

# Lab 0.5最小执行内核

小组成员：张高，张铭，黄贝杰

## 练习1：使用GDB验证启动流程

### 一.环境准备：

首先，这里实验用的是Ubuntu系统，确保系统安装有QEMU和GDB工具以及RISC-V相关的交叉编译工具链。

### 二.启动流程：

RISC-V架构下，系统上电后会进行CPU自检，然后跳转到bootloader处，bootloader在这里是OpenSBI,bootloader会初始化硬件并设置好内核运行的环境配置，接下来从硬件加载内核（kernel）到内存，最后跳转到内核（kernel）入口地址。

在这里，bootloader就是OpenSBI,它被加载到0x80000000，OpenSBI会探测硬件并初始化内核的环境变量，之后OpenSBI会从硬盘加载Kernel到0x80200000,开始执行内核代码

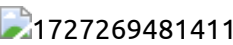
### 三.实验验证

在lab0实验路径下

```
make debug

make gdb
```

启动gdb！



#### 1. 初始化系统

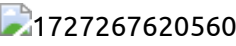
输入指令

```
x/10i $pc
```

查看接下来10条汇编指令

```
l
```

得到执行的源码：



**\*从上图中不难看出RISC-V的复位地址是0x1000,每次启动,系统都会从复位地址开始运行,以进行相关准备工作\***

上述这些是系统刚刚上电后执行的指令,这段代码的作用主要是在进行内核栈的分配

```
la sp, bootstacktop
```

, 然后跳转到内核初始化函数(tail kern\_init)

## 2.bootloader开始运行


我们连续执行

```
si
```

运行到

```
0x1010 jr t0
```

跳转到0x80000000。从这里执行bootloader开始运行的代码,跳转后程序在分配内核栈

1727270121394

(.global bootstack 定义内核栈)

(.global bootstacktop 之后内核栈将要从高地址向低地址增加,初始时内核栈为空)

OpenSBI.bin 作为 Bootloader 固件被加载到 0x80000000。OpenSBI.bin被成功加载后,主要负责初始化硬件和设备,包括定时器、中断和串口,配置内存布局和清理内存等,加载操作系统内核到0x80200000

## 3.内核入口

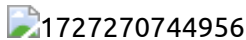
### 1.运行

```
break *0x80200000
```

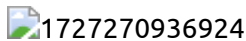
指令设置个断点,并运行

```
c
```

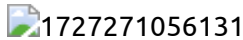
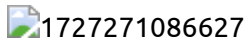
指令,程序会一直运行到0x80200000,此时执行到了la sp, bootstacktop这一步。

1727270744956

可以发现OpenSBI已经启动，但还处于 entry.S 文件当中而并非 init.c 文件当中

1727270936924

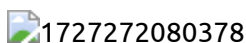
2.继续执行，我们可以发现程序真正进入 init.c 文件当中的 kern\_init() 函数，也就是进入到了程序内核，开始准备打印字符 ' (THU.CST) os is loading ... \n '

17272710561311727271086627

2.执行n指令，即将运行当前源程序中的cprint函数。

cprint.png

3.执行'n'指令，cprint函数运行完毕，可以发现在make debug终端中成功输出了cprint要输出的内容。

1727272080378

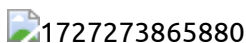
### 三.实验分析以及知识点总结

#### 问题

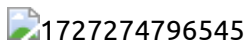
**说明RISC-V硬件加电后的几条指令在哪里，完成了哪些功能？**

RISC-V的启动顺序分为三个阶段

第一阶段：将必要的文件载入到 Qemu 物理内存之后，pc会被初始化为 复位地址0x1000，然后它将执行几条指令并跳转到物理地址 0x80000000 对应的指令处并进入第二阶段

