

中国科学技术大学计算机学院
《操作系统原理与设计》实验报告



实验题目：lab3_Switch2Shell

学生姓名：胡毅翔

学生学号：PB18000290

完成日期：2020 年 4 月 4 日

计算机实验教学中心制

2019 年 09 月

实验目的

1. 实现简单的 shell 程序，提供 cmd 和 help 命令，允许注册新的命令。
2. 实现中断机制和中断控制器 i8259A 初始化。
3. 实现时钟 i8253 和周期性时钟中断。
4. 实现 VGA 输出的调整：
 - 左下角：时钟中断之外的其他中断，一律输出“Unknown interrupt1”。
 - 右下角：从某个时间开始，大约每秒更新一次，格式为：HH:MM:SS。
5. 提供脚本完成编译和执行。

实验环境

1. PC 一台
2. Windows 系统
3. Ubuntu
4. QEMU
5. Xserver

软件框图

本实验的软件框图如图 1 所示。

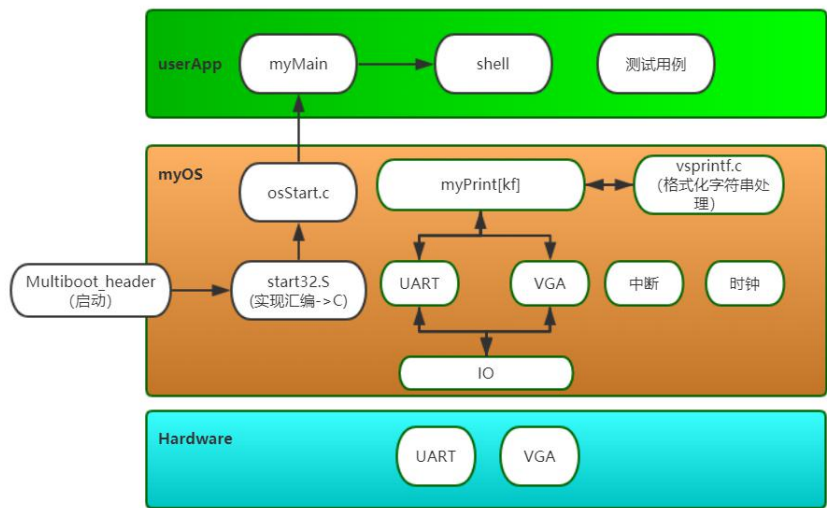


图 1

软件层次分为 multiboot_header、myOS 和 userApp 三部分。multiboot_header 为系统启动部分，系统启动后进入 myOS，在 osStart.c 中调用 myMain.c 进入 userApp 部分。若 userApp 部分，myMain 及其调用的程序需要输出字符串时，通过 myPrintk/f，将经 vsprintf 处理后的字符串通过 IO 接口，在 VGA 和 UART 上完成输出。在 myMain 中，调用 startShell 函数，进入 shell 程序。

主流程

本实验的主流程如下图（图 2）所示：



图 2

1. 在 `multiboot_header` 中完成系统的启动。
2. 在 `start32.S` 中准备好上下文，最后调用 `osStart.c` 把进入 `c` 程序。
3. 在 `osStart.c` 中完成清屏，初始化 `8259`，初始化 `8253`，设置 `WallClock` 初始时间等初始化操作，调用 `myMain`，进入 `userApp` 部分。
4. 运行 `myMain` 中的代码，进入 `shell` 程序。
5. 等待用户输入命令，并对命令进行处理。

主要功能模块及其实现

中断机制及其初始化

该功能模块用于建立中断机制，并完成初始化。

首先，为中断描述符表分配(IDT)一块内存，并将所有中断处理程序初始化为合适的缺省处理函数，例如 `ignore_int1`，使得中断调用 `ignoreIntBody`，在屏幕左下角，输出 “Unknown interrupt1”。

