OS_Lab_03 Report

PB21111686_赵卓

实验题目

- Start2Shell

实验目的

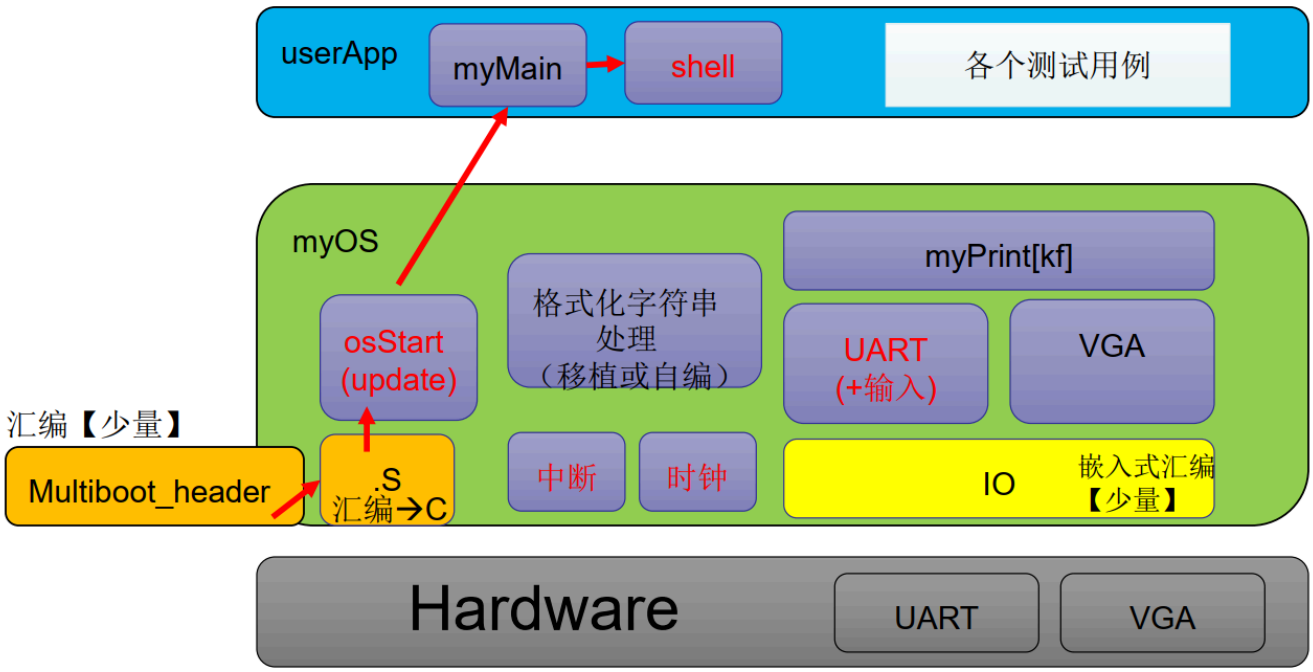
- 在之前搭建的内核基础上添加对时钟和中断的支持，同时添加简单的Shell。

实验内容

- 在给出的框架基础上，补全框架空缺的文件，包括Start32.S，i8253.c，i8259A.c，irq.S，tick.c，wallClock.c，startShell.c。

实验框架

- 本次实验框图如下：



- 框图概述：本次实验结构与实验二基本相同，在UserApp模块增加了Shell模块，同时在myOS内核增加了中断和时钟模块。

实验模块实现

- Start32.S
  Start32.S模块缺少time_interrupt和ignore_int1两部分，以实现中断机制。按照实验讲义的提示，补全如下：
  - time_interrupt处理时钟中断，调用tick()函数实现：

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

  - ignore_int1调用ignoreIntBody()函数，补全如下：

```
ignore_int1:
  cld
  pusha
  call ignoreIntBody
  popa
  iret
```

- i8253.c和i8259A.c
  i8253和i8259A都是可编程芯片，只需要按照PPT上给出的配置要求在对应位置输入对应内容即可：
  - 对于i8253：
    - 配置要求如下：

