源代码层次分开。因此，我们具有一个源代码根目录和 3 个源代码子目录。这 3 个源代码子目录分别对应上述三个部分。

另外，我们将编译过程中所有要生成的目标文件都单独存放到 output 子目录下，以便于我们要删除编译生成的中间结果的时候，可以简单的删除 output 目录下的所有内容即可。

下面是目录划分的简单示意：

```
--源代码根目录（例如可以命名为 lab2）
  --Multibootheader 子目录
  --内核子目录
  --userApp 子目录
  --output 子目录
```

我们的源代码文件将按照上述划分分别组织到对应的子目录下。

可以根据需要决定是否在某个子目录下进一步划分子目录。

我们在源代码根目录下提供根 Makefile 和一个用于编译 OS 和测试运行 OS 的脚本。

4.1.1 Multiboot_head.S 的变化

实验一中我们编写了 multiboot_head.S 文件，这个文件中既包含了启动头，也包含了我们的操作系统，尽管这个操作系统非常简单。现在我们要将启动头和我们的操作系统代码分开，并建立启动头和我们的操作系统之间的关联关系，使得我们能够从启动头转入我们的操作系统代码中运行。

我们约定，操作系统的第一个入口的符号是“_start”。因此，我们需要修改启动头后的代码。

首先，将原来进行输入输出的代码删除。如果你想保留这段代码也是可以的，那就不要删除。

然后，加入一条函数调用指令：“call _start”。这样，启动头启动后，通过 call 指令转入

操作系统的_start 符号处开始执行。

最后，为了防止代码从_start 返回后无代码执行，我们在 call 指令后提供一条 hlt 指令，令其在执行到此处时“暂停”执行，即不执行任何指令。

启动头后的代码示例如下：

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

4.2 从汇编到 C 函数，要准备什么？如何准备？

我们约定，从汇编到 C 函数的接口是 osStart。

从汇编到 C 函数，一条 call 指令就能完成。但是 C 函数的运行，需要栈的辅助。兵马未动粮草先行。因此，在进入 C 函数之前，要先准备并初始化栈。

我们需要在物理内存的合理位置安排这个栈。要注意，在 os 的运行中，这个栈会不断的 push/pop，因此要留够足够的空间给这个栈，以防在运行过程中栈溢出。在 X86 中，我们需要为栈初始化两个寄存器：esp 和 ebp。其中，ebp 是栈基址，esp 是栈针。

此外，我们还需要准备 BSS 段。BSS 段用于存放未初始化的数据。在使用 BSS 段之前，最好对 BSS 段清 0。好处是，万一我们没有遵循常规编程规范，在初始化一个未初始化的数据之前，先行访问了这个变量，那么我们将得到一串 0。如果没有清 0，我们将得到一个未知的任意的数据，给调试带来困惑。

上述两个工作完成之后，我们就可以开始正式调用 C 函数入口 osStart 了。

参考代码的片段如下：

```
.text
.code32
_start:
      jmp establish_stack

dead:     jmp dead              # Never here

# Set up the stack
establish_stack:
      movl      $0x？？？？, %eax

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

```
# 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
      shrl %ecx                     # size of .bss in longs
      shrl %ecx

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


# Transfer control to main
to_main:
      call   osStart

shut_down:
      jmp  shut_down      # Never here
```

由于 osStart 是一个 C 函数，有可能会从这个函数返回。为了防止返回后无代码执行而导致出错，建议在最后增加一个死循环。即上最后一行代码“shut_down: jmp shut_down”。

4.3 从 OS 到 userApp 的 main

我们约定，从 OS 到 userApp 的 main 的转接接口是 userApp 中 myMain 入口。

我们的操作系统没有实现双模式，即没有区分用户态和内核态，但我们在源代码层面通过文件和目录的组织以及限定 userApp 中代码只能使用 OS 接口来形式地区分内核代码和用户代码。

因此，从 OS 到 userApp 的 main 的转接，非常简单，只需要在操作系统初始化完成之后，直接调用 userApp 的 main 函数入口即可。代码示例如下：

```
extern int myPrintk(int color,const char *format, ...);
extern void clear_screen(void);
extern void myMain(void);

void osStart(void){
        clear_screen();