而后，完成寄存器 `IDTR` 的初始化。

之后，初始化中断控制器 `PIC i8259`，分别对主片，从片初始化，设置从几号中断开始，设置接入和输出引脚，设置中断结束方式，同时设置屏蔽所有中断源。

最后，调用 `enable_interrupt` 函数，启动中断。



图 3

Tick 的实现

该功能模块用于产生周期性时钟中断，主要目的是完成 `i8253` 的初始化，实现 `tick` 函数。

`8253` 的频率为 `1,193,180Hz`，先除以 `100`，得到分频参数，将 `0x34` 写入端口 `0x43`，再分别将分频参数的低字节，高字节，写入端口 `0x40`，最后通过 `8259` 控制，将最低位置 `0`，允许时钟中断。

时钟中断会调用 `tick` 函数，`tick` 函数中的计数器初始置为 `0`，每次调用时，计数器加一，当调用满一百次时，计数器复位为 `0`，并调用 `maybeUpdatewallClock` 函数，更新时钟。

示意图如图 4.



图 4

wallClock 的实现

该功能模块用于时钟控制，及与时钟相关的钩子函数，主要目的是实现 `setWallClock`，`getWallClock`，`setWallClockHook` 以及 `maybeUpdateWallClock` 函数。

时钟的时分秒分别用全局变量 `hh`，`mm`，`ss` 表示，因此 `setWallClock` 和 `getWallClock` 函数，只需对这三个全局变量进行赋值或取值即可。

为实现设置钩子函数，建立了函数指针数组 `func_array`，及表示函数指针数组中函数个数全局变量 `No_func_in_Hook`，每次设置时，只需将函数指针赋值到数组中，并将 `No_func_in_Hook` 加一即可。

在 `maybeUpdatewallClock` 函数中，根据 `No_func_in_Hook`，逐一调用钩子函数，最后根据时分秒的前一状态，更新下一时刻的时分秒，并调用显示函数 `wallclock_append`，刷新时钟。

示意图如图 5。

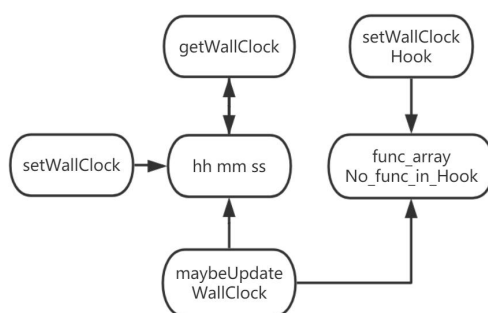


图 5

Shell 的实现

该模块的主要功能是实现简单的 `shell` 程序，与用户进行交互。

本次主要实现 `startShell` 函数，以及 `cmd` 和 `help` 指令。

在 `startShell` 函数中，完成以下内容：

1. 输出提示内容。
2. 等待用户输入。
3. 若输入为“Enter”，对缓冲区中的字符串进行处理，判断是否为已定义的命令：
 - ① 若是，输出命令描述，调用命令处理函数，转 1。
 - ② 否则，输出错误信息，转 1。

- 4. 若输入为“BackSpace”，执行相应操作，转 2。
- 5. 若输入为其他字符，转 2。

PS: TODO: 上下左右等其他输入控制。

在 help 命令处理函数中，判断 help [cmd]中的 cmd 是否为已定义命令，且该指令有 help_func，则调用该命令的帮助函数，否则，输出错误信息。

在 cmd 命令处理函数中，将命令列表中的命令及命令描述，逐一输出。

源代码说明

目录组织

目录组织如图 6 所示：

```
brx@DESKTOP-LPIA2G2: ~/Github/os2020-labs/lab3/src$ tree
.
├── Makefile
├── README
├── shell
├── interrupt_timer.txt
├── start32
├── multibootHeader.S
├── bios
├── i8253
├── i8259A
├── uart
├── vga
├── io
├── irq
├── tick
├── wallClock
├── string
├── myOS.ld
├── osStart
├── myPrintk
├── types
├── vsprintf
├── start32.S
├── source2img.sh
├── main
├── shell
└── 3 directories, 32 files
```

