**OS Lab**

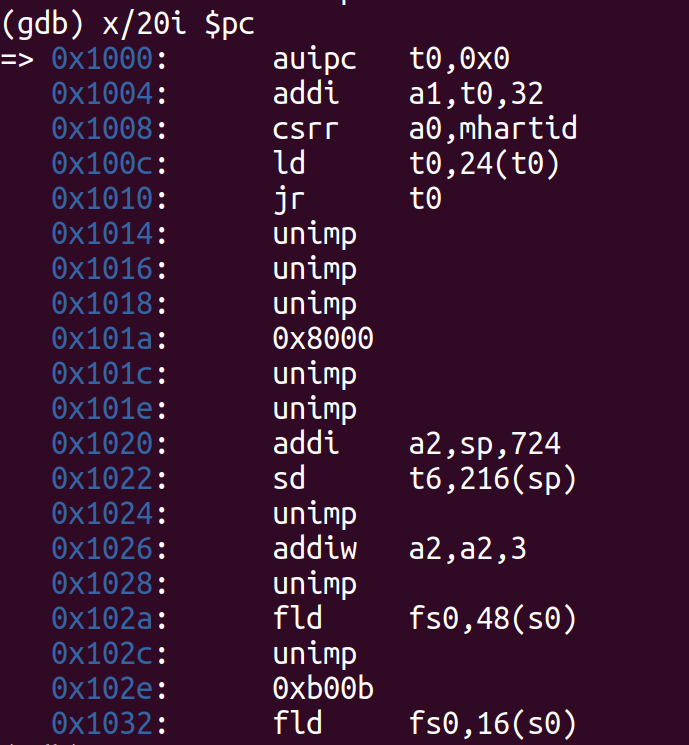
**Lab 0.5**

**练习1 使用GDB验证启动流程**

为了熟悉使用qemu和gdb进行调试工作,使用gdb调试QEMU模拟的RISC-V计算机加电开始运行到执行应用程序的第一条指令（即跳转到0x80200000）这个阶段的执行过程，说明RISC-V硬件加电后的几条指令在哪里？完成了哪些功能？要求在报告中简要写出练习过程和回答。

首先，通过cd命令进入操作系统内核代码的目录，然后使用make debug命令启动QEMU虚拟机进行调试。为了使系统在第一条指令处暂停，需要在另一个终端中使用make gdb命令。程序在地址0x1000处暂停，这是由于0x1000是程序的入口点。复位过程中，所有系统组件包括CPU、内存和外围设备都被重置到初始状态。Bootloader负责初始化硬件、配置系统参数并加载主操作系统。

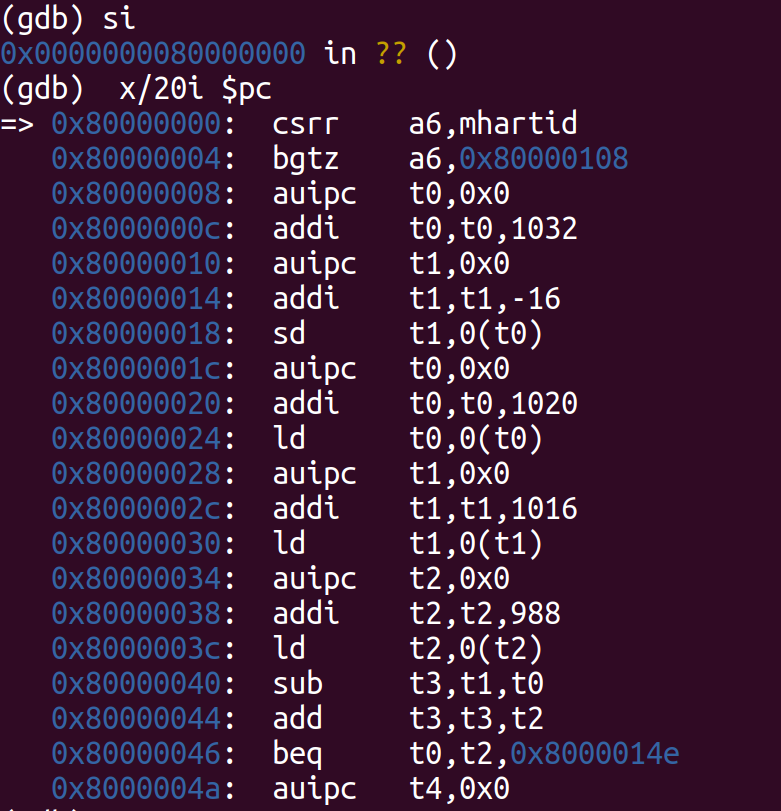
通过GDB命令x/20i $pc查看程序计数器PC当前指向的地址0x1000后的20条汇编指令。这些指令是系统复位后首先执行的代码，它们的作用如下：