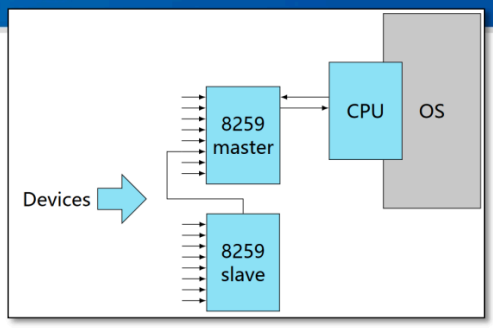
- 需要对PIT: i8253进行初始化，接口：void init8253(void)
  - 端口地址 0x40~0x43；
  - 14,3178 MHz crystal
    4,772,727 Hz system clock
    1,193,180 Hz to 8253
  - 设定时钟中断的频率为100HZ，分频参数是多少？
  - 初始化序列为：
    - 0x34 ==》端口0x43（参见控制字说明）
    - 分频参数==》端口0x40，分两次，先低8位，后高8位
  - 通过8259控制，允许时钟中断
    - 读取原来的屏蔽字，将最低位置0

- 根据配置编写代码将对应端口内容写入即可，其中时钟中断的频率为100Hz，而8253的频率约为1193180Hz，因此分频参数为1193180/100：

```
void init8253(void)
{
  short temp = 1193180 / 100;
  outb(0x43, 0x34);
  outb(0x40, (unsigned char)temp);
  outb(0x40, (unsigned char)(temp >> 8));
  outb(0x21, inb(0x21) & 0xFE);
 }
```

- 对于i8259A:
  - 配置要求如下：

可编程中断控制器 PIC i8259



- 两个8259级联
- 需要对i8259进行初始化
  接口：void init8259A(void);
  - 端口地址：主片0x20~0x21
    从片0xA0~0xA1
  - 屏蔽所有中断源：0xFF==》0x21和0xA1
  - 主片初始化：ICW1：0x11 ==》0x20
    ICW2：起始向量号0x20 ==》0x21
    ICW3：从片接入引脚位 0x04 ==》0x21
    ICW4：中断结束方式 AutoEOI 0x3 ==》0x21
  - 从片初始化： ICW1：0x11 ==》0xA0
    ICW2：起始向量号0x28==》0xA1
    ICW3：接入主片的编号0x02==》0xA1
    ICW4： 中断结束方式 0x01==》0xA1
- 读/写i8259的当前屏蔽字节：即读写主片0x21或从片0xA1

- 根据配置要求编写代码在对应端口写入内容即可：

```
void init8259A(void)
{
  outb(0x21, 0XFF);
  outb(0xA1, 0xFF);

  outb(0x20, 0x11);
  outb(0x21, 0x20);
  outb(0x21, 0x04);
  outb(0x21, 0x3);

  outb(0xA0, 0x11);
  outb(0xA1, 0x28);
  outb(0xA1, 0x02);
  outb(0xA1, 0x01);
 }
```

- irq.S
  irq.S为中断的开关，根据实验讲义提示，irq.S编写如下：

```
.text
.code32
_start:
  .globl enable_interrupt
 enable_interrupt:
    sti
    ret

  .globl disable_interrupt
 disable_interrupt:
    cli
    ret
```

- tick.c和wallClock.c
  - 对于tick.c：
    - tick函数由i8253引起的时钟中断而引起的中断子程序time_interrupt处理中调用，由于 tick 函数的调用是固定100Hz的，所以我们可以借此来进行时钟的输出。使用全局变量system_ticks来记录tick发生的次数。
    - 同时根据实验讲义的提示，我们可以采用hook机制实现tick函数，对于hook机制，我们需要有一些了解：

      > hook机制是一种编程手段，它可以在不修改源代码的情况下，通过在应用程序的关键步骤之前或之后插入特定的代码，来实现对应用程序的功能增强、维护或检测。它类似于一种“挂钩”机制，能够在应用程序运行时，将第三方软件插入原来的程序中，从而实现功能拓展。Hook机制是一种常用的API和操作系统级别的编程技术。