1727273865880

第二阶段：bootloader opensbi.bin 放在以物理地址 0x80000000 开头的物理内存中，在这一阶段，bootloader 负责进行设置硬件环境、加载内核等工作，并加载到下一阶段软件的入口，在 Qemu 上即可实现将计算机控制权移交给我们的内核镜像 os.bin。

1727274796545

第三阶段：需要先运行到真正的内核入口，一旦 CPU 开始执行内核的第一条指令，证明计算机的控制权已经被移交给我们的操作系统内核，也就完成了加载操作系统的目标

#### 知识点总结

##### 1. makefile(初步)

target:目标文件 (object or executable)

prerequisites:生成target所需文件或目标

command:make需要执行的命令

```
target ....: prerequisites ...
```

```
command
```

```
...
```

```
...
```

```
.PHONY: qemu
```

```
qemu:${UCOREIMG} ${SWAPIMG} ${SFSIMG}
```

```
\# $(V)$(QEMU) -kernel ${UCOREIMG} -nographic
```

```
$(V)$(QEMU) \
```

```
-machine virt \
```

```
-nographic \
```

```
-bios default \
```

```
-device loader,file=${UCOREIMG},addr=0x80200000
```

## 2. 内存管理

比如：bootloader分配内存栈（内核栈将从高地址向低地址增加，初始时内核栈为空），加载物理内存等，对应OS原理中内存分配的知识点

## 3.地址相关 和 地址无关

**地址相关**：某些代码或数据在使用时，必须在特定的地址或特定的地址范围内执行或存储。如果位置发生改变，代码或数据可能会无法正常工作。 **地址无关**：代码或数据在使用时，不依赖于其所在的具体内存地址，可以在任意内存位置运行。地址无关的代码或者数据可以更灵活地在不同地址执行和存储。

## 4.ELF 和 BIN 文件

**ELF 文件**：包含元数据（Metadata），任务是描述文件如何加载到内存，包括各个段的位置和长度。包含入口点（Entry Point）信息，指明执行的第一条指令位置。 **BIN 文件**：纯粹的二进制文件，没有元数据。不包含位置信息和入口点信息。需要额外的机制指定加载地址。

**加载和跳转的机制**：操作系统BIN文件需要是地址相关的，这意味着它的代码中包含了具体的内存地址，这些地址在编译时已经确定。为了正确地对接 OpenSBI 和 QEMU，需要确保内核镜像预先加载到 **0x80200000**。而 ELF 文件格式通常用来解决这个问题，因为它包含了详细的加载地址和入口点信息。

# Lab 1: 中断处理机制

## 练习1：理解内核启动中的程序入口操作

问题：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？

答：

kern/init/entry.S 代码如下：

```
#include <mmu.h>
#include <memlayout.h>

.section .text, "ax", %progbits
.globl kern_entry

kern_entry:
    la sp, bootstacktop

    tail kern_init

.section .data
    # .align 2^12
    .align PGSIZE
    .global bootstack
bootstack:
    .space KSTACKSIZE
    .global bootstacktop
bootstacktop:
```

`la sp, bootstacktop` 是一条地址加载指令用于将 `bootstacktop` 符号的地址加载到栈顶指针 `sp` 当中。"`la`" 是一个汇编指令，用于加载一个地址到寄存器。"`bootstacktop`" 是一个地址标号，指向操作系统内核为栈分配的顶部位置。目的是将 `bootstacktop` 所表示的内存地址当作内核堆栈的顶部，以便在内核执行时正确使用栈内存。

`tail kern_init` 指令用于跳转到 "`kernel_init`" 函数的入口点，目的是为了将控制转移给内核初始化函数，并开始操作系统内核的正常运行。"`tail`" 是一个汇编指令，它会将当前函数的返回地址保存在链接寄存器中，并跳转到目标函数的入口点。通过跳转到 "`kernel_init`" 函数，操作系统开始执行初始化过程。

