

lab3 实验文档

1 实验说明

实验名: Start2Shell

本实验是系列实验中的第三个。在内核中，增加对中断和时钟的支持。在用户程序中，增加 shell。这是一个简单的 shell。Shell 中的命令，以函数的形式被调用，而不是以一个被调度的单位来运行。

1.1 实验基础

本实验在实验 2 的基础上进行。在实验 2 提交的截止时间过后，同学们可以就实验 2 的内容互通有无。实验 3 可以在其他同学实验 2 的基础上进行：

- 无论你使用哪一个（包括自己的），请在实验报告中标注，实验 2 的基础来自哪个同学（可以自己）；
- 给你使用的实验 2 打分。
- 也可以参考助教提供的实验 2（或实验 3）框架，这个框架已经帮同学们完成了较为关键的部分代码。若直接使用了助教提供的框架，请说明。

2 实验目的

实现一个具有简单 shell 功能、能显示墙钟的 OS。

3 实验要求

- **【必须】**简单的 shell 程序，提供 cmd 和 help 命令，允许注册新的命令
- **【必须】**中断机制和中断控制器初始化
- **【必须】**时钟和周期性时钟中断
- **【必须】**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 interrupt”
 - 右下角：从某个时间开始，大约每秒更新一次
- 格式为：HH:MM:SS
- **【必须】**采用自定义测试用例和用户（助教）测试用例相结合的方式进行验收
- **【必须】**提供脚本完成编译和执行

- Offset
- P: Present; Set to 0 for unused interrupts or for Paging.
- DPL: Gate call protection.
- S: Set to 0 for interrupt gates
- Type:
 - 0b0101: 32bit task gate
 - 0b1110: 32-bit interrupt gate
 - 0b1111: 32-bit interrupt gate

- Offset 是 32 位的中断处理例程入口地址（段内偏移），被分成两个部分，offset 的低 16 位对应于中断描述符 8 字节的低 16 位，offset 的高 16 位对应于中断描述符 8 字节的高 16 位。
- Selector 是段选择子，用来确定是哪个段，要配合 GDT/LDT 一起使用。我们的实验中段选择子取值 0x8。关于段选择子，参见 4.1.3。
- 其他几个标志位，取值为 0b1000 1110 0000 0000=0x8E00，除了低 8 位全 0，其他几位的含义分别是：
 - Type 位，类型位，4 位。在 32 位保护模式下，用作中断门时，取值 0b1110。若用作陷阱门（trap gate），取值 0b1111。我们很粗暴地将所有的描述符都初始化为中断门。后续，若有需要，可以根据实际情况进行调整。
 - S 位，是否系统段。用作中断门时，设为 0。
 - DPL（Descriptor privilege level，描述符特权级），2 位。用于保护对中断门的访问。只允许内核态访问中断门，因此，此处取值 00。（若有其他考虑，可能要取不同的值。）
 - P 位，存在位。我们总是将其设为 1，表示这个中断描述符使用中，是有效的。

4.2.2 如何定义 IDT 表并为其分配存储空间

我们尚未对内存进行动态管理，因此我们提供静态分配的方法。下面是参考代码：

```
# IDT
    .p2align 4
    .globl IDT
IDT:
    .rept 256
    .word 0,0,0,0
    .endr
```

注意，此处我们定义了一个全局量 IDT 表，这是一个变量，应该放置在数据区。

我们在定义 IDT 表时，将这个表填充为全 0。在正式使用 IDT 表之前，即开中断之前，还需要进一步初始化这个 IDT 表。

4.2.3 将 IDT 表信息告知 CPU

IDTR（IDT register）用于存放 IDT 表信息。CPU 在检测到中断发生的时候，将通过 IDTR 寄存器获得 IDT 表信息，进而根据中断向量号，查表。

汇编指令 LIDT 和 SIDT 分别用于 load/store 这个寄存器的内容。其中，LIDT 指令只可以在内核态执行，用于存储 IDT 表信息到 IDTR 寄存器中。SIDT 指令用于将 IDTR 寄存器中的内容保存到内存中，该指令可以在任何特权级调用。我们只使用 LIDT 指令。参考 Intel 手册 3A 6.10。

IDT 表信息包括两个内容：IDT 表基地址（base address）和 IDT 限长（limit）。参考定义如下：

```
idtptr:
    .word (256*8 - 1)
    .long IDT
```

注意，上述定义仅仅将 IDT 表的基地址和长度信息拼装到 idtptr 内存变量中，放置在数据区。还需要执行 lidt 指令来将这个信息 load 到 IDTR 寄存器中，这样才能建立动态的中断机制与中断描述符表之间的关联关系。参考代码如下：

```
lidt idtptr
```