图 6

Makefile 组织

Makefile 组织结构如图 7 所示：

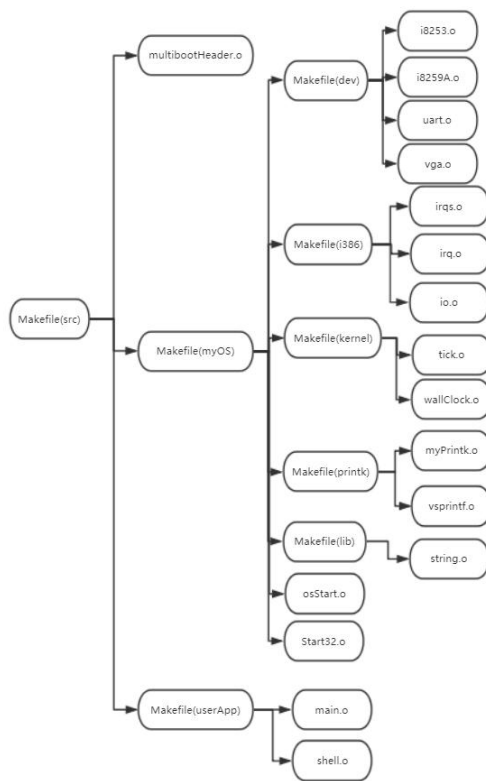


图 7

代码布局说明（地址空间）

从物理内存 1M 的位置开始放代码和数据，前面 12 个字节为 `multiboot_header`，向后对齐 8 个字节，放代码。再向后对齐 16 个字节，用于放初始化的数据（数据段）。在数据段之后，再向后对齐 16 个字节。之后为 BSS（Block Started by Symbol）段，用于存放程序中未初始化的全局变量和静态变量。并在 BSS 段后，再向后对齐 16 个字节。剩余部分为堆栈段。

示意图如图 8 所示：

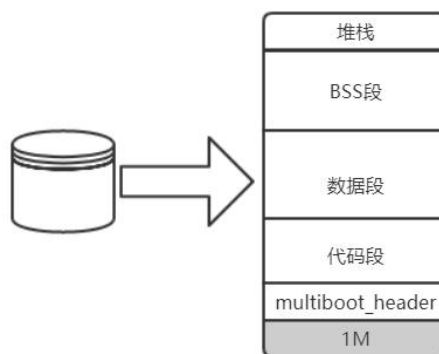


图 8

编译过程说明

`ASM_FLAGS=-m32 --pipe -Wall -fasm -g -O1 -fno-stack-protector`

-m32 :用 32 位机器的编译器来编译这个文件

--pipe:使用管道代替编译中临时文档

-Wall :打开警告选项

-fasm :识别 asm 关键字

-g :使用调试器 GDB

-O1 :优化生成代码

-fno-stack-protector :停止使用 stack-protector 功能

`C_FLAGS = -m32 -fno-stack-protector -fno-pic -fno-builtin -g`

-fno-pic :-fpic 选项为“如果支持这种目标机,编译器就生成位置无关目标码.适用于共享库(shared library) ” -fno-pic 则反之。此选项使得一个 warning 消失, 但通过所搜索到的资料未能了解到消除这一 warning 的原理

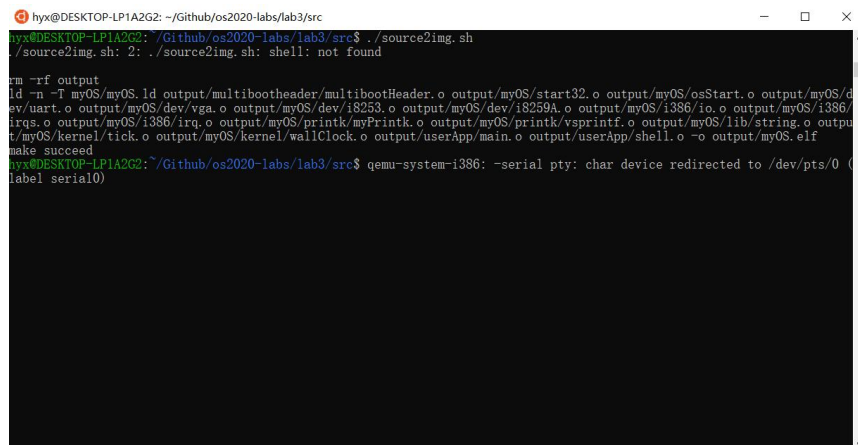
-fno-builtin :不使用 C 语言的内建函数

编译生成 multiHeader.o、osStart.o、start32.o、uart.o、vga.o、io.o、myPrintk.o、vsprintf.o 及 main.o 等目标文件, 由链接器链接生成 myOS.elf 文件。

