**浙江大学实验报告**

**课程名称：**操作系统

**实验项目名称：**RV64内核引导与时钟中断处理

**学生姓名：**  **学号：**

**电子邮件地址：**

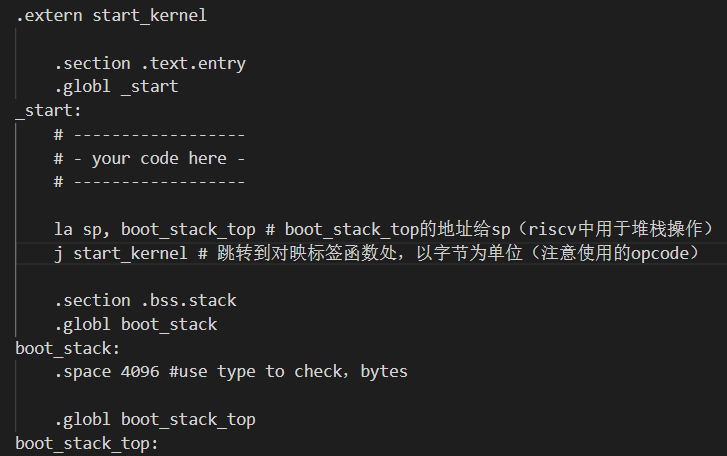
**实验日期：** 2023年 10 月 8 日

1. **实验内容**
   1. RV64内核引导

Part 1

最开始，我们需要在start\_kernel函数，即运行第一个C函数之前，设置程序栈的大小，在本实验中我们将此栈的大小设置为4KB，并将该栈放置于.bss

.stack段，再通过跳转指令，运行main.c中的start\_kernel函数。具体head.S文件设置代码如下：

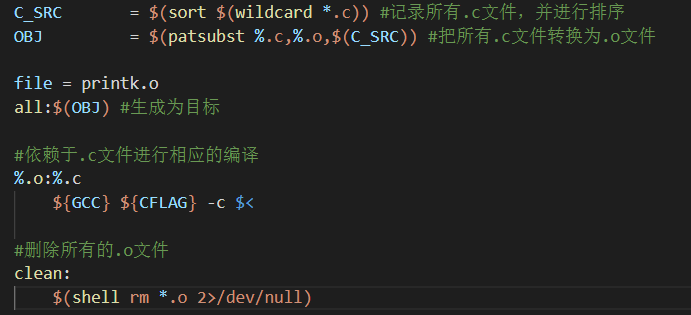


其中，,section表示riscv汇编语言的程序段，我们将boot\_stack，与boot\_stack\_top这两个global变量放置于.bss.stack段（本来就用于存储未定义的数据），并将需要执行的代码放置于.text.entry段。我们均设置了对映的标签，便于我们后续的访问。

另外，对于.bss.stack而言，我们声明它的space大小为4096（数值默认设定为0），这以字节为单位，对映于我们4KB的栈内存需求。最后，在代码实现部分，我们需要将book\_stack\_top标签对映的地址赋值给sp寄存器，用于确定使用栈的区域，再使用伪指令j跳转至我们需要运行的start\_kernel函数处。

Part 2

在本文件夹中，我们存有大量的.c和.h文件，我们可以学习makefile规则后，用其来实现我们的编译过程，生成目标vmlinux运行文件。



这里使用的makefile文件代码，借鉴了同一父文件夹下其他子文件夹的makefile文件。在代码实现中，我们先记录所有.c文件并排序，便于后续的统一声明处理；接着，我们在命名方面将所有的.c文件替换为.o文件，因为.c文件的编译产物往往为.o文件，因而在此我们可以用统一的变量名称来便于后续操作；进而，我们可以使.c文件进行相应的编译操作，生成对映名称的.o文件；最后，我们额外设定一个伪操作，用于make调用时清除所有的编译中间产物。