这条指令是一条伪调用指令，伪调用可以使得跳转更加高效，因为它避免了额外的函数调用/返回开销。

## 练习2：完善中断处理（需要编程）

### 相关函数实现

在 `trap.c` 中补充全局变量 `PRINT_NUM`，记录打印了多少次 "100 ticks"：

```
static int PRINT_NUM=0;
```

随后，填写 `kern/trap/trap.c` 函数中处理时钟中断的部分：

```

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 2213219 : */
    clock_set_next_event();
    if(++ticks%TICK_NUM == 0){
        print_ticks();
        PRINT_NUM++;
        if(PRINT_NUM == 10){
            sbi_shutdown();
        }
    }
    /*(1)设置下次时钟中断- clock_set_next_event()
    *(2)计数器 ( ticks ) 加一
    *(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
    * (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
    */
    break;

```

## 实现过程

### 1. 引入计数变量：

- 需要两个全局变量 `ticks` 和 `PRINT_NUM`。
- `ticks` 用来计数时钟中断次数。
- `PRINT_NUM` 记录打印了多少次 "100 ticks"。

### 2. 设置定时器：

- 在时钟中断处理函数中，需要调用 `clock_set_next_event` 来设置下一次时钟中断，这样时钟中断可以一直发生。

### 3. 中断处理逻辑：

- 每次触发时钟中断，先增加 `ticks` 计数器。
- 使用 `ticks` 检查是否达到了 100 次中断，如果是则调用 `print_ticks` 打印 "100 ticks"。不要忘记同步增加 `PRINT_NUM` 计数器。
- 检查 `PRINT_NUM` 是否达到了 10，如果达到了则调用 `sbi_shutdown` 进行关机操作。

## 定时器中断处理的流程

`kern/init/entry.S`是OpenSBI启动之后将要跳转到的一段汇编代码。在这里进行内核栈的分配，然后转入C语言编写的内核初始化函数。

`kern/init/init.c`是C语言编写的内核入口点，即C语言编写的内核初始化函数。主要包含`kern_init()`函数，从`kern/init/entry.S`跳转过来完成其他初始化工作。

`init.c`的`kern_init`函数中有如下代码：

```
clock_init(); // init clock interrupt
```

`clock_init`函数的作用为初始化一个时钟中断系统，具体在`kern/driver/clock.c`中有该函数的定义：

```
void clock_init(void) {
    // enable timer interrupt in sie
    set_csr(sie, MIP_STIP);
    // divided by 500 when using Spike(2MHz)
    // divided by 100 when using QEMU(10MHz)
    // timebase = sbi_timebase() / 500;
    clock_set_next_event();

    // initialize time counter 'ticks' to zero
    ticks = 0;

    cprintf("++ setup timer interrupts\n");
}
```

这个函数负责配置和初始化系统的定时器中断，使系统能够基于定时器生成周期性中断，并按照指定的时间间隔执行某些任务。

具体来说，`set_csr(sie, MIP_STIP)`使用了一个内嵌汇编函数 `set_csr`，将 `MIP_STIP` 标志位写入名为 `sie` (Supervisor Interrupt Enable Register) 的CSR (Control and Status Register) 寄存器以使能Supervisor级别的定时器中断。`clock_set_next_event()`调用用于设置定时器的下一个事件时间。

我们来详细分析`clock_set_next_event()`的作用，首先找到该函数的定义：

```
void clock_set_next_event(void) { sbi_set_timer(get_cycles() + timebase); }
```

`get_cycles()`函数被调用，获取当前的时间戳计数器值。`timebase`通常表示定时器中断的间隔时间。`sbi_set_timer()`是RISC-V中的一个SBI调用，用来设置下一次硬件定时器触发的时间点。

总的来说，`clock_set_next_event()`通过`get_cycles()`获取当前时间，然后加上一个预设的`timebase`，并通过`sbi_set_timer`将这个时间点设置为硬件定时器的下一个中断时间。

