

中国科学技术大学计算机学院

实验报告



实验题目: lab3 shell & interrupt & timer

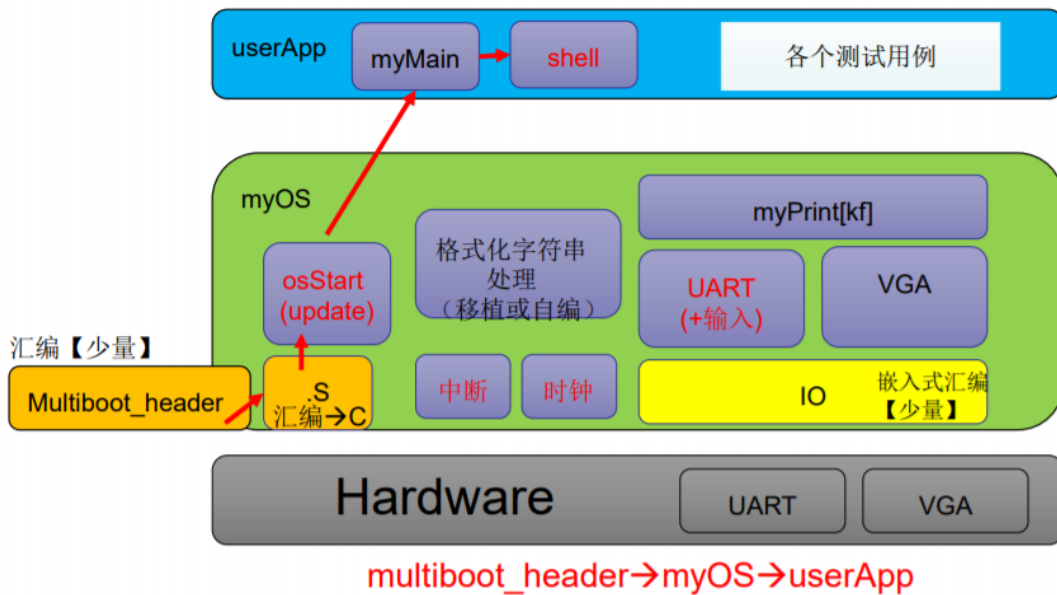
学生姓名: 王舒

学生学号: PB19071472

完成日期: 2021.5.5

一、软件框图

采用老师提供的软件框图进行说明。

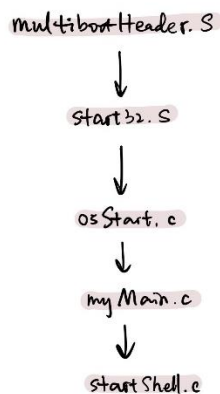


本次实验在 lab2 的基础上，在 OS 中主要添加了中断的处理，其中一个特殊的中断是时钟中断，并根据时钟中断维护了一个 WallClock。在 userApp 中，添加了 shell 程序，它可以提供 cmd 和 help 命令。在 osStart 中，只需要对中断控制器 i8259A、时钟 i8253、WallClock 初始化，调用 myMain 函数，通过 myMain 再进入 shell 即可。

二、主流程及其实现

本次实验的主流程在 lab2 的基础上稍作修改，主要增加了中断处理和 shell 的部分。

首先，用 multibootHeader.S 启动整个程序，在这段代码的最后有一句“call _start”，可以跳入到 start32.S 中的_start 入口，从而用汇编语言为 C 语言准备上下文。其中，在 setup_idtptr 中，call setup_idt 初始化了中断向量表（包括特别初始化时钟的中断），再用 lidt 指令加载 idtptr。然后，利用“call osStart”跳转到 osStart.c 程序中。在 osStart.c 中，完成操作系统的初始化、调用用户态程序 myMain.c 等功能。这样，就转入了用户态运行。在 myMain 中，调用 startShell 进入 shell 运行。



三、主要功能模块及其实现

1. 中断机制及其初始化

① 可编程中断控制器 PIC i8259

通过 `outb` 函数直接写端口，按照芯片协议约定的顺序依次写入相应的数，即可修改 i8259 对中断的控制。

② 中断描述符表 IDT

在 `start32.S` 中的 `data` 部分，可以看到，我们为 IDT 静态分配了一块内存，然后先把所有的中断处理例程设置为调用 `ignoreIntBody` 函数（在 `irqs.c` 中）。再特殊处理时钟中断，每次产生时钟中断时，将调用 `tick` 函数。