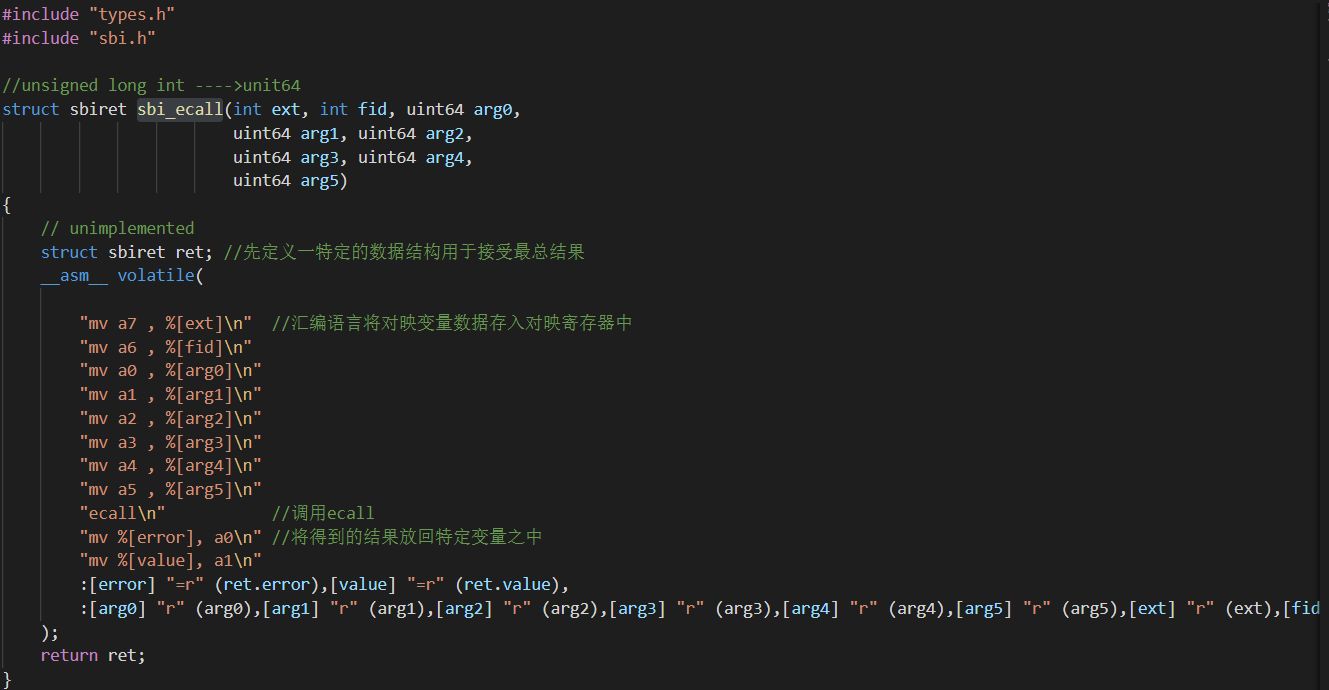
Part 3

因为本实验我们需要借助OpenSBI平台，但OpenSBI运行在M态，因而我们需要为S态提供一些接口，帮助其借助OpenSBI的功能。在这里，我们以完成sbi\_ecall函数为例。在完成sbi\_ecall函数时，我们需要使用内联汇编指令来完成寄存器内容的读写、M模式下特定功能的调用，以及函数返回值的返回。

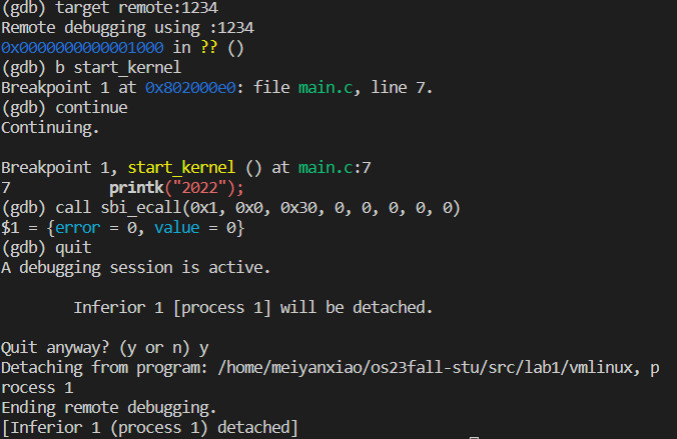
另外，sbi\_ecall函数在后续实验中可以借助使用的功能如下，其中Function ID对映于int fid输入参数，Extension ID对映于int ext输入参数。



函数具体的内联汇编代码实现如下，我们首先定义函数返回值结构，并将对映的函数输入参数与返回值结构属性连接至汇编语言中；再使用riscv汇编语言，对我们已经记录的数据进行指定功能的操作。其中，ecall伪指令会自动为我们转交OpenSBI并完成相应的操作。



此时，我们使用 sudo make debug指令尝试运行我们编译好的vmlinux。并使用gdb进行对映的调试。在远程连接，断点设置以及continue运行之后，我们可以call sbi\_ecall函数，并传递相应的参数（第三个参数表示我们需要输出的信息对映的ASCII编号）：



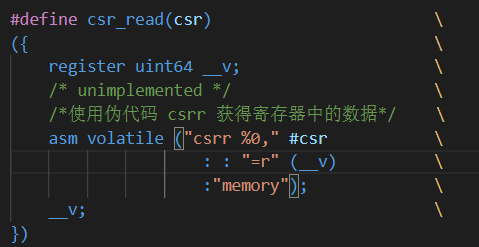
此时我们可以在OpenSBI界面得到如下结果：



由结果我们可以看到，OpenSBI输出结果为0，是ASCII码对映0x30的数值，这在一定程度上证明了我们程序代码编写的正确性。

Part 4

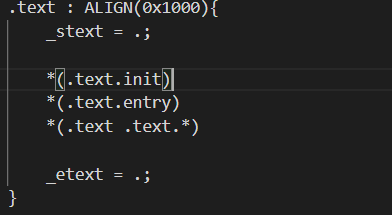
（此模块感觉本实验没有显示的使用）

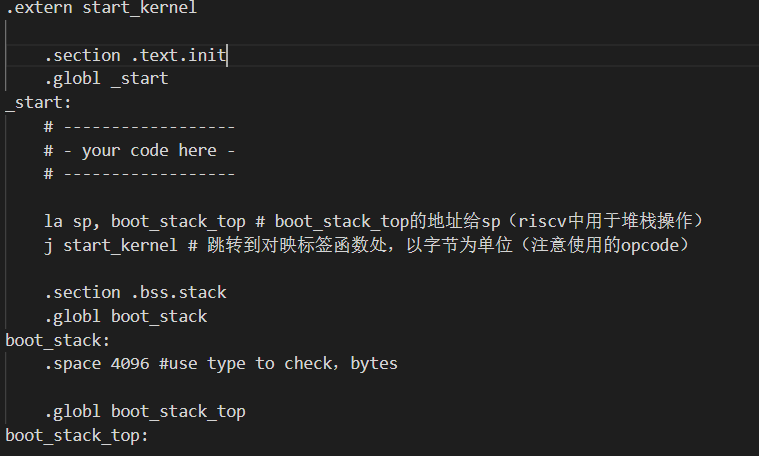
另外，我们需要使用内联汇编代码，填充def.h文件，在实现csr寄存器写的基础上，实现寄存器读这一功能。结果如下：

由结果看出，我们只是效仿csr\_write函数，使用csrr伪指令读取了对映寄存器中的数值，并将其存储于\_\_v变量之中。

1.2 RV64 时钟中断处理

准备工作，修改vmlinux.lds以及head.S文件。修改结果如下：

vmlinux.lds修改结果

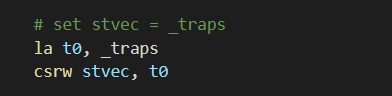
head.S修改结果

此次修改的结果，表示我们将全局变量\_start以及对映的准备程序代码，重新定位于.text.init代码段。

Part 1

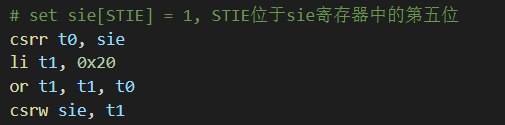
在运行start\_kernel函数之前，我们需要对上面提到的CSR寄存器进行初始化。