1. auipc t0, 0x0：这条指令将程序计数器（PC）的高20位与立即数0x0相加，结果存入寄存器t0。
2. addi a1, t0, 32：将t0寄存器的值加上立即数32，结果存入寄存器a1。
3. csrr a0, mhartid：从控制状态寄存器mhartid读取当前硬件线程的ID，并将其存储到寄存器a0中。
4. ld t0, 24(t0)：从t0寄存器指向的内存地址加上偏移24的位置加载数据到t0寄存器。
5. jr t0：这条指令使得程序跳转到0x8000地址，是Bootloader的起始位置。

接下来的指令（从0x1014开始）不再是系统复位上电时执行的代码。

接着通过GDB的si命令单步执行，直到执行jr t0指令，跳转到物理地址0x80000000，这是Bootloader的启动位置。然后，使用x/20i $pc命令查看该地址后的20条汇编指令。



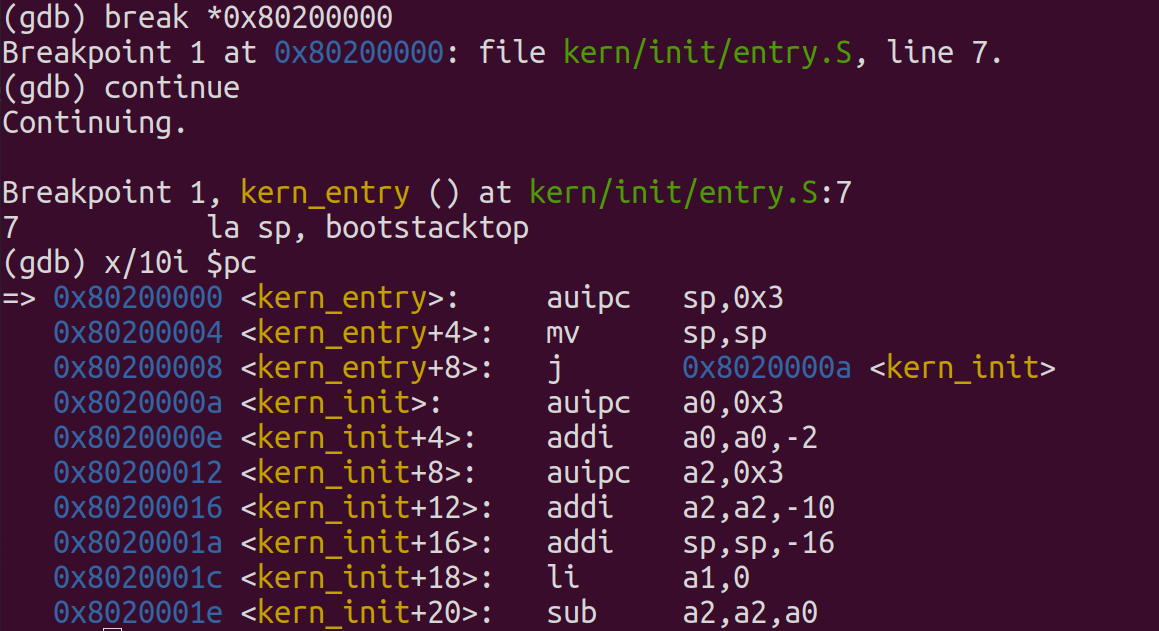
我们对汇编指令分析如下：

1. csrr a6, mhartid：从CSR寄存器读取当前硬件线程的ID到a6。
2. bgtz a6, 0x80000108：如果a6的值大于0，跳转到地址0x80000108。
3. auipc t0, 0x0 和 addi t0, t0, 1032：组合用于计算内存地址。
4. sd t1, 0(t0)：将t1的值存储到t0指向的内存地址。

通过GDB调试，可以在0x80200000处设置断点，0x80200000是QEMU模拟器为操作系统内核指定的加载地址。执行到该地址时，可以看到系统启动的标志。在kern\_entry函数中，第一条指令是auipc sp,0x3，它将栈指针设置到合适的位置。接下来的指令包括跳转、加载地址、调整栈指针等操作。

|  |  |
| --- | --- |
|  |  |

kern\_entry()是操作系统内核的入口点，位于内核的启动代码中。当程序执行到地址0x80200000时，GDB命中了之前设置的断点，并显示了断点信息。这里显示Breakpoint 1, kern\_entry () at kern/init/entry.S:7，表明断点位于kern\_entry函数，该函数定义在文件kern/init/entry.S的第7行。在kern/init/entry.S文件的第7行，代码是la sp, bootstacktop，这条指令的作用是初始化栈指针，使其指向内核启动时的栈顶。



1. 硬件复位：当系统加电时，硬件组件（包括CPU、内存和外设）被复位到初始状态。
2. 固件启动：固件（BIOS）开始执行，进行基本的硬件检测，并定位Bootloader的位置。
3. Bootloader执行：

* Bootloader是一个小型软件程序，它继续初始化硬件。
* 它负责加载操作系统内核到预定的内存地址，如RISC-V架构中常见0x80200000。
* Bootloader跳转到内核的入口点，开始执行内核代码。

