Lab0&Lab1

sscratch的作用

`sscratch`寄存器在处理用户态程序的中断时才起作用

在这段代码中，`sscratch`寄存器的作用是作为临时存储，在异常或中断处理过程中保存和恢复某些重要的寄存器值。具体来说，`sscratch`寄存器在异常处理过程中用于以下目的：

1. **保存原始的栈指针**：在`SAVE_ALL`宏中，当前的栈指针（`sp`）被保存到`sscratch`寄存器。这是因为在异常处理过程中，栈指针可能会被修改，所以需要在`sscratch`中保留一个原始的栈指针的副本，以便在异常处理结束后能够正确地恢复堆栈状态。
2. **避免递归异常**：`sscratch`寄存器还用于防止递归异常的发生。在异常处理过程中，如果再次发生异常，处理器会检查`sscratch`寄存器的值。如果`sscratch`已经被设置为非零值，这意味着一个异常处理正在执行中，此时如果再次发生异常，处理器会意识到这是一个递归异常，并采取相应的措施。
3. **恢复状态**：在`RESTORE_ALL`宏中，`sscratch`寄存器的值被恢复到`s0`寄存器，这样可以在异常处理程序中访问原始的栈指针。

```
SAVE_ALL:
    csrw sscratch, sp  # 将当前栈指针 sp 保存到 sscratch 寄存器
    ...
    csrrw s0, sscratch, x0  # 在异常发生时，将 sscratch 寄存器的值（原始 sp）保存到 s0
    ...
RESTORE_ALL:
    LOAD s0, 2*REGBYTES(sp)  # 从栈上恢复 s0（包含原始 sp）
    ...
```

它为异常处理提供了一个安全的方式来保存和恢复现场，确保了异常处理的可靠性。

stevc的前两位(中断向量表基址)

在中断产生后，应该有个**中断处理程序**来处理中断。CPU怎么知道中断处理程序在哪？实际上，RISCV架构有个CSR叫做`stvec`(Supervisor Trap Vector Base Address Register)，即所谓的"中断向量表基址"。中断向量表的作用就是把不同种类的中断映射到对应的中断处理程序。如果只有一个中断处理程序，那么可以让`stvec`直接指向那个中断处理程序的地址。

对于RISCV架构，`stvec`会把**最低位的两个二进制位用来编码一个"模式"**：

- 如果是"00"就说明更高的SXLEN-2个二进制位存储的是唯一的中断处理程序的地址(SXLEN是`stval`寄存器的位数)
- 如果是"01"说明更高的SXLEN-2个二进制位存储的是中断向量表基址，通过不同的异常原因来索引中断向量表。

但是怎样用62个二进制位编码一个64位的地址？RISCV架构要求这个地址是四字节对齐的，**总是在较高的62位后补两个0。**

sstatus寄存器

所以，`sstatus`寄存器(Supervisor Status Register)里面有一个二进制位`SIE`(supervisor interrupt enable，在RISCV标准里是2^1 对应的二进制位)，数值为0的时候，如果当程序在S态运行，将禁用全部中断。（对于在U态运行的程序，SIE这个二进制位的数值没有任何意义），`sstatus`还有一个二进制位`UIE`(user interrupt enable)可以在置零的时候禁止用户态程序产生中断。

为什么触发下一个时钟时，ecall不会重复执行

> **ecall**(environment call)，当我们在 S 态执行这条指令时，会触发一个 ecall-from-s-mode-exception，从而进入 M 模式中的中断处理流程（如设置定时器等）；当我们在 U 态执行这条指令时，会触发一个 ecall-from-u-mode-exception，从而进入 S 模式中的中断处理流程（常用来进行系统调用）。

```
void clock_init(void) {
    // sie这个CSR可以单独使能/禁用某个来源的中断。默认时钟中断是关闭的
    // 所以我们要在初始化的时候，使能时钟中断
    set_csr(sie, MIP_STIP); // enable timer interrupt in sie
    //设置第一个时钟中断事件
    clock_set_next_event();
    // 初始化一个计数器
    ticks = 0;

    cprintf("++ setup timer interrupts\n");
}
//设置时钟中断：timer的数值变为当前时间 + timebase 后，触发一次时钟中断
//对于QEMU，timer增加1，过去了10^-7 s，  也就是100ns
```

STIP会被复位，所以不会重复执行时钟中断

kernel里执行的第一条指令（真正的入口点）

我们在`kern/init/init.c`编写函数`kern_init`，作为“真正的”内核入口点。为了让我们能看到一些效果，我们希望它能在命令行进行格式化输出。