    - 简单来说，hook机制就是通过函数指针数组方便地调用所需要的函数，类似API的调用。我们将tick函数调用时需要的函数指针都加入到这个数组中，在tick函数调用时直接调用即可。这样增强了代码的可维护性和模块性。
    - 编写代码如下：

```
#define TICK_HOOK_VOL 10

int system_ticks = 0;
int hook_func_num = 0;
void (*hook_list[TICK_HOOK_VOL])(void);

void setWallClockHook(void (*func)(void))
{
   hook_list[hook_func_num++] = func;
}

void tick(void)
{
  system_ticks++;
  for (int i = 0; i < hook_func_num; i++)
      hook_list[i]();
 }
```

- 对于wallClock.c：
  - 有了tick.c的铺垫，现在wallClock.c只需要维护tick需要的函数即可，即设置墙钟，显示墙钟，更新墙钟。
  - 显示墙钟，只需要将对应的时间通过vga输出在界面上即可。我们用一个12位字符数组保存时间，选择最后一行的最后12个位置，用亮绿色将时间显示出来。为了方便地显示时间，我们在vga.c中增加了一个append2screen_clock函数，直接将时间显示出来而不会改变光标位置：

```
void displayWallClock(void) {
char str[12];
str[0] = '0' + HH / 10;
str[1] = '0' + HH % 10;
str[2] = ' ';
str[3] = ':';
str[4] = ' ';
str[5] = '0' + MM / 10;
str[6] = '0' + MM % 10;
str[7] = ' ';
str[8] = ':';
str[9] = ' ';
str[10] = '0' + SS / 10;
str[11] = '0' + SS % 10;
append2screen_clock(str, 0x2f, 25 - 1, 80 - 12);
}
```

  - 更新墙钟，只需要根据时分秒的进位关系在边界时更新改变之后输出即可：

```
void updateWallClock(void) {
if (system_ticks % HZ_INTERRUPT != 0)
    return;

SS = (SS + 1) % 60;

if (SS == 0)
    MM = (MM + 1) % 60;

if (MM == 0 && SS == 0)
    HH = (HH + 1) % 24;

displayWallClock();
}
```

- 设置墙钟，同时将更新墙钟的函数加入函数指针数组，然后显示墙钟：

```
void setWallClock(int h, int m, int s)
{
  if (h < 0)
      HH = 0;
  else if (h > 23)
     HH = 23;
  else
      HH = h;

  if (m < 0)
      MM = 0;
  else if (m > 59)
      MM = 59;
  else
      MM = m;

  if (s < 0)
      SS = 0;
  else if (s > 59)
      SS = 59;
  else
      SS = s;

  setWallClockHook(updateWallClock);
  displayWallClock();
}
```

- startShell.c

  根据框架，shell模块用大小位256的字符数组作为缓冲区存储输入的指令，同时给出了得到输入指令的代码。只需要实现cmd，help，help cmd三个指令的识别并输出对应的提示，同时识别其他非法指令。因此我们还需要实现分割指令，执行指令的函数。为了提高代码的模块化，我们将得到输入指令的代码也作为一个函数来实现。

  - 为了得到输入的指令，我们只需要从uart的输入中取出输入值存入BUF数组，在识别到换行符时停止。同时根据实验要求将指令回显在vga上，因此在框架提供的得到输入功能代码上稍作修改得到如下函数：

```
void get_input_cmd(char *buf)
{

 char *str = buf;
 unsigned char input;

 while ((input = uart_get_char()) != '\r')
 {
     if (input == '\r')
         break;
     else
     {
         *str++ = input;
         myPrintk(0x7, "%c", input);
         uart_put_char(input);
     }
 }
 *str = '\0';

 myPrintk(0x7, " -pseudo_terminal\0");
 myPrintk(0x7, "\n");
 uart_put_chars("\r\n");
}
```

  - 分割指令，由于我们输入的指令之间有空格隔开，因此我们要将其分割成有效的部分。只需将BUF中的内容识别，在空格处分开，非空格内容分组存储即可：

```
int devide_command(const char *str, char **str_devided)
{
 int devidednum = 0;
 int i = 0;

 while (*str++ == ' ')
     ;
 str--;

 while (*str)
 {
     if (*str == ' ')
     {
         while (*str++ == ' ')
             ;
         if (*--str)
         {
             str_devided[devidednum++][i] = '\0';
             i = 0;
         }
     }
     else
         str_devided[devidednum][i++] = *str++;
 }
 str_devided[devidednum++][i] = '\0';
 return devidednum;
}
```

