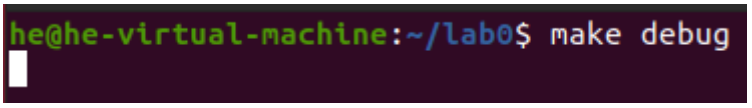
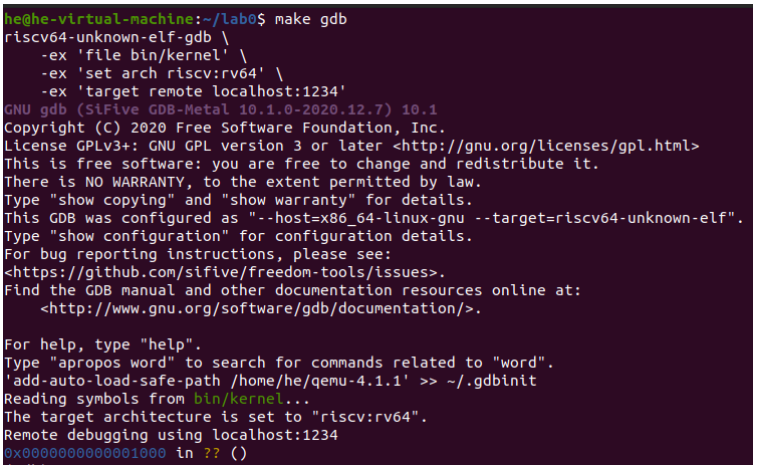
操作系统第一次实验报告

lab0.5 最小可执行内核

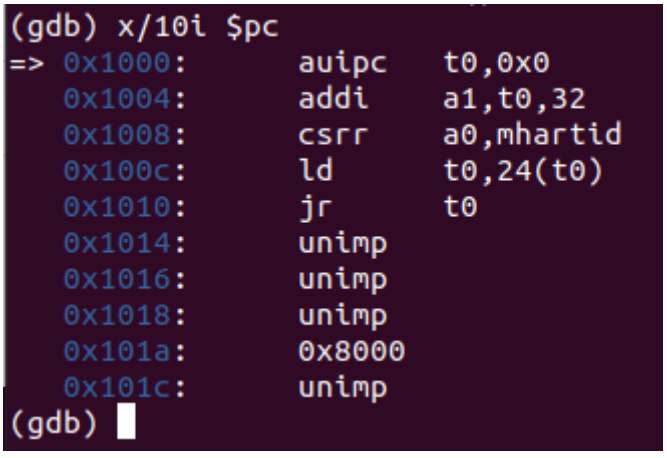
练习1：使用GDB验证启动流程

启动调试，在Makefile文件目录下，创建两个终端，分别输入make debug启动qemu，make gdb启动gdb，如图所示





输入命令x/10i $pc : 显示即将执行的10条汇编指令。



该段汇编代码展示了RISC-V架构下的一段启动代码。这段代码通常位于ROM或初始化代码中，用于执行一些基本的初始化任务。

0x1000: auipc t0,0x0，将PC（程序计数器）的高20位与立即数0x0组合，并存储结果到寄存器`t0`中。0x1000+0x0=0x1000，所以此时t0的值为0x1000。

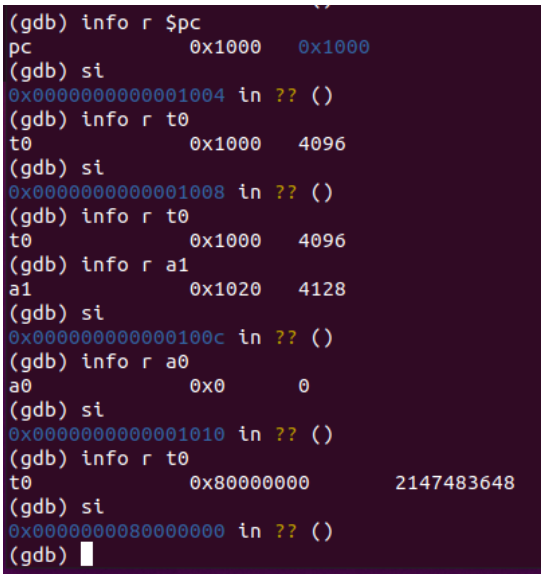
0x1004: addi a1,t0,32，将`t0`寄存器的值与立即数32相加，并存储结果到寄存器`a1`中。0x1000+0x0020=0x1020，所以此时a1的值为0x1020。