如果我们在linux下运行一个C程序，需要格式化输出，那么大一学生都知道我们应该`#include<stdio.h>`。于是我们在`kern/init/init.c`也这么写一句。且慢！linux下，当我们调用C语言标准库的函数时，实际上依赖于`glibc`提供的运行时环境，也就是一定程度上依赖于操作系统提供的支持。可是我们并没有把`glibc`移植到ucore里！

怎么办呢？只能自己动手，丰衣足食。QEMU里的OpenSBI固件提供了输入一个字符和输出一个字符的接口，我们一会把这个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。这里格式化输出函数的名字不使用原先的`printf()`，强调这是我们在ucore里重新实现的函数。

函数具体内容：

```
// kern/init/init.c
#include <stdio.h>
```

```c
#include <string.h>
//这里include的头文件， 并不是C语言的标准库，而是我们自己编写的!

//noreturn 告诉编译器这个函数不会返回
int kern_init(void) __attribute__((noreturn));

int kern_init(void) {
    extern char edata[], end[];
    //这里声明的两个符号，实际上由链接器ld在链接过程中定义，所以加了extern关键字
    memset(edata, 0, end - edata);
    //内核运行的时候并没有c标准库可以使用，memset函数是我们自己在string.h定义的

    const char *message = "(THU.CST) os is loading ...\n";
    cprintf("%s\n\n", message); //cprintf是我们自己定义的格式化输出函数
       while (1)
        ;
}
```

格式化输出在哪一态，为什么

因为格式化输出是在硬件层面，所以是固件，应该是在M态

makefile的意义（要知道代码的意思）

我们需要：编译所有的源代码，把目标文件链接起来，生成elf文件，生成bin硬盘镜像，用qemu跑起来

这一系列复杂的命令，我们不想每次用到的时候都敲一遍，所以我们使用~~魔改的祖传~~ `Makefile`。

make命令执行时，需要一个 makefile （或Makefile）文件，以告诉make命令需要怎么样的去编译和链接程序

makefile的基本规则简介

在使用这个makefile之前，还是让我们先来粗略地看一看makefile的规则。

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

target也就是一个目标文件，可以是object file，也可以是执行文件。还可以是一个标签（label）。prerequisites就是，要生成那个target所需要的文件或是目标。command也就是make需要执行的命令（任意的shell命令）。 这是一个文件的依赖关系，也就是说，target这一个或多个的目标文件依赖于prerequisites中的文件，其生成规则定义在 command中。如果prerequisites中有一个以上的文件比target文件要新，那么command所定义的命令就会被执行。这就是makefile的规则。也就是makefile中最核心的内容

内部中断和外部中断

异常(Exception)，指在执行一条指令的过程中发生了错误，此时我们通过中断来处理错误。最常见的异常包括：访问无效内存地址、执行非法指令(除零)、发生缺页等。他们有的可以恢复(如缺页)，有的不可恢复(如除零)，只能终止程序执行。

陷入(Trap)，指我们主动通过一条指令停下来，并跳转到处理函数。常见的形式有通过ecall进行系统调用(syscall)，或通过ebreak进入断点(breakpoint)。

外部中断(Interrupt)，简称中断，指的是 CPU 的执行过程被外设发来的信号打断，此时我们必须先停下来对该外设进行处理。典型的有定时器倒计时结束、串口收到数据等。

外部中断是异步(asynchronous)的，CPU 并不知道外部中断将何时发生。CPU 也并不需要一直在原地等着外部中断的发生，而是执行代码，有了外部中断才去处理。我们知道，CPU 的主频远高于 I/O 设备，这样避免了 CPU 资源的浪费。

由于中断处理需要进行较高权限的操作，中断处理程序一般处于**内核态**，或者说，处于“比被打断的程序更高的特权级”。注意，在RISCV里，中断(interrupt)和异常(exception)统称为"trap"。

> 1. **来源不同**：
>   - **外部中断**：通常由外部设备（如键盘、鼠标、磁盘驱动器等）或外部事件（如定时器、电源故障等）触发。
>   - **内部中断**：由CPU内部事件触发，如算术溢出、除零错误、非法指令、硬件故障等。
> 2. **处理方式**：
>   - **外部中断**：通常需要操作系统的设备驱动程序来处理，这些驱动程序负责与硬件设备通信，并处理来自设备的输入/输出请求。
>   - **内部中断**：通常由操作系统的异常处理程序处理，这些程序负责处理错误和异常情况，并可能触发系统崩溃、重启或恢复操作。
> 3. **优先级**：
>   - **外部中断**：优先级可能较低，因为它们通常与用户输入或设备状态变化有关，可以稍微延迟处理。
>   - **内部中断**：优先级通常较高，因为它们涉及到系统稳定性和数据完整性，需要立即处理。

