

**本科实验报告**

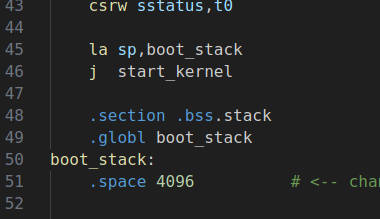
|  |  |
| --- | --- |
| 课程名称： | 操作系统 |
| 姓 名： | 董佳鑫 |
| 学 院： | 计算机学院 |
| 系： | 计算机系 |
| 专 业： | 计算机科学与技术 |
| 学 号： | 3210102181 |
| 指导教师： | 寿黎但 |

# Lab 1: RV64内核引导与时钟中断处理

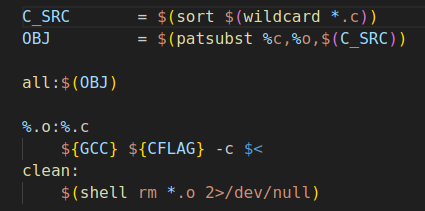
实验过程和操作步骤

1.RV64内核引导

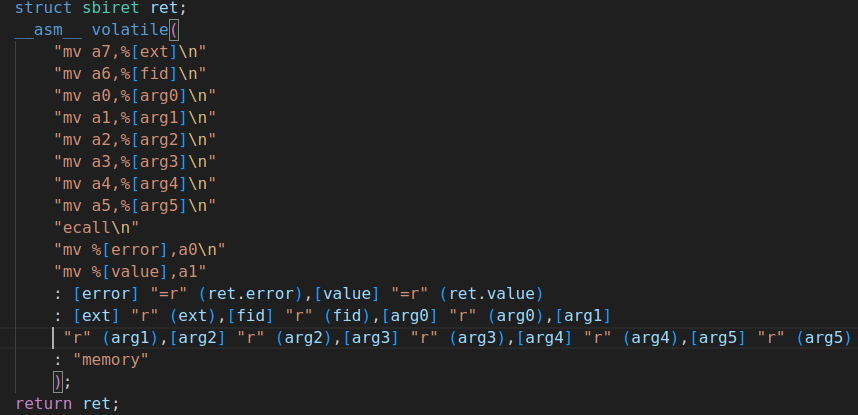
首先学习汇编指令，完成head.s汇编代码。



然后学习Makefile基本语法格式，编写lib/Makefile。



根据指导书的讲解，完成sbi.c中ecall函数的实现，学习并使用内联汇编完成C语言函数。



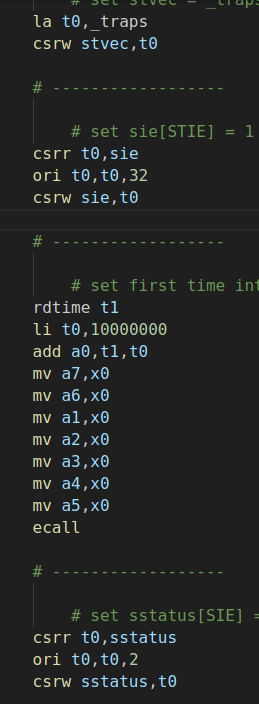
补充完成defs.h文件，使用内联汇编完成csr\_read宏定义。



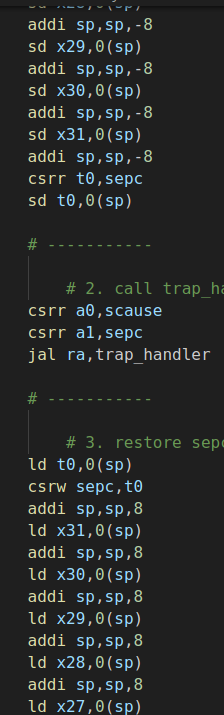
1. RV64时钟中断处理

首先按照手册完成准备工作，将对应文件内容进行修改。

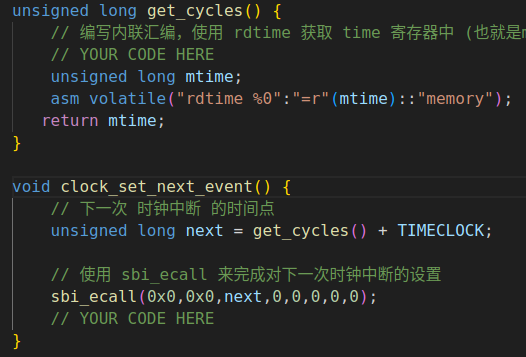
然后继续修改head.s文件，设置stvec，开启时钟中断使能，设置第一次时钟中断。