运行和运行结果说明

输入 ./source2img.sh 指令后, 编译, 链接, 生成 myOS.elf 文件并根据运行指令“qemu-system-i386 -kernel myOS.elf -serial pty &”运行之。运行结果如下图:

Ubuntu 中:



```
hyx@DESKTOP-LP1A2G2: ~/Github/os2020-labs/lab3/src
./source2img.sh
./source2img.sh: 2: ./source2img.sh: 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/dev/i8253.o output/myOS/dev/i8259A.o output/myOS/i386/io.o output/myOS/i386/irqs.o output/myOS/i386/irq.o output/myOS/printk/myPrintk.o output/myOS/printk/vsprintf.o output/myOS/lib/string.o output/myOS/kernel/tick.o output/myOS/kernel/wallClock.o output/userApp/main.o output/userApp/shell.o -o output/myOS.elf
make succeed
hyx@DESKTOP-LP1A2G2:~/Github/os2020-labs/lab3/src$ qemu-system-i386: -serial pty: char device redirected to /dev/pts/0 (label serial0)
```

图 9

QEMU 中:

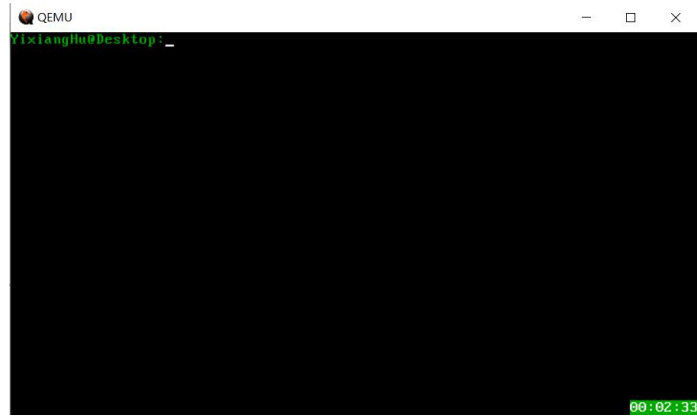


图 10

在 Ubuntu 中输入“`sudo screen /dev/pts/0`”，通过 Ubuntu，输入 shell 中的命令，运行结果如图。

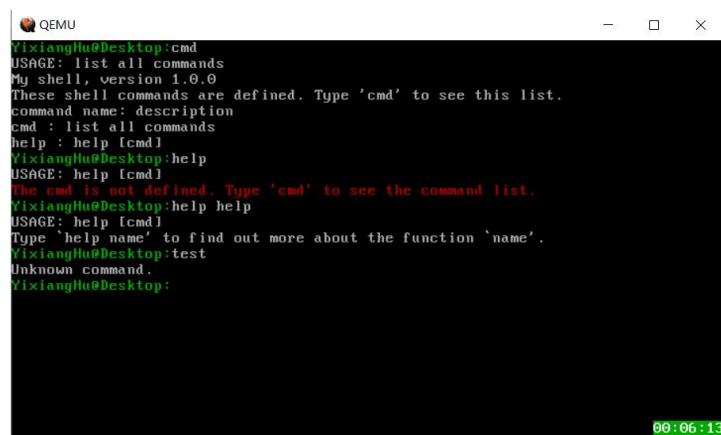


图 11

遇到的问题 and 解决方案说明

1. 编译时出现报错“对‘_GLOBAL_OFFSET_TABLE_’未定义的引用”。

解决方案：在 `src` 目录下的 `Makefile` 文件的 `CFLAGS` 变量中添加 `-fno-pic`。

2. 编译时出现 `warning: conflicting types for built-in function`。

解决方案：在 `src` 目录下的 `Makefile` 文件的 `CFLAGS` 变量中添加 `-fno-builtin`。

3. 除零操作后，程序剩余部分无法运行

解决方案：除零操作可产生中断，但会造成死循环。

PS：除零操作可导致中断，并调用 `ignoreIntBody`，其他中断并未测试。