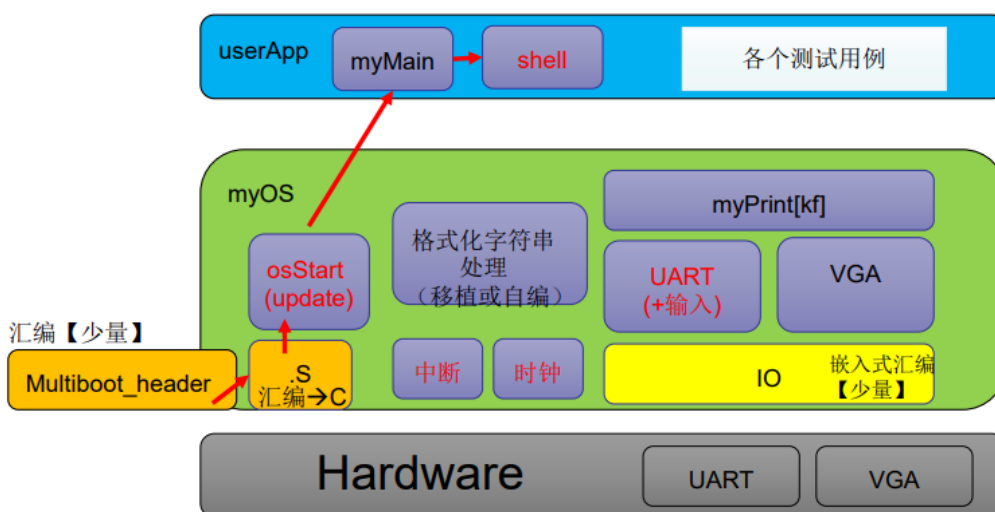


Lab 3 实验报告

I. 程序框图

整体框架如下图：



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，`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`myOS\start32.S`文件。

相比上一阶段的实验，本次实验的`myOS\start32.S`文件完成了更多的任务。除了初始化堆区、栈区和清零BSS数据段，它还为IDT（中断描述符表）分配内存并进行初始化。本次实验中，只处理时钟信号中断，其它中断均使用定义的缺省函数。时钟中断的处理函数和缺省中断的处理函数均在`myOS\start32.S`实现主函数。同样地，最后通过指令`call osStart`进入OS主体(`myOS\osStart.c`)。

在`myOS\osStart.c`中，先初始化i8253时钟源和i8259A中断控制器，然后初始化系统时间，并打开中断，再清空QEMU窗口，输出启动信息后进入用户程序`myMain()`，它位于`\userApp\main.c`。

在 `\userApp\main.c` 中直接启动shell程序 `startshell()`，它定义于 `userApp\startShell.c` 中。

在开中断后，对于运行过程中地每个中断，OS会暂时离开用户代码，查询IDT来找到对应的中断处理程序并执行。本实验中，主要的中断是定时产生的时间信号中断，OS借助该中断来实现墙钟。

II. 实验二代码选择与修改

综合提供代码与本人的实验二代码，联系本次实验的需求进行修改。

注意到当QEMU窗口发生滚动时，最后一行的时钟会被复制到上一行，可以让墙钟单独占最后一行，所以修改 `myOS\include\vga.h` 中 `#define VGA_ROW 25` 为 `#define VGA_ROW 24`。

格式化输出函数改为同时往UART串口的VGA进行输出。为了后续的方便，在本人实验二代码的基础上拓展 `vsprintf` 的功能，使其能处理部分转义字符和 `%c`、`%s`。

III. 中断原理和实现

1. 实现时钟中断的原理：

在配置好 `i8253` 和 `i8259A` 后，`i8253` 就相当于一个特定频率的时钟源，而且输出的时钟信号就当作中断信号挂载在 `i8259A` 上，`i8259A` 作为一个中断控制器，会使 CPU 去执行相应中断号的中断子程序。