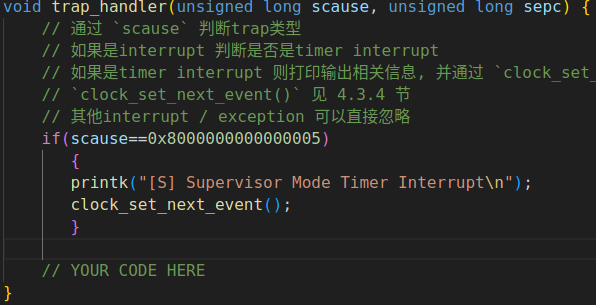


编写entry.s汇编代码，完成时钟中断处理操作。（部分代码如下）

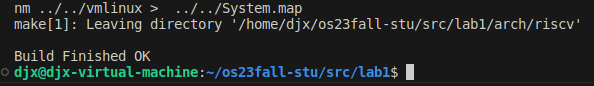


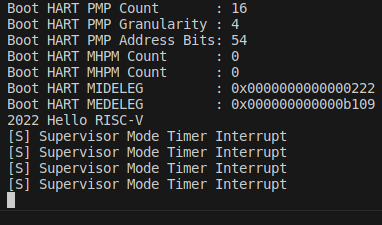
最后，实现trap处理函数和时钟函数。





至此，时钟中断部分完成，下面进行编译运行。





1. 其他架构的交叉编译——以aarch64为例

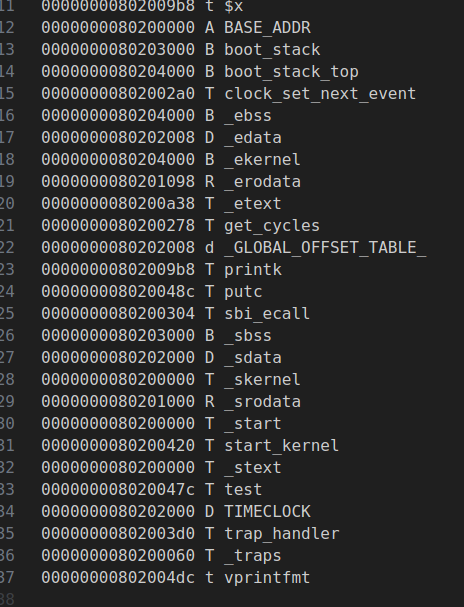
这一部分的操作过程及截图在思考题5中展示，这里不再赘述。

思考题

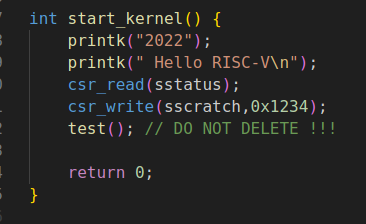
1. riscv的call convention：在调用函数前，会先把函数参数传递给特定的寄存器a0-a7，以便函数内部使用这些值。而其他不参与函数内部的值需要进行保存，将值保存到stack中，如ra寄存器的值，在调用函数时会把返回地址传给ra，所以在调用前会先把ra原先的值保存到stack中，函数返回后再把ra的值从stack中取出。

call convention会涉及到两种特殊的寄存器，Caller-Saved和callee-saved寄存器。其中Caller-Saved寄存器是被调用者不需要保持这些寄存器的值不变，而调用者在调用其他函数之前，要把其中一些必要的寄存器保存到stack中，以便于函数结束调用后能顺利找到调用者的地址。而callee-saved寄存器则是在被调用者在使用完这些寄存器后应该把这些寄存器的值恢复到原先的状态，保证不因为函数调用而改变这些寄存器的值。

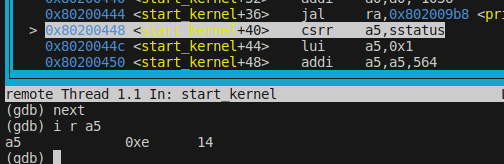
2.打开System.map ，可以看到vmlinux.lds中符号的值。

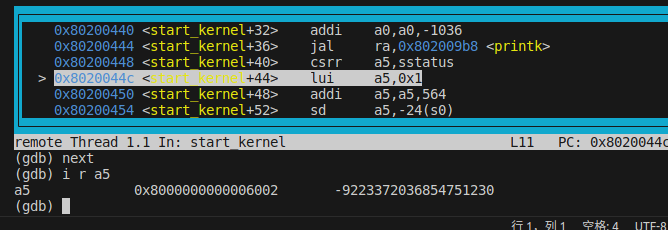


3.

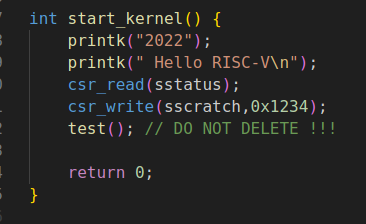


从上图可以看到，在main函数内调用了csr\_read宏和csr\_write宏，下面使用gdb调试来观察这两个宏的功能是否正确。



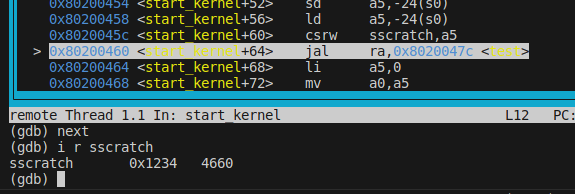


4.



