OS_Lab_02 Report

PB21111686_赵卓

实验题目

- Multiboot2myMain

实验目的

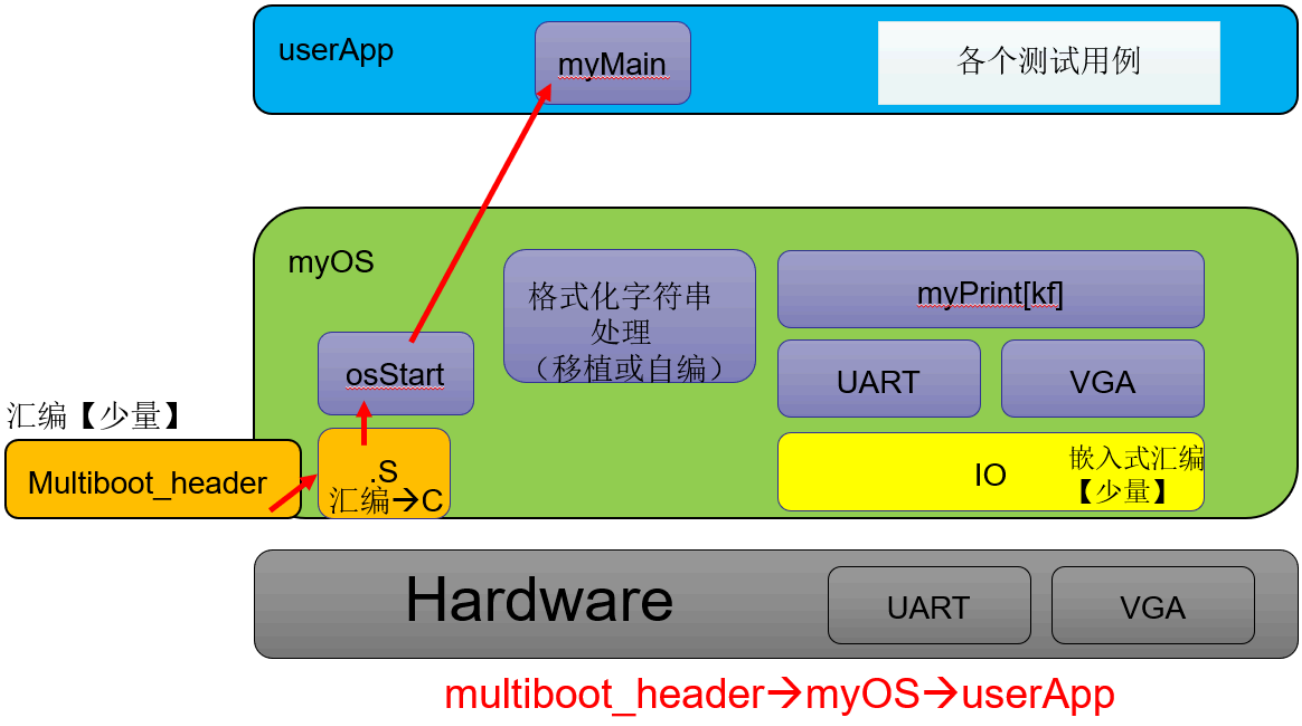
- 在Multiboot协议启动后，从汇编语言编程进入C语言编程，区分内核代码和用户代码，并从内核启动转入用户代码运行。