（1）设置stvec寄存器，stvec寄存器可以在trap发生时，记录trap对映的信息，便于对映的处理函数进行相应的程序处理。因而，我们在编写时可以将\_traps所表示的地址写入stvec寄存器中；这里我们采用Direct模式,而\_traps则表示trap处理入口函数的基地址。



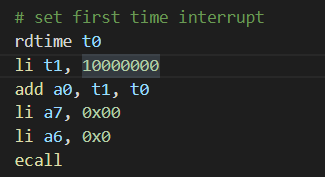
对于stvec这一寄存器而言，我们不能直接使用la指令进行数值的复制；因而我们只能采取这样间接的方式，借助于csrw伪指令，来介导\_trap信息的载入。

（2）接着我们需要将SIE寄存器的STIE为置位为1。其中SIE寄存器表示时钟中断的开启与否，而STIE位对应于其中的第5位。



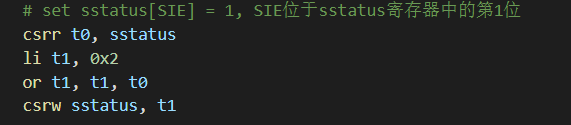
同理于stvec，sie等特殊的寄存器，还是需要使用间接的方式，借助于csrw伪指令来写入内容；且我们需要修改的只有sie寄存器的第五位，因而我们在写入时需要保证sie寄存器的其他bit不被随意修改。

（3）设置本实验中第一次时钟中断。类似于clock\_set\_next\_event()函数（后续会详细展开）的实现，只是在此我们需要使用汇编语言进行编写。



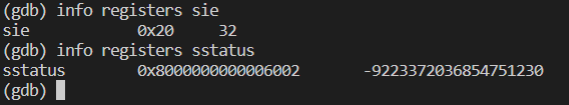
我们可以使用rdtime伪指令获取时钟寄存器中时钟，因为QEMU中时钟的频率是10MHz，相当于1秒钟内含有10000000个时钟周期；所以我们在获取时钟周期数的基础上，可以加上10000000个时钟周期，表示加上1秒钟后，触发第一次时钟中断。

由上给出的sbi\_ecall函数的解析可知，我们需要为Function ID与Extension ID设置对映参数（对映于寄存器a6与a7），以使其触发时钟中断的处理程序。最后，我们使用ecall伪指令，让OpenSBI帮助我们完成相应的操作。

1. 开启S态下的中断响应，即将sstatus状态寄存器的STI标志置1，其中STI为sstatus寄存器的第2位。sstatus寄存器主要负责跟踪记录处理器当前的运行状态。

同理，我们也需要借助csrw伪指令来改变sstatus寄存器中的内容。

我们可以借助gdb工具进行寄存器已有值的验证。

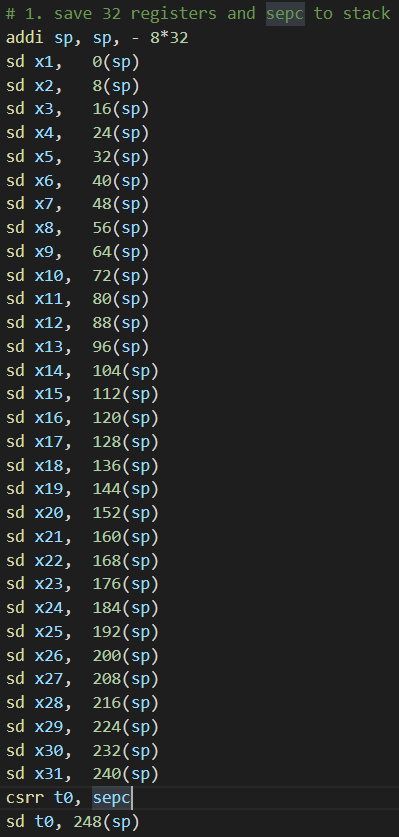


由gdb反映的数据我们可以看出，sie寄存器的第5位，以及sstatus寄存器的第1位，都被我们成功的置为1。

Part 2

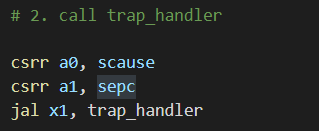
在触发中断之后，我们需要记录当前运行状态下的上下文信息，以至在中断处理结束后，我们可以恢复这些上下文，继续执行指令。

我们可以额外添加一个汇编语言文件，来实现识别中断并进行跳转的功能。

1. 我们需要在介导中断处理程序前，保存各个通用寄存器以及sepc寄存器的信息，以至中断处理结束后，我们可以继续执行原有指令。

存入时无需特别关注x2寄存器的存储次序（x2寄存器即sp寄存器，用于管理栈的各项操作），特殊寄存器sepc需要使用csrr进行间接读取。

（2）然后，我们需要将scause和sepc寄存器中的值传入trap处理函数trap\_handler（后续会详细介绍），并在trap\_handler中实现对trap的处理。



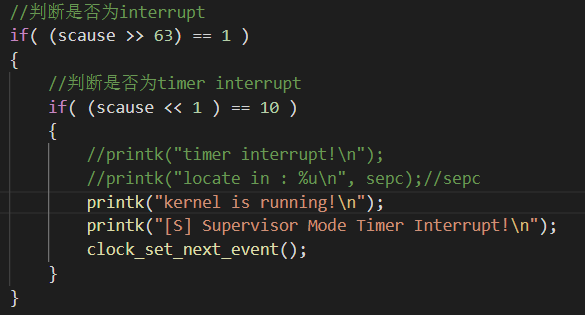
经过尝试我们察觉了，对于trap\_handler函数而言，a0、a1两个寄存器分别用于传递scause和sepc寄存器中的存储数据（对映参数）。存储完函数参数后，我们直接使用jal指令进行跳转即可。