特权级以及特权指令

> RISCV有四种**特权级（privilege level）**。
>
> | Level | Encoding | 全称 | 简称 |
> | --- | --- | --- | --- |
> | 0 | 00 | User/Application | U |
> | 1 | 01 | Supervisor | S |
> | 2 | 10 | Reserved(目前未使用，保留) | |
> | 3 | 11 | Machine | M |
>
> 粗略的分类：
>
> U-mode是用户程序、应用程序的特权级，S-mode是操作系统内核的特权级，M-mode是固件的特权级。

在计算机中，**固件(firmware)**是一种特定的计算机软件，它为设备的特定硬件提供低级控制，也可以进一步加载其他软件。固件可以为设备更复杂的软件（如操作系统）提供标准化的操作环境。对于不太复杂的设备，固件可以直接充当设备的完整操作系统，执行所有控制、监视和数据操作功能。 在基于 x86 的计算机系统中, BIOS 或 UEFI 是固件；在基于 riscv 的计算机系统中，OpenSBI 是固件。OpenSBI运行在**M态（M-mode）**，因为固件需要直接访问硬件。

RISCV支持以下和中断相关的特权指令：

- **ecall**(environment call)，当我们在 S 态执行这条指令时，会触发一个 ecall-from-s-mode-exception，从而进入 M 模式中的中断处理流程（如设置定时器等）；当我们在 U 态执行这条指令时，会触发一个 ecall-from-u-mode-exception，从而进入 S 模式中的中断处理流程（常用来进行系统调用）。
- **sret**，用于 S 态中断返回到 U 态，实际作用为pc←sepc，回顾**sepc**定义，返回到通过中断进入 S 态之前的地址。
- **ebreak**(environment break)，执行这条指令会触发一个断点中断从而进入中断处理流程。
- **mret**，用于 M 态中断返回到 S 态或 U 态，实际作用为pc←mepc，回顾**sepc**定义，返回到通过中断进入 M 态之前的地址。（一般不用涉及）

解释代码（进入kernel init）

init.c文件中的定义，首先定义了一个noreturn，告诉编译器这个函数不会返回

然后声明两个符号，edata和end，由链接器ld在链接过程中进行定义

将内存区域进行初始化，将edata到end部分的内存全部清空

然后定义一个指针，输出我们的内容，然后之后就直接进入死循环，不输入内容

```
#include <stdio.h>
#include <string.h>
#include <sbi.h>
int kern_init(void) __attribute__((noreturn));
//是kern_init函数的声明，告诉编译器这个函数不会返回值。
//__attribute__((noreturn))是一个GCC扩展属性，用于指示函数不会返回。
```

```
int kern_init(void) {//c语言编写的内核入口点，从entry.s跳转过来，完成其他初始化工作
    extern char edata[], end[];//声明了两个变量edata和end
    memset(edata, 0, end - edata);//初始化，使用memset函数将内存中edata到end部分的内存全部清空

    const char *message = "(THU.CST) os is loading ...\n";
    //定义了一个指向字符串的指针，表示操作系统正在加载
    cprintf("%s\n\n", message);//输出message指向的字符串到控制台
   while (1)//死循环，使内核在完成初始化之后不会退出
        ;
}
```