实验内容

- 本次实验在提供框架下进行，补全start32.S，io.c，uart.c，vga.c文件，以实现从汇编到C的转变，用C语言实现VGA和uart串口输出。

实验框架

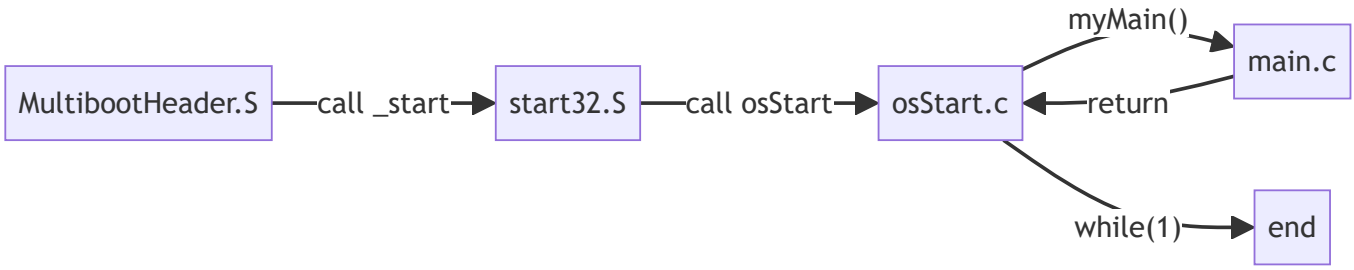
- 本次实验软件框图如下：



- 框图概述：
  - 第一层：userApp，是用户模块，可以通过myMain函数调用第二层中的库函数myPrintk/f，实现输出功能。
  - 第二层：库函数，myPrintk/f和vsprintf，调用第三层的UART和VGA进行输出。
  - 第三层：UART和VGA输出模块，通过uart.c和vga.c实现，调用第四层的IO实现输出。
  - 第四层：IO模块，通过少量的C语言内嵌汇编实现。

实验流程

- 本次实验流程图如下：



- 流程概述：
  - 在MultibootHeader.S汇编执行后，通过调用call _start指令转入start32.S。
  - 汇编执行start32.S，为程序执行进行准备工作，设置栈，并且将栈中内容清0初始化，接着调用call osStart转入osStart.c。
  - 在osStart.c中，调用myMain()函数转入main.c。
  - 在main.c中，正常执行用户程序，实现用户需求，然后return回到osStart.c。
  - 接着执行osStart.c中的死循环，停机结束。

实验模块实现

- multibootHeader.S
  multibootHeader.S与实验一没有变化，将实验一的输出部分去除，同时加入call _start指令即可。

```
.globl start

MAGIC = 0x1BADB002 # we use version 0.6.96 not version 2 (magic = 0xe85250d6)
FLAGS = 0
CHECKSUM = 0xE4524FFE   #(magic + checksum + flags should equal 0)

.section ".multiboot_header"
.align 4
  .long MAGIC
  .long FLAGS
  .long CHECKSUM

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

- start32.S
  - 这部分框架中已经给出大部分内容，只需要补全空缺部分即可。
  - 根据对框架和原理的理解，start32.S在为程序执行开辟栈空间，且不能覆盖OS程序代码。因此取$_end，即bss段的结尾地址，作为我们栈空间的开始地址即可。将$_end补全到空缺处即可完成start32.S。

```
establish_stack:
        movl    $_end, %eax             # eax = end of bss/start of heap #
        addl    $STACK_SIZE, %eax       # make room for stack
        andl    $0xfffffc0, %eax        # align it on 16 byte boundary

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

- io.c
  - 这是我们需要补全的部分，需要实现inb和outb函数，供uart.c和vga.c调用。inb函数从port_from地址中取出value并返回，outb函数则将value的值存入port_to地址中。我们需要通过通过C语言内嵌汇编来实现与底层硬件的交互，完成我们通过C语言无法直接完成的低级操作。
  - 对于C语言内嵌汇编，我们需要有一些了解，以inb为例：
    - _asm_或asm通常用来标识内嵌汇编。
    - __volatile__通常用来指示编译器不要优化这部分代码，保持原指令。