2. 初始化 `i8253` 和 `i8259A`

首先说明 `i8253` 和 `i8259A` 的初始化，分别通过 `myOS\dev\i8253.c` 中的 `init8253()` 函数和 `myOS\dev\i8259A.c` 中的 `init8259A()` 实现。`myOS\dev\i8259A.c` 的实现较为简单，只需按照文档用 `outb()` 函数往相应端口发送特定数据即可；而 `init8253()` 中值得注意的是分频参数的设置，我使用的是20Hz的时钟频率，由于 `i8253` 的原频率为1,193,180Hz，所以分频参数为 `0xe90b`。

3. IDT的实现

然后回到`myOS\start32.s`中，它实现了中断机制的重要部分：IDT。

```
IDT:
    .rept 256
    .word 0,0,0,0
    .endr
```

这段代码为IDT分配内存空间，最多256个中断描述符，每个占8个字节。根据中断描述符的结构，`setup_idt`代码段实现了未定义中断的中断描述符（其中基址加上偏移量指向处理程序`ignore_int1`），分别存储于`eax`和`edx`两个寄存器中；然后在`rp_sidt`段用该描述符初始化整个IDT；`call setup_time_int_32`进入的程序段则是将时间信号中断的处理程序设置为`time_interrupt`。

4. 时间信号中断的处理程序

```
time_interrupt:
    cld
    pushf
    pusha
    call tick
    popa
    popf
    iret
```

发生中断时会由用户程序进入内核代码，这需要切换上下文。在调用中断处理程序前通过`pushf`，`pusha`将各种寄存器存入栈中，调用返回后通过`popa`，`popf`进行恢复。需要注意的是暂存和恢复的次序需要一致，即先执行`pushf`，则恢复时需后执行`popf`。

`call tick`进入`myOS\kernel\tick.c`中的`tick()`函数，它用于在QEMU窗口通过VGA显示墙钟。程序中使用全局变量`system_ticks`记录时钟信号中断的次数（初始为0），`HH,MM,SS`分别记录墙钟时间。初始调用需要初始化时间为00:00:00；而每满20个周期秒数加1（注意某一位满的时候需要往前进一位）；每次调用都需使`system_ticks`增加1。代码如下：

```
void tick(void) {
```

```

if (system_ticks == 0) {    //初始化00:00:00(第一次调用)
    HH = MM = SS = 0;
    setwallClock(HH, MM, SS);
}
else if (system_ticks % 20 == 0) {    //20个周期时时间增加1s
    SS++;
    if (SS == 60) MM++, SS = 0;
    if (MM == 60) HH++, MM = 0;
    setwallClock(HH, MM, SS);
}
// else setwallClock(HH, MM, SS);
system_ticks++;
return;
}

```

`setwallClock()` 接收HH,MM,SS三个参数，用于往QEMU窗口右下位置输出时间，通过vga基址加上偏移计算地址，按照`HH:MM:SS`的格式逐个写入即可。

5.未定义中断的处理程序

调用过程类似。处理程序输出"Unknown interrupt"。

IV. shell的实现

通过指令结构体类型的全局数组存储指令，一个数记录指令数目：

```

typedef struct myCommand {
    char name[80];    //命令名(可以作为唯一标识符使用)
    char help_content[200];    //该命令的使用说明
    int (*func)(int argc, char(*argv)[8]);
}myCommand;

myCommand commands[64];    //所有命令
int N;    //命令总数

```

在shell主函数`void startShell(void)`中，先调用指令集初始化函数`initCommands()`，该函数按照确定的顺序将所有内置指令放入`commands`数组；然后进入迭代，每次迭代有如下步骤：

1. 读取用户输出。通过`uart_get_char()`检测用户输入，并回显到QEMU窗口和UART重定向到达的伪终端，直到用户输入回车符。
2. 分割字符串得到token。解析第1步得到的字符串，得到若干个token。
3. 解析token流得到指令。先查看token个数，若为0，说明用户没有输入指令，进入下一次迭代；否则，在`commands`指令数组中搜索第一个token，找到则执行相应的函数，否则进行未定义指令的处理。