4.2.4 IDT 表的进一步初始化

我们不知道操作系统内核会为哪些中断/异常提供进一步处理。我们将这件事情交给未来可能会实现的各个模块自行决定。但现在，我们先对所有的中断提供缺省的处理。

缺省的处理方式很简单，即，一旦发生某个中断，就显示“Unknown interrupt”。我们使用一个 C 函数 ignoreIntBody() 作为处理函数，具体如下：

```
void ignoreIntBody(void){
    put_chars("Unknown interrupt\0",0x4,24,0);
}
```

函数 ignoreIntBody() 需要调用 put_chars() 这个接口。

在处理中断前后，需要保存上下文，因此我们提供中断入口处理程序如下：

```
.p2align 4
ignore_int1:
    cld
    pusha
    call    ignoreIntBody
    popa
    iret
```

其中：

- 第 2 行是一个 label，为中断入口处理程序的符号名，“ignore_int1”。
- 第 3 行，汇编指令 cld，将标志寄存器 eflags 的方向标志位 DF 清 0。
- 第 4 行和第 6 行，汇编指令 pusha 和 popa，分别用于保存寄存器上下文和恢复寄存器上下文。
- 第 7 行，汇编指令 iret，从中断返回。

在处理器检测到中断，进入这段程序运行之前，处理器会在当前栈上保存一部分上下文，我们称之为硬件保存的上下文。在进入这段程序运行之后，这段程序会进一步保存上下文，即上述代码第 4 行保存的上下文，我们称之为软件保存的上下文。这两部分上下文结合在一起，是完整的现场。

在我们的实验中，没有特权级的切换，因此没有栈的切换。如果我们对代码进行调试，可以观察到栈中保存的上下文的信息。

调用 iret 从中断返回之前，首先要调用 popa 将软件保存的上下文恢复，然后调用 iret 指令。IRET 指令将恢复硬件保存的上下文。

准备好缺省的中断处理程序之后，我们用它来填写 IDT 表。参考代码如下：

```
setup_idt:
    movl $ignore_int1,%edx
    movl $0x00080000,%eax /* selector */
    movw %dx,%ax
    movw $0x8E00,%dx      /* interrupt gate — dpl=0, present */

    movl $IDT,%edi
    mov $256,%ecx
rp_sidt:
    movl %eax,(%edi)
    movl %edx,4(%edi)
    addl $8,%edi
    dec %ecx
    jne rp_sidt
```

- 本代码中，在填写 IDT 表时，eax 和 edx 寄存器分别用来临时存放中断描述符的低 4 字节和高 4 字节。在此之前，先要准备这两个寄存器中的值。
- 第 2 行，取缺省的中断处理程序的首地址（32 位），到 edx 寄存器中。
- 第 3 行，将 0x00080000 写入 eax 寄存器中。这是为了将段选择子的内容写入 eax 寄存器的高 16 位。
- 第 4 行，将 edx 寄存器（这是一个 32 位寄存器）的低 16 位，即 dx 寄存器（这是一个 16 位寄存器）的值，也即中断处理程序首地址的低 16 位，写入 eax 寄存器的低 16 位。如此，我们就准备好 eax 寄存器的内容。
- 第 5 行，我们在 edx 寄存器的低 16 位，写入了各个标志位。如此，我们就准备好了 edx 寄存器的内容。
- 第 7 行，我们要填写 IDT 表了，先将 IDT 表基地址写入 edi 寄存器，这个地址也是 IDT 表中第 0 个表项的首地址。
- 第 8 行，我们将采用循环 256 次的方法，挨个写中断描述符到 IDT 表相应的表项中。因此，我们将 256 写入 ecx 寄存器，ecx 寄存器常常用作循环次数的计数。
- 第 9 行，循环体的标号。
- 第 10 行，填写 eax 寄存器中的内容到 IDT 表当前表项的低 4 字节，表项的首地址是 edi 寄存器的内容。
- 第 11 行，填写 edx 寄存器中的内容到 IDT 表当前表项的高 4 字节。
- 第 12 行，更新 edi 寄存器指向 IDT 表下一个表项的首地址。
- 第 13 行，递减循环次数。
- 第 14 行，当递减为 0 时，循环结束。循环未结束时，回到循环体首地址循环执行。