（3）在完成对trap的处理之后，我们需要恢复通用寄存器以及sepc寄存器中的原有数值。特别注意，因为x2寄存器对映于sp寄存器，需要进行栈的管理，因而在从栈中取出对映的寄存器数据时，我们需要最后读取x2寄存器的数据，保证sp寄存器管理栈的正确性。

（4）最后，我们需要从trap函数中返回。一般而言，我们会选择ret与jal进行配套使用，这样可以使程序自动跳回x1寄存器处的地址。但是在S态内核态时，我们需要使用sret伪指令，才可以完成这个跳回动作。

Part 3

在这一部分，我们需要实现trap的处理函数。具体实现如下：

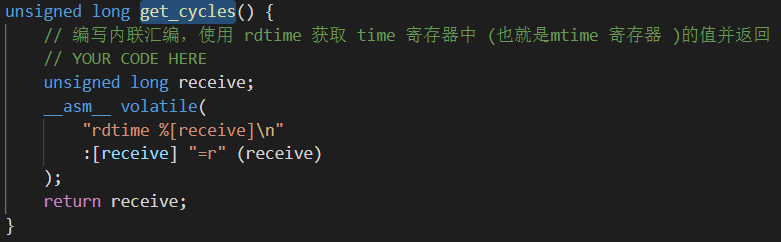


首先，这个函数含有两个输入参数，对映于scause和sepc寄存器中的数据。根据获得的知识，我们首先需要判断scause寄存器中最高位是否为1，只有当它的最高位为1时，才表示发生了interrupt。接下来，我们需要判断scause寄存器的后63位，只有当scause寄存器的后63位表现为5时，才表明系统发生timer interrupt。此时，我们可以进行对映的时钟中断处理操作。

Part 4

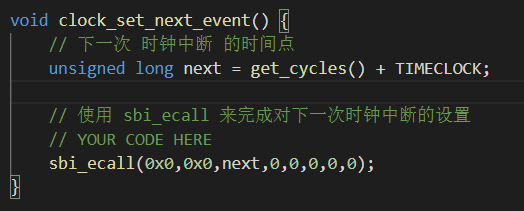
在最后，我们需要补充完成实现时钟中断的相关函数。

首先是get\_cycles()函数，我们需要使用内联汇编语言代码，去获取时钟寄存器中断时钟周期数。



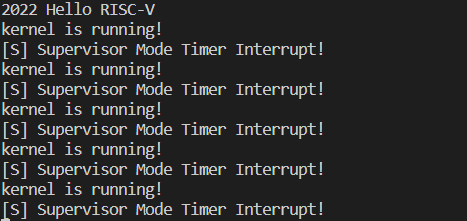
使用rdtime伪指令，读入receive变量中即可。

接下来是clock\_set\_next\_event()函数，用于定义下一次时钟中断发生时间。



在此已完成下一轮时钟中断的时间设定（1秒钟之后），现在只需要为sbi\_call()函数设置对映的参数，再进行简单的调用即可。

完成上述实验步骤后，我们使用sudo make run指令，可以得到以下结果：

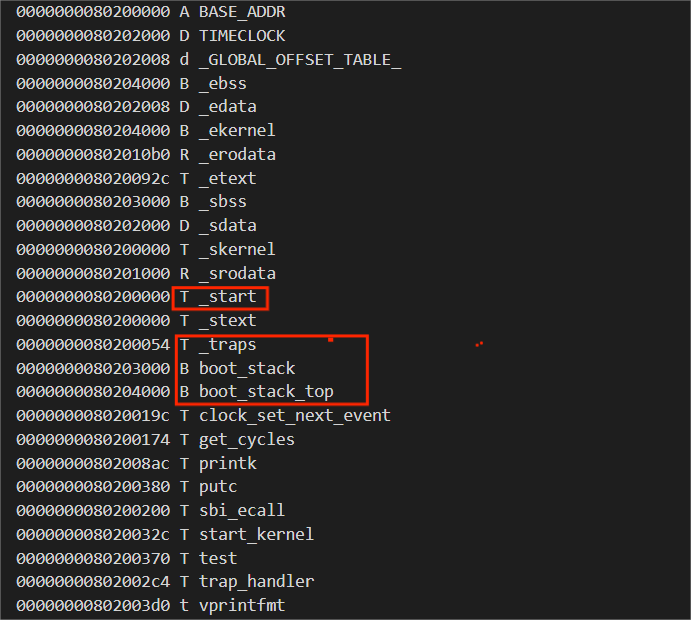


我们的linux内核确实实现了大约1秒1次的时钟中断处理响应。

1. **思考题**
2. RISC-V的调用约定（Calling Convention）是一套规则和约定，用于定义函数之间如何传递参数、返回值以及保存寄存器状态等。RISC-V的调用约定通常称为ABI（Application Binary Interface），它有不同的规范级别（例如，GABI、EABI、ILP32、LP64等），常见的规定有：通用寄存器使用规范（可以接受临时变量与否），参数传递规范（通常由x10与x11传递参数），返回值传递（通常使用x10作为返回值接收者），通常使用x2寄存器为栈指针等。

其中，Caller Saved Register表示需要由caller函数保存的寄存器，因为在进行调用的过程中，这些“临时”寄存器可能会被调用程序给修改，因而caller需要保存这些寄存器的数值。对于Callee Saved Register而言，caller认为这些寄存器在函数调用前后不会发生改变，因而caller不会主动保存这些寄存器，对于这些寄存器而言，就需要callee进行对映的保存。