1. 内核入口点：

* 内核的第一条指令通常位于物理地址0x80200000。内核入口点的代码负责初始化必要的寄存器和栈指针，设置系统运行的基础环境。
* 执行跳转到内核初始化函数。

1. 内核初始化：

* 内核初始化代码配置系统环境，包括内存管理、设备驱动初始化、文件系统挂载等。
* 内核初始化完成后，它会创建第一个用户空间进程，通常是系统初始化进程（如init进程）。

1. 用户空间进程启动：

* 系统初始化进程继续启动其他服务和应用程序，如窗口系统、网络服务等。
* 系统达到完全运行状态，准备用户登录和交互。

总之，操作系统的启动是一个从硬件加电到系统完全启动的连续过程，涉及多个组件的初始化和配置。每个阶段都为下一个阶段的成功执行奠定基础，确保了系统的平稳启动和运行。

**Lab 1**

**练习1：理解内核启动中的程序入口操作**

阅读kern/init/entry.S 内容代码，结合操作系统内核启动流程，说明指令la sp,bootstacktop 完成了什么操作， 目的是什么？tail kern\_init 完成了什么操作，目的是什么？

la即load指令，这句指令相当于把bootstacktop的值传递给了sp。bootstacktop是栈顶。操作系统初始化需要bootloader引导，而bootstacktop就是引导栈的起始地址。我们这两行指令实际上是给操作系统分配堆栈环境。

tail是一种特殊的函数调用指令，即尾部调用。尾部调用是当前函数的最后一个操作，直接跳转到 kern\_init函数，当前函数的上下文不再需要。

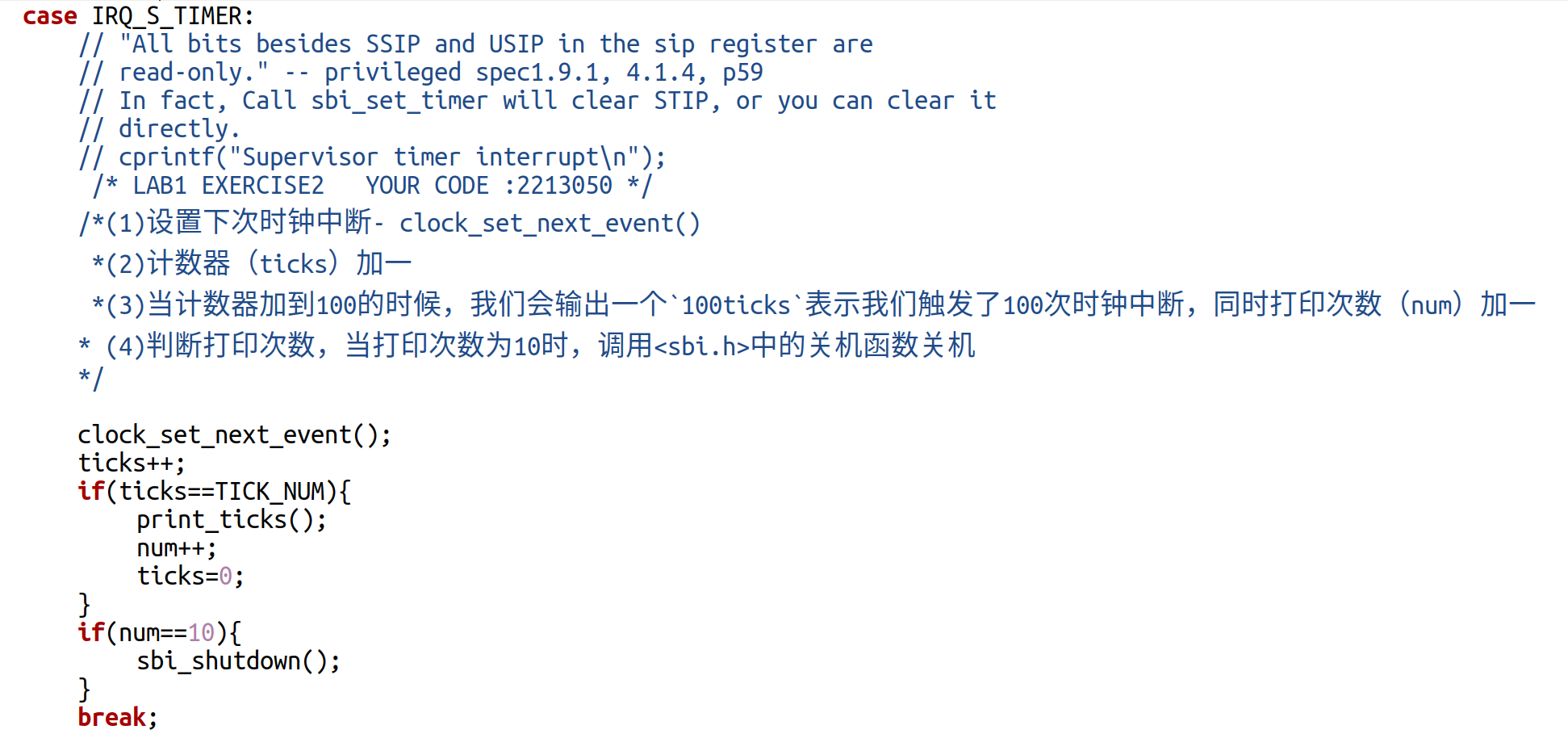
**练习2：完善中断处理**

请编程完善trap.c 中的中断处理函数trap，在对时钟中断进行处理的部分填写kern/trap/trap.c函数中处理时钟中断的部分，使操作系统每遇到100次时钟中断后，调用print\_ticks子程序，向屏幕上打印一行文字”100 ticks”，在打印完10行后调用sbi.h中的shut\_down()函数关机。