0x1008: csrr a0,mhartid，从CSR mhartid中读取当前Hart（核心）的ID，并存储到寄存器`a0`中。

0x100c: ld t0,24(t0)，从`t0`寄存器指向的地址加上偏移量24处加载一个双字（64位）的数据，并存储到`t0`寄存器中。0x1000+0x1018（偏移量24）=0x2018

0x1010: jr t0，跳转到`t0`寄存器中存储的地址。

输入命令info r显示寄存器的值，输入命令si单步执行一条汇编指令，进行代码调试，结果如图所示



从调试结果可以看出，初始程序计数器（PC）的值为0x1000。

执行si单步调试后（auipc t0,0x0），PC值更新为0x1004，`t0`寄存器的值为0x1000（即4096）。

继续执行si单步调试后（addi a1,t0,32），PC值更新为0x1008，`t0`寄存器的值为0x1000（即4096），`a1`寄存器的值为0x1020（即4128）。

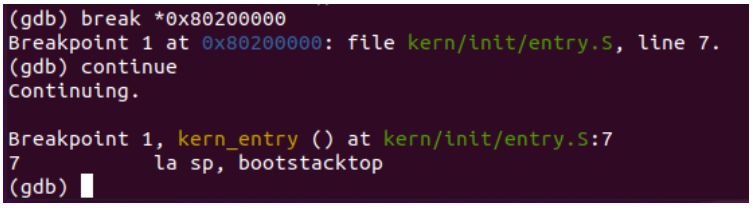
继续执行si单步调试后（csrr a0,mhartid），PC值更新为0x100c，`a0`寄存器的值为0x0（即0）。

继续执行si单步调试后（ld t0,24(t0)），PC值更新为0x1010，`t0`寄存器的值为0x80000000（即2147483648）。

最后执行si单步调试（jr t0），PC值更新为0x80000000。

综上可知，使用gdb调试QEMU模拟的RISC-V计算机加电开始运行时的地址为0x1000，这五条指令是地址跳转到0x80000000，即作为 bootloader 的 OpenSBI.bin 被加载到物理内存以物理地址 0x80000000 开头的区域。

输入命令break *0x80200000，在 0x80200000 处设置断点，输入命令continue，执行直到碰到断点。



有结果可以看出在0x80200000执行的第一条指令，也是entry.S文件入口点的第一条指令，为la sp，bootstacktop，即内核镜像 os.bin 被加载到以物理地址 0x80200000 开头的区域上，此时make debug也显示了OpenSBI的输出，如图所示

实验知识点

本实验主要学习最小可执行内核和启动流程，内核主要在Qemu模拟器上运行，它可以模拟一台64位RISC-V计算机，我们需要了解从机器启动到操作系统运行的过程。

我们使用QEMU自带的bootloader——OpenSBI固件，把操作系统加载到内存。

链接脚本tools/kernel.ld中把程序的入口点定义为kern_entry, 所以在kern/init/entry.S编写一段汇编代码, 作为整个内核的入口点，作用就是分配好内核栈，然后跳转到kern_init。在kern/init/init.c编写函数kern_init, 作为真正的内核入口点。由于调用C语言标准库的函数所需glibc提供运行环境，但glibc并没有被移植到ucore里，所以需要把QEMU里的OpenSBI固件提供的输入字符和输出字符的接口一层层封装起来，提供stdio.h里的格式化输出函数cprintf()来使用。