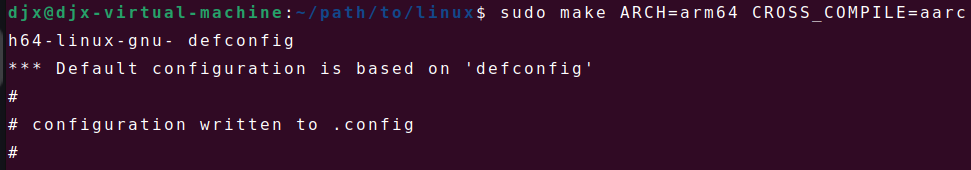
如上图，我们成功调用了宏csr\_read读到了寄存器sstatus的值，对照riscv手册可得知，sstatus的第二位（即SIE位）为1，说明此时CPU允许时钟中断；sstatus的第63位为1，表示是在RV64架构中。

根据上面的main函数截图可以看到，我们试图往寄存器sscratch中写入0x1234。

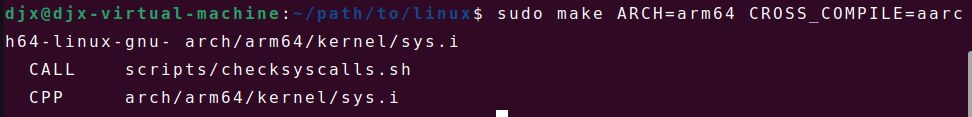


可以看到，在执行完csrw sscratch,a5后，sscratch寄存器的值的确变成了0x1234。

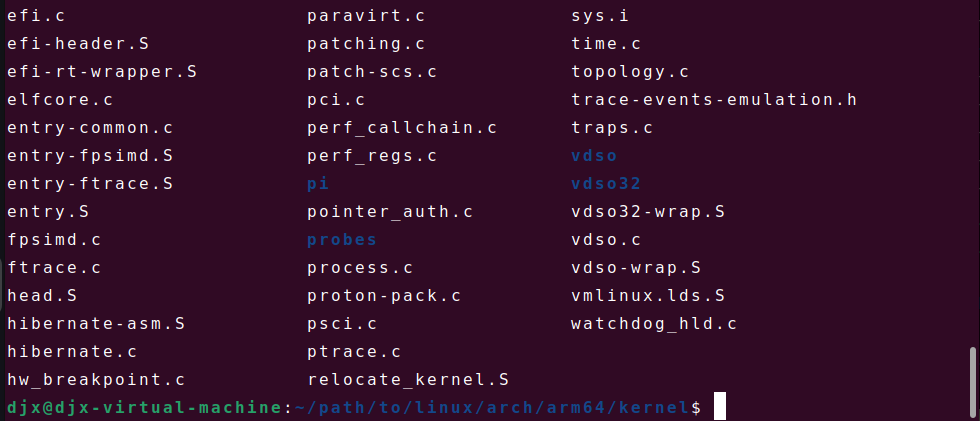
5.与lab0中的操作类似，执行make指令进行交叉编译，然后指定生成的文件sys.i，如下图。首先进入Linux内核目录，然后执行如下图的make指令。



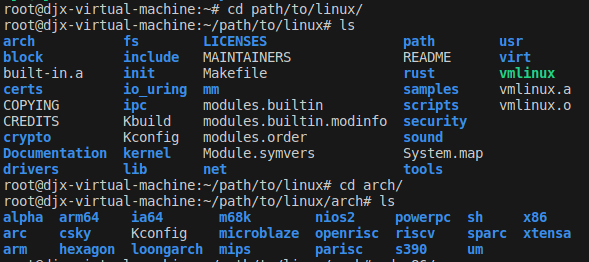
选择正确的路径进行编译，指定生成文件sys.i。



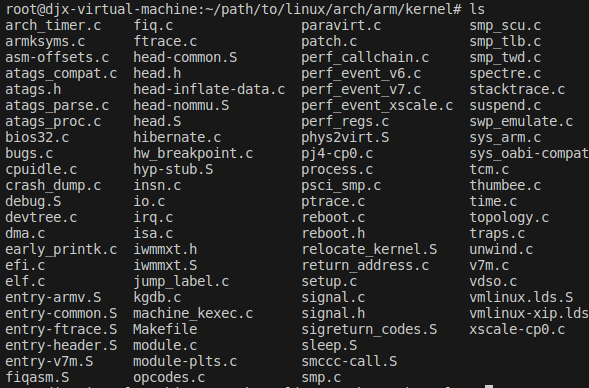
进入arch/arm64/kernel/，可以看到sys.i成功生成。（见下图右上角）



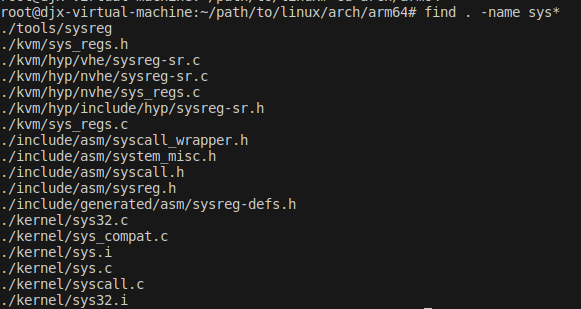
6.