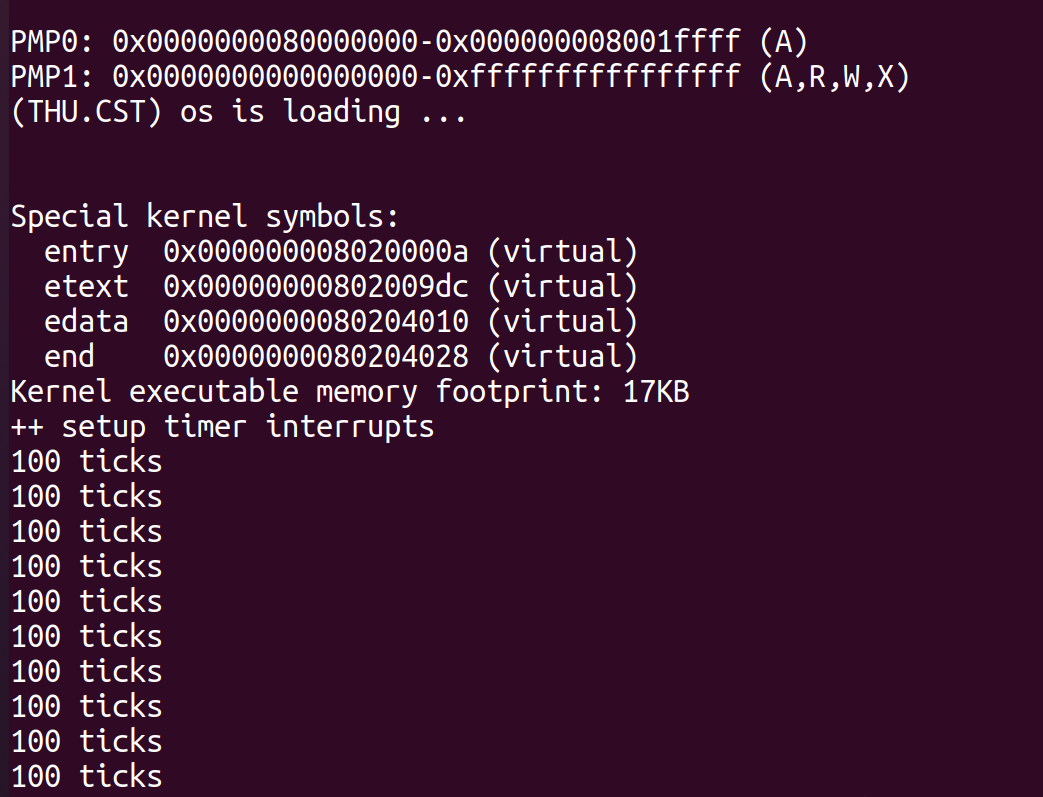
要求完成问题1提出的相关函数实现，提交改进后的源代码包（可以编译执行），并在实验报告中简要说明实现过程和定时器中断处理的流程。实现要求的部分代码后，运行整个系统，大约每1秒会输出一次” 100 ticks”，输出 10 行。

我们编写的代码如下：

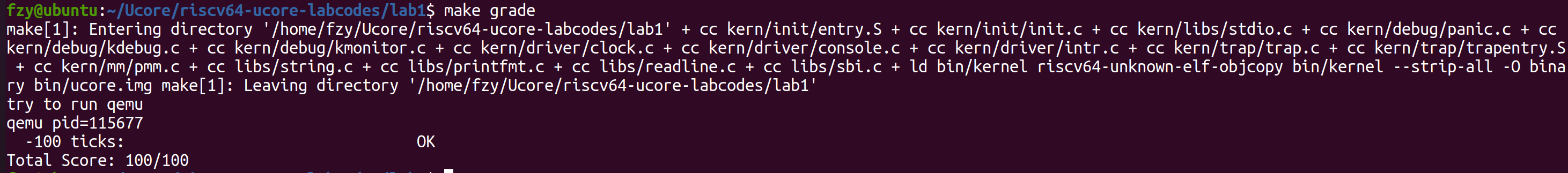
在trap.c的case语句中，找到对应的时钟中断处理。我们在这里每次先设置下次时钟中断，然后ticks++；每当触发了100次时钟中断之后，调用print\_ticks()函数打印输出。当打印十次之后，我们调用shutdown()函数关机。