## 扩展练习Challege1:描述和理解中断流程

问题：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。

`trapentry.S`相关部分代码如下：

```

#include <riscv.h>

    .globl __alltraps

.align(2)
__alltraps:
    SAVE_ALL

    move  a0, sp
    jal trap
    # sp should be the same as before "jal trap"

    .globl __trapret

__trapret:
    RESTORE_ALL
    # return from supervisor call
    sret

```

```

# return from supervisor call

```

sret

问题1：描述ucore中处理中断异常的流程（从异常的产生开始）

答：

当异常或中断产生时，CPU跳转至中断向量表基址（stvec）中找到相应的中断处理程序，在中断处理之前，先保存上下文（context），即保存所有相关数据、指令的寄存器到栈顶（把一个trapFrame结构体放到了栈顶），然后处理中断，处理完成后，恢复context。

在trap.c中的中断处理函数中将异常处理的工作分发给了interrupt\_handler()，exception\_handler()两个函数，这两个函数再根据中断或异常的不同类型来处理。在执行完相应的操作后，恢复保存的寄存器的状态，以便返回到正常的程序执行流程。

问题2：mov a0, sp的目的是什么？

答：

mov a0, sp汇编指令将栈指针(sp)的值保存到a0寄存器中。目的是将当前的栈指针保存下来，以便在异常处理过程中可以访问栈中的数据。

问题3：SAVE\_ALL中寄存器保存在栈中的位置是什么确定的？

答：

```

.macro SAVE_ALL

    csrw sscratch, sp

```



```
addi sp, sp, -36 * REGBYTES
```

在 `__alltraps` 的入口处，通过 `SAVE_ALL` 宏来保存所有寄存器的值。这个宏的作用是在当前栈顶分配空间来保存寄存器，并且将新的栈顶地址保存到 `sscratch` 寄存器中。这样，在中断或异常处理程序中，可以通过读取 `sscratch` 寄存器来获取原始的栈顶地址，从而正确地恢复寄存器。