③ 寄存器 IDTR

在 `start32.S` 中，我们还通过 `lidt` 指令，把用 `word` 和 `long` 拼接好的 `idtprtr` 加载到特定的寄存器 IDTR 中，这样就能通过访问它了解到 IDT 表的起始地址和表长。

④ 开关中断

在 `irq.S` 中，用汇编写了两个函数 `enable_interrupt` 和 `disable_interrupt`，可以用来开关中断。这样，就可以通过调用这两个函数来开关中断。

2. Tick 的实现

① 可编程间隔定时器 PIT i8253

该模块通过产生晶振来产生一定频率的震荡。按照芯片的协议，我们将特定的数输入到指定端口，再将分频参数输入，并调整 8259 来设定允许时钟中断（读出原先的屏蔽字，将最低位设为 0，再传回去），i8253 就可以产生我们需要的 100HZ 的震荡，也就是每秒触发 100 次时钟中断。

② Tick

`Tick` 函数的作用是每次时钟中断时计数一次，并根据这个计数器来更新 `WallClock`、显示 `WallClock`。这里使用了钩子函数，也就是两个函数指针，调用了在 `WallClock` 中设定好的函数。

```
void tick(void) {
    ms = cnt_tick / 10;
    cnt_tick++;
    if (cnt_tick % 100 == 0) {
        updateClockHook();
        displayClockHook();
    }
}
```

3.WallClock 的实现



在 wallclock 相关的文件中，提供了可以读取当前时间的接口 `getWallClock` 和由用户可以自己设置时间的接口 `setWallClock`。

此外，还有根据 `tick` 更新 wallclock 的 `updateWallClock`，以及控制显示模块的 `displayWallClock`。采用机制与策略相分离的思想，我们不直接用 `tick` 调用这两个函数，而是设定两个函数指针，使他们的默认值指向这两个函数。这样，如果用户想要用 `tick` 调用其他函数，不必在 `tick` 中修改，而是在更高层的 `wallclock` 中，通过调用 `setUpdateWallClockHook` 和 `setDisplayWallClockHook`（两个可以修改函数指针指向的函数），把需要用到函数名传入进去，就可以完成功能。在这个例子中，“机制”是 `tick` 调用特定函数，而“策略”是根据需求修改被调用函数实际的内容。

```
void (*updateClockHook)(void) = updateWallClock;
void (*displayClockHook)(void) = displayWallClock;
```

```
//修改函数指针
void setUpdateWallClockHook(void (*func)(void)) {
    updateClockHook = func;
}

void setDisplayWallClockHook(void (*func)(void)) {
    displayClockHook = func;
}
```

在这次实验中，我们采用钩子的默认值，没有进行修改。

4.shell 的实现

在 shell 中，首先需要做到将输入的命令存到数组里，并在 VGA 和串口回显，这里还要

在 VGA 中能够回车、退格。

在 startShell 中，对于可显示字符，直接显示；对于退格键，在 VGA 输出中写一个 backspaceVGA 函数来处理，该字符不录入数组；对于回车，处理显示后，说明命令已经输入完成，进入处理阶段，即 solve 函数。

这里需要解释一下每条命令行的保存形式。如下：

```
typedef struct {
    char name[80];
    char help_content[200];
    int (*func)(char* argv);
} myCommand;
myCommand command[COMMAND_NUM];
```

其中，函数指针指向每个命令行的处理函数。当检测到输入的命令中有与已有命令匹配的部分时，就进入相应命令的处理函数。这里要注意，对于多余的空格，需要省略掉。

```
void solve(char *str) {
    int i = 0;
    while (str[i] == ' ') {
        i++;
    }
    int flag = 0;
    for (int j = 0; j < COMMAND_NUM; j++) {
        if (myStrcmp(&str[i], command[j].name)) {
            command[j].func(str);
            flag = 1;
            break;
        }
    }
    if(flag==0) {
        myPrintk(0xF, "no such command");
        myPrintk(0xF, "\n");
    }
}
```