我们输入编译指令make qemu即可看到打印的输出：

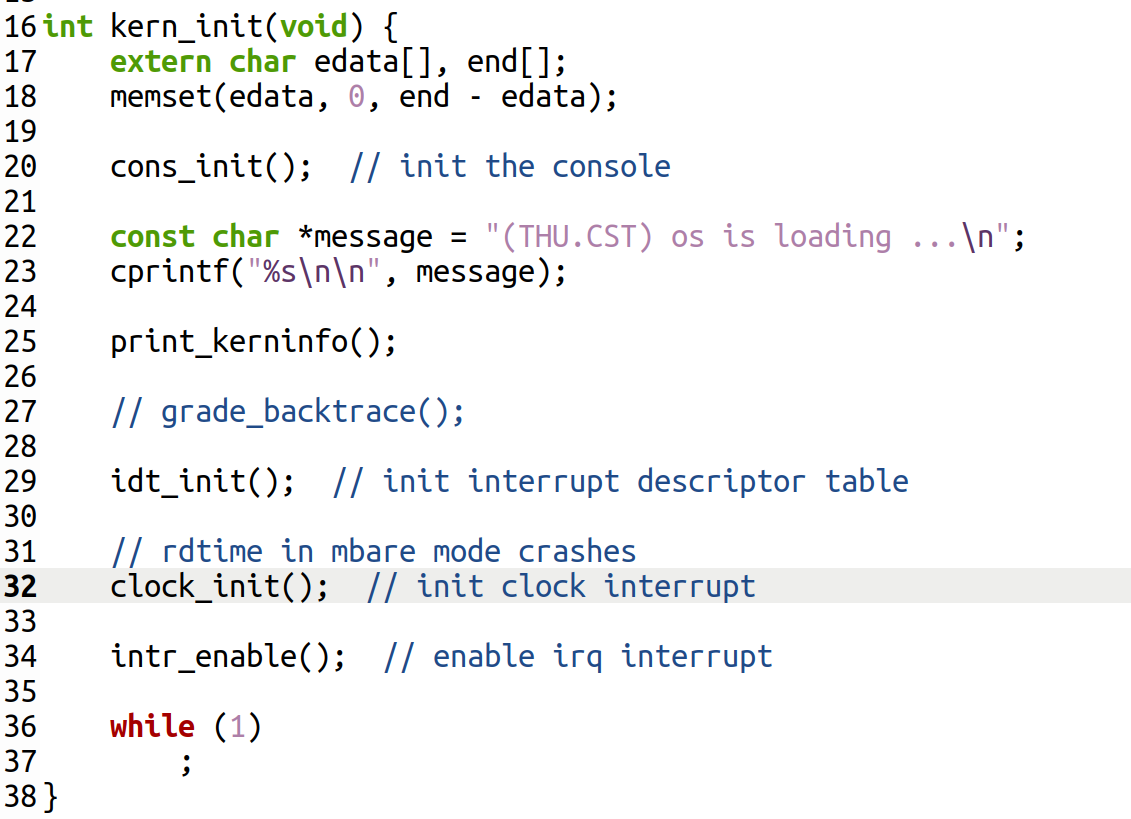


输入编译指令make grade即可获得评分：

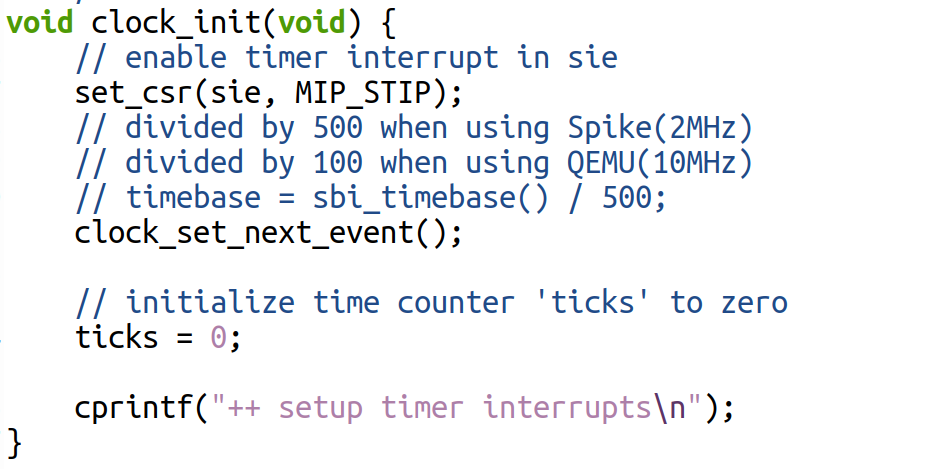


定时器中断处理的流程为：

在kern\_init函数里面，clock\_init()函数产生了第一次时钟中断，然后操作系统查找中断向量表，调用trap函数，根据中断类型交给不同的函数处理。