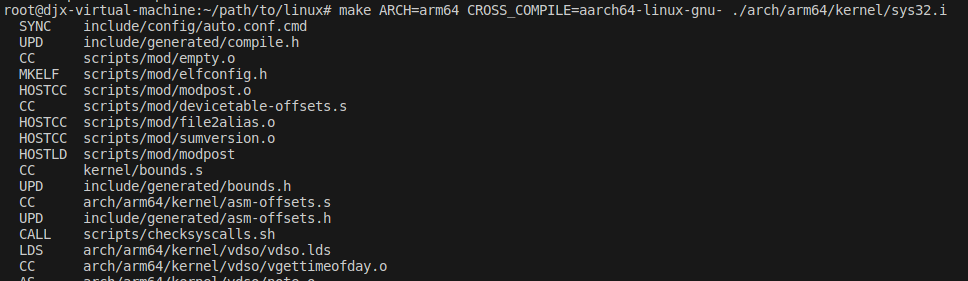
进入arch文件夹，可以看到各种架构。

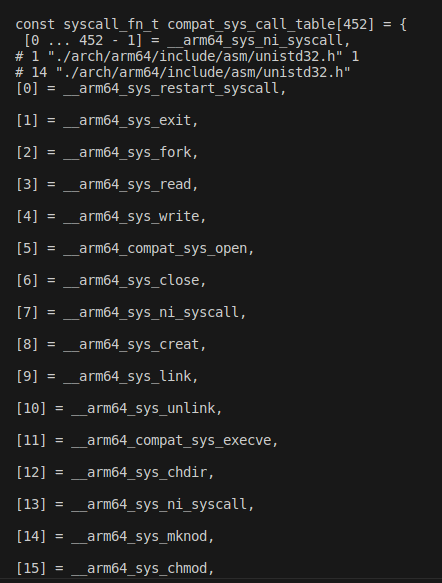


我们首先进入arm64架构，然后使用find命令找到系统调用表sys，并进行查看，如下图。

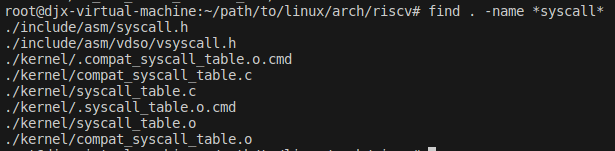


可以看到sys.c是arm64的系统调用表，sys32.c是arm32的系统调用表，下面我们对sys32.c进行预编译并查看系统调用表。

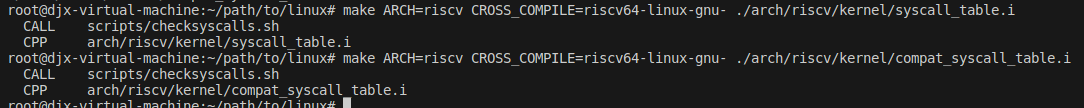




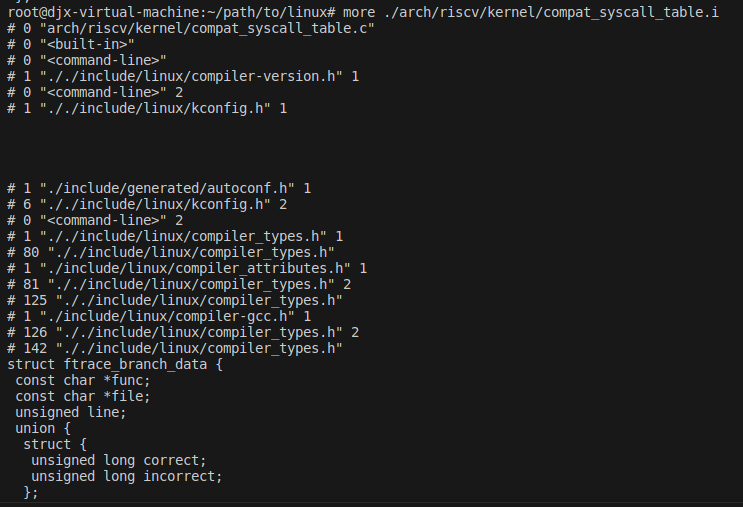
Riscv架构也类似，我们使用find命令找到所有和syscall有关的文件如下。