- 执行指令，在识别到指令并分组之后，我们需要识别不同指令并执行（在本实验中即输出对应的内容），即执行cmd和help（包括help和help cmd）指令。由于没有glibc库，我们可以编写一个自己的string库并实现需要的函数便于我们进行字符串比较。实现如下：

```
int execute_command(int argc, char *argv[])
{
 if (!myStrlen(argv[0]))
     return 1;
 for (int i = 0; myStrcmp(cmds[i].name, "NULL") != 0; i++)
 {
     if (!myStrcmp(argv[0], cmds[i].name))
     {
         cmds[i].func(argc, argv);
         return 0;
     }
 }
 myPrintk(0x7, "The cmd is not defined\n");
   return 1;
 }

 int func_help(int argc, char *argv[])
 {
   if (argc == 1)
   {
     myPrintk(0x2, "Usage: help [command]\nDisplay info about[command]\n");
     return 1;
   }

 for (int i = 0; myStrcmp(cmds[i].name, "NULL") != 0; i++)
 {
     if (!myStrcmp(argv[1], cmds[i].name))
     {
         myPrintk(0x2, "%s", cmds[i].help_content);
         return 1;
     }
 }

 myPrintk(0x7, "The cmd is not defined\n");
 return 0;
 }

 int func_cmd(int argc, char *argv[])
 {
   for (int i = 0; myStrcmp(cmds[i].name, "NULL") != 0; i++)
       myPrintk(0x2, "%s\n", cmds[i].name);
   return 0;
 }

 int myStrcmp(const char *str1, const char *str2)
 {
```

```
    while (*str1 && *str2 && *str1++ == *str2++)
        ;
    if (!*str1 || !*str2)
        return *str1 - *str2;
    else
        return *--str1 - *--str2;
}

int myStrlen(const char *str)
{
    int len = 0;
    while (*str++)
        len++;
    return len;
}
```

- 实现各函数后，在startShell中调用即可：

```
void startShell(void)
{
 char BUF[256];
 char agrv[8][8];
 char *cmd[8];
 int argc;
 for (int i = 0; i < 8; i++)
     cmd[i] = agrv[i];

 while (1)
 {

     myPrintk(0x7, "Student>>");

     get_input_cmd(BUF);

     argc = devide_command(BUF, cmd);

     execute_command(argc, cmd);
 }

 return;
}
```

源代码说明

- 代码布局如下：

```
src
├──multibootheader
│  └──multibootHeader.S ├──myOS
│  ├──dev
│  │  ├──i8253.c
│  │  ├──i8259A.c
│  │  ├──uart.c
│  │  └──vga.c
│  ├──i386
│  │  ├──io.c
│  │  ├──irq.S
│  │  ├──irqs.c
│  ├──include
│  │  ├──i8253.h
│  │  ├──i8259A.h
│  │  ├──io.h
│  │  ├──irqs.h
│  │  ├──myPrintk.h
│  │  ├──string.h
│  │  ├──tick.h
│  │  ├──wallClock.h
│  │  ├──uart.h
│  │  ├──vsprintf.h
│  │  └──vga.h
│  ├──kernel
│  │  ├──tick.c
│  │  └──wallClock.c
│  ├──osStart.c
│  ├──printk
│  │  └──myPrintk.c
│  ├──string
│  │  └──string.c
│  └──start32.S
└──userApp
   ├──main.c
   └──startShell.c
```