OpenSBI的接口一层层封装到格式化输入输出函数

分析sbi_call()一步步封装实现cputs()进而实现cprintf()函数的过程：

libs/sbi.c 可以通过sbi_console_putchar()来输出一个字符。

kern/driver/console.c 利用cons_putc(int c)做简单的封装，并将int强制转换成unsigned char。

kern/libs/stdio.c 实现了ucore版本的puts函数cputs()，在libs/printfmt.c实现了一些复杂的格式化输入输出函数。最后得到的cprintf()函数仍在kern/libs/stdio.c定义，功能和C标准库的printf()基本相同。使用qemu语句进行qemu启动加载内核，ucore就跑起来了，在kern/init/init.c中的kern_init函数完成格式化输出cprintf()后，它输出一行(THU.CST) os is loading, 然后进入死循环。

makefile自动化编译链接工具

Makefile是一种用于自动化构建过程的脚本文件，常用于编译源代码文件并链接它们以创建可执行程序或库。Makefile 是 make 工具的核心配置文件，该工具读取 Makefile 文件中的指令来构建最终的应用程序或库。

lab1 中断处理机制

练习1：理解内核启动中的程序入口操作

la sp, bootstacktop指令设置了一个初始的栈指针，意味着内核将使用这个栈来进行函数调用和其他需要栈空间的操作，如果没有这一步就没有正确的栈指针，内核就无法正确地执行函数调用或处理中断等需要栈空间的任务。

tail kern_init用于跳转到kern_init函数，目的是开始执行内核初始化过程。kern_init函数通常包含了一系列初始化步骤，它为内核的进一步运行准备了必要的环境，一旦sp被正确设置，内核就可以通过调用kern_init来完成更多的初始化工作。

综上，la sp, bootstacktop设置了栈指针到预定的栈顶地址，为后续的函数调用提供了工作环境。tail kern_init 则开始了内核的初始化过程，为内核的完全启动做准备。

练习2：完善中断处理

函数完善

根据题目完善函数内容如下：

```
case IRQ_S_TIMER:
          // "All bits besides SSIP and USIP in the sip register are
          // read-only." -- privileged spec1.9.1, 4.1.4, p59
          // In fact, Call sbi_set_timer will clear STIP, or you can clear it
          // directly.
          // cprintf("Supervisor timer interrupt\n");
           /* LAB1 EXERCISE2   2211532 :  */
          //(1)设置下次时钟中断- clock_set_next_event()
          clock_set_next_event();
          //(2)计数器（ticks）加一
          ++ticks;
          //(3)当计数器加到100的时候，我们会输出一个`100ticks`表示我们触发了100次时钟中断，同时打印次数（num）加一
          if (ticks == TICK_NUM) {
              print_ticks();
              ticks = 0;
              ++num;
          }
          //(4)判断打印次数，当打印次数为10时，调用<sbi.h>中的关机函数关机
          if(num == 10){
              sbi_shutdown();
          }
          break;
```

实现过程：

- 本次时钟中断发生时，先使用`clock_set_next_event();`函数设置下一次时钟中断。
- ticks计数器加一。
- `TICK_NUM`在宏定义中被定义为100，将ticks与其做相等判断。当计数器的值为100，使用`print_ticks()`打印“100 ticks”，将ticks重置为0。最后打印次数num加一。
- 条件判断num是否等于10，如果相等，则调用<sbi.h>中的关机函数`sbi_shutdown()`关机。

在终端运行 `make qemu` ，运行结果如下：

约每1s打印一次，打印十次后关机。

```
Special kernel symbols:
  entry  0x000000008020000a (virtual)
  etext  0x00000000802009a8 (virtual)
  edata  0x0000000080204010 (virtual)
  end    0x0000000080204028 (virtual)
Kernel executable memory footprint: 17KB
++ setup timer interrupts
100 ticks
100 ticks
100 ticks
100 ticks
100 ticks
100 ticks
100 ticks
100 ticks
100 ticks
100 ticks
yayaa@yayaa-virtual-machine:~/riscv64-ucore-labcodes/lab1$ ^C
```