在本实验中，我们只有 cmd 和 help 命令。如果需要注册新的命令，需要静态写进去。

对于 cmd 的处理比较简单，列出所有的命令即可。而对于 help，有三种不同的有效情况，需要分别讨论，即 help、help cmd、help help。

5.QEMU 重定向

按照所给的操作要求，下载 screen 指令后，在脚本中改为 -serial pty &，根据运行时给出的伪终端号码，就可以进入交互界面。这里因为在 VGA 和串口都有回显，所以两边都有显示。

四、源代码说明

1.目录组织 (tree 指令打印)

```
|—— Makefile
|—— multibootheader
|   |—— multibootHeader.S
|—— myOS
|   |—— dev //与 IO 有关
|   |   |—— i8253.c
|   |   |—— i8259A.c
|   |   |—— Makefile
|   |   |—— uart.c
|   |   |—— vga.c
|   |—— i386 //与体系结构有关
|   |   |—— io.c
|   |   |—— io.h
|   |   |—— irq.S
|   |   |—— irqs.c
|   |   |—— Makefile
|   |—— include //头函数
|   |   |—— i8253.h
|   |   |—— i8259A.h
|   |   |—— io.h
|   |   |—— myPrintk.h
|   |   |—— string.h
|   |   |—— tick.h
|   |   |—— uart.h
|   |   |—— vga.h
|   |   |—— wallClock.h
|   |—— kernel //与具体功能有关
|   |   |—— Makefile
|   |   |—— tick.c
|   |   |—— wallClock.c
|   |—— lib //字符串处理
|   |   |—— Makefile
|   |   |—— string.c
|   |—— Makefile
|   |—— myOS.ld
|   |—— osStart.c
|   |—— printk //输出打印
|   |   |—— Makefile
|   |   |—— myPrintk.c
|   |—— start32.S
```

```

├── output
│   ├── multibootheader
│   │   └── multibootHeader.o
│   ├── myOS
│   │   ├── dev
│   │   │   ├── i8253.o
│   │   │   ├── i8259A.o
│   │   │   ├── uart.o
│   │   │   └── vga.o
│   │   ├── i386
│   │   │   ├── io.o
│   │   │   ├── irq.o
│   │   │   └── irqs.o
│   │   ├── kernel
│   │   │   ├── tick.o
│   │   │   └── wallClock.o
│   │   ├── lib
│   │   │   └── string.o
│   │   ├── osStart.o
│   │   ├── printk
│   │   │   └── myPrintk.o
│   │   └── start32.o
│   ├── myOS.elf
│   └── userApp
│       ├── main.o
│       └── shell.o
├── README_shell_interrupt_timer.txt
├── test.sh
└── userApp
    ├── main.c
    ├── Makefile
    └── shell.c

```

2.Makefile 组织

在 lab2 的基础上，对于新增的一些文件和文件夹，修改了他们对应的 Makefile。这里在 src 目录下的 Makefile 中增添了 include path，并做了一些修改，这样可以使用头函数。

src 目录下的 Makefile 将各种其他子 Makefile 文件串联起来。在 output/myOS.elf 中，将 OS_OBJS 包含进去，而 OS_OBJS 将 MYOS_OBJS 和 USER_APP_OBJS 包含进去。MYOS_OBJS 中又把 myOS 目录下每个文件夹对应的 Makefile 包含进去。这样就把所有需要编译成可执行文件的内容包含进去了。

