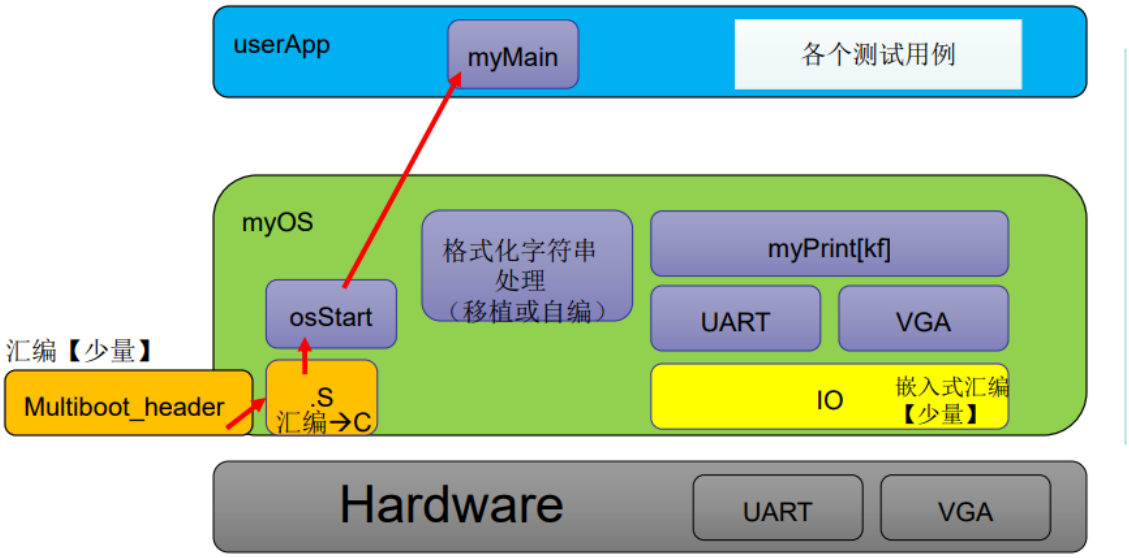
Lab2实验报告

1. 程序框架

整体框架如下图：



源代码目录的文件树如下：

```
└─src
    |  Makefile
    |  source2run.sh
    |
    ├─multibootheader
    |      multibootHeader.S
    |
    ├─myOS
    |  |  Makefile
    |  |  myOS.ld
    |  |  osStart.c
    |  |  start32.S
    |  ├─dev
    |  |      Makefile
    |  |      uart.c
    |  |      vga.c
    |  ├─i386
```

```
|  |      io.c
|  |      io.h
|  |      Makefile
|  └─printk
|          Makefile
|          myPrintk.c
|          vsprintf.c
|
├─output
|          ....
|
└─userApp
        main.c
        Makefile
```

整个程序是由一个我们编写的简单OS和用户程序组成，以下结合程序运行过程和具体文件来解释。

首先，`src\multibootheader\multibootHeader.S`中的`Multiboot_header`部分使得该OS的加载引导支持Multiboot协议，便于使用qemu模拟器来运行OS。之后的汇编指令`call _start`则使程序进入到`src\myOS\start32.S`文件。

在`src\myOS\start32.S`文件中，主要完成内存的初始化。首先设置堆区加上栈区的大小，并且初始化寄存器`esp`和`ebp`；然后将内存中的BSS段全部初始化为0；最后通过汇编指令`call  osStart`进入OS的主体部分，这部分在文件`src\myOS\osStart.c`中实现。

在`src\myOS\osStart.c`中，先清空屏幕，再调用自编的内核级打印API`myPrintk()`来输出`"START RUNNING......\n"`，提示OS启动；然后调用预先约定好的`myMain()`函数端口进入用户程序（位于`src\userApp\main.c`）；返回后输出OS停止的提示信息。

OS分别为内核和用户提供了打印格式化字符串的接口，即`myPrintk()`和`myPrintf()`。这两个函数的实现又依托于格式化字符串的解释程序`vsprintf()`、VGA打印程序`append2screen()`，其中VGA打印涉及光标位置的处理，这又需要使用`src\myOS\i386\io.c`中实现的与端口交互的功能。