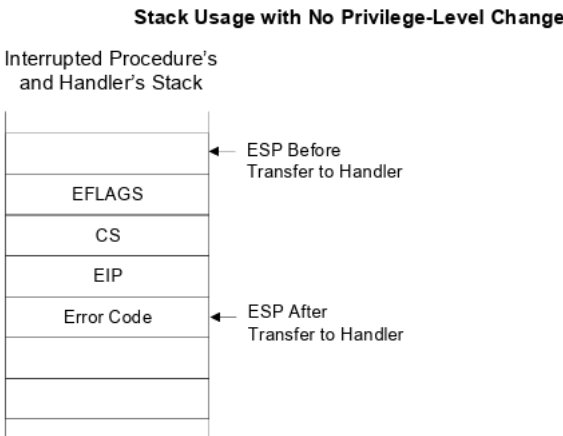
4.2.5 中断发生时由硬件保存的上下文

参考 Intel 手册第 1 卷 6.5.1。

当处理器检查到中断时，处理器（硬件）将依次保存 EFLAGS、CS 和 EIP 寄存器中的内容到当前的栈上。如果是异常，还会保存错误码。我们这里不考虑异常。紧接着，处理器会根据 IDTR 中的 IDT 表信息和中断向量号，找到当前中断相关的中断描述符，从中取出段选择子和中断处理程序的首地址，加载到 CS:EIP 寄存器中，从而转入中断处理程序中运行。这个中断处理程序正是我们之前填写到 IDT 表的中断处理入口所对应的程序。

在我们的实验中，没有特权级的切换，因此没有栈的切换。如果我们对代码进行调试，可以观察到栈中保存的上下文的信息。

图 1: 无特权级变化时，硬件上下文的保存

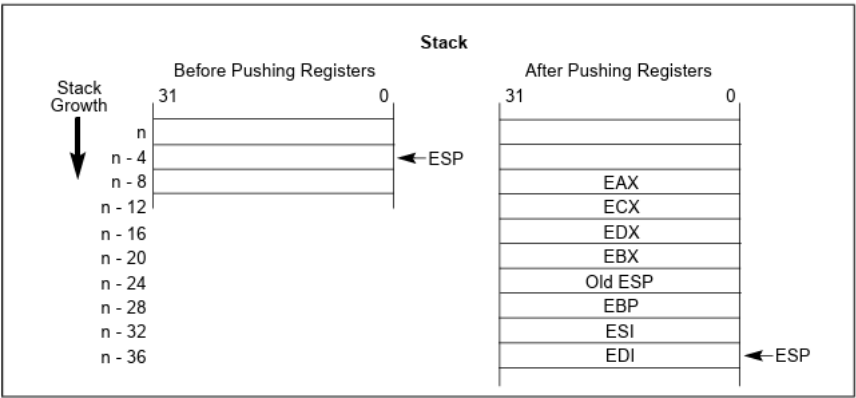


4.2.6 保存和恢复寄存器上下文 pusha 和 popa

如果我们没有特别的寄存器上下文的定义，可以直接调用 pusha 和 popa 来保存和恢复上下文。

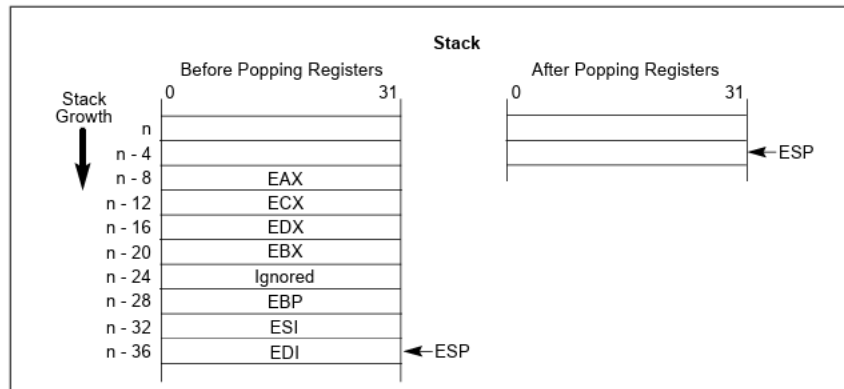
汇编指令 pusha 将所有通用寄存器入栈。在 32 位保护模式下，pusha 的入栈顺序是：eax, ecx, edx, ebx, esp, ebp, esi, edi。每个大小 4 个字节。入栈后，esp 指向新的栈顶。

图 2: PUSH A 指令执行前后



汇编指令 popa 从栈顶出栈 8 个数据到通用寄存器中。在 32 位保护模式下，出栈的数据每个大小 4 字节。出栈顺序是：edi, esi, ebp, esp/跳过, ebx, edx, ecx, eax。出栈后，esp 指向新的栈顶（而不是出栈的值）。

图 3: POPA 指令执行前后



4.2.7 标志位寄存器 eflags

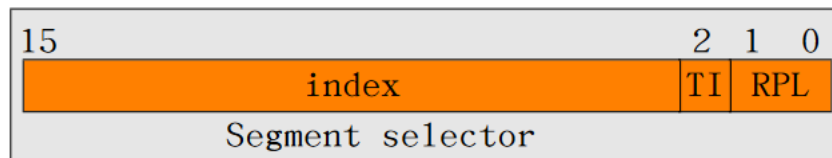
我们可能用到标志位寄存器 eflags 中的一些。

- 第 9 位, IF 位, Interrupt enable flag, 中断使能标志位。汇编指令 cli/sti 分别将这个位清 0 或置 1。置 1 时中断被使能, 即开中断。清 0 时, 中断被禁止, 即关中断。
- 第 10 位, DF 位, direction flag, 方向标志位。汇编指令 cld/std 分别将这个位清 0 或者置 1。