五、代码布局说明（地址空间）

本次实验的地址空间与 lab2 相同，ld 文件未作修改。

该 ld 文件将各可执行文件中的 text、data 和 bss 段分别拆开，然后拼接到一起。

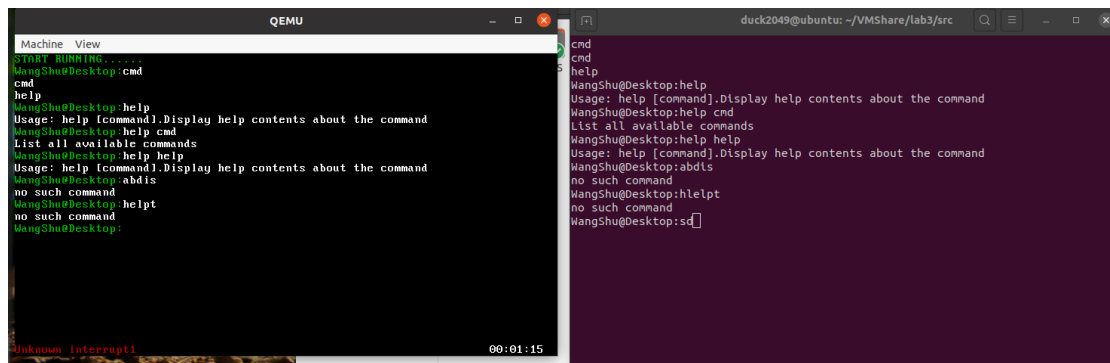
在 .text 段中，先把 multiboot_header 的启动头写进去，然后对齐，再把 text 部分写入。再对齐后，把 data 部分写入。再次对齐，把 bss 段写入，再经过一些对齐的处理。经过编译，就可以按照 ld 文件生成 elf 文件。

六、编译过程说明

Src 目录下的 Makefile 将所有 .S 文件和 .c 文件按照 gcc 编译为可执行文件，然后按照 ld 文件的指示生成 elf 文件。（所有生成的可执行文件和 elf 文件都被放到了 output 文件夹中）。

七、运行结果

用修改后的脚本运行，并进行串口重定向，结果如下



The image shows a QEMU terminal window with two panes. The left pane shows the execution of the 'cmd' command and its help text. The right pane shows the execution of the 'help' command and its help text. The terminal output is as follows:

```
Machine View
START RUNNING...
WangShu@Desktop: cmd
cmd
help
WangShu@Desktop: help
Usage: help [command].Display help contents about the command
WangShu@Desktop: help cmd
List all available commands
WangShu@Desktop: help help
Usage: help [command].Display help contents about the command
WangShu@Desktop: abdis
no such command
WangShu@Desktop: hlept
no such command
WangShu@Desktop:
Unknown interrupt! 00:01:15
```

```
cmd
cmd
help
WangShu@Desktop: help
Usage: help [command].Display help contents about the command
WangShu@Desktop: help cmd
List all available commands
WangShu@Desktop: help help
Usage: help [command].Display help contents about the command
WangShu@Desktop: abdis
no such command
WangShu@Desktop: hlept
no such command
WangShu@Desktop: sd
```

即在右边串口输入，在左边 VGA 中和右边串口输出。当产生非时钟中断时，会有相应的提示。

八、遇到的问题和解决方案

做 lab 已经做麻了，不管遇到什么困难都能微笑面对：)

首先，本次实验的文件目录又复杂了许多，对于许多内容之间的联系很难把握，导致在刚开始做的时候一头雾水不知所措，连抄都不知道从哪里抄到哪里。后来揣摩研究多日，终于搞清楚了各个部件之间的关系。

在开始着手写代码后，一个难点是 tick 和 wallclock 之间怎样联系、怎样利用 hook 机制来实现，这一点的理解对我来说很困难。由于我对函数指针也不是很熟悉，写起来更是雪上加霜，只能从头开始学（以及求助可靠的大腿），最后才搞明白这是在干嘛。

另一个难点是 shell。要对输入的字符串做恰当的处理，还要解析字符串，做一个不对称的匹配。其实总体难度不大，但是因为情况比较多，写起来很麻烦，不是什么特别愉悦的体验。关于为什么不能直接用自带的 string.h 库函数，我猜测是因为这个库的链接方式不是静态的，按照我们现有的 ld 文件没法处理这种东西。（只是大胆口胡）

三次 lab 做完，我感觉我变强了:)，奇怪的知识增长了很多。或许是因为奇怪的困难遇到太多，实验报告里这个模块越来越短了。越来越多的问题通过百度和自己摸索都可以解决掉（当然还有靠谱的助教哥哥，/heart），也不再像一开始遇到什么问题都觉得昏天黑地的，虽然过程还是很痛苦。