代码布局

- 代码布局由myOS.ld规定：

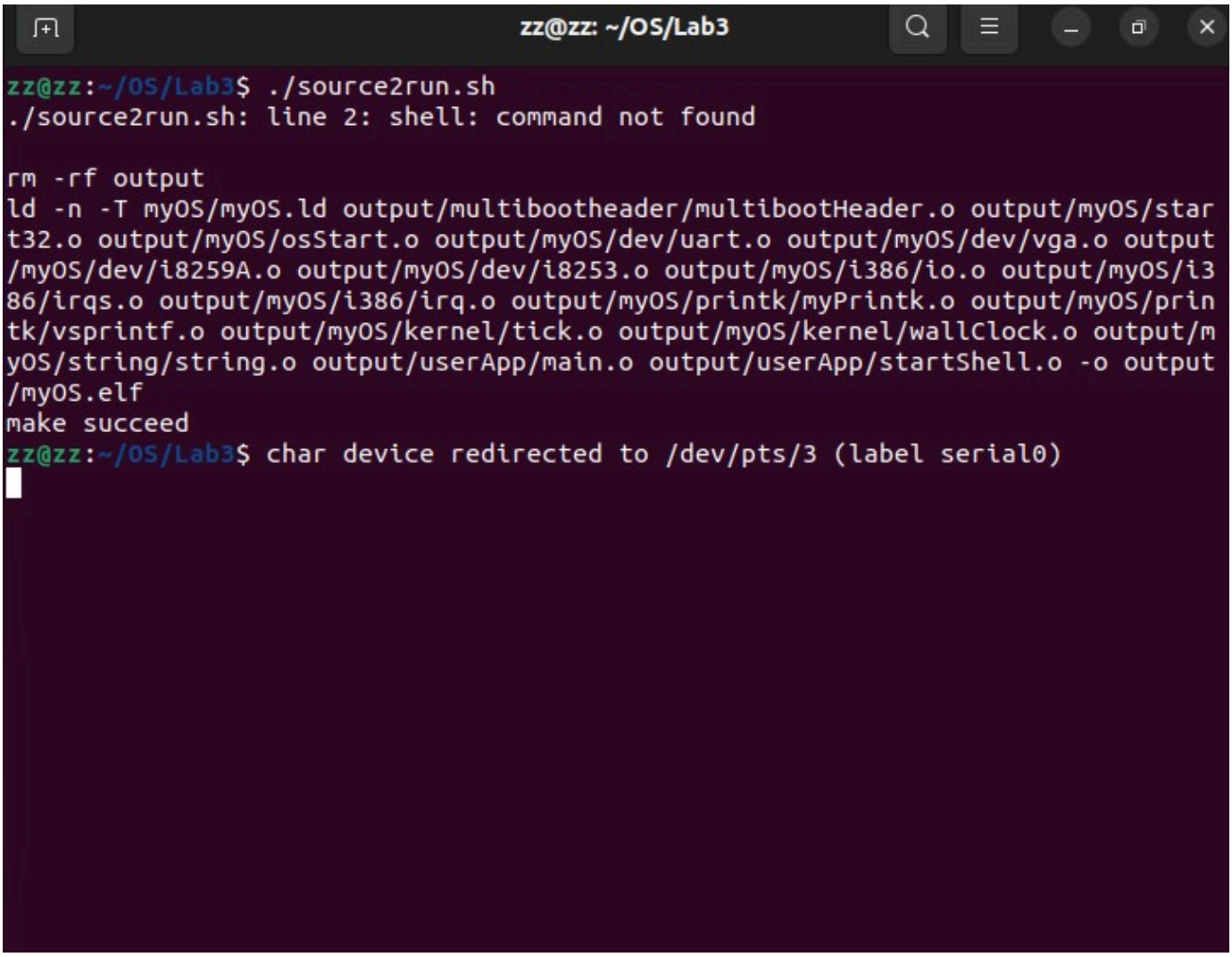
```
OUTPUT_FORMAT("elf32-i386", "elf32-i386", "elf32-i386")
OUTPUT_ARCH(i386)
ENTRY(start)

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);
}
```