        //此处可以根据需要进行 OS 各个模块的初始化
```

```
        myPrintk(0x2,"START RUNNING......\n");
        myMain();
        myPrintk(0x2, "STOP RUNNING......ShutDown\n");
        while(1);
}
```

4.4 怎么实现 myPrint[fk]

代码框架中已经实现，也可以自己编写，需要理解 c 语言可变参数原理。

4.5 VGA 和串口提供哪些接口给 myPrint[fk]

这一部分是实验的重点。

在 src/myOS/dev/vga.c 中我们要实现的任务：

更新当前光标的位置 update cursor

获取当前光标的位置 get cursor position

清屏函数 clear screen

在 VGA 屏幕上显示字符串 append2screen

光标位置是由一个一维偏移量决定的（并不是（x,y）这样子的二维坐标）。在这种情况下，第 0 行第 0 列的偏移量是 0，第 1 行第 0 列的偏移量是 80，第 2 行第 0 列的偏移量是 160 。 这个一维偏移量由高 8bit 和低 8bit 构成。具体地讲:

当我们往 port=0x3D4 送 0x0F 后，我们与 port=0x3D5 交互的就是这个一维偏移量低 8bit 的数据。

当我们往 port=0x3D4 送 0x0E 后，我们与 port=0x3D5 交互的就是这个一维偏移量高 8bit 的数据。

我们 VGA 屏幕是可以显示 25 行×80 列个字符，但是需要注意的是一个字符占 16bit（16bit 中的一半对应 ASCII 码，另一半对应颜色）。

关于“在 VGA 屏幕上显示字符串”编写的注意事项：

当字符串中含有’\n’抑或是光标已经在行尾时，我们需要进行换行操作。

当前光标位于第 25 行时，如果需要换行，则需要实现 VGA 屏幕的滚屏操作。

在 src/myOS/dev/uart.c 我们将调用 src/myOS/i386/io.c 中的 inb 和 outb 函数，来实现 UART 串口的数据传输。UART 串口的 port 的值为 0x3F8，所以我们要做的事情就是“向

port=0x3F8 发送数据”和“从 port=0x3F8 获取数据”

4.6 如果要显示光标，怎么办

通过读取或者更改数据端口 0x3D5 的数值，就可以做到读取以及设置光标的行列值，进而确定光标在屏幕上的位置

4.7 我们的 Makefile 怎么写？

可以考虑每个目录下提供一个 Makefile【非必须】，并在上层目录下的 Makefile 中通过使用 include 命令来引入下层目录的 Makefile。

关于 Makefile 如何编写的进一步学习，可以参考 GNU make 手册。参考网址如下：https://www.gnu.org/software/make/manual/make.html。

关于上述 include 命令，参见手册中的 3.3 Including other Makefiles。

4.7.1 源代码根目录下的 Makefile

通常源代码根目录下的 Makefile 最为复杂，它给出了编译整个操作系统的规则。下面仅简单介绍示例中的根 Makefile。

```
1	SRC_RT=$(shell pwd)
2
3	CROSS_COMPILE=
4	ASM_FLAGS= -m32 --pipe -Wall -fasm -g -O1 -fno-stack-protector
5	C_FLAGS = -m32 -fno-stack-protector -g
6
7	.PHONY: all
8	all: output/myOS.elf
9
10	MULTI_BOOT_HEADER=output/multibootheader/multibootHeader.o
11	include $(SRC_RT)/myOS/Makefile
12	include $(SRC_RT)/userApp/Makefile
13
14	OS_OBJS = ${MYOS_OBJS} ${USER_APP_OBJS}
15
16	output/myOS.elf: ${OS_OBJS} ${MULTI_BOOT_HEADER}
17		${CROSS_COMPILE}ld -n -T myOS/myOS.ld ${MULTI_BOOT_HEADER} ${OS_OBJS} -o output/myOS.elf
18
19	output/%.o : %.S
20		@mkdir -p $(dir $@)
21		@${CROSS_COMPILE}gcc ${ASM_FLAGS} -c -o $@ $<
22
23	output/%.o : %.c
```