4.2.8 GDT/LDT 和段选择子 selector

在 32 位保护模式下, 16 位的段寄存器用作段选择子。段选择子配套 GDT/LDT 使用。其中, GDT/LDT 是全局/局部段描述符表 (Global/Local descriptor table)。LDT 一般用作进程的段描述符表。我们的实验中不使用 LDT 表, 只使用 GDT 表。

GDT/LDT 各是一张表, 每个表项对应于一个段描述符。每个段描述符的长度是 8 个字节。16 位段选择子用于在 GDT/LDT 表中选定一个段描述符。如下图所示。



- Index, 高 13 位, 是选定的段描述符在 GDT/LDT 表中的索引号, 即下标。
- TI 位, 用于在 GDT 和 LDT 中进行选择。TI=0, 表示选择 GDT 表。
- RPL 是请求的特权级, Requested privilege level。我们取值 00, 表示内核级。

因此, 在我们的实验中, 段选择子的低 3 位取值 0b000。

(关于段选择子的另一个理解: 段选择子存放的内容是段描述符在段表 GDT/LDT 中的偏移量, 由于段描述符的首地址是 8 字节对齐的, 因此这个偏移量的低 3 位总是全 0。系统正好利用这 3 位来存放 TI 信息和 RPL 信息。)

4.2.9 开中断和关中断

我们约定, 开中断和关中断的接口如下:

```
void enable_interrupt(void);
void disable_interrupt(void);
```

如何实现这两个接口？使用汇编指令 cli/sti。参考代码如下：

```
        .globl enable_interrupt
enable_interrupt:
        sti
        ret

        .globl disable_interrupt
disable_interrupt:
        cli
        ret
```

4.2.10 可编程中断控制器 PIC i8259 及其初始化

参考 8259 手册可以获得 8259 相关的信息。

本讲义中仅涉及我们用到的部分。

我们需要对 8259 进行初始化。

两片 8259 采用级联的方式连接。一为主片，一为从片。主片的端口地址的范围是 0x20-0x21，从片为 0xA0-0xA1。

中断屏蔽寄存器（Interrupt Mask Register），8 位。每一位对应于一个中断请求线（Interrupt request line, IRL）。当对应位置 1 时，该中断请求线上发生的中断被屏蔽。我们通过对端口地址 0x21 和 0xA1 进行读写，可以设置或者读取当前的中断屏蔽寄存器的值。一开始，所有中断都被屏蔽，因此初始化为 0xFF。

后续如果需要取消屏蔽某一位或者再次屏蔽某一位，可以先读取中断屏蔽寄存器，然后对读出来的值的对应位执行置 1 或者清 0 操作，再将修改后的值写回中断屏蔽寄存器。

中断控制器的读写控制逻辑是可编程的。CPU 端使用初始化命令字（Initialization Command Word, ICW）来初始化中断控制器。我们按以下顺序对主片进行初始化：

- ICW1: 0x11，写入端口地址 0x20。最低位为 1 表示 “ICW4 needed”。另一个 1 是 ICW1 的标志性比特位之一。次低位为 0 表示有多片 8259。
- ICW2: 0x20 写入端口地址 0x21。0x20 是主片的起始中断向量号，对应于 0 号中断请求线。
- ICW3: 0x04 写入端口地址 0x21。对于主片，写入的值，每一位对应于一个中断请求线，置 1 的那个表示有从片接入。0x04 表示，在 0-7 引脚中，2 号引脚上接入从片。
- ICW4: 0x3 写入端口地址 0x21。表示 Auto EOI。EOI 为 end of interrupt。

我们按以下顺序对从片进行初始化：

- ICW1: 0x11 写入端口地址 0xA0。
- ICW2: 0x28 写入端口地址 0xA1。从片的起始向量号为 0x28。

- ICW3: 0x02 写入端口地址 0xA1。从片的 ICW3 定义与主片不同。高 5 位全 0。低 3 位能表达 0-7，用于表示从片连接到主片的哪个引脚上。从片的 ICW3 和主片的 ICW3 要相呼应。因此，我们这里写入 0x02。
- ICW4: 0x1 写入端口地址 0xA1。AEOI 不置位。