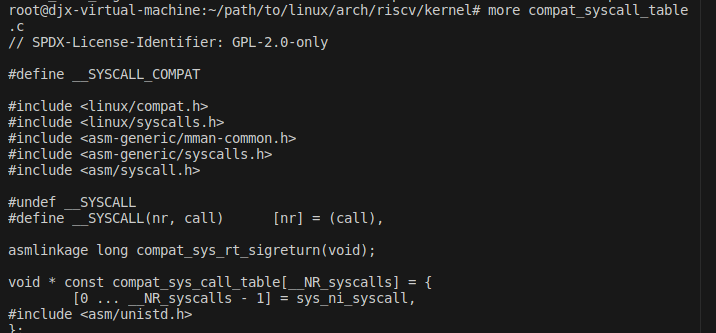


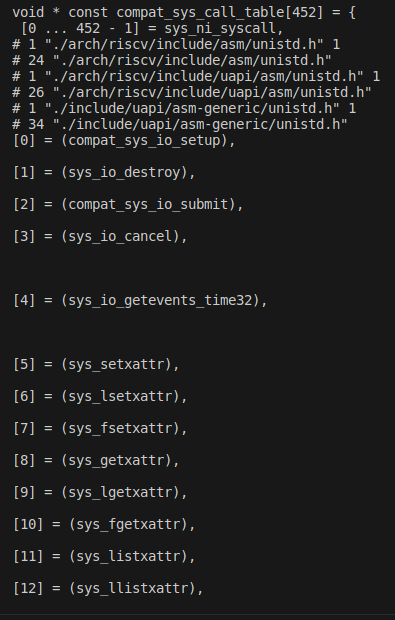
文件名称为syscall\_table。下面进行编译获得中间产物进行查看。



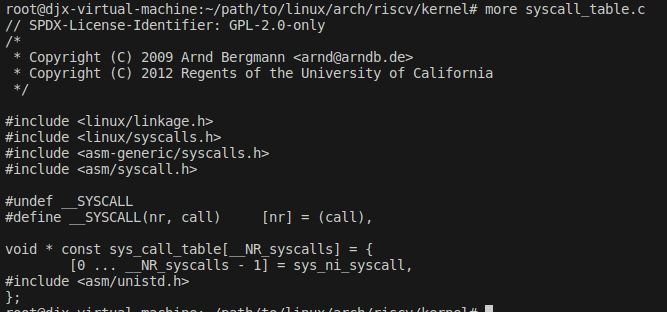
下图是riscv32的系统调用表文件compat\_syscall\_table.c 和 compat\_syscall\_table.i。

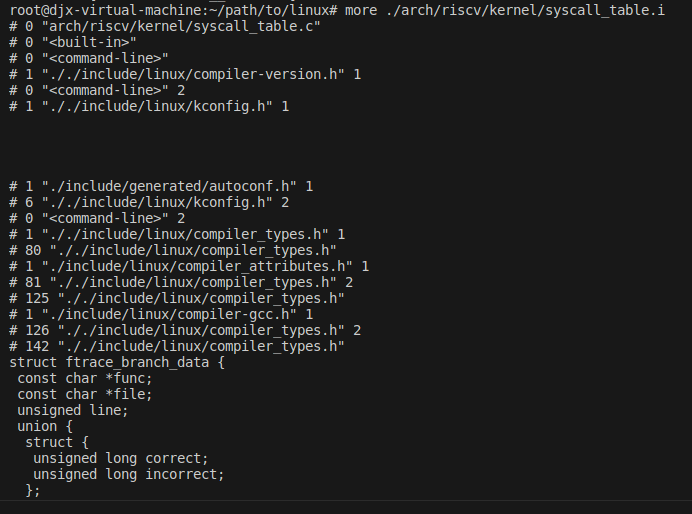


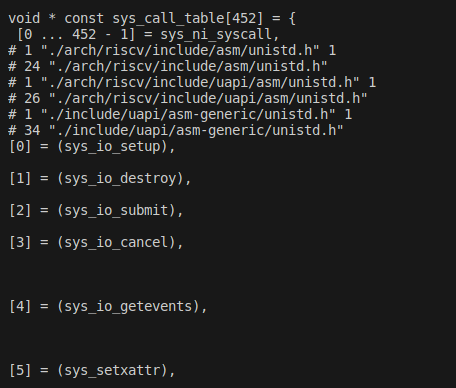




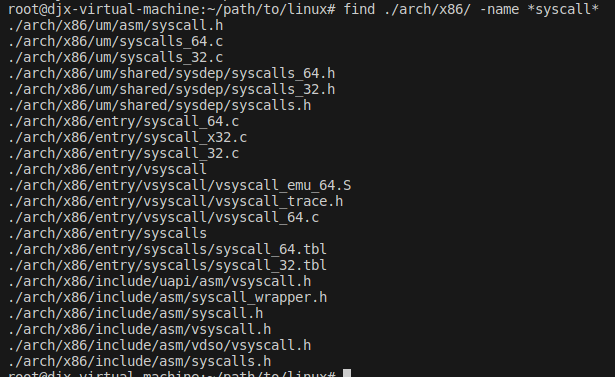
下面是riscv64的系统调用表文件syscall\_table.c 和 syscall\_table.i。