定时器中断中断处理的流程

1. **初始化时钟中断：** `init.c` 文件中调用 `clock_init()` 函数初始化时钟中断。在 `sie` 寄存器中设置 `MIP_STIP` 位，使能时钟中断。 `clock_set_next_event()` 函数设置时钟中断事件。之后初始化计数器 `ticks` 为0。初始化完成后，打印字符串“`++ setup timer interrupts\n`”。
2. **`clock_set_next_event()` 函数**：`sbi_set_timer(get_cycles() + timebase);` timer的数值变为当前时间 + timebase 后，触发一次时钟中断。
3. **定时器中断处理：**先使用 `clock_set_next_event();` 函数设置下一次时钟中断。更新ticks计数器加一。 `TICK_NUM` 在宏定义中被定义为100，将ticks与其做相等判断。当计数器的值为100，使用 `print_ticks()` 打印 `100 ticks` ，并将ticks重置为0， `num++` 。当 `num` 为10时，调用<sbi.h>中的关机函数 `sbi_shutdown()` 关机。

扩展练习1：描述与理解中断流程

处理中断异常的流程

1. **中断处理初始化：** `init.c` 文件中调用 `idt_init()` 函数初始化中断。将 `sscratch` 寄存器赋为0，表示现在是内核态； `trapentry.S` 中的中断入口点处定义了一个全局标签 `__alltraps` ，这里还将 `stvec` 寄存器置为 `&_alltraps` ，即中断入口点的地址。
2. **初始化时钟中断：** `init.c` 文件中调用 `clock_init()` 函数初始化时钟中断。在 `sie` 寄存器中设置 `MIP_STIP` 位，使能时钟中断。 `clock_set_next_event()` 函数设置时钟中断事件。之后初始化计数器 `ticks` 为0。初始化完成后，打印字符串“`++ setup timer interrupts\n`”。
3. **初始化使能中断：** `intr_enable()` 函数用于使能中断，它将 `sstatus` 寄存器的值置为 `SSTATUS_SIE` 位，使能中断。
4. **跳转到入口：**异常产生后，CPU跳转到 `stvec` 寄存器中的中断入口地址。

5. **保存上下文**：通过汇编宏`SAVE_ALL`将一个保存了所有寄存器`trapFrame`的结构体放入栈中，实现保存上下文。
6. **进入中断处理**：保存上下文后，将`trapFrame`结构体送到`trap.c`文件中的中断处理函数`trap`中。
7. **分类**：函数`trap`会根据`trapframe`结构体中的`cause`字段来区分中断和异常。中断调用`interrupt_handler()`继续处理；其他情况则调用`exception_handler()`来处理。
8. **执行对应异常处理程序**：`interrupt_handler()`与`exception_handler()`中`switch`的多个`case`会对中断再次分类。并调用对应的处理程序。
9. **中断结束**：对应中断处理程序完成后，会回到`trapentry.S`文件中的`__trapret`标签位置，使用汇编宏`RESTORE_ALL`恢复上下文，继续执行中断前的任务。

mov a0，sp的目

中断入口点处，在保存上下文之后进行了`mov a0，sp`，这一步指令将当前的栈顶指针传递给`trap`函数。

这是因为之前的上下文保存操作中将寄存器状态都打包成`trapframe`结构体保存在栈中，将sp存入寄存器a0后，按照RISCV calling convention，a0寄存器会传递参数给接下来调用的函数trap。也就相当于将保存了寄存器状态的`trapframe`结构体指针，作为参数传递给`trap`函数，使异常处理函数可以访问到栈中寄存器的状态。

SAVE_ALL中寄存器保存在栈中的位置是什么确定的

观察汇编宏SAVE_ALL。可以发现寄存器保存的位置是通过在栈顶指针sp的基础上偏移来决定的。

宏中先保存了原先的栈顶指针到`sscratch`，然后让栈顶指针向低地址空间延伸 36 个寄存器的空间（一个寄存器占`REGBYTES`个字节），可以放下一个trapFrame结构体。

然后依次保存32个通用寄存器，如`x0`存入`0*REGBYTES(sp)`，`x1`存入`1*REGBYTES(sp)`。

为保存4个和中断有关的CSR，需要csrr把CSR读取到通用寄存器，再从通用寄存器STORE到内存，如sscratch先读取到s0中，再将s0存入位置`2*REGBYTES(sp)`