而代码中让栈顶指针向低地址空间延伸 36 个寄存器的空间，可以放下一个 `trapFrame` 结构体。所以位置由 `sp` 和偏移量确定。

问题4：对于任何中断，`__alltraps` 中都需要保存所有寄存器吗？请说明理由。

答：

原因一：有的异常处理会具有复杂性，不仅仅会导致涉及个别寄存器内容被改变，甚至有可能会影响到更多寄存器。哪个寄存器是否需要保存不好判断，代码实现难，容易出错。

原因二：寄存器数量不是很多，而且保存起来对内存空间要求小，为了安全性和一致性，全部保存

## 扩展练习 Challenge2：理解上下文切换机制

`trapentry.S` 相关部分代码如下：

```
#include <riscv.h>
```

```
.macro SAVE_ALL

    csrw sscratch, sp

    addi sp, sp, -36 * REGBYTES
    # save x registers
    STORE x0, 0*REGBYTES(sp)
    STORE x1, 1*REGBYTES(sp)
    STORE x3, 3*REGBYTES(sp)
    STORE x4, 4*REGBYTES(sp)
    STORE x5, 5*REGBYTES(sp)
    STORE x6, 6*REGBYTES(sp)
    STORE x7, 7*REGBYTES(sp)
    STORE x8, 8*REGBYTES(sp)
    STORE x9, 9*REGBYTES(sp)
    STORE x10, 10*REGBYTES(sp)
    STORE x11, 11*REGBYTES(sp)
    STORE x12, 12*REGBYTES(sp)
    STORE x13, 13*REGBYTES(sp)
    STORE x14, 14*REGBYTES(sp)
    STORE x15, 15*REGBYTES(sp)
    STORE x16, 16*REGBYTES(sp)
    STORE x17, 17*REGBYTES(sp)
    STORE x18, 18*REGBYTES(sp)
    STORE x19, 19*REGBYTES(sp)
    STORE x20, 20*REGBYTES(sp)
```

```
STORE x21, 21*REGBYTES(sp)
STORE x22, 22*REGBYTES(sp)
STORE x23, 23*REGBYTES(sp)
STORE x24, 24*REGBYTES(sp)
STORE x25, 25*REGBYTES(sp)
STORE x26, 26*REGBYTES(sp)
STORE x27, 27*REGBYTES(sp)
STORE x28, 28*REGBYTES(sp)
STORE x29, 29*REGBYTES(sp)
STORE x30, 30*REGBYTES(sp)
STORE x31, 31*REGBYTES(sp)

# get sr, epc, badvaddr, cause
# Set sscratch register to 0, so that if a recursive exception
# occurs, the exception vector knows it came from the kernel
csrrw s0, sscratch, x0
csrr s1, sstatus
csrr s2, sepc
csrr s3, sbadaddr
csrr s4, scause

STORE s0, 2*REGBYTES(sp)
STORE s1, 32*REGBYTES(sp)
STORE s2, 33*REGBYTES(sp)
STORE s3, 34*REGBYTES(sp)
STORE s4, 35*REGBYTES(sp)
.endm

.macro RESTORE_ALL

LOAD s1, 32*REGBYTES(sp)
LOAD s2, 33*REGBYTES(sp)

csrw sstatus, s1
csrw sepc, s2

# restore x registers
LOAD x1, 1*REGBYTES(sp)
LOAD x3, 3*REGBYTES(sp)
LOAD x4, 4*REGBYTES(sp)
LOAD x5, 5*REGBYTES(sp)
LOAD x6, 6*REGBYTES(sp)
LOAD x7, 7*REGBYTES(sp)
LOAD x8, 8*REGBYTES(sp)
LOAD x9, 9*REGBYTES(sp)
LOAD x10, 10*REGBYTES(sp)
LOAD x11, 11*REGBYTES(sp)
LOAD x12, 12*REGBYTES(sp)
LOAD x13, 13*REGBYTES(sp)
LOAD x14, 14*REGBYTES(sp)
LOAD x15, 15*REGBYTES(sp)
LOAD x16, 16*REGBYTES(sp)
LOAD x17, 17*REGBYTES(sp)
LOAD x18, 18*REGBYTES(sp)
```

```
LOAD x19, 19*REGBYTES(sp)
LOAD x20, 20*REGBYTES(sp)
LOAD x21, 21*REGBYTES(sp)
LOAD x22, 22*REGBYTES(sp)
LOAD x23, 23*REGBYTES(sp)
LOAD x24, 24*REGBYTES(sp)
LOAD x25, 25*REGBYTES(sp)
LOAD x26, 26*REGBYTES(sp)
LOAD x27, 27*REGBYTES(sp)
LOAD x28, 28*REGBYTES(sp)
LOAD x29, 29*REGBYTES(sp)
LOAD x30, 30*REGBYTES(sp)
LOAD x31, 31*REGBYTES(sp)
# restore sp last
LOAD x2, 2*REGBYTES(sp)
#addi sp, sp, 36 * REGBYTES
.endm
```

问题1：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？

- `csrw sscratch, sp`

这条指令的作用是将当前的栈指针（SP）值保存到 `sscratch` 寄存器中。这样，在中断或异常处理期间，可以通过读取 `sscratch` 来获取原始的栈顶地址。通过将栈指针存储在 `sscratch` 寄存器中，可以在处理异常或中断时使用备份的栈指针，而不会干扰当前正在使用的栈指针。

- ``csrrw s0, sscratch, x0``

`csrrw s0, sscratch, x0` 用于原子性地读取 `sscratch` 寄存器的值到 `s0`，并清零 `sscratch` 寄存器，通常是为了保护和恢复用户模式下的栈指针。目的就是用于异常或中断发生的恢复现场，而将 `sscratch` 清零可以避免对保存的栈指针的误用。