4.3 时钟相关

我们使用 8253 实现周期性时钟中断，即嘀嗒。

我们需要对 8253 编程，以获得周期性时钟中断。

我们需要提供并注册周期性时钟中断的处理程序。

4.3.1 可编程间隔定时器 PIT i8253 及其初始化

参考 8253 手册可以获得 8253 相关的信息。同学们也可以参考这个网页：

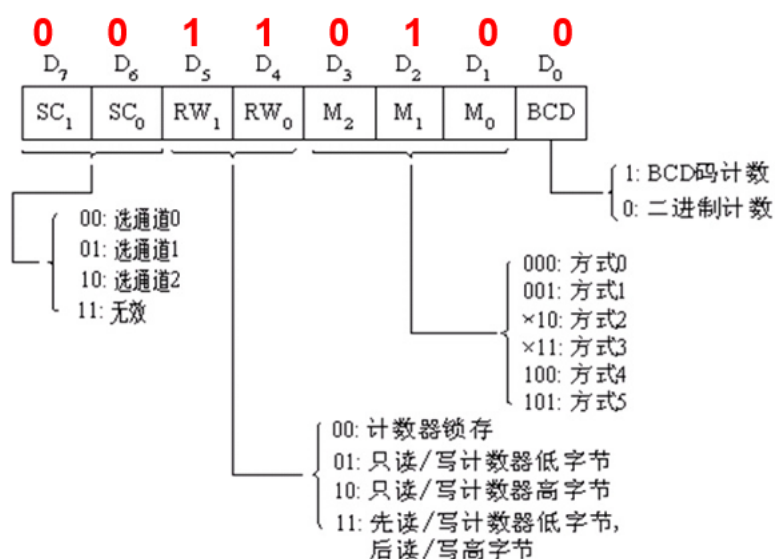
https://wiki.osdev.org/Programmable_Interval_Timer

我们只涉及用到的部分。

8253 的端口地址是 0x40-0x43。

我们需要对 8253 分频以触发我们需要的周期性时钟中断。在我们的实验中，可以考虑设定周期性时钟中断的频率为 100Hz。分频参数为 8253 的频率除以 100 后取整。8253 的频率是一个常数，查手册可以查到，约为 1193182Hz。

对 8253 进行初始化时，我们先将一个值 0x34 写入端口 0x43。值 0x34 的含义如下图所示：



然后我们依次将分频参数的低字节和高字节写入端口地址 0x40。

这样我们就完成了 8253 的初始化。8253 就会周期性的产生时钟中断了。时钟中断的向量号为 0x20，对应于 8259 的 0 号引脚。但此时我们还没有设置好中断处理程序，也还没有解开对时钟中断的屏蔽。

4.3.2 周期性时钟中断 Tick（嘀嗒）

我们定义一个全局变量来维护 tick 发生的次数。每当产生一次时钟中断，这个全局变量就递增 1。

我们用 C 语言来定义 tick() 函数。在这个函数中维护上述全局变量。此外，其他需要随时间变化的维护，也可以在 tick() 中进行。例如，维护墙钟。

周期性时钟中断的处理程序，先保存上下文，再调用 tick()，然后恢复上下文。参考代码如下：

```
        .p2align 4
time_interrupt:
        cld
        pushf
        pusha
        call tick
        popa
        popf
        iret
```

定义好时钟中断处理程序后，我们用其重新初始化对应的中断描述符：

```
setup_time_int_32:
        movl $time_interrupt,%edx
        movl $0x00080000,%eax      /* selector: 0x0010 = cs */
        movw %dx,%ax
        movw $0x8E00,%dx          /* interrupt gate - dpl=0, present */

        movl $IDT,%edi
        addl $(32*8), %edi
        movl %eax,(%edi)
        movl %edx,4(%edi)

        ret
```

代码及其含义类似前面填写 IDT 表。

在 OS 完成初始化的适当时机，我们要开中断，否则系统将不会相应任何中断。

4.3.3 维护和显示墙钟

本讲义中，墙钟使用 wallClock 表示。

墙钟 wallClock 具有四个时间分量：

- 时。可以考虑使用 hh 表示，取值范围为 [0-23]。
- 分。可以考虑使用 mm 表示，取值范围为 [0-59]。
- 秒。可以考虑使用 ss 表示，取值范围为 [0-59]。
- 毫秒。可以考虑使用 ms 表示，取值范围为 [0-999]。

我们约定，墙钟的设置和读取接口如下：

```
void setWallClock(int h, int m, int s); //设置墙钟的时间值
void getWallClock(int *h, int *m, int *s); //读取墙钟
```

墙钟时间值的初始化。通常，墙钟跟真实的世界保持一致。本实验中，不做此严格要求。可以将墙钟初始化为任意合理的时间值，但建议最好不要是 00:00:00:00，以显示一个经过初始化的时钟。请思考，何时进行墙钟初始化？