表达式`REGBYTES(sp)`会根据一个寄存器的字节大小（`REGBYTES`），计算从栈指针`sp`开始的内存偏移量。

对于任何中断，__alltraps 中都需要保存所有寄存器吗

在本次实验的代码中，并没有看到根据中断类型来决定保存寄存器种类的代码。都是直接调用汇编宏SAVE_ALL来保存所有寄存器。出于稳健型的考虑，这种方法是一定可行且不会出错的，可以完整的保存和恢复上下文。

但在一些特殊情况下，也可以考虑不保存所有寄存器。比如对于一些简单的中断处理程序中，有些寄存器没有被使用，就可以不必保存。

扩展练习2：理解上下文机制

`trapentry.S`中汇编代码

`csrw sscratch, sp`:这条指令主要出现在文件的开头位置，其中`csrw`指令是用于向控制状态寄存器（Control and Status Register, `CSR`）写入数据的指令，整条指令执行的操作是将栈指针`sp`的内容写入`sscratch`中, `sscratch`是一个在RISC-V架构中定义的特殊系统寄存器，其用途是提供一个临时存储位置，以便在需要时保存和恢复重要的状态信息，如栈指针或其他上下文信息。通过此操作，在擦破做

系统中，当任务被抢占或终端时，当前任务的上下文可以被保存，之后可以恢复并继续执行。此外，该操作只需要从一个已知的位置读取并写回指针`sp`中，就能恢复原来的栈环境。

`csrrw s0, sscratch, x0`：这句指令在一系列将当前CPU的整数寄存器（x寄存器）的内容保存到栈上之后，这些寄存器是RISC-V架构中用于存储数据和地址的主要寄存器。其中`csrrw`用于原子地读取-修改-写入控制状态寄存器，整条指令的操作是从`sscratch`寄存器读取值到`s0`寄存器，同时，将`x0`（即0）写入到`sscratch`寄存器中。目的在恢复寄存器状态之前检查或清除`sscratch`寄存器中的值，通过将其设置为0，可以确保在恢复过程中不会受到之前保存的栈指针或其他信息的干扰。

store的意义：

`scause`这个`CSR`包含了触发当前异常或中断的原因，保存了这个控制状态寄存器可以让程序在返回的时候知道触发异常的具体原因。不需要恢复的原因是，这些`CSR`中的信息主要用于调试或记录目的，一旦异常被处理，这些寄存器的使命就结束了。因此，保存他们的意义在于，当遇到了异常发生时，可以了解发生的原因和上下文的重要信息，从而决定采取何种错误恢复措施。

扩展练习3：完善异常中断

当异常中断出现时，首先输出异常的类型，并通过改变`epc`的值，跳过异常指令，具体内容见代码。

```
        case CAUSE_ILLEGAL_INSTRUCTION:
             // 非法指令异常处理
             /* LAB1 CHALLENGE3   2211849 :  */
            /*(1)输出指令异常类型（ Illegal instruction）
             *(2)输出异常指令地址
             *(3)更新 tf->epc寄存器
            */
            cprintf("exception type:Illegal instruction\n");
            cprintf("Illegal instruction address: 0x%016llx\n", tf->epc);
            tf->epc += 4;//跳过导致异常的指令
            break;
         case CAUSE_BREAKPOINT:
            //断点异常处理
            /* LAB1 CHALLLENGE3   2211849 :  */
            /*(1)输出指令异常类型（ breakpoint）
             *(2)输出异常指令地址
             *(3)更新 tf->epc寄存器
            */
            cprintf("exception type:breakpoint\n");
            cprintf("Illegal instruction address: 0x%016llx\n", tf->epc);
            tf->epc += 2;//跳过导致异常的指令
            break;
```

为了测试代码的功能，我在`init.c`文件中增加了两句指令：

```
asm("mret")
asm("mret")
```

分别测试非法指令异常和断点异常，接着运行`make qemu`得到输出：

```
exception type:Illegal instructionI
llegal instruction address: 0x0000000080200046
exception type:breakpointI
llegal instruction address: 0x0000000080200052
```