2.自定义符号的值（如\_start，book\_stack，book\_stack\_top）如下：



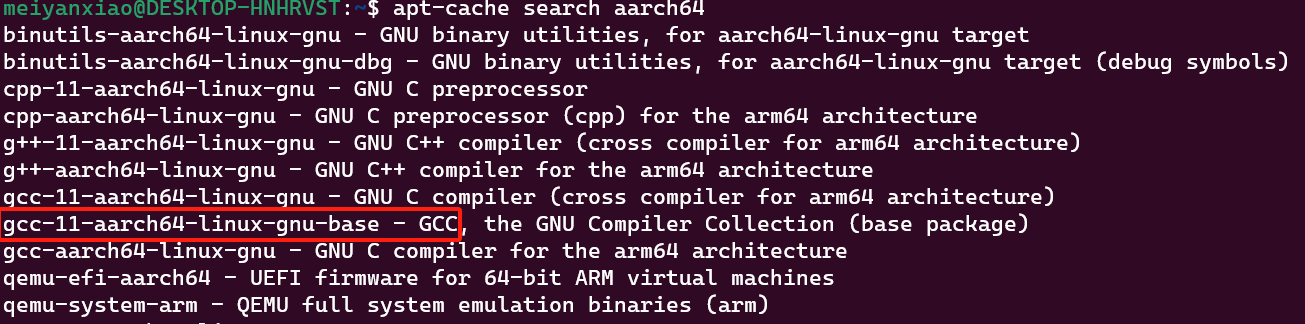
810b2bab6fdd6498681a4434dd761a43.使用csr\_read从sstatus寄存器中读出数据。

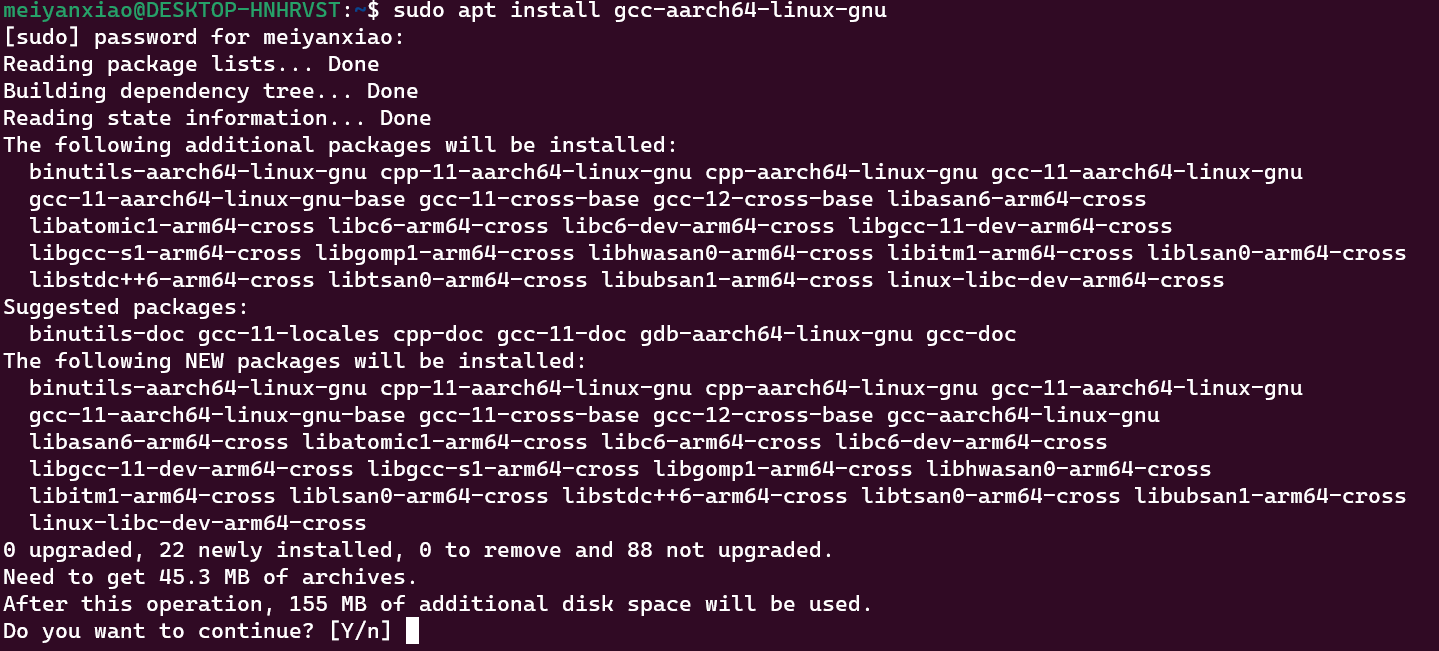
sstatus寄存器的数值为0x00006002，意味着SIE位置被置1，UBE、SPIE位置也同样被置1；SPIE位指示在捕获到监控模式之前同样启用了监控中断。UBE 位则是RISC-V的sstatus寄存器中的一个字段，用于启用或禁用用户态下的中断处理。当 UBE 位为 1 时，允许用户态下的中断触发和处理。

4.使用csr\_write向sscratch寄存器中写入数据。

a73f8426951f0bdb734222fad7f6f41我们先使用csr\_read函数读取sscratch寄存器中的数值，发现数据为0；接着，使用csr\_write向其中写入数据1；再次使用csr\_read从中读出数据，发现数值为1，说明我们写入的成功。

1. 首先，我们使用“apt-cache search aarch64”指令搜索包含 aarch64 的软件包，一般是交叉编译工具。



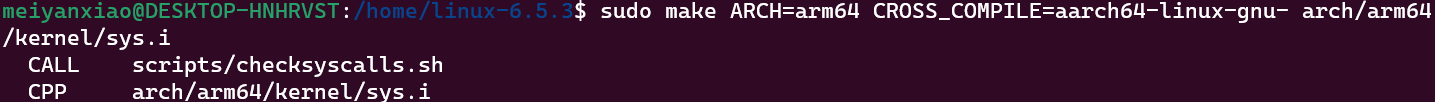
我们发现在搜索结果中含有gcc-11-aarch64-linux-gnu，直接使用“sudo apt install gcc-aarch64-linux-gnu”指令安装。