墙钟时间值的不断维护。一旦墙钟初始化完成，则依靠周期性时钟中断来不断维护墙钟。比如，随着时钟中断的发生，根据 ms 和 tick 之间的时间比例关系，来确定是否要更新 ms，即 ms 的数值是否要加 1，如果溢出，则进而考虑是否要更新 ss, mm, hh。建议将维护墙钟时间值的功能，编写成一个独立的内部函数，在需要维护墙钟之处调用这个内部函数，以体现一定的模块化。至于墙钟维护函数的调用点，请结合“依靠周期性时钟中断来不断维护墙钟”这句话来考虑。

如何显示墙钟（例如，墙钟的外观），是用户程序要考虑的事情。内核只负责维护墙钟并提供初始化和读取墙钟相关的接口。墙钟应当随着墙钟的变化，不断更新显示。

直观来看，似乎在用户程序中提供一个不断运行（如一个死循环）的程序，该程序不断地调用 getWallClock() 来获取当前的墙钟值，并将其与之前读取的墙钟值进行比较，若发现墙钟值发生了变化，就更新显示。同学们可以尝试一下这种方法，看看这种方法的优劣。

另外一种更新显示墙钟的方法，可以考虑使用 hook 机制。即，在维护墙钟的过程中，若发现墙钟被更新，并且界面上显示的墙钟也要随着更新时，就更新显示墙钟。这里的问题是，内核不知道用户程序更新显示墙钟的接口是什么。使用 hook 机制，用户程序在初始化的过程中，可以注册一个函数，该函数在内核检查到要更新显示墙钟时，将被调用。我们约定，注册更新显示墙钟函数的接口为：

```
void setWallClockHook(void (*func)(void));
```

上面的接口中，func 就是用户自定义的函数。setWallClockHook() 用于注册用户自定义的函数。请思考，何时注册？何时调用？

5 Shell

从本实验开始，我们有了一个 shell。这个 shell 提供一个交互式界面，用户可以在这个交互式界面中，输入并执行命令。

5.1 哪些命令

至少提供两个命令。

一个是“cmd”命令，该命令用于罗列所有当前支持的命令。

另一个是“help [cmd_name]”命令，该命令用于显示某个命令的帮助信息。如果没有指定命令，则显示 help 的帮助信息。

因此，一个命令包含的要素主要有：

命令名。给这个命令起一个名字。函数指针。调用这个函数，就是执行这个命令。命令的 help 函数指针。调用这个函数，就显示本命令的帮助信息。描述命令的字符串。除了上述必须的两个命令之外，用户可以自定义其他的命令。

5.2 命令的维护

Shell 要管理所有的命令。可以采用静态定义或者动态注册的方式管理命令。

5.3 Shell 的入口

我们约定，调用 `startShell()` 来进入 shell 交互界面，用户可以执行命令。

```
void startShell(void);
```

5.4 Shell 的输入输出设备

结合 Qemu，我们使用串口作为输入设备，使用 uart 和 VGA 作为回显和输出设备。

Uart 的端口区间为 0x3F8-0x3FF；假设已经初始化完毕。数据输入输出的端口地址为 0x3F8；访问状态寄存器的端口地址为 0x3FD，状态寄存器值的最低位为 1，则表示数据可读。

从 Uart 读取字符的参考代码如下：

```
unsigned char uart_get_char(void){
    while (!(inb(uart_base+5)&1));
    return inb(uart_base);
}
```

5.5 Qemu 串口重定向

处理键盘中断较为繁琐，我们选择使用串口输入。QEMU 和主机之间可以使用伪终端来模拟串口连接。

修改运行脚本中执行 qemu 的命令为：

```
qemu-system-i386 -kernel output/myOS.elf -serial pty &
```

运行时，qemu 会报告使用的具体的伪终端设备，例如，可能是 “/dev/pts/1”（后面的数字可能会变化）。为便于描述，我们使用 N 来表示这个数字。

我们打开另一个命令窗口，执行命令：

```
sudo screen /dev/pts/1
```

这个命令中的数字，要跟上面的数字 N 保持一致。这样才能建立主机和 qemu 之间的连接。

6 实验内容

程序阅读流程:

multibootheader/multibootHeader.S → myOS/start32.S → myOS/osStart.c → userApp/startShell.c

6.1 myOS/start32.S 中的 time_interrupt 和 ignore_int1 的填写

6.1.1 在该模块中你需要用到的汇编语言:

- cld: 将标志寄存器的方向标志位 DF 清零
- pushf: 将标志寄存器 (Flag) 中的值存入栈中
- popf: 将栈中的内容存入标志寄存器
- pusha: 将通用寄存器 (eax, ebx 等) 中的值存入栈中
- popa: 将栈中的内容存入通用寄存器中
- call: 函数调用 (可以调用 C 语言程序中的函数)
- iret: 返回调用该函数的函数

关于 call 和 iret: 本质上 call 和 iret 就是 CS 和 IP 寄存器的入栈和出栈, 对于 CS 和 IP 寄存器的理解可以对应你们计组课上的 PC(Program Counter, 程序计数器)

6.1.2 如何编写这两个函数?