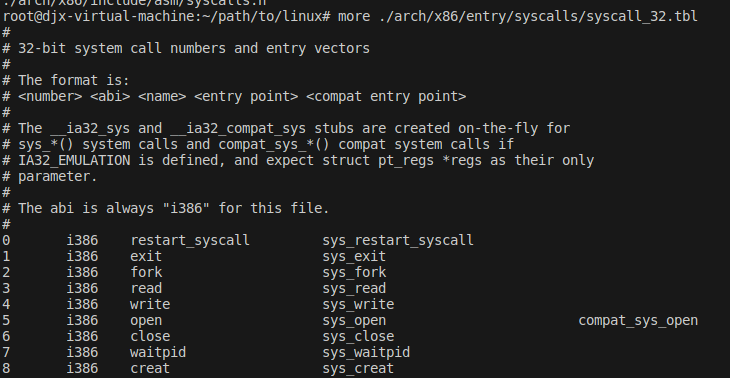


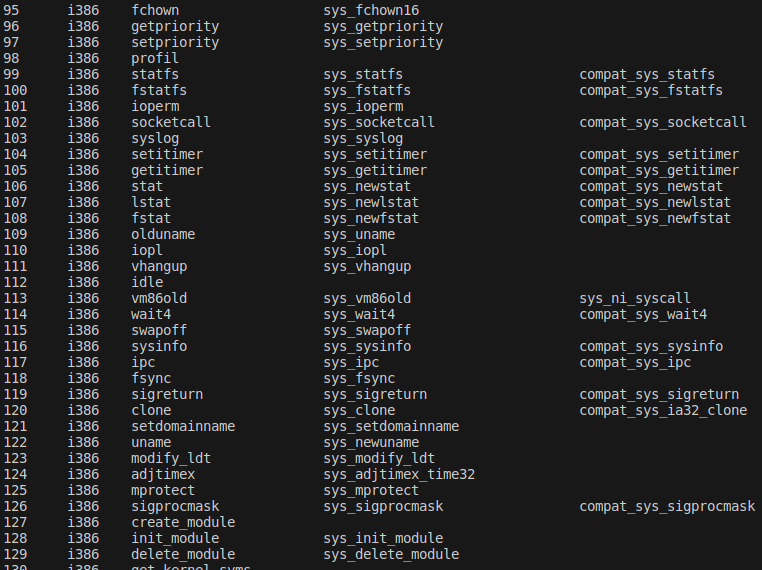


下图为使用find命令寻找x86架构的系统调用表位置。

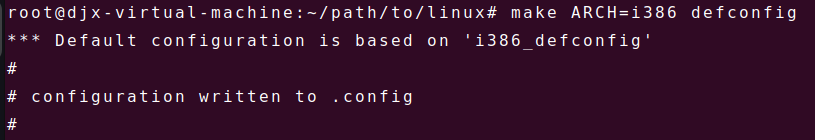


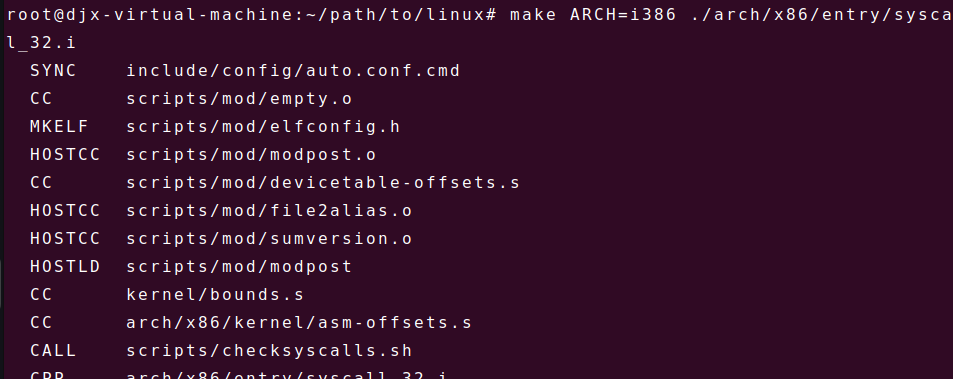
查看syscall\_32.tbl文件，即x86（32位）系统调用表宏拓展.