    - "inb %w1,%0"是汇编指令模板。表示inb指令需要两个操作数：一个是I/O端口号，另一个是目标寄存器，用于存放读取到的字节。%w1和%0是占位符，它们分别代表后面的输入和输出操作数。
    - "=a"(value) 是输出操作数部分。"=a"约束指示输出应该被存放在一个累加器寄存器（通常是AL寄存器）中。value是一个C语言变量，它的值将被设置为从I/O端口读取的字节。
    - : "Nd"(port_from) 是输入操作数部分。"Nd"约束通常用于表示一个无符号整数，并且这个整数是一个有效的I/O端口号。port_from是一个C语言变量，它包含要从中读取的I/O端口的地址。
    - 内嵌汇编格式和基本汇编没有太大区别，outb指令类似。

```
unsigned char inb(unsigned short int port_from)
{
    unsigned char value;
    __asm__ __volatile__("inb %w1,%0" : "=a"(value) : "Nd"(port_from));
    return value;
}

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

- uart.c
  uart.c由三个函数组成：
    - uart_put_char：通过uart输出字符c，调用已经实现的outb函数即可，将port_to设置为uart_base即可。
    - uart_get_char：从uart中取出字符c，调用已经实现的inb函数即可，将port_from设置为uart_base即可。
    - uart_put_chars：通过uart输出字符串str，只需将str中的每个字符依次输出，到'\0'停止即可。

```
extern unsigned char inb(unsigned short int port_from);
extern void outb(unsigned short int port_to, unsigned char value);

#define uart_base 0x3F8

void uart_put_char(unsigned char c)
{
        outb(uart_base, c);
}

unsigned char uart_get_char(void)
{
        return inb(uart_base);
}

void uart_put_chars(char *str)
{
        for (int i = 0; str[i] != '\0'; i++)
                outb(uart_base, str[i]);
}
```

- vga.c
  vga内容较为复杂，我们需要实现update_cursor，get_cursor_position，clear_screen和append2screen四个函数功能。
  - update_cursor：更新光标位置。
    - 光标其实是一维的，并不是二维。也就是说实际上光标只有一个16bit数存储，表示vga的偏移量。而行和列都是我们自己定义的，类似于二维数组的存储，实际上还是一维数组。要通过当前行和列得到光标，则要将当前行数乘80再加上当前列数，这样得到光标的绝对偏移量。
    - 然后将得到的光标偏移量，16bit数，拆分成两个8bit数存入光标地址寄存器中。将光标偏移量右移8位，得到高8bit数，先将0x0E（光标的高8位寄存器）存入0x3D4（光标的索引端口）中，这样我们在0x3D5（光标的数据端口）中写入的就是0x0E，实现光标的高8位存储，低8位同理。

```
void update_cursor(void)
{
  outb(0x3D4, 0x0E);
  outb(0x3D5, ((cur_line * 80 + cur_column) >> 8) & 0xFF);
  outb(0x3D4, 0x0F);
  outb(0x3D5, (cur_line * 80 + cur_column) & 0xFF);
}
```

- get_cursor_position：得到光标位置。
  - 理解update_cursor之后，实现get_cursor_position就比较简单，只需要将update_cursor的过程反过来即可。
  - 具体来说，先将0x0E写入0x3D4，然后从0x3D5中取出高8位bit，低8位同理。然后将高8位左移和低8位拼接起来就得到一维偏移量。然后转换成二维偏移，除80得到行数，模80得到列数。

```
short get_cursor_position(void)
{
  unsigned int high8_bit, low8_bit, location;
  outb(0x3D4, 0x0E);
  high8_bit = inb(0x3D5);
  outb(0x3D4, 0x0F);
  low8_bit = inb(0x3D5);
  location = high8_bit << 8 + low8_bit;
  cur_line = location / 80;
  cur_column = location % 80;
}
```

- clear_screen：清空屏幕。
  - vga开始地址为0xB8000，我们给其开辟了0x1000的栈空间。
  - 要清空屏幕，只需要从0xB8000开始，每两个字节写入0x0F20（空）即可，设置行列为0并更新光标即可。

```
void clear_screen()
{
 int *p;
 for (int i = 0; i < 0x1000; i += 4)
 {
        p = (int *)(vga_init_p + i);
        *p = 0x0F200F20;
 }
 cur_line = 0;
 cur_column = 0;
 update_cursor();
}
```

- append2screen：实现内容输出并且可滚屏。
  - 为实现append2screen，添加了两个辅助函数：
    - putsinglechar：输出单个字符，用于实现append2screen的输出功能。在当前行和列对应的地址分别写入颜色和字符内容即可。然后更新行和列，光标位置。