2. 功能实现的说明

在给定文件目录下，添加的源代码及相应的阐述如下：

1. src\myOS\start32.S

只需要填写`movl $????, %eax`中缺省的堆区起始地址（或bss区域的结束地址）。查看.ld链接脚本，在该文件中定义的变量是可以在目标文件中使用的，`__bss_start = .`使得`__bss_start`取值为bss区域的初始地址，而`__bss_end = .`则使`__bss_end`指向bss区域末尾。故上述汇编指令确实项应填入`$__bss_end`。

2. 理解src\myOS\i386\io.c

该文件使用C语言内嵌汇编实现了端口的IO操作。以函数`unsigned char inb(unsigned short int port_from)`为例：

`__asm__ __volatile__("inb %w1,%0":"=a"(value) : "Nd"(port_from));`提供标准格式内嵌了一条汇编指令`inb %w1,%0`，其中`%0`寄存器关联到变量`value`，`%w1`寄存器关联到端口号`port_from`，这样就实现了从`port_from`端口获取一个字节到`value`中。

3. src\myOS\dev\uart.c

实现了与 UART 相关的输出。由于已知UART端口号0x3F8，所以直接调用函数`outb()`并提供端口号与输出字符即可实现往UART端口发送一个字节(字符)。至于往UART端口发送字符串，可以多次调用已实现的`uart_put_char()`函数。

4. src\myOS\dev\vga.c

该文件实现了VGA相关的功能，设置全局变量`cur_line`和`cur_column`共同标识光标位置，五个函数功能及实现如下：

1. `void update_cursor(void)`

   功能：通过当前行值`cur_cline`与列值`cur_column`回写光标。

实现：先计算偏移值`offset`，由此得到偏移值的高低位字节；然后通过`outb()`往0x3D5端口分别发送0x0F、0x0E，并分别将地位字节和高位字节发送到0x3D5端口。这样就更新了显存中存储偏移值的两个寄存器，从而更新光标的显示位置。

2. `short get_cursor_position(void)`

   功能：获得当前光标，计算出`cur_line`和`cur_column`的值。

   实现：实现`update_cursor()`的逆过程，其中从0x3D5端口接收字节需要使用`inb()`函数。

3. `void clear_screen(void)`

   功能：清空整个屏幕（并将光标置于开始位置(0,0)）。

   实现：先调用`get_cursor_position()`更新光标行列值，计算出偏移量`offset`；然后从VGA基址0xB8000开始写入`0x00`实现清除，直到到达`0xB8000+2*offset`位置；最后调用`update_cursor()`重置光标位置。

4. `void scrolling(void)`

   功能：往上滚屏一行（用于`append2screen()`).

   实现：滚屏实际上是显示字符的平移，由于VGA屏幕位置与内存地址严格对应，所以将每个显示字符[attr,char]在内存中往前迁移80*2个字节即可（第一行的原字符不需要迁移，迁移后最后一行为空）。

5. `void append2screen(char* str, int color)`

   功能：按指定属性输出字符串str到VGA屏幕。

   实现：首先由参数`color`获取一字节属性参数`attr`，然后获取光标位置，计算偏移量`offset`，再从当前光标位置逐个往后填写字符串str中的字符，直到字符串末尾(即遇到`'\0x0'`)。填写要显示的字符时，如果该字符为换行符，不需要往内存写入如何数据，只需更新光标偏移值到下一行的开头。此外每次循环中还需检测下一个字符的显示位置是否超出屏幕，如果超出，需要先向上滚屏一行，再更新偏移值指向最后一行开头。

5. 理解`src\myOS\printk\myPrintk.c`及编写`src\myOS\printk\vsprintf.c`

理解`src\myOS\printk\myPrintk.c`主要是需要了解C语言中实现可变参数的方式。函数声明中，`...`代表可变参数列表；`va_start(argptr,format);`通过宏的方法使`argptr`指向`format`后的第一个参数，即第一个可变参数；之后，通过`vsprintf(kBuf,format,argptr)`将格式化字符串转化为可直接输出的字符串，调用`append2screen()`即可输出至VGA屏幕。