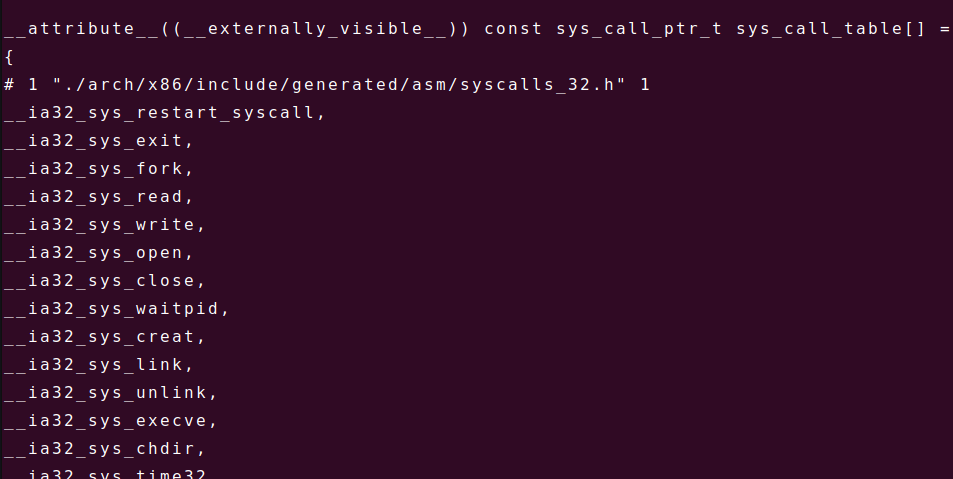


或者也可以编译系统调用表.c文件获得宏展开.i文件，如下图。

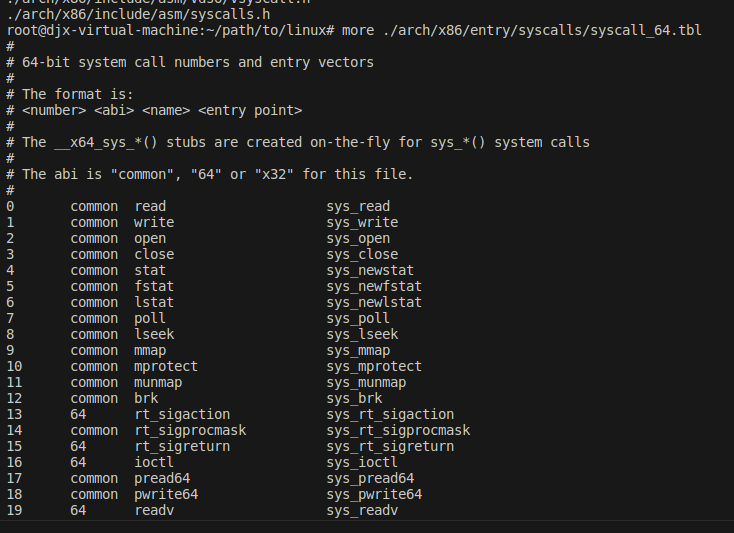


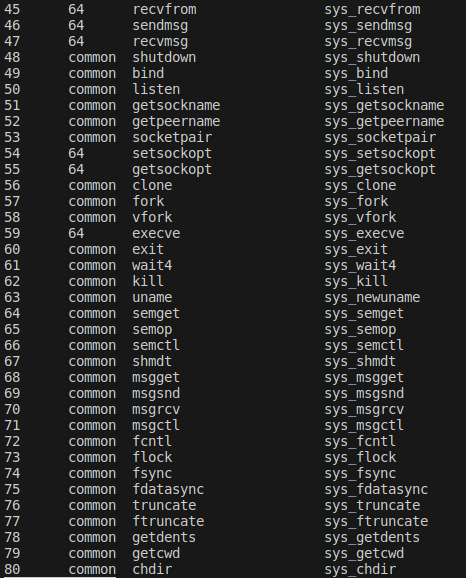


查看该syscall\_32.i文件，查看系统调用表宏展开。

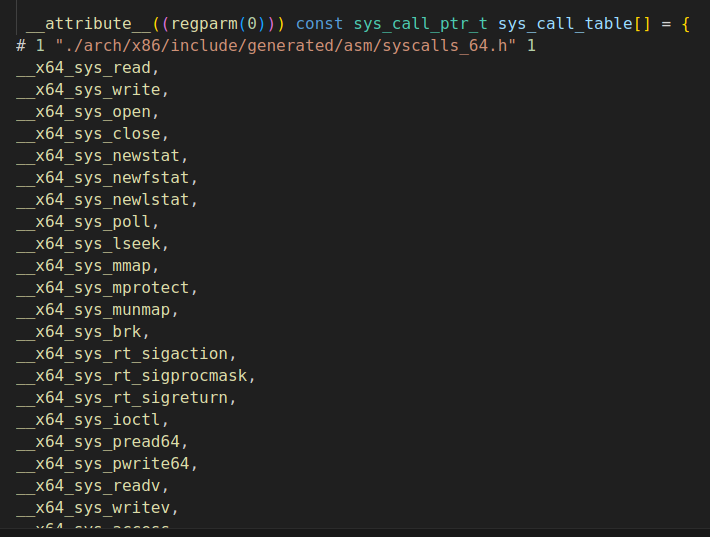


下图为x86\_64的系统调用表。

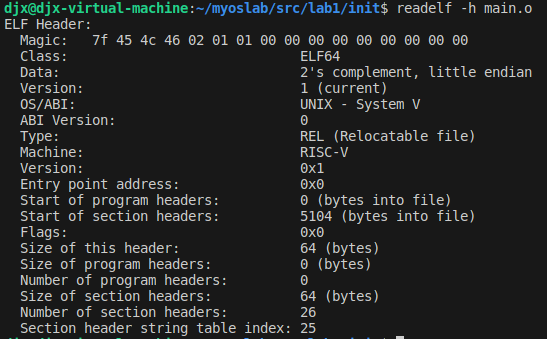




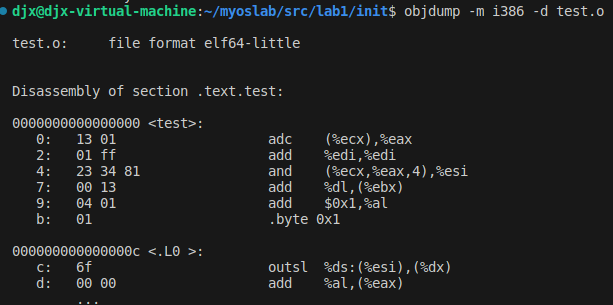
同样可以使用编译系统调用表.c文件获得宏展开.i文件，如下图。



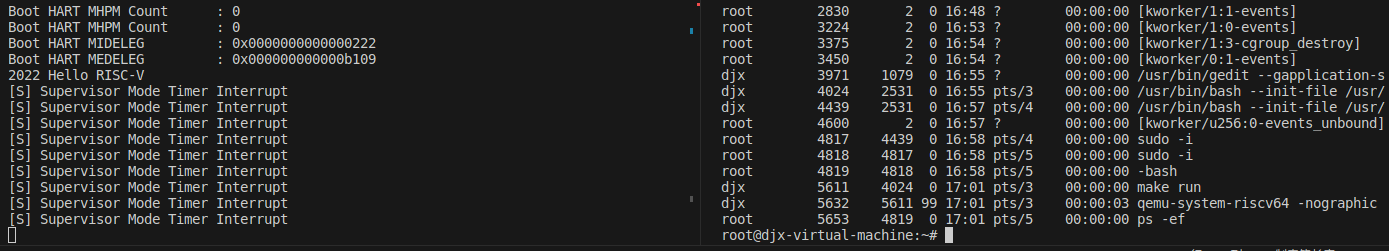
7.ELF，全称为Executable and Linkable Format，是在Linux系统中的一种可执行可链接的二进制文件。下面以lab1中的一些elf格式文件为例。