```
24              @mkdir -p $(dir $@)
25              @${CROSS_COMPILE}gcc ${C_FLAGS} -c -o $@ $<
26
27      clean:
28              rm -rf output
```

第 1 行：找到当前目录。结果记录在 SRC_RT 变量中。

第 3 行：用于定义 CROSS_COMPILE 变量，这通常用于交叉编译，分别在用到交叉编译工具的地方使用，如第 17、21、25 行。由于我们没有使用交叉编译，此变量为空。

第 4-5 行：分别定义了汇编文件和 C 文件在编译时所使用的编译选项。分别在第 21 行和 25 行使用。

第 7 行：定义了一个目标名为“.PHONY”的规则。关于“.PHONY”名的进一步说明，参见 make 手册 4.9 Special Build-in Target Names。本实验中，该规则将无条件运行 all 规则。

第 8 行：定义了一个目标名为“all”的 phony 规则。关于 phony 规则的进一步说明，参见 make 手册 4.6 Phony Targets。本实验中，本实验中将检查依赖文件“output/myOS.elf”是否存在，若不存在，则找到一个目标名为“output/myOS.elf”的规则来运行（对应于本文件中第 16-17 行，以生成这个文件。

第 10 行：将 Multiboot_header 将要生成的.o 文件名罗列到变量 MULTI_BOOT_HEADER 中。

第 11-12 行：包含两个子目录下的 Makefile。我们在这两个 Makefile 中将所有源代码所对应的.o 文件都罗列在了一个变量中。

第 14 行：将内核的.o 文件和 userApp 的.o 文件的文件名罗列到变量 OS_OBJS 中。

第 16-17 行：定义了 output/myOS.elf 文件的生成规则，该规则使用所有依赖的.o 文件，按照链接描述文件，将这些.o 文件链接成我们的 myOS.elf 文件。

第 19-21、23-25 行：分别给出了从.S 文件（汇编语言写的源代码文件）和.c 文件（C 语言写的源代码文件）到对应的.o 文件（目标文件）的生成规则。当我们需要某个.o 文件的时候，make 会尝试找到对应的.S 或.c 文件，并使用 gcc 命令将找到的源代码文件编译成.o 文件。

第 27-28 行：定义 clean 规则。在命令行使用 make clean 将找到 clean 规则并执行第 28 行的命令。第 28 行命令用于清除上次 make 时生成的所有中间或结果文件。

4.7.2 子目录下的 Makefile（不含有进一步子目录）

在示例的 Makefile 中，userApp 目录下的 Makefile 最为简单。由于目前该目录下只有一

个 main.c 文件，因此 Makefile 中仅仅罗列了该文件对应的、要生成的.o 文件，并将其罗列到变量 USER_APP_OBJS 中，本实验中该变量的定义如下所示。

```
USER_APP_OBJS = output/userApp/main.o
```

源代码根目录下的 Makefile 中，将 include 这个 Makefile，从而将变量 USER_APP_OBJS 传递过去。

4.7.3 子目录下的 Makefile（含有进一步子目录）

如果一个子目录下还进一步包含一个或者多个子目录，那么 Makefile 中可以使用 include 来引入嵌套子目录下的 Makefile。下面的示例包含 3 个子目录。每个子目录贡献了一个变量，变量将在 MYOS_OBJS 中被展开，从而将子目录下的所有.o 文件带入 MYOS_OBJS 变量中。

```
include $(SRC_RT)/myOS/dev/Makefile
include $(SRC_RT)/myOS/i386/Makefile
include $(SRC_RT)/myOS/printk/Makefile


MYOS_OBJS = output/myOS/start32.o \
            ${DEV_OBJS} \
            ${I386_OBJS} \
            ${PRINTK_OBJS}
```

这种方法的好处是，我们修改子目录下的源代码时，可以根据需要修改子目录下的 Makefile，但，一般情况下，无需修改上层目录下的 Makefile，除非上述文件中所引用的变量的名字被改写了。

4.8 链接脚本的变化

与实验一相比，链接脚本略复杂一些。关于链接脚本的进一步学习，参见 https://sourceware.org/binutils/docs-2.42/ld.html。若你知道自己使用的 binutils 是哪个版本，可以找到你所使用的那个版本的手册来看。但我们所涉及的学习内容与版本关系不大。

本实验的链接脚本中 SECTIONS 定义示例如下：

```
1   SECTIONS {
2           . = 1M;
3           .text : {
4                   *(.multiboot_header)
5                   . = ALIGN(8);
6                   *(.text)
```

```
7                   }
8                   . = ALIGN(16);
9                   .data                   : { *(.data*) }
10                  . = ALIGN(16);
11                  .bss : {
12                          __bss_start = .;
13                          _bss_start = .;
14                          *(.bss)
15                          __bss_end = .;
16                  }
17                  . = ALIGN(16);
18                  _end = .;
19                  . = ALIGN(512);
20      }
```

SECTIONS 命令用于告知链接器，如何从输入文件的各个 sections 来生成目标文件的各个 sections。至于有哪些输入文件，在本实验中，参见根 Makefile 中对应的规则，即其第 16-17 行。在命令行，使用“ld –help”命令，可以获得 ld 的各个选项的说明。根 Makefile 第 17 行中使用的各个选项的含义，请自行查看。除去各个选项后，剩下的就是输入文件列表了。根 Makefile 第 17 行中，输入文件列表由 MULTI_BOOT_HEADER 和 OS_OBJS 这两个变量罗列。

下面简单说明本实验链接脚本（仅 SECTIONS 部分）：

第 2 行：指明当前链接的位置。说明代码链接的地址从 1M 开始。在我们的实验中，这个 1M 对应于物理地址 1M 的位置，代码将被加载到这个地址开始的物理内存中。

第 3-7 行：定义了操作系统的 text 段，即代码段。代码段由 Multiboot 头、各个输入文件的 text 段链接而成。

第 9 行：定义了操作系统的 data 段，由所有输入文件的 data 段链接而成。

第 11-16 行：定义了操作系统的 bss 段，由所有输入文件的 bss 段链接而成。其中，为了后续便于将 bss 段清 0，在 bss 段前和段后定义了几个变量，这几个变量可能在其他地方用到。注意：这里定义的变量可以在源代码中访问。

4.9 编译和运行脚本

为了减少命令行的输入，可以考虑使用一个脚本来辅助。

下面是一个简单的 source2run.sh 脚本的内容：