```
void putsinglechar(unsigned char c, int color)
{
while (cur_line >= 25)
    roll_screen();
unsigned char *p;
p = (unsigned char *)(vga_init_p + (cur_line * 80 + cur_column) * 2);
*p = c;
p = (unsigned char *)(vga_init_p + (cur_line * 80 + cur_column) * 2 + 1);
*p = (unsigned char)color;
if (cur_column == 80)
{
          cur_line++;
    cur_column = 0;
}
else
    cur_column++;
update_cursor();
}
```

- roll_screen：翻滚屏幕，用于实现append2screen的翻滚功能。只需要在行号≥25（从0开始计算，因此最后一行是24）时，将下一行的内容复制到上一行，并且将当前行号减一即可。

```
void roll_screen()
{
   int *pread, *pwrite;
   int i = 0;
   for (int i = 0; i < (cur_line * 80 + cur_column); i += 2)
   {
       pwrite = (int *)(vga_init_p + i * 2);
      if (i < (cur_line * 80 + cur_column - 80))
      {
            pread = (int *)(vga_init_p + i * 2 + 160);
            *(pwrite) = *(pread);
      }
      else
            *(pwrite) = 0x0F200F20;
   }
      cur_line--;
      update_cursor();
}
```

- 有了上述两个函数，append2screen只需在对每个字符调用putsinglechar输出，当行号超过24时翻滚屏幕之后再输出即可。

```
void append2screen(char *str, int color)
{
   for (int i = 0; str[i] != '\0'; i++)
   {
           if (str[i] == '\n')
           {
                   cur_line++;
                   cur_column = 0;
                   update_cursor();
                   while (cur_line >= 25)
                        roll_screen();
           }
           else
                   putsinglechar(str[i], color);
   }
}
```

源代码说明

- 目录组织如下：

```
 src
├──Makefile
├──multibootheader
│   └──multibootHeader.S
├──myOS
│   ├──dev
│   │   ├──uart.c
│   │   └──vga.c
│   ├──i386
│   │   ├──io.c
│   │   └──io.h
│   ├──osStart.c
│   ├──printk
│   │   ├──myPrintk.c
│   │   └──vsprintf.c
│   └──start32.S
├──output
└──userApp
```

- Makefile组织如下：

```
src
|__myOS
|   |__dev
|   |__i386
|   |__printk
|__userApp
```

代码布局

- 代码空间由myOS.ld规定：
  - .text：1M空间，存放multibootheader和代码。
  - .data：16字节align，存放程序中已初始化的全局变量。
  - .bss: 16字节align，存放程序中未初始化的全局变量。
  - 堆栈空间

```
OUTPUT_FORMAT("elf32-i386", "elf32-i386", "elf32-i386")
OUTPUT_ARCH(i386)
ENTRY(start)

SECTIONS {
        . = 1M;
        .text : {
            *(.multiboot_header)
                . = ALIGN(8);
                *(.text)
        }