原子性：整个读取-写入操作是作为一个单独的指令执行的，这在多核系统中可以避免竞态条件。

问题2： `save all`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？

答：

- `scause` 用于记录中断发生原因、记录是不是外部中断，用以中断处理。
- `stval` 用于记录处理中断的辅助信息，指令获取(instruction fetch)、访存、缺页异常等，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。

`stval` 和 `scause` 这些 csr，它们通常用于记录异常的原因和相关的错误，它们为异常处理程序提供了必要的信息。而在异常处理完成后，这些寄存器的值可能不再需要恢复，因为它们的目的已经达到，或者它们可以在异常处理程序中被重新设置。

## 扩展练习Challenge3: 完善异常中断

### 中断处理流程

1. **IDT初始化**: 内核加载启动后, 通过`tail kern_init`指令跳转到内核的入口点, 即`init.c`文件中的`kern_init`函数。在这个函数中, 我们调用了`idt_init`函数对中断描述符表(Interrupt Descriptor Table, IDT)进行初始化 具体包括将`sscratch`置零、并将中断处理程序`__alltraps`的地址写入`stvec`(中断向量表基址)

```
void idt_init(void) {
    extern void __alltraps(void);
    //约定: 若中断前处于S态, sscratch为0
    //若中断前处于U态, sscratch存储内核栈地址
    //那么之后就可以通过sscratch的数值判断是内核态产生的中断还是用户态产生的中断
    //我们现在是内核态所以给sscratch置零
    write_csr(sscratch, 0);
    //我们保证__alltraps的地址是四字节对齐的, 将__alltraps这个符号的地址直接写到
    stvec寄存器
    write_csr(stvec, &__alltraps);
}
```

2. **中断处理程序**: `__alltraps`首先将所有通用目的寄存器(GPRs)的值保存到内存中, 并将这些寄存器的值以及`scause`, `sepc`, `sstatus`, `stval`封装为一个名为`trapframe`的结构体 将这个结构体作为参数传给`trap.c`中的`trap`函数进行处理 处理完毕后再将所有寄存器的值restore, 最终使用`sret`指令将`sepc`的值写入`pc`, 返回到中断之前的地址, 从S态返回至U态。

### 编程实现

为了处理非法指令异常和断点异常, 我们需要修改`trap.c`中`exception_handler`函数的对应部分, 如下所示。

```
void exception_handler(struct trapframe *tf) {
    switch (tf->cause) {
        case CAUSE_ILLEGAL_INSTRUCTION:
            // 非法指令异常处理
            /* LAB1 CHALLENGE3 YOUR CODE : */
            /*(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");
            //输出指令异常类型为非法指令 ( Illegal instruction )
            cprintf("Illegal instruction caught at0x%x\n", tf->epc);
            //输出触发异常的指令地址
            tf->epc+=4;
            //更新 tf->epc寄存器, 指向下一条指令
            break;
        case CAUSE_BREAKPOINT:
            //断点异常处理
```

```

/* LAB1 CHALLENGE3 YOUR CODE : */
/* (1)输出指令异常类型 ( breakpoint )
 * (2)输出异常指令地址
 * (3)
 */
cprintf("Exception type: breakpoint\n");
//输出指令异常类型为断点异常 ( breakpoint )
cprintf("ebreak caught at0x%x\n", tf->epc);
//输出触发异常的指令地址
tf->epc+=4;
//更新 tf->epc寄存器, 指向下一条指令
break;
...
}
}

```

这里我们通过`trapframe`的`epc`获取到了触发异常的指令地址, 并连带异常类型输出到命令行 之后`epc+4`使其指向触发异常的指令的下一条指令, 从而保证`sret`时能够返回到中断之前的地址

### 实验验证

我们可以使用内联汇编来添加非法指令和`ebreak`指令 特别要注意的是, `intr_enable`函数将`sstatus`的`SIE`置位后, 异常处理程序才会开始工作 因此这些指令必须在之后添加在`intr_enable`后