然后我们通过设置下一次时钟中断时间来继续产生中断。SIE（Supervisor Interrupt Enable，监管者中断使能）用于控制和管理处理器的中断使能状态。因此在初始化clock时，需要先开启时钟中断的使能。接着调用 clock\_set\_next\_event(void)设置时钟中断事件，使用 sbi\_set\_timer()接口，将timer的数值变为当前时间 + timebase，设置下次时钟中断时刻。



**扩展练习 Challenge1：描述与理解中断流程**

描述ucore中处理中断异常的流程（从异常的产生开始），其中mov a0，sp的目的是什么？SAVE\_ALL 中寄存器保存在栈中的位置是什么确定的？对于任何中断，\_\_alltraps中都需要保存所有寄存器吗？请说明理由。

1. 异常产生，被操作系统捕捉到。
2. CPU跳到中断向量表基址stvec，stvec最低两位是00，那么更高的二进制位存放的是中断处理程序的地址；如果是01那么存放的是中断向量表基址。操作系统通过不同的异常原因来索引中断向量表。本次实验我们是Direct模式，stvec直接跳到中断处理程序的入口点。
3. 我们定义真正的中断入口点\_\_alltraps，在这里调用SAVE\_ALL宏，保存临时寄存器和控制状态寄存器，进入trap函数进行处理，将sp栈指针作为参数传递给不同的中断处理函数。
4. 根据tf->cause的值，trap函数把中断处理分给interrupt\_handler()和exception\_handler()，区分是异常还是中断分别进行处理。
5. 恢复寄存器，RESTORE\_ALL，并且恢复sp栈指针，通过sret指令把sepc的值赋给pc，便于处理异常之后的其它指令。

mov a0,sp：将sp栈指针的值放到a0中，a0是参数寄存器，这样相当于把栈指针 的值作为函数参数传递给中断处理程序，从而实现对中断的处理。

SAVE\_ALL中寄存器保存位置的确定：我们先执行csrw sscratch ,sp;将sp栈指针寄存器保存到sscratch寄存器里。然后addi sp,sp,-36\*REGBYTES;栈地址向低空间增长36个寄存器的空间，并且额外保存了4个CSR寄存器。我们通过寄存器编号、REGBYTES、sp即可获得寄存器保存的位置。

\_\_alltraps中不需要保存所有寄存器。一方面，对于某些中断，其处理程序可能只会用到几个寄存器，我们只需要将这些寄存器保存下来即可，若所有中断都保存所有寄存器，那会使得系统的性能和效率大大降低；另一方面，有很多寄存器的值实际上是不会受中断影响而改变的，对于这部分寄存器我们完全可以不用保存，减少程序的空间和时间开销。

**扩增练习 Challenge2：理解上下文切换机制**

在trapentry.S 中汇编代码csrw sscratch, sp；csrrw s0, sscratch, x0 实现了什么操作，目的是什么？saveall 里面保存了stval、scause 这些csr，而在restore all 里面却不还原它们？那这样store的意义何在呢？

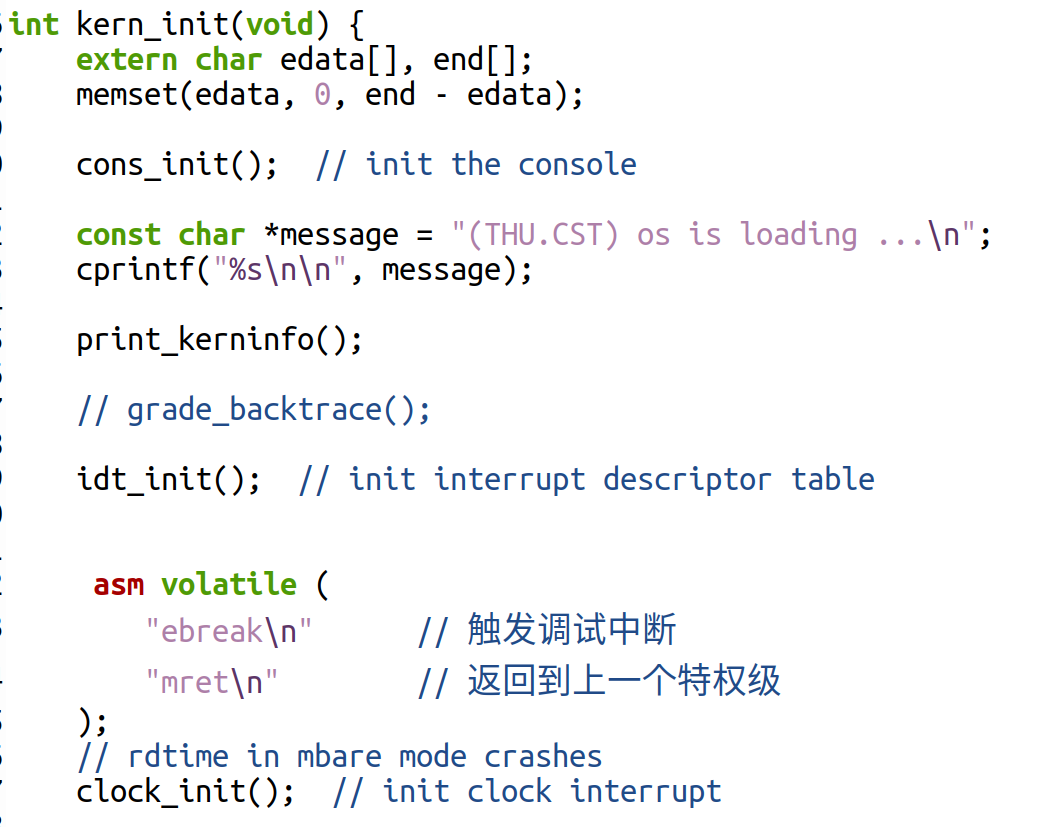
csrw sscratch,sp; 这条指令将栈指针sp的值写入控制状态寄存器sscratch。sscratch寄存器用于保存临时状态信息，将sp指针的值写入该寄存器，可以确保上下文切换时不丢失当前的栈指针，处理完异常之后可以返回当前位置执行剩余的代码。