```
#!/bin/sh

make clean
```

```
make

if [ $? -ne 0 ]; then
        echo "make failed"
else
        echo "make succeed"
        qemu-system-i386 -kernel output/myOS.elf -serial stdio
fi
```

其中，“make clean”用来清理上一次 make 生成的中间或最后的文件。

“make”用来开始一次新的 OS 编译，以生成 output/myOS.elf。

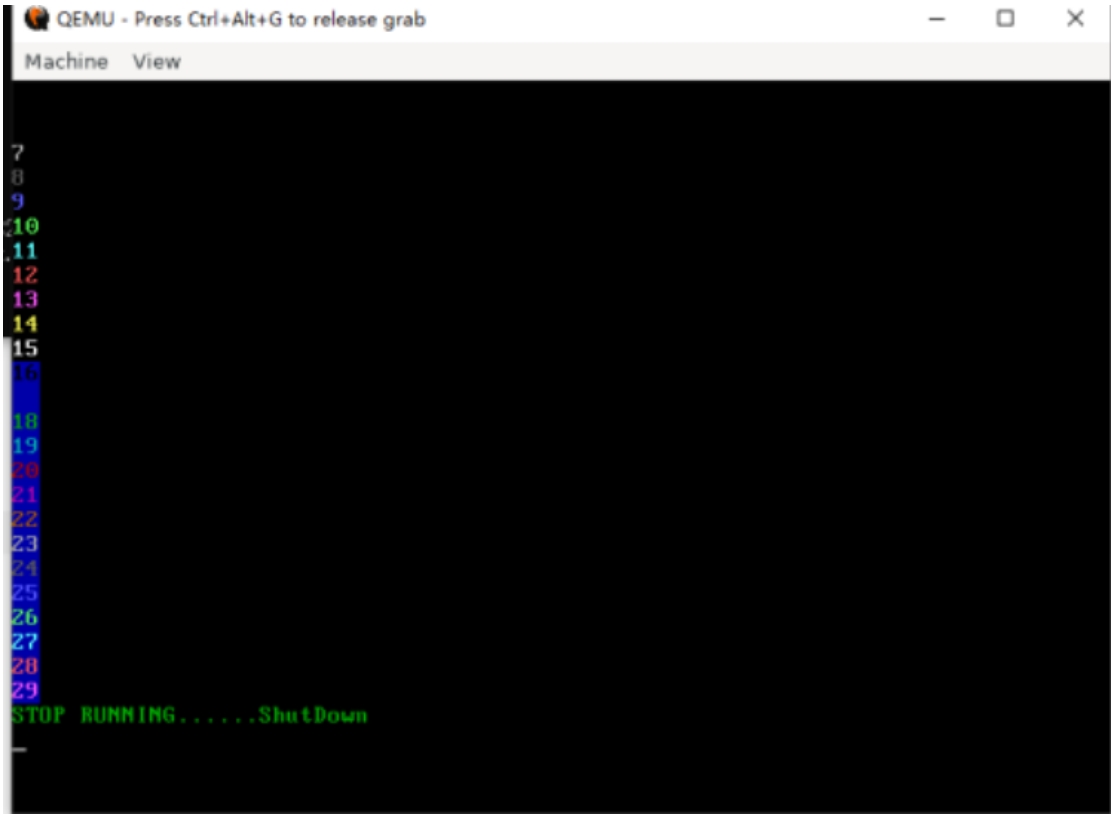
若编译成功，则运行“qemu-system-xxx …”命令来加载运行 myOS。

5 实验内容

src/myOS/start32.S 的编写
src/myOS/dev/uart.c 和 src/myOS/dev/vga.c 的编写
src/myOS/printk/myPrintk.c 和 src/myOS/printk/vsprintf.c 的理解（也可以自己编写）

6 实验示例

我们提供了 source2run.sh 文件，所以只需在命令行键入”./source2run.sh”即可运行程序。运行结果如下：



7 提交要求

提交到 BB 平台。提交内容要求同实验 1

实验报告要求：

- 给出软件的框图，并加以概述
- 详细说明主流程及其实现，画出流程图
- 详细说明主要功能模块及其实现，画出流程图
- 源代码说明（目录组织、Makefile 组织）
- 代码布局说明（地址空间）
- 编译过程说明
- 运行和运行结果说明
- 遇到的问题和解决方案说明