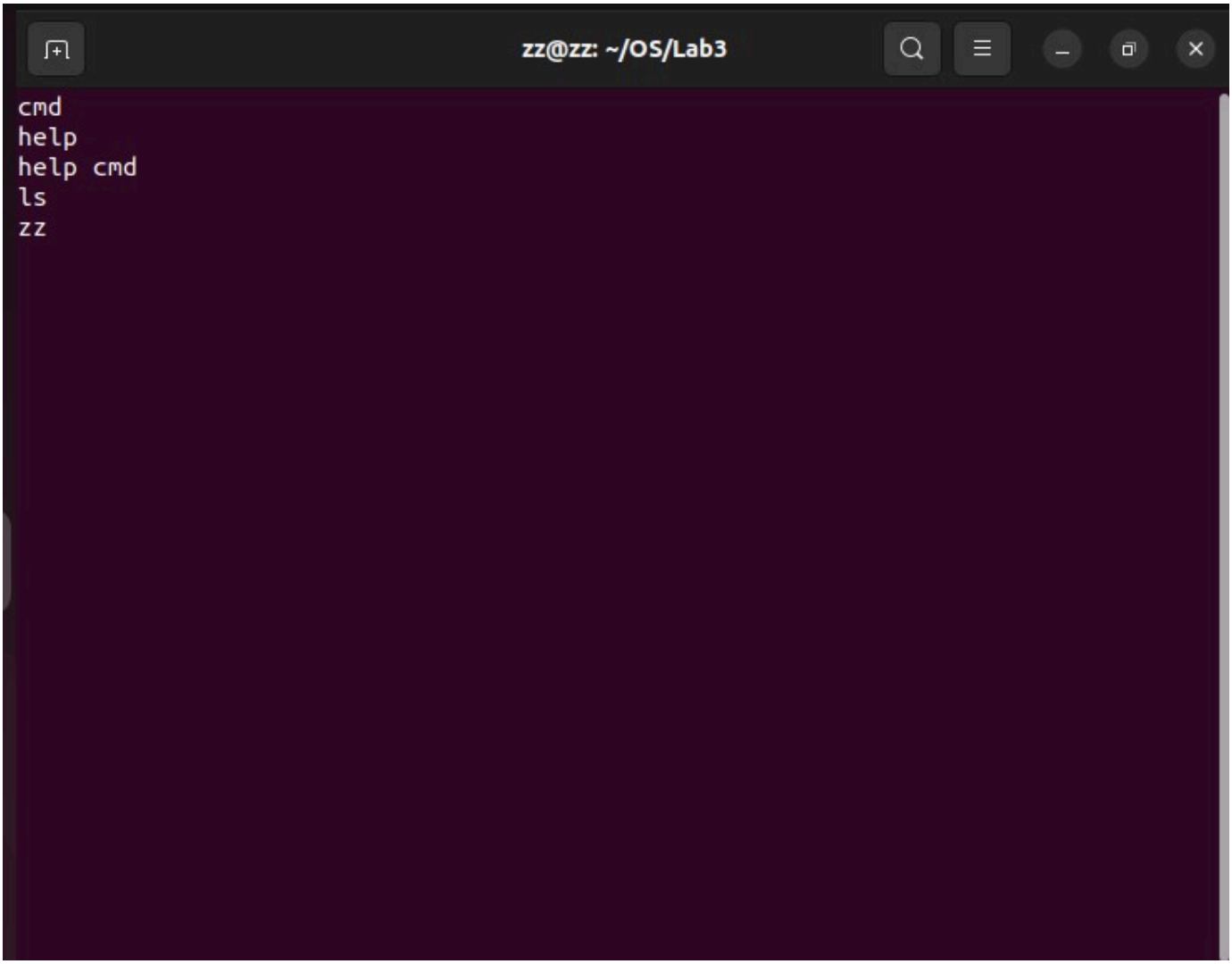
编译过程说明

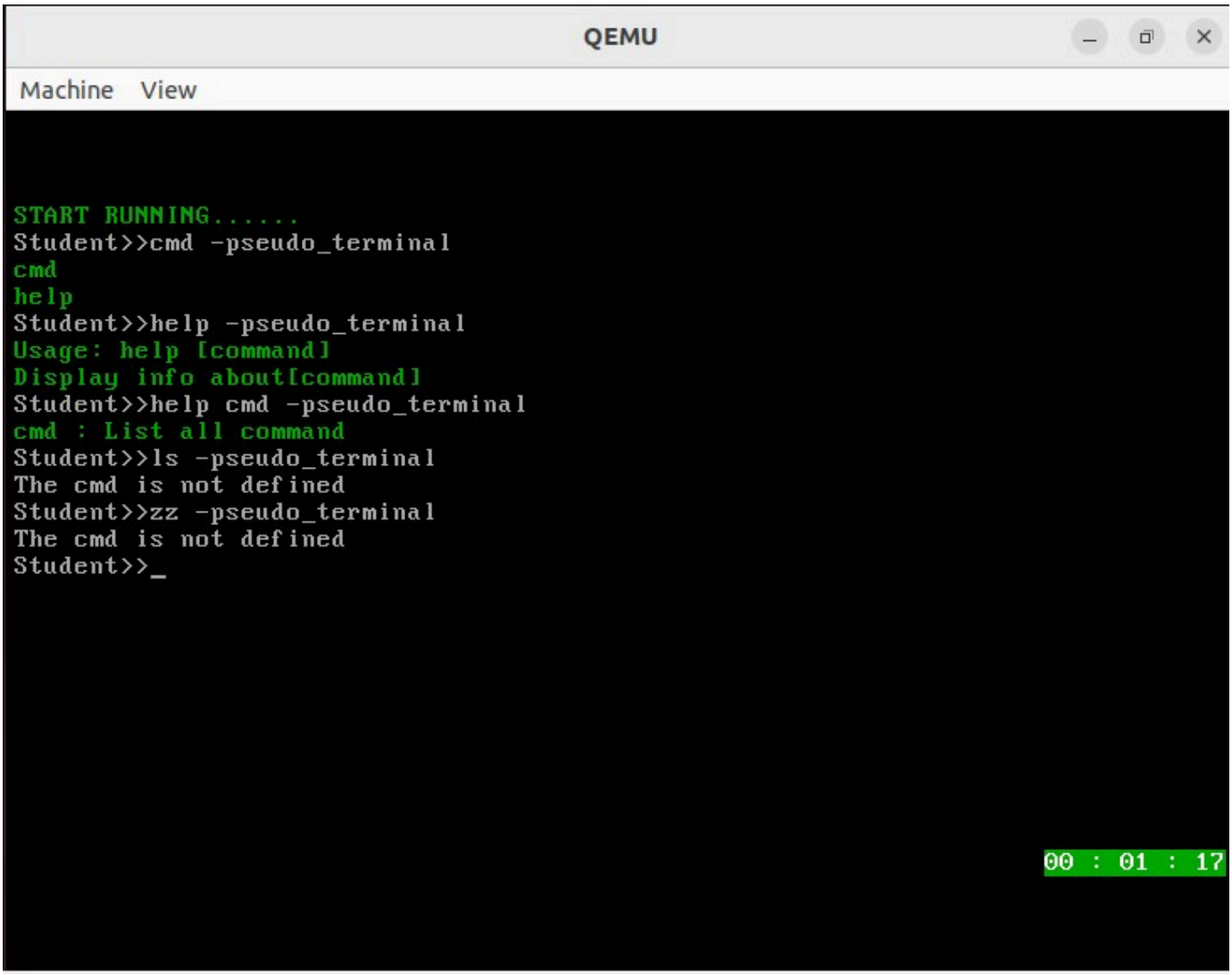
- 由Makefile文件描述控制汇编文件和C文件编译生成.o中间文件。
- 链接器链接各.o文件生成可执行文件。

运行结果

- 根据实验讲义运行脚本并输入screen指令后测试cmd，help，help cmd，ls，zz指令结果如下，符合要求：


zz@zz: ~/OS/Lab3
zz@zz:~/OS/Lab3$ ./source2run.sh
./source2run.sh: line 2: shell: command 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/i8259A.o output/myOS/dev/i8253.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/kernel/tick.o output/myOS/kernel/wallClock.o output/myOS/string/string.o output/userApp/main.o output/userApp/startShell.o -o output/myOS.elf
make succeed
zz@zz:~/OS/Lab3$ char device redirected to /dev/pts/3 (label serial0)


zz@zz: ~/OS/Lab3
cmd
help
help cmd
ls
zz



问题和解决方法

- 墙钟实现中的hook机制，在检索相关知识了解后理解并实现了hook机制。
- 为了将时间输出在屏幕上，在vga.c中增加了append2screen_clock函数来实现。
- 为了在shell指令识别中方便比较字符串，增加了一个简单的string库来模仿glibc的string库。