csrrw s0,sscratch, x0;先把sscratch寄存器里存储的上文栈指针的值写入s0寄存器，用于后续存入内存实现上文的保存；然后把零寄存器x0复制给sscratch，实现sscratch的清零，以便后续标识中断前程序处于S态。中断前处于S态，那么sscratch寄存器值为0；若处于U态，那么sscratch存储内核栈地址。

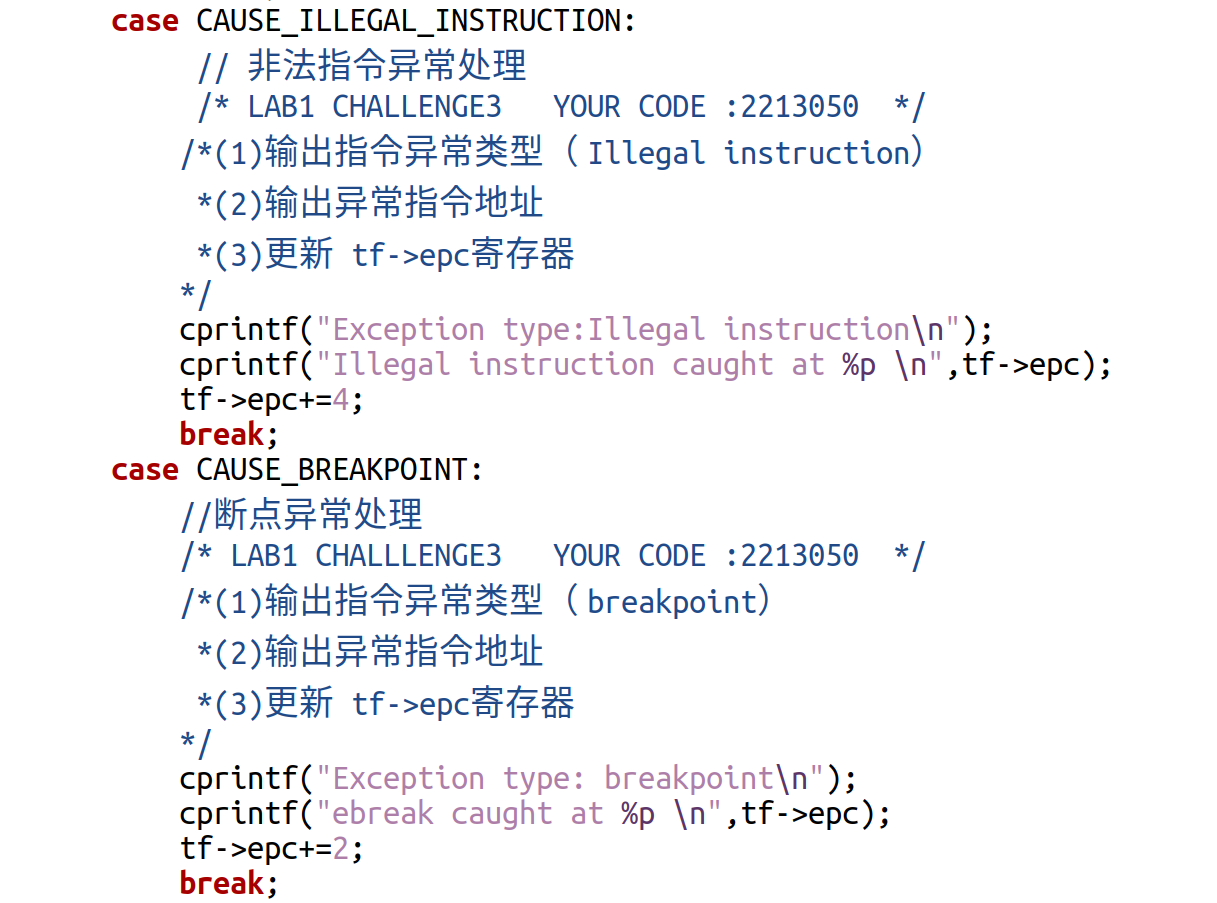
stval、scause仅仅是用来查询中断产生的具体的类别以及相关的原因，用于中断处理程序的执行。而在中断处理程序结束后，我们便不再需要这些信息，并且之后也可以安全地覆盖这两个寄存器，所以就不再进行恢复。