        . = ALIGN(16);
        .data           : { *(.data*) }

        . = ALIGN(16);
        .bss            :
        {
                __bss_start = .;
                _bss_start = .;
                *(.bss)
                __bss_end = .;
        }
        . = ALIGN(16);
        _end = .;
        . = ALIGN(512);
}
```

编译过程说明

- 由Makefile文件描述控制汇编文件和C文件编译生成.o中间文件。
- 链接器链接各.o文件生成可执行文件。

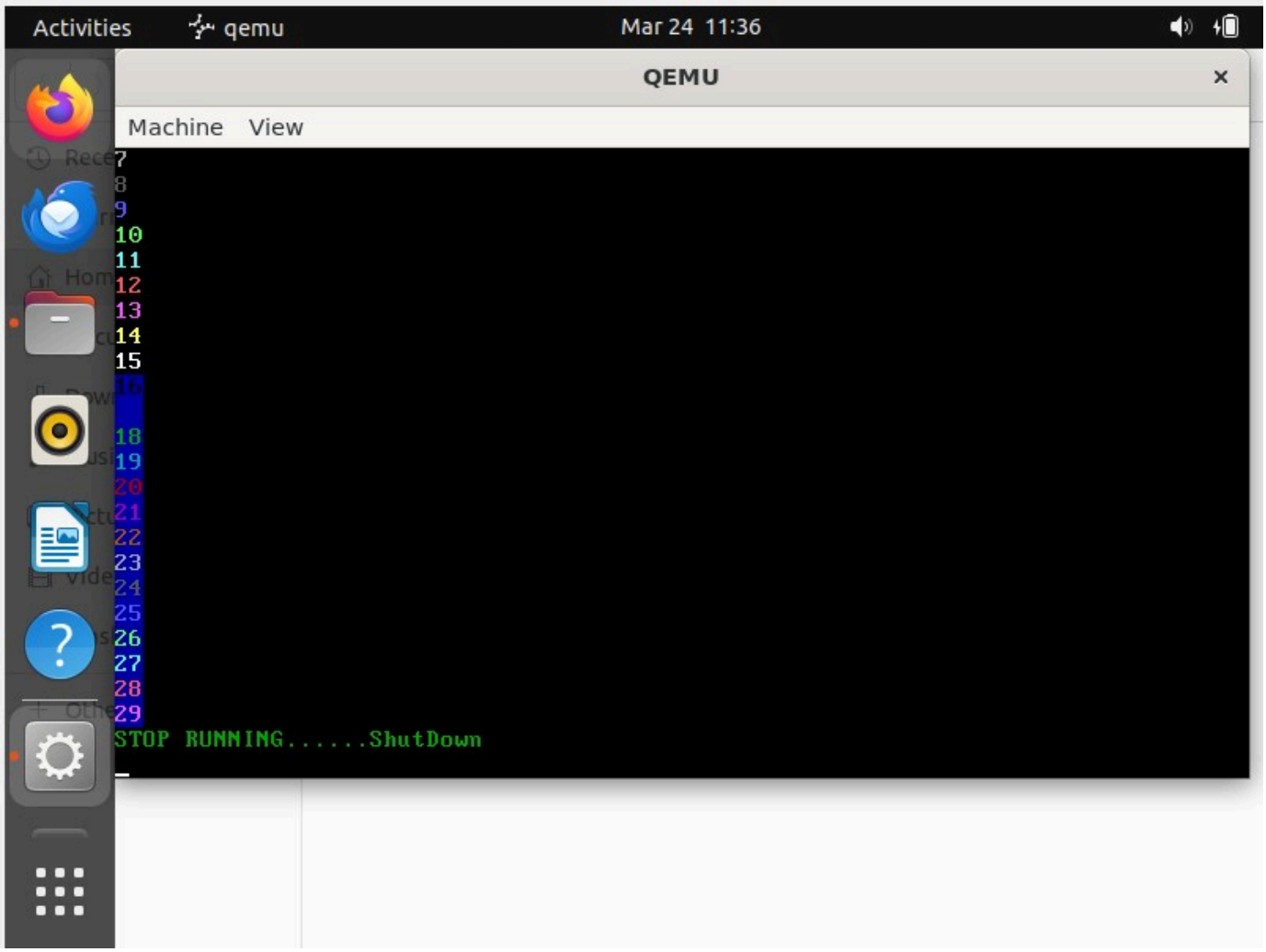
运行和运行结果

- 命令行输入指令运行bash脚本：

```
zz@zz:~/OS/Lab2/src$ ./source2run.sh
./source2run.sh: 2: shell: not found

rm -rf output
ld -n -T myOS/myOS.ld output/multibootheader/multibootHeader.o output/myOS/start32.o output/myOS/osStart.o output/myOS/dev/uart.o output/myOS/dev/vga.o output/
myOS/i386/io.o output/myOS/printk/myPrintk.o output/myOS/printk/vsprintf.o output/userApp/main.o -o output/myOS.elf
make succeed
```

- 得到正确输出结果：



问题和解决方法

- 在实现vga.c的append2screen的滚屏功能时，未注意滚屏之后设置光标导致光标位置错误，debug后解决问题。