函数`startShell()`代码如下：

```
//shell程序的主函数
void startShell(void) {
    initCommands();

    char BUF[256]; //输入缓存区
    int BUF_len=0; //输入缓存区的长度

    int argc;
    char argv[8][8];

    do{
        BUF_len=0;
        myPrintk(0x07, "WJL>>\0");
        //读取命令行输入至BUF缓冲区,并同时通过VGA和UART回显
        while ((BUF[BUF_len] = uart_get_char()) != '\r') {
            myPrintf(0x7, "%c", BUF[BUF_len]);
            BUF_len++; //BUF数组的长度加1
        }
        uart_put_chars(" -pseudo_terminal\0");
        myPrintf(0x7, "\n");

        //将BUF缓冲区中的字符串解析为单词存入argv
        argc = 0;
        int wlen = 0;
        for (int i = 0; i <= BUF_len; i++) {
```

```

        if (BUF[i] == ' ' || BUF[i] == '\r') {    //遇到空格或行尾
            if (wlen == 0) continue;
            else {    //终止某个单词
                argv[argc][wlen] = '\0';
                argc++, wlen = 0;
            }
        }
        else {
            argv[argc][wlen] = BUF[i];
            wlen++;
        }
    }

    //从各个单词解析出指令并执行
    if (argc == 0) continue;    //没有指令
    else {
        int num;
        for (num = 0; num < N; num++) //匹配指令名
            if (strcmp(commands[num].name, argv[0]) == 0)
                break;

        if (num == N) //未定义指令
            myPrintf(0x02, "The cmd is not defined\n");
        else if (commands[num].func(argc, argv) == 0)
            myPrintf(0x02, "Wrong arguments.\n");
    }
}while(1);
}

```

cmd、help指令的函数都较为简单，调用myPrintf()进行输出即可。

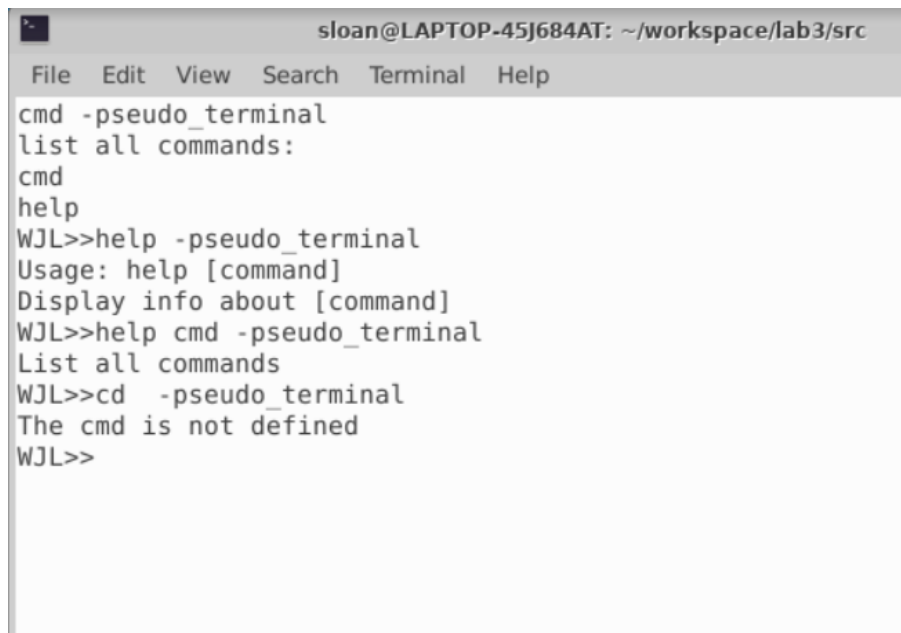
V. 其它

myOS\i386\irq.s中定义了开中断和关中断的过程，在正式启动OS前，应开启中断。

程序结构基本继承了实验二的结构，因此MakeFile文件变化较小，不过需要注意的是大量头文件的加入，-I\${INCLUDE_PATH}指定头文件的优先寻找路径为myOS/include。