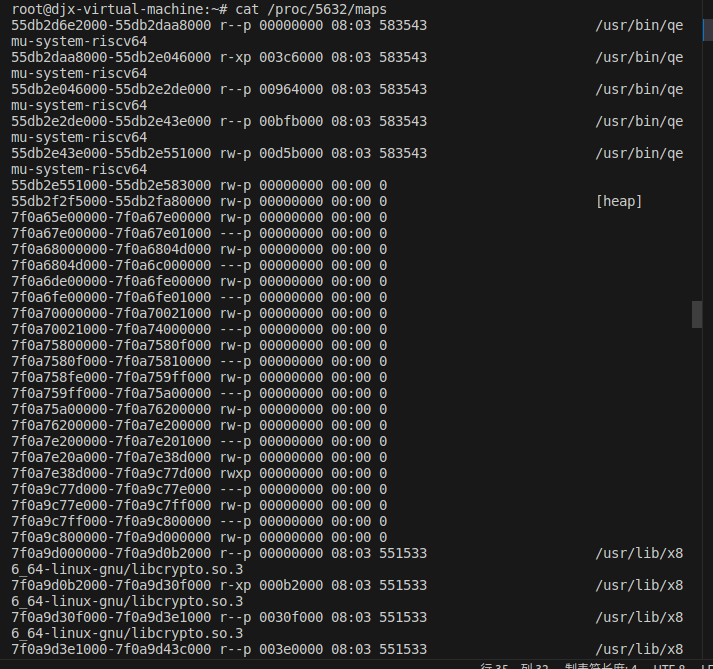
上图显示了elf文件main的文件头信息。



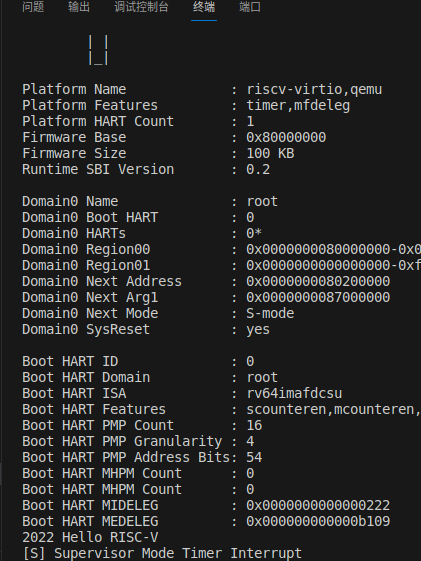
Objdump命令是检查二进制文件，可用于反汇编和检查二进制代码。

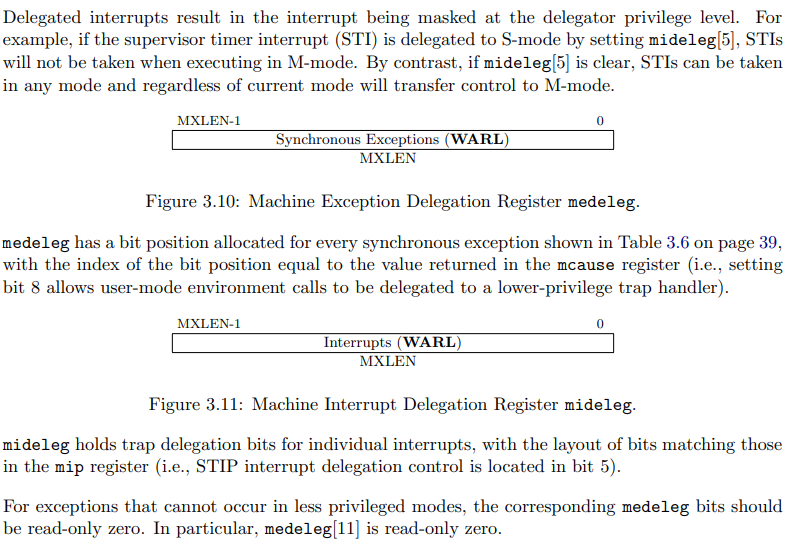


运行内核程序，执行ps命令查看进程号，然后执行cat命令查看内存布局。



8.





通过查阅手册可以得知，MIDELEG和MEDELEG分别代表了M mode中断委托寄存器（Machine Interrupt Delegation Register）和M mode异常委托寄存器（Machine Exception Delegation Register）。这两个寄存器决定了哪些中断异常可以被HART（处理器核）处理。

下面解释MIDELEG 值的含义。可以看到，该寄存器第1位，第5位和第9位（最低位为0位），为1，其他为0。第1位为1表示S mode下软件中断处理开启。第5位为1表示U mode下计时器中断开启，第9位为1表示S mode下计时器中断开启。

讨论心得

本次实验任务较多，需要学习较多的前置知识。在这个过程中我遇到的问题有不少，首先是在编写汇编代码时， #return form traps 部分一开始并不知道该写什么，甚至试图使用jalr x0,ra指令，后来在了解ecall机制后，正确使用sret返回用户模式。还有个问题是head.s中一开始忽略的栈顶指针sp的作用因此无法正确启动kernel，最终在查阅各种资料学习过后加上la sp,boot\_stack后正确运行。其他的类似问题也都能通过查阅手册或者上网搜索得到解决。这次实验使我对于特权模式的切换以及环境调用的机制、OS启动原理有了更深入的了解，也让我学会了更多的汇编指令，对我大有裨益。