entry一些内容，栈的定义与大小，栈顶和栈底

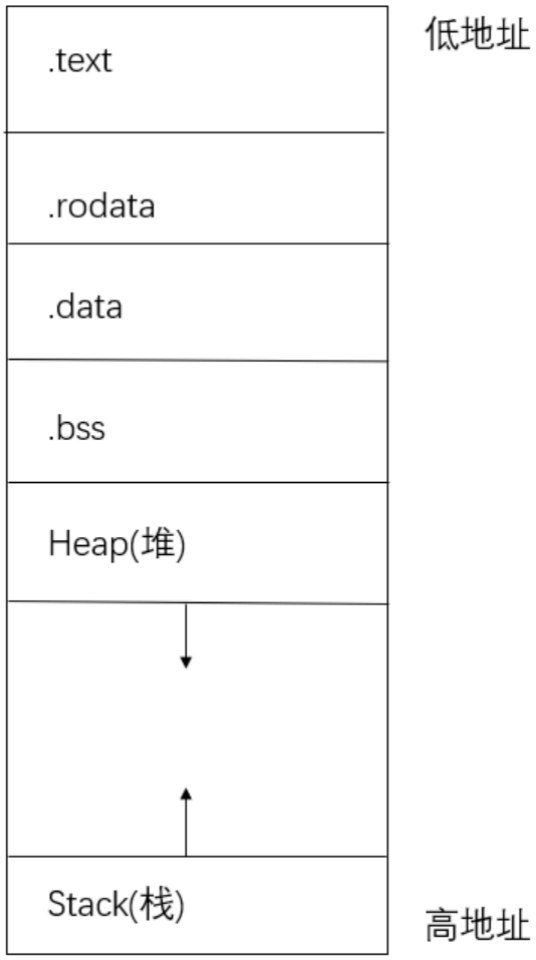
entry.s的内容

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

    .section .text,"ax",%progbits
    ;指定接下来的代码应该放在名为.text的段中。"ax"表示该段是可读(a)、可执行(x)的，%progbits表示程序的代码和数据。
    .globl kern_entry;声明一个全局函数kern_entry
kern_entry:;这是定义内核入口点的标签
    la sp, bootstacktop;将bootstacktop的地址加载到栈指针sp中。这意味着设置栈的起始位置。

    tail kern_init
    ;用于跳转到另一个函数并传递参数。这里它跳转到kern_init函数，并且由于tail调用的特性，它还会清理栈，因为它假定调用的函数会返回。

.section .data;数据段
    # .align 2^12
    .align PGSHIFT;定义内存中的对齐量，确保完成页对齐
    .global bootstack;声明全局变量函数
bootstack:
    .space KSTACKSIZE;为内核栈分配了空间
    .global bootstacktop;声明全局符号
bootstacktop:;定义内核栈顶的标签
```

数据段中，定义了内存中的页对齐量，2的12次，也就是4096B(4KB)

然后一共定义的是2页，所以大小应该是8KB

栈的组成方式：



栈的定义和大小：

```
#ifndef __KERN_MM_MEMLAYOUT_H__
#define __KERN_MM_MEMLAYOUT_H__

#define KSTACKPAGE          2                           // # of pages in kernel
stack
#define KSTACKSIZE          (KSTACKPAGE * PGSIZE)       // sizeof kernel stack

#endif /* !__KERN_MM_MEMLAYOUT_H__ */

/*
定义了一个宏KSTACKPAGE，它的值为2。
这个宏表示内核栈使用的页数。
在操作系统中，内存通常被分割成固定大小的块，称为“页”（page）。
每页的大小通常是4KB。
*/
```

```
#ifndef __KERN_MM_MMU_H__
#define __KERN_MM_MMU_H__

#define PGSIZE          4096                    // bytes mapped by a page
#define PGSHIFT         12                      // log2(PGSIZE)

#endif /* !__KERN_MM_MMU_H__ */
```

```
/*
定义了一个宏PGSIZE，它的值为4096字节。
这表示操作系统内存分页机制中每页的大小。
在许多操作系统和硬件架构中，内存页的大小通常为4KB。

定义了一个宏PGSHIFT，它的值为12。这个宏表示PGSIZE的对数值，即log2(PGSIZE)。
由于4096是2^12，所以PGSHIFT的值为12。这个宏常用于计算地址到页的偏移量。

*/
```

是不是所有的中断都能复原

不是

1. **可恢复的中断**：
   - **硬件中断**：大多数硬件中断（如I/O设备请求）是可以恢复的。处理程序完成后，设备状态被更新，处理器可以恢复执行被中断的指令。
   - **软件中断**：如系统调用，通常也可以恢复。系统调用完成后，控制权返回给用户程序。
2. **不可恢复的中断**：
   - **致命错误**：如硬件故障或严重的程序错误（如内存管理错误），可能无法恢复。这些错误可能需要重启系统或采取其他恢复措施。
   - **非屏蔽中断（NMI）**：某些类型的中断（如NMI）可能指示严重的问题，不一定能完全恢复到中断前的状态。

为什么store的时候没有保存x2，x2去哪了

Register	ABI Name	Description	Saver
x0	zero	Hard-wired zero	—
x1	ra	Return address	Caller
x2	sp	Stack pointer	Callee
x3	gp	Global pointer	—
x4	tp	Thread pointer	—
x5	t0	Temporary/alternate link register	Caller
x6–7	t1–2	Temporaries	Caller
x8	s0/fp	Saved register/frame pointer	Callee
x9	s1	Saved register	Callee
x10–11	a0–1	Function arguments/return values	Caller
x12–17	a2–7	Function arguments	Caller
x18–27	s2–11	Saved registers	Callee
x28–31	t3–6	Temporaries	Caller

如上所示，x2寄存器是sp寄存器，代表callee，sp寄存器是在最后才进行恢复的，因此暂时没有在此处进行保存

trapframe的结构