安装完成后，我们先使用“make ARCH=arm64 CROSS\_COMPILE=aarch64

-linux-gnu- defconfig”指令进行默认配置的设定。

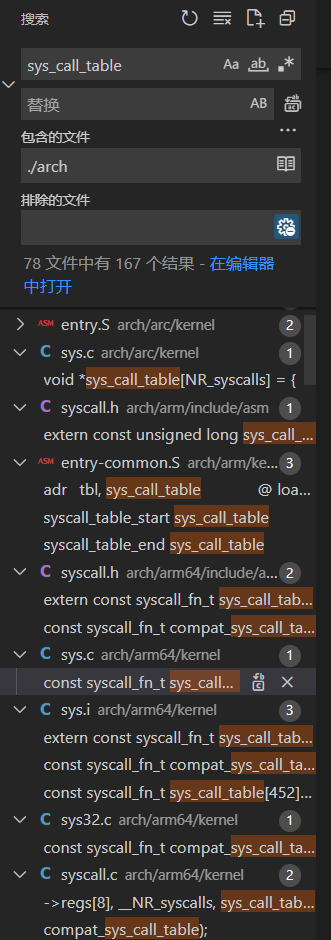
接着，我们使用“make ARCH=arm64 CROSS\_COMPILE=aarch64-linux-gnu-

arch/arm64/kernel/sys.i”指令生成目标文件。

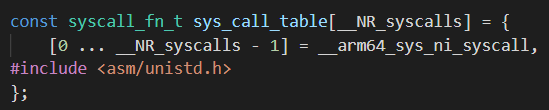


1. 我们的目标是寻找linux内核在不同架构下的system call table，一方面，我们可以在相应的文件夹下，使用“system call table”指令进行寻找。

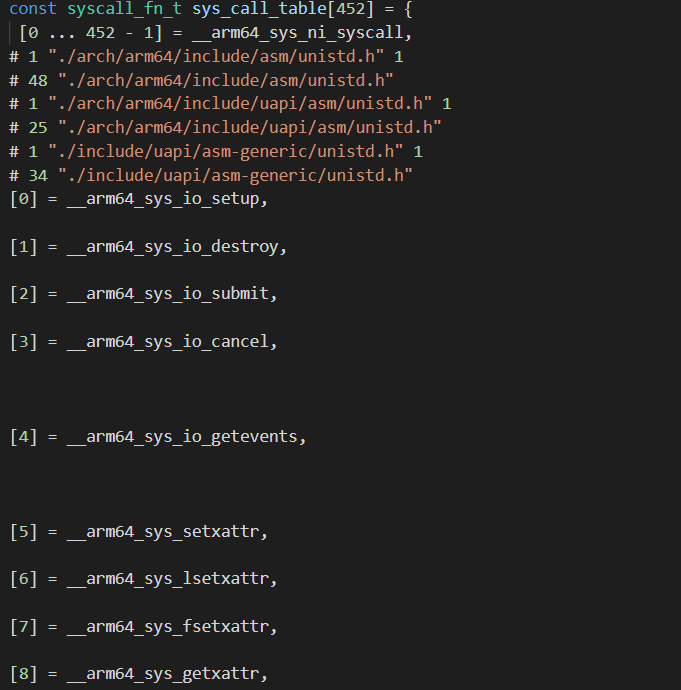
我们以arm64架构进行演示。



接着打开对映的文件（sys.c），我们可以从代码中寻得system call table的痕迹。



接着我们联想到，在5中我们对sys.c文件的预处理文件sys.i进行了获取，而C语言文件的预处理过程会进行宏展开，因而在sys.i文件中，我们应该可以寻得arm64架构下system call table的全部宏展开形式。

部分宏展开结果

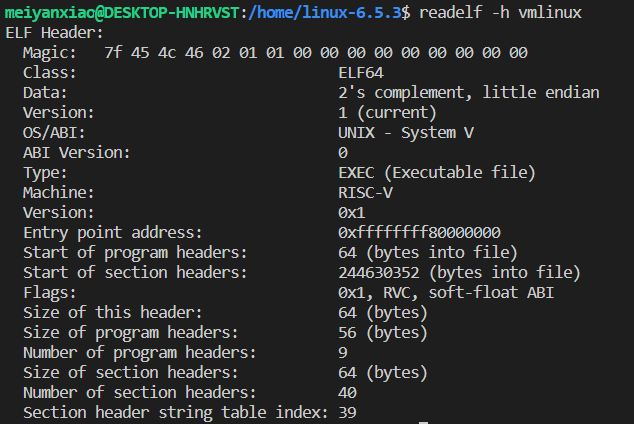
1. ELF的英文全称是The Executable and Linking Format，最初是由UNIX系统实验室开发、发布的ABI(Application Binary Interface)接口的一部分，也是Linux的主要可执行文件格式。主要的ELF文件的种类主要有三类：