我们在`kern_init`函数中分别添加了两条`ebreak`指令和两条非法指令来触发断点异常和非法指令异常, 如下所示

```

int kern_init(void) {
    .....
    .....

    intr_enable(); // enable irq interrupt

    asm volatile(".word 0x00100073"); //ebreak指令的指令码
    asm volatile(".word 0x00100073");
    asm volatile(".word 0xffffffff"); //非法指令码
    asm volatile(".word 0xffffffff");
    while (1)
        ;
}

```

我们可以通过gdb调试来查看添加的这4条指令对应的内存地址, 结果如下: `0x8020004e <kern_init+68>: ebreak 0x80200052 <kern_init+72>: ebreak 0x80200056 <kern_init+76>: 0xffff 0x80200058 <kern_init+78>: 0xffff 0x8020005a <kern_init+80>: 0xffff 0x8020005c <kern_init+82>: 0xffff`

命令行输出结果如下所示。可以看到我们成功捕获了以上异常并成功输出了引发异常的指令地址  
 Exception type: breakpoint ebreak caught at0x8020004e Exception type: breakpoint ebreak caught at0x80200052 Exception type:Illegal instruction Illegal instruction caught at0x80200056 Exception type:Illegal instruction Illegal instruction caught at0x8020005a

# 知识点总结

## 1.了解了中断分类：

- **异常 (exception)**：正在执行的指令发生错误。如“访问无效地址”，“0作除数”，“缺页”。其中，“缺页”可以恢复，“0作除数”不能恢复。
- **陷入 (Trap)**：主动通过一条指令停下来，跳转到处理函数。如“通过 `ecall` 的 `syscall`”，“通过 `ebreak` 的 `breakpoint`”。
- **外部中断 (Interrupt)**：CPU 执行过程被外设信号打断，我们必须停下来先对外设进行处理。如：定时器倒计时结束，串口收到数据。这是异步的，CPU并不知道外部中断何时发生，在正常执行代码时，有了中断才去处理。中断处理需要较高权限，中断处理程序在内核态。

## 2.riscv64 的权限模式

- **Machine mode**: Risc-v 硬件线程最高权限的模式。一旦发生异常，就会被移交到M模式处理程序，所有 Riscv 处理器都必须实现的唯一权限模式
- **S mode**: Unix 系统中的大多数exception都应该进行 S 模式下的系统调用。M 模式的异常处理程序可以将异常重新导向 S 模式，也支持通过**异常委托机制** (Machine Interrupt Delegation, 机器中断委托) 选择性地将中断和同步异常直接交给 S 模式处理,而完全绕过 M 模式。

## 3.中断处理机制：

- **实验知识点**：中断处理机制是操作系统中用于响应外部事件（如硬件中断、软件中断）的一种机制。在 RISC-V中，中断处理涉及到中断向量表、中断服务例程、中断优先级等。
- **操作系统原理**：中断处理是操作系统内核的一部分，用于处理硬件设备发出的信号，如I/O操作完成、时钟中断等。

## 4.上下文环境的保存与恢复：

- **实验知识点**：中断发生时，需要保存当前的执行状态（即上下文），以便中断处理完毕后能够恢复到中断发生前的状态。这通常通过保存寄存器状态到中断帧 (TrapFrame) 来实现。
- **操作系统原理**：上下文切换是操作系统调度进程时必须进行的操作，它涉及到保存当前进程的状态，并加载新进程的状态。

## 5.中断帧 (TrapFrame)：

- **实验知识点**：中断帧是一个数据结构，用于在中断发生时保存CPU的状态，包括通用寄存器、程序计数器等。
- **操作系统原理**：在操作系统中，中断帧的概念与进程的上下文切换密切相关，进程的上下文切换同样需要保存和恢复寄存器状态。