```
struct pushregs {
    uintptr_t zero;  // Hard-wired zero
    uintptr_t ra;    // Return address
    uintptr_t sp;    // Stack pointer
    uintptr_t gp;    // Global pointer
    uintptr_t tp;    // Thread pointer
    uintptr_t t0;    // Temporary
    uintptr_t t1;    // Temporary
    uintptr_t t2;    // Temporary
    uintptr_t s0;    // Saved register/frame pointer
    uintptr_t s1;    // Saved register
    uintptr_t a0;    // Function argument/return value
    uintptr_t a1;    // Function argument/return value
    uintptr_t a2;    // Function argument
    uintptr_t a3;    // Function argument
    uintptr_t a4;    // Function argument
    uintptr_t a5;    // Function argument
    uintptr_t a6;    // Function argument
    uintptr_t a7;    // Function argument
    uintptr_t s2;    // Saved register
    uintptr_t s3;    // Saved register
    uintptr_t s4;    // Saved register
    uintptr_t s5;    // Saved register
    uintptr_t s6;    // Saved register
    uintptr_t s7;    // Saved register
    uintptr_t s8;    // Saved register
    uintptr_t s9;    // Saved register
    uintptr_t s10;   // Saved register
    uintptr_t s11;   // Saved register
    uintptr_t t3;    // Temporary
    uintptr_t t4;    // Temporary
    uintptr_t t5;    // Temporary
    uintptr_t t6;    // Temporary
};

struct trapframe {
    struct pushregs gpr;
    uintptr_t status; //sstatus
    uintptr_t epc; //sepc
    uintptr_t badvaddr; //sbadvaddr
    uintptr_t cause; //scause
};
```

C语言里面的结构体，是若干个变量在内存里直线排列。也就是说，一个`trapFrame`结构体占据36个`uintptr_t`的空间（在64位RISCV架构里我们定义`uintptr_t`为64位无符号整数），里面依次排列通用寄存器`x0`到`x31`,然后依次排列4个和中断相关的CSR, 我们希望中断处理程序能够利用这几个CSR的数值。

就是说一共有32个寄存器，还有四个和中断相关的CSR

```
uintptr_t status; //sstatus  sstatus 寄存器定义了当前的处理器状态和控制位
uintptr_t epc; //sepc  sepc 寄存器在异常发生时保存了异常发生的位置，即当前指令的地址（PC）
uintptr_t badvaddr; //sbadvaddr  sbadaddr 寄存器用于存储导致访问故障的虚拟地址。如果异常是由于加载、存储或其他内存访问指令试图访问非法地址引起的，这个寄存器将包含尝试访问的地址。
uintptr_t cause; //scause  scause 寄存器指示了导致异常的具体原因。这个值是一个编码后的数字，表示不同类型的异常或中断
```

Makefile文件里第114行的那个变量是什么作用

- 在这一行`KOBJS = $(call read_packet,kernel libs)`中，`KOBJS`是一个变量，它被赋予了函数`read_packet`的返回值。这个函数调用的目的是读取`kernel`和`libs`目录下的所有源代码文件，并返回这些文件的列表。

作用：

里的`$(call read_packet,kernel libs)`调用会执行以下步骤：

1. **查找源文件**：`read_packet`函数会在`kernel`和`libs`目录下查找所有符合编译要求的源文件。这通常是`.c`、`.S`（汇编）等类型的文件。
2. **生成对象文件列表**：对于找到的每个源文件，`read_packet`函数会生成相应的对象文件（`.o`文件）的名称。对象文件是源文件编译后的中间产物。
3. **返回结果**：函数将生成的对象文件列表返回给`KOBJS`变量。
4. **赋值**：`KOBJS`变量被赋值为`read_packet`函数返回的文件列表。这样，`KOBJS`就包含了所有需要编译的源文件对应的对象文件的名称。

在后续的编译过程中，`KOBJS`变量会被用来指定哪些对象文件需要被链接成最终的内核文件。例如，链接命令可能会像这样使用`KOBJS`：

```
$(kernel): $(KOBJS)
    @echo + ld $@
    $(V)$(LD) $(LDFLAGS) -T tools/kernel.ld -o $@ $(KOBJS)
    @$(OBJDUMP) -S $@ > $(call asmfile,kernel)
    @$(OBJDUMP) -t $@ | $(SED) '1,/SYMBOL TABLE/d; s/ .* / /; /^$$/d' > $(call
symfile,kernel)
```