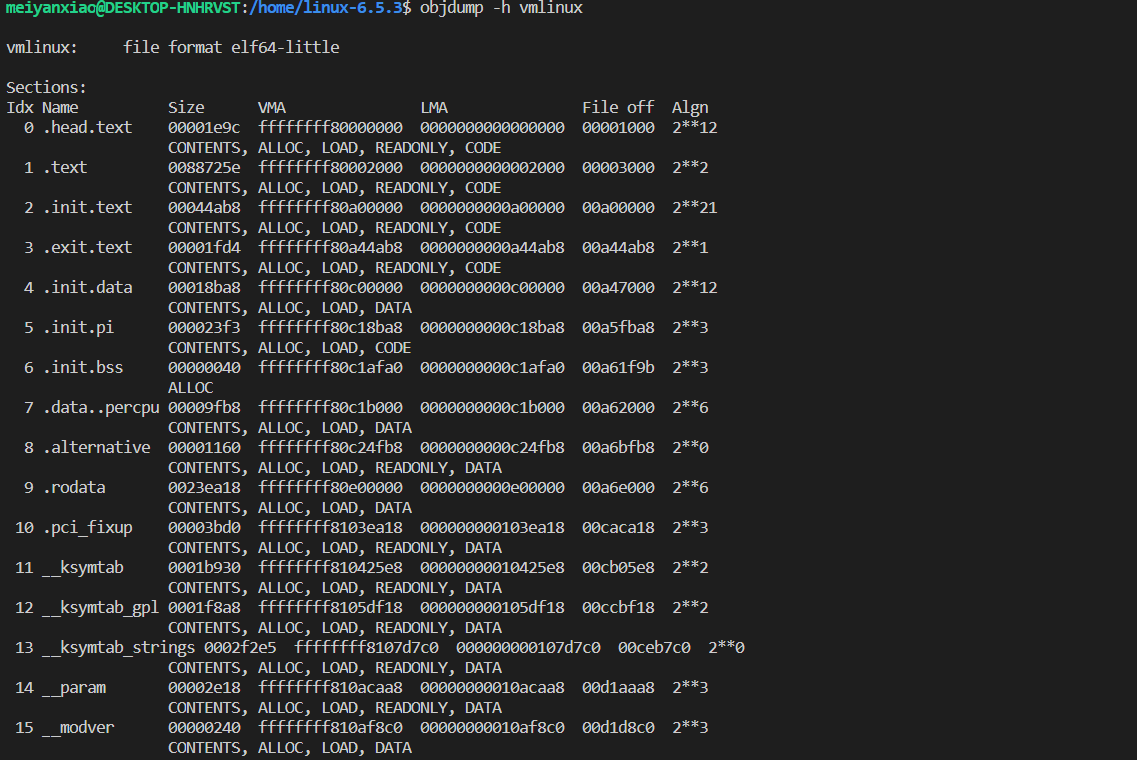
可执行文件（.out）：包含代码和数据，是可以直接运行的程序。其代码和数据都有固定的地址，系统可根据这些地址信息把程序加载到内存执行。

可重定位文件（.o文件）：包含基础代码和数据，但它的代码及数据都没有指定绝对地址，因此它适合于与其他目标文件链接来创建可执行文件或者共享目标文件。

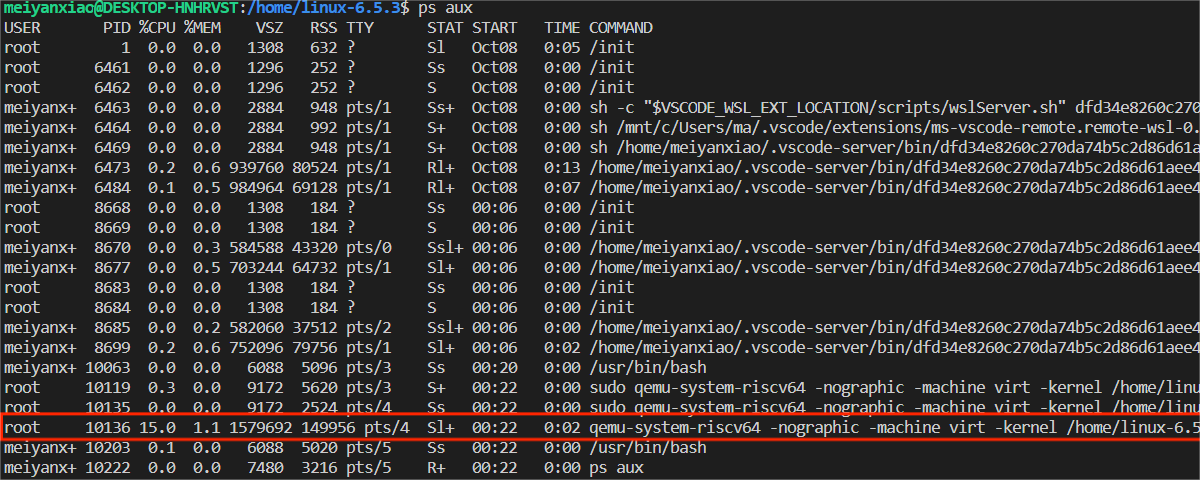
共享目标文件（.so）：也称动态库文件，包含了代码和数据，这些数据是在链接时被链接器（ld）和运行时动态链接器（ld.so.l、libc.so.l、ld-linux.so.l）使用的。