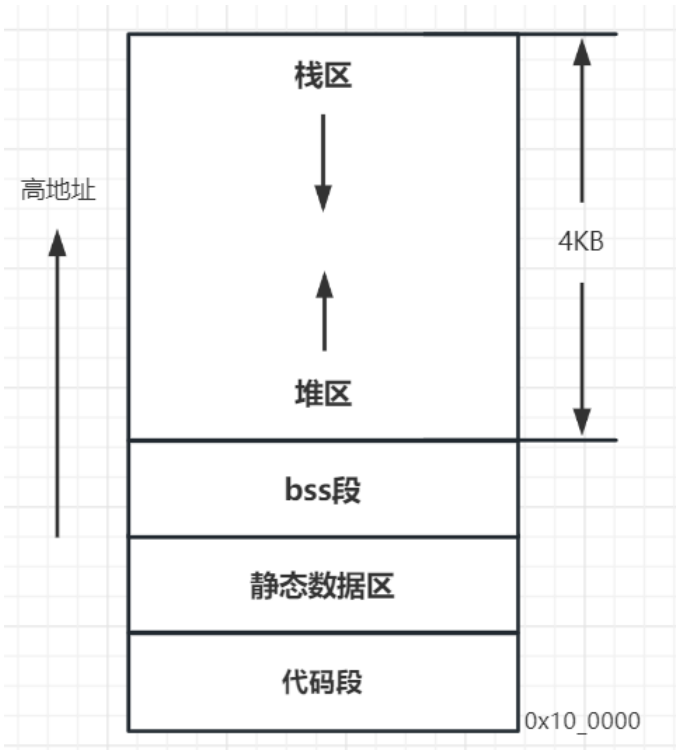
`int vsprintf(char* buf, const char* fmt, va_list argptr)`函数逐个扫描格式化字符串`fmt`中的每个字符，直到遇到结束标志`'\0'`。每次迭代中，

1. 如果遇到`'\'`字符，需要查看下一个，如果下一个字符为`n`，往`buf`中添加一个换行符，其它转移字符的情况类似；
2. 如果当前字符为`%`，查看下个字符，若下个字符为`d`，说明需要匹配可变参数列表指定位置的一个整数，那么先通过`va_arg(argptr, int)`获取整数类型的可变参数的值（并更新`argptr`指向下一个可变参数），再编写函数`itoa()`实现整数到可输出字符串的转化，然后写入`buf`即可，其它类型匹配的情况类似；
3. 除了以上两种情况，都视作可直接显示的字符，直接把字符写入`buf`即可。

迭代结束后记得再在`buf`末尾添加`'\0'`，以使`append2screen()`检测到字符串结尾。

3. 地址空间

由链接脚本`src\myOS\myOS.ld`可以了解程序地址空间的分配，如下图：



4. 编译过程说明

去掉每条指令前的`@`符号可以开启回声，这时执行shell脚本文件（其中包含`make`命令）可以看到所有编译指令，如下：

```
rm -rf output
mkdir -p output/myOS/
gcc -m32 --pipe -Wall -fasm -g -O1 -fno-stack-protector  -c -o output/myOS/start32.o myOS/start32.S
mkdir -p output/myOS/
gcc -m32 -fno-stack-protector -g -c -o output/myOS/osStart.o myOS/osStart.c
mkdir -p output/myOS/dev/
gcc -m32 -fno-stack-protector -g -c -o output/myOS/dev/uart.o myOS/dev/uart.c
mkdir -p output/myOS/dev/
gcc -m32 -fno-stack-protector -g -c -o output/myOS/dev/vga.o myOS/dev/vga.c
mkdir -p output/myOS/i386/
gcc -m32 -fno-stack-protector -g -c -o output/myOS/i386/io.o myOS/i386/io.c
mkdir -p output/myOS/printk/
gcc -m32 -fno-stack-protector -g -c -o output/myOS/printk/myPrintk.o myOS/printk/myPrintk.c
mkdir -p output/myOS/printk/
gcc -m32 -fno-stack-protector -g -c -o output/myOS/printk/vsprintf.o myOS/printk/vsprintf.c
mkdir -p output/userApp/
gcc -m32 -fno-stack-protector -g -c -o output/userApp/main.o userApp/main.c
mkdir -p output/multibootheader/
gcc -m32 --pipe -Wall -fasm -g -O1 -fno-stack-protector  -c -o output/multibootheader/multibootHeader.o mult
ibootheader/multibootHeader.S
ld -n -T myOS/myOS.ld output/multibootheader/multibootHeader.o output/myOS/start32.o output/myOS/osStart.o o
utput/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
```

与`src`目录相对应，先创建`output`目录，然后逐一创建子目录，并将对应的源代码文件编译生成目标文件放入对应目录中，最后根据`src\myOS\myOS.ld`进行链接，生成文件`myOS.elf`。

5. 运行结果

在`src/`目录下，执行shell脚本文件`source2run.sh`，成功后可以在shell中看到`make succeed`的输出，打开的QEMU窗口如下图：