1. 利用 push 来进行现场保护
2. call 相应的函数 (可以调用 C 语言程序中的函数)
3. 利用 pop 来进行恢复现场
4. 利用 iret 来返回调用该函数的函数

6.2 myOS/dev/i8253.c 和 myOS/dev/i8259A.c 的填写

i8253 和 i8259A 都是可编程芯片, 什么叫可编程逻辑芯片? 就意味着它可以在配置后进行使用, 至于如何配置他们呢? 我们只需要用 outb 函数往相应的地址输出相应的数值即可 (ppt 中有详细的配置地址与相应配置数值说明)

6.2.1 关于 i8253 和 i8259A 的工作方式说明:

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 (也即 time_interrupt 和 ignore_int1)

6.3 myOS/i386/irq.s 的填写

在该模块中你需要用到的汇编语言：

- ret: 返回调用该函数的函数
- sti: 开中断
- cli: 关中断

6.4 Tick 的实现

我们会在由 i8253 引起的时钟中断而引起的中断子程序 time_interrupt 处理中调用 tick 函数，由于 tick 函数的调用是有固定频率的，所以我们可以用它来进行时钟的输出

6.5 WallClock 的实现

实现通过 vga 往合适的位置输出 HH:MM:SS，即显示时钟；根据 vga 显存中的数值，返回时钟，并存到相应的指针指向位置中。

```
void setWallClock(int HH,int MM,int SS){
    //通过 vga 往合适的位置输出 HH:MM:SS，即显示时钟
}
void getWallClock(int *HH,int *MM,int *SS){
    //根据 vga 显存中的数值，返回时钟，并存到相应的指针指向位置中
}
```

6.6 Shell 的实现

功能：显示交互界面；接收并处理命令行。

一个命令的元信息为（命令名，描述命令的字符串，func）本实验至少需要提供 2 个命令：

- cmd：列出所有命令
- help [cmd]：调用指定命令的 help 函数，若没有指定命令，则调用 help 的 help 函数

添加命令可以使用静态的方式手动添加，也可以用数组等方式动态注册。

源代码如下：

```
typedef struct myCommand {
    char name[80];
    char help_content[200];
    int (*func)(int argc, char (*argv)[8]);
}myCommand;

int func_cmd(int argc, char (*argv)[8]){

}

myCommand cmd={"cmd\0","List all command\n\0",func_cmd};

int func_help(int argc, char (*argv)[8]){

}

myCommand help={"help\0","Usage: help [command]\n\0Display info about [command]\n\0",func_help};

void startShell(void){
//我们通过串口来实现数据的输入
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,"Student>>\0");
        while((BUF[BUF_len]=uart_get_char())!='\n'){
            uart_put_char(BUF[BUF_len]); //将串口输入的数存入BUF数组中
            BUF_len++; //BUF数组的长度加
        }
        uart_put_chars("-pseudo_terminal\0");
        uart_put_char('\n');

        //OK,助教已经帮助你们实现了“从串口中读取数据存储到BUF数组中”的任务，接下来你们要做
        //的就是对BUF数组中存储的数据进行处理(也即，从BUF数组中提取相应的argc和argv参
        //数)，再根据argc和argv，寻找相应的myCommand ***实例，进行***.func(argc,argv)函数
        //调用。

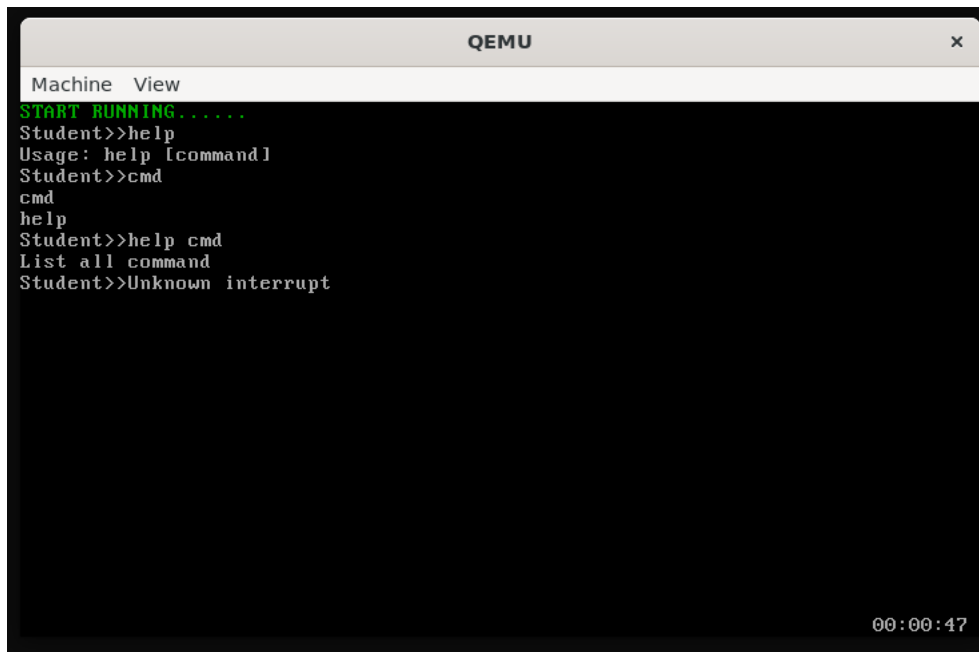
        //比如BUF中的内容为 “help cmd”
        //那么此时的argc为2 argv[0]为help argv[1]为cmd
        //接下来就是 help.func(argc, argv)进行函数调用即可

    }while(1);
}
```