接着，我们以vmlinux（elf文件）为例，进行其header内容的展示。我们可以使用“readelf -h vmlinux”指令对其头信息进行查看。

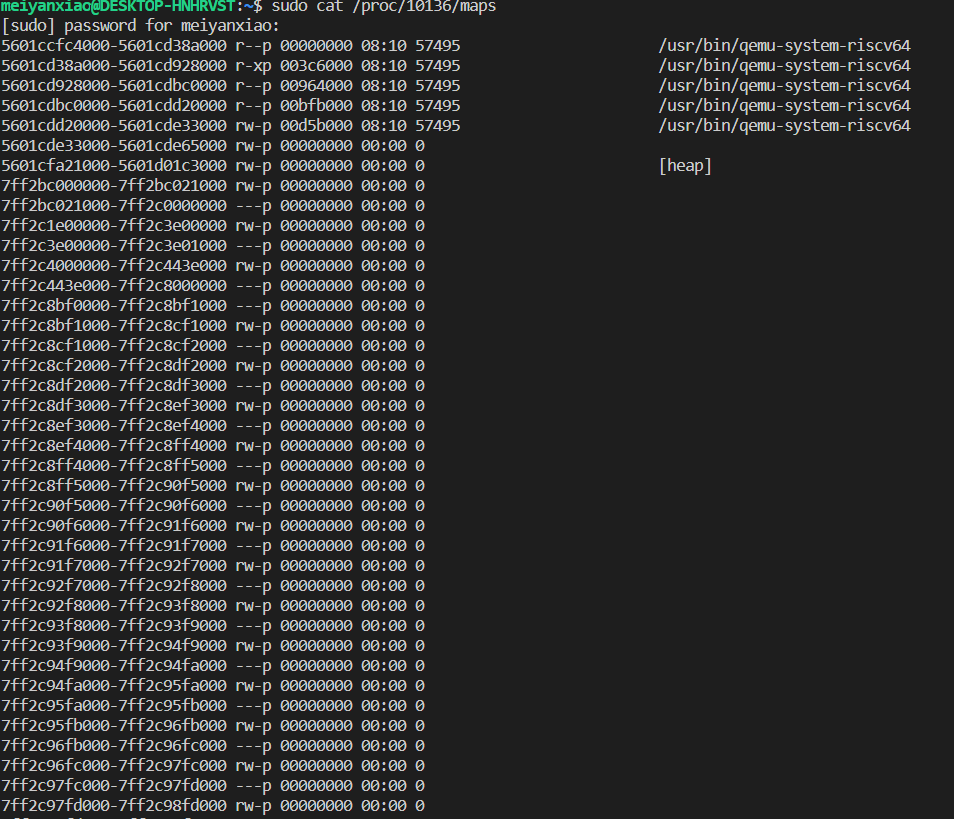


由结果我们可以看出，使用指令后，我们可以得到一些关于elf文件的头信息，如数据的表示与存储方式、ABI接口版本等内容。同样，我们也可以使用“objdump -h vmlinux”指令进行头信息展示。

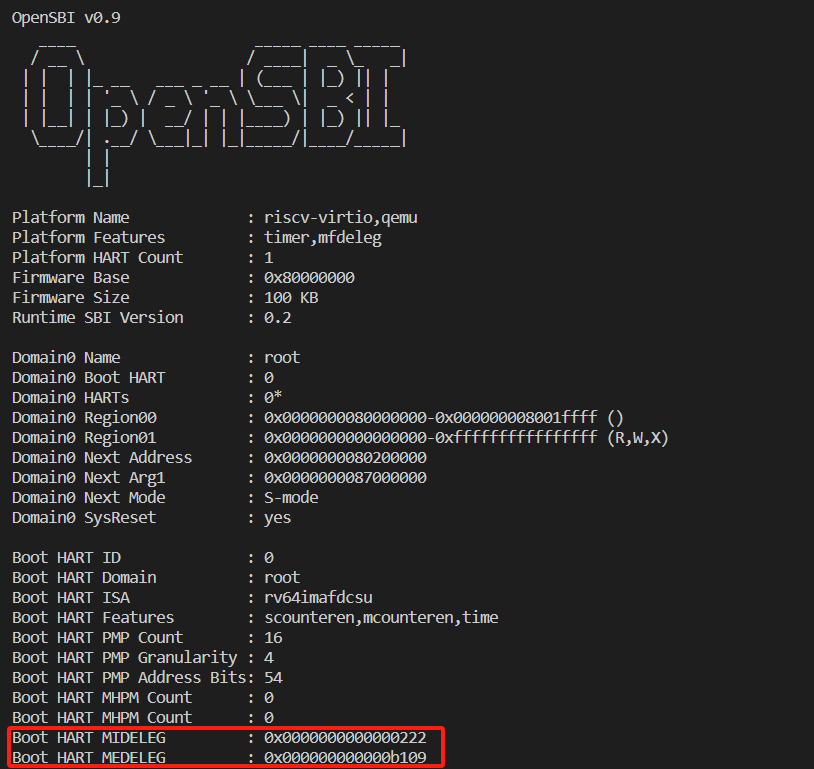
我们借助qemu平台对vmlinux进行运行。此时使用“ps aux”指令查看系统正在运行的进程，并寻找我们需要程序运行时对映的PID。



从结果中我们可以看到，vmlinux运行的PID信息为10136，接下来，我们可以使用“cat /proc/10136/maps”指令查看对映的内存使用。

 部分内存使用信息

8.我们得到的结果如下。在OpenSBI中，Boot HART MIDELEG 是控制特权级切换时 Machine-Level的中断委托设置。MIDELEG 寄存器用于控制哪些异常或中断可以被委托给更低特权级的处理器模式执行。这在处理器启动和初始化过程中非常重要，因为在启动时需要确保正确的异常处理机制。

****

1. **讨论、心得**

首先，还是想浅浅Q一下makefile规范的学习文件。对于给出的链接，我进行了认真的学习，但在实际makefile文件编写的过程中，我却发现学习与实践完全是两个level；虽然规则中已经提出了很多便捷的处理办法，但是对于实际的makefile文件运用来说，通用性上还是大大不足的。

另外，在本次实验中，我也碰到了很多的问题，在此一一总结，也算是警醒自己，下次不再犯这些简单的细节错误。

最主要的是第一次接触的内联汇编语言编写，对于输入输出以及可能需要内存/寄存器的三个“:”，需要特别注意其中的使用规范，位置标椎，这些都会影响C语言文件的正常编译。另外，内联汇编语言的“，”使用也十分讲究，在编写输入输出的变量/寄存器时，对于最后的变量，我们不能擅自添加“，”，不然也会导致编译错误的发生。

另外，在完成本次实验后，我也尚存一些不理解的问题。

1. 我们可以看到，在main.c文件的start\_kernel()函数中，存在一个死循环函数test()，用于把控程序的不退出无限循环状态，以保证时钟中断的正常触发。但是，我在注释掉test()函数后，程序依然正常的进行时钟中断触发操作，可能PC已经漫天飞舞但是对这个设定的时钟中断影响不大，还想请教其中的原因。
2. 在其他架构交叉编译的测试模块，我们对于预处理产物的生成，需要使用“arch/arm64/kernel/sys.i”指令；在这里，路径使用需要特别注意，只能使用相对路径，使用绝对路径会导致“target不存在”等奇怪的错误，多次尝试测试才发觉出了这个问题，也想请教一下。