同时我们注意到，在系统中初始化中断和异常处理时，通常会通过修改 `stvec` 寄存器的值来设置异常向量表的基地址，这个基地址是异常向量表中第一个条目的地址，而异常向量表中的每个条目都指向一个特定的异常处理函数。因此，只有在`stvec`初始化之后，才能正常执行异常中断的问题，而只有在 `idt_init();` 函数调用中设置 `stvec` 的值，实际上是在为系统配置一个异常处理的"入口点"。因此我们终端测试的代码应该放在该函数之后。

实验知识点

关于ＲＩＳＣ－Ｖ

与中断机制有关的控制状态寄存器Control and Status Registers

`sscratch` :一个在RISC-V架构中定义的特殊系统寄存器，是一个读写寄存器,其用途是提供一个临时存储位置，以便在需要时保存和恢复重要的状态信息，如栈指针或其他上下文信息。当用户程序执行系统调用时，会触发`ecall`指令，导致hart（RISC-V中的处理器核心）从用户模式陷入Supervisor模式,同时,`sscratch`寄存器的访问和修改通常受到Supervisor模式的权限控制,此时,`sscratch`寄存器通常用于保存一个指向当前用户上下文（如`trapframe`）的指针。这样，在Supervisor模式下处理完trap后，可以方便地恢复用户模式的上下文。

`sstatus`寄存器:决定`cpu`能否被打断

`stvec`:中断向量表基址,通过映射处理不同种类的中断处理程序.

`epc`:记录触发中断的那条指令的地址,通过该寄存器可以设置跳过异常指令直接执行下一条.

`cause`:记录中断发生的原因，还会记录该中断是不是一个外部中断.

`tval`:记录一些中断处理所需要的辅助信息，比如指令获取(instruction fetch)、访存、缺页异常，它会把发生问题的目标地址或者出错的指令记录下来，这样我们在中断处理程序中就知道处理目标了。

利用好上述寄存器,通过寄存器直接的存储和传递关系,我们可以正确的实现中断处理程序的上下文切换以及确定所需的处理机制.

关于中断的特权指令

`ebreak`:这条指令会触发一个断点中断从而进入中断处理流程,在调试模式下，这可以暂停程序的执行，允许调试器检查程序的状态

`mret`用于 从机器模式（Machine Mode）的异常处理程序中返回到 S 态或 U 态，实际作用为`pc←mepc`，回顾`epc`定义，返回到通过中断进入 M 态之前的地址,当机器模式异常发生时（如中断或异常），处理器会跳转到异常处理程序中。在异常处理程序执行完毕后，使用 `mret` 指令可以返回到发生异常时的指令的下一条指令继续执行。

在本次实验中,我们使用这两种指令检测我们的断点和异常中断处理是否能够正常运行,不过在使用中需要注意`stvec`寄存器是否已经赋值,以实现正确的中断程序.

关于`CSR`的指令

`csrrw`:用于原子地读取-修改-写入控制状态寄存器

`csrw`:是用于向控制状态寄存器（Control and Status Register, `CSR`）写入数据的指令

关于上下文切换以及栈的保存

在`trap.c`文件中,我们定义了`.macro SAVE_ALL`宏,主要涉及到当前状态的保存,除了上述已经提到过的几个特殊寄存器之外,还将当前CPU的整数寄存器（x寄存器）的内容保存到栈上,这些寄存器是RISC-V架构中用于存储数据和地址的主要寄存器.主要的操作过程有将指定的x寄存器（如`x0`, `x1`, ..., `x31`）的内容读取出来;计算栈指针（`sp`）加上一个偏移量;将寄存器的内容存储到计算出的栈地址上.在发生中断、异常或任务切换时，能够保存当前执行环境的上下文，以便之后能够恢复并继续执行。在RISC-V架构中，x寄存器包含了程序执行过程中所需的大部分数据和地址，因此保存它们是上下文保存过程中非常重要的一步。

此外,该文件的函数还实现了打印寄存器状态,时钟中断管理,中断向量初始化,陷入内核状态判断等功能,完成了一个基本的异常和中断处理框架，使其能够响应并处理各种硬件和软件事件。