7 实验示例

输入 `./source2run.sh` 指令后，编译，链接，生成 `myOS.elf` 文件。输入指令 `qemu-system-i386 -kernel myOS.elf -serial pty &` 运行。

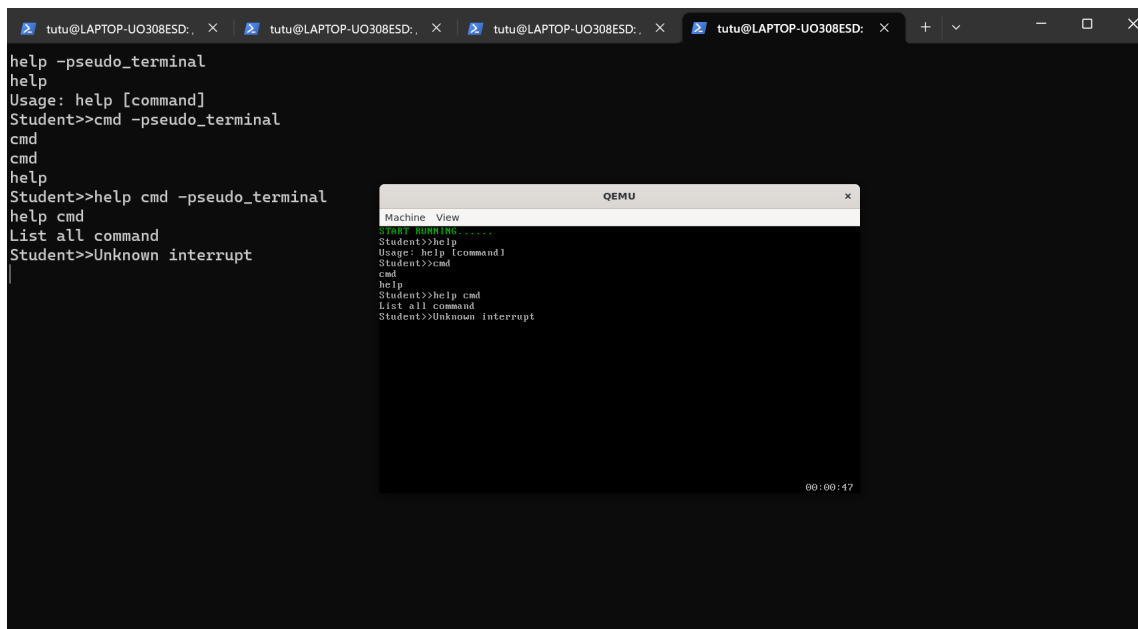
使用 `screen` 命令进入交互界面（与 QEMU）`sudo screen /dev/pts/1` 时钟如右下角所示：



```
QEMU
Machine View
START RUNNING.....
Student>>help
Usage: help [command]
Student>>cmd
cmd
help
Student>>help cmd
List all command
Student>>Unknown interrupt

00:00:47
```

执行 `cmd`，`help`，`help cmd` 和未知命令，结果如下：



```
tutu@LAPTOP-UO308ESD: x tutu@LAPTOP-UO308ESD: x tutu@LAPTOP-UO308ESD: x tutu@LAPTOP-UO308ESD: x + - □ ×

help -pseudo_terminal
help
Usage: help [command]
Student>>cmd -pseudo_terminal
cmd
cmd
help
Student>>help cmd -pseudo_terminal
help cmd
List all command
Student>>Unknown interrupt

QEMU
Machine View
START RUNNING.....
Student>>help
Usage: help [command]
Student>>cmd
cmd
help
Student>>help cmd
List all command
Student>>Unknown interrupt

00:00:47
```

8 提交要求

请将代码打包压缩（包含报告）提交，并且将报告再单独提交一份。两份文件一次性提交。命名为学号 _ 姓名 _lab3