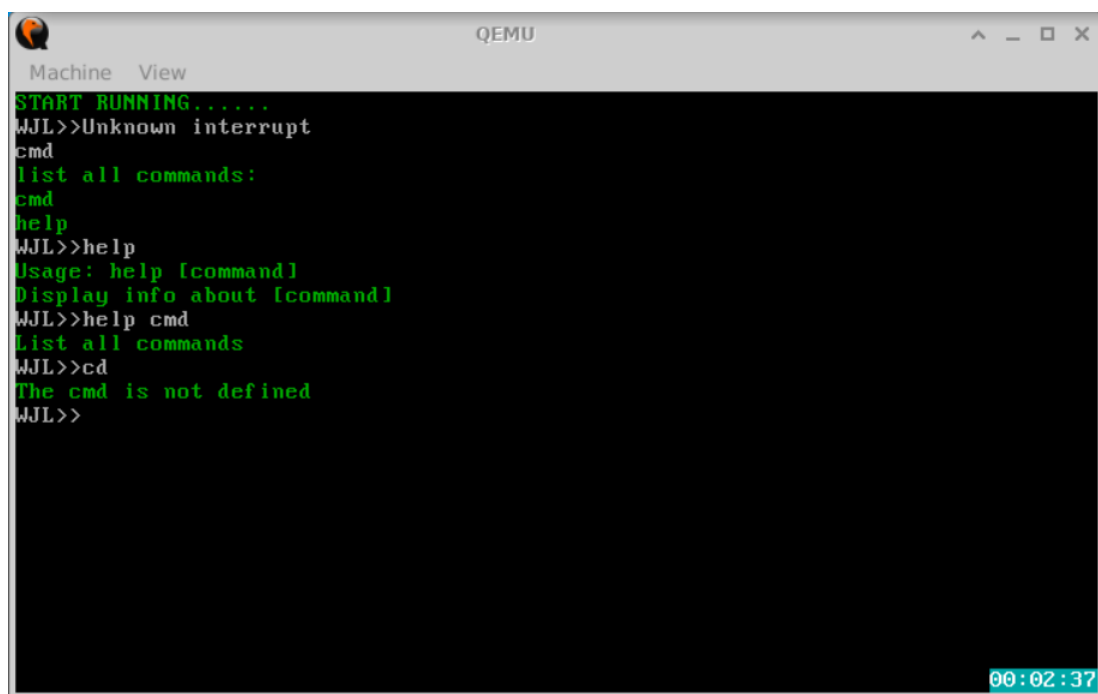
VI. 实验结果

编译并运行程序后，输入指令重定向串口到伪终端，在伪终端输入 `cmd`, `help`, `help cmd`, `cd`(`cd`指令未定义)，伪终端输入如下结果：



```
sloan@LAPTOP-45J684AT: ~/workspace/lab3/src
File Edit View Search Terminal Help
cmd -pseudo_terminal
list all commands:
cmd
help
WJL>>help -pseudo_terminal
Usage: help [command]
Display info about [command]
WJL>>help cmd -pseudo_terminal
List all commands
WJL>>cd -pseudo_terminal
The cmd is not defined
WJL>>
```

QEMU窗口显示结果如下：



```
QEMU
Machine View
START RUNNING.....
WJL>>Unknown interrupt
cmd
list all commands:
cmd
help
WJL>>help
Usage: help [command]
Display info about [command]
WJL>>help cmd
List all commands
WJL>>cd
The cmd is not defined
WJL>>
00:02:37
```