**扩展练习Challenge3：完善异常中断**

编程完善在触发一条非法指令异常mret和ebreak，在kern/trap/trap.c的异常处理函数中捕获，并对其进行处理，简单输出异常类型和异常指令触发地址，即“Illegal instruction caught at 0x(地址)”，“ebreak caught at 0x（地址）” 与“Exception type:Illegal instruction”，“Exception type: breakpoint”。

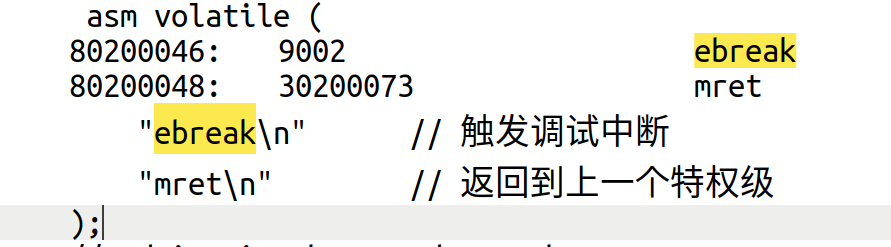


我们在kern\_init函数里加入汇编代码，触发ebreak和mret中断，接着调用对应的中断处理函数。



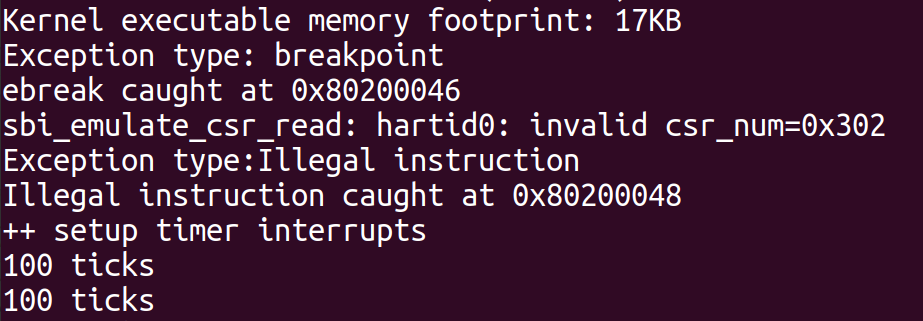
在代码中，我们主要更改是：打印输出异常类型和异常指令地址。最关键的操作是更新tf->epc寄存器。我们主要来探讨为什么一个+4，一个+2。

在 kern\_init()函数内部添加汇编代码，分别调用ebreak和mert指令，在 trap.c的 exception\_handler()内部分别添加完打印语句后，打开kernel.asm，搜索ebreak，发现如下代码：



* 80200046:9002：这是ebreak指令的机器码表示，它触发一个调试异常。
* 80200048:30200073：这是mret指令的机器码表示，它从异常或中断处理程序返回到上一个特权级。
* mret指令占用了4个字节，而ebreak指令仅占用2个字节。经过查阅资料发现，在设置环境断点时调用的ebreak实际上是16位的指令c.ebreak，而非32位的break指令。因此，在打印语句之后，应该分别将epc更新为+4和+2的位置。

做完以上更改后，我们make qemu得到的输出如下：



**知识点总结**

1. 异常与中断

* **异常**：指在程序执行过程中由于内部错误而触发的事件，如执行了非法指令或缺页错误。
* **中断**：由外围设备发出的信号导致CPU暂停当前任务，例如时钟中断。

1. 异常与中断的处理流程

* **异常处理**：
* 系统进入S态并触发异常。
* 系统将当前上下文保存到栈中。
* 转入异常处理代码。
* 根据异常类型进行处理，并更新结构体中的epc值。
* 处理完毕后返回到异常发生位置的下一条指令继续执行。
* **中断处理**：
* 时钟中断或其他中断信号触发。
* 系统将当前上下文保存到栈中。
* 转入中断处理代码。
* 根据中断类型进行处理，并更新结构体中的epc值。
* 处理完毕后返回到中断发生位置的下一条指令继续执行。

1. 系统调用（System Call）

* **系统调用**：是用户程序请求操作系统服务的机制，如文件操作、进程控制等。
* **触发方式**：通过中断触发系统调用，使程序切换到内核态以执行请求的服务。

1. 上下文切换（Context Switching）

* **上下文切换**：操作系统内核中用于保存和恢复进程或线程执行状态的过程。
* **应用场景**：通常在中断处理中需要进行上下文切换，确保处理程序完成后能返回到被中断的程序继续执行。

1. 中断向量表（Interrupt Vector Table）

* **中断向量表**：一个存储了所有可能中断类型及其处理程序入口地址的数据结构。
* **作用**：当中断发生时，CPU利用此表查找并执行对应的中断处理程序。