Lab 1: RV64 内核引导与时钟中断

1 实验目的

学习 RISC-V 相关知识，了解 OpenSBI 平台，实现 sbi 调用函数，封装打印函数，并利用 Makefile 来完成对整个工程的管理。

2 实验内容及要求

- 阅读 RISC-V 中文手册，学习 RISC-V 相关知识
- 学习 Makefile 编写规则，补充 Makefile 文件使得项目成功运行
- 了解 OpenSBI 的运行原理，编写代码通过 sbi 调用实现字符串的打印
- 利用 OpenSBI 平台的 SBI 调用触发时钟中断，并通过代码设计实现定时触发时钟中断的效果

请各位同学独立完成实验，任何抄袭行为都将使本次实验判为0分。

请跟随实验步骤完成实验并根据文档中的要求记录实验过程，最后删除文档末尾的附录部分，并命名为**“学号*姓名*lab1.pdf”**，你的代码请打包并命名为**“学号*姓名*lab1.zip/tar/..”**，文件一并上传至学在浙大平台。

3 实验步骤

3.1 搭建实验环境

本实验提供的代码框架结构如图，你可以点击 lab1.zip 进行下载。首先，请下载相关代码，并移动至你所建立的本地映射文件夹中（即`lab0`中创建的`os_experiment`文件夹）。

```
Lab1
├── □ arch
|   └── □ riscv
|       ├── □ boot
|       ├── □ kernel
|       |   ├── □ clock.c
|       |   ├── □ entry.S
|       |   ├── □ head.S
|       |   ├── □ init.c
|       |   ├── □ main.c
|       |   ├── □ Makefile
|       |   ├── □ print.c
|       |   ├── □ sbi.c
|       |   ├── □ trap.c
|       |   └── □ vmlinux.lds
|       └── □ Makefile
├── □ include
|   ├── □ defs.h
```

```
|   ├── □ riscv.h
|   └── □ test.h
└── □ Makefile
```

3.2 了解项目框架，编写 MakeFile（10%）

3.2.1 编写 Makefile 文件

1. 请参考【附录A.Makefile介绍】学习 Makefile 的基本知识。
2. 阅读项目中的 Makefile 文件，确保你理解了 Makefile 文件中每一行的作用（一些参数配置等不做要求）。

注意：在 `Lab1/Makefile` 中已经帮助你预定义好了文件的 `include` 地址，编译参数等等，你再编写下面的 Makefile 的时候是需要用到的，如果不知道如何使用，可以阅读代码框架里面的其他 Makefile 文件（有参考），仔细做了解。

> Makefile 部分是需要大家自己学习的，请务必阅读附录部分提供的教程，这里简单讲一下本项目的 Makefile 结构，最外层的 Makefile 定义了一些基础变量以供使用，包括本项目使用的编译器，编译的参数，头文件所在路径，另外定义了一些基本的编译目标，如 all，vmlinux，run，debug，其中 vmlinux 就是编译出 vmlinux 这个本项目用的程序， run 则是编译出的基础上再运行，debug 则是编译出的基础上以 debug 模式运行（即 Lab 0 中介绍的调试内核部分）在编译 vmlinux 的时候，最外层 Makefile 会跳转到内层的一些 Makefile 去执行，内层的 Makefile 文件干的事情基本就是把各文件夹的 .c 文件编译成 .o 文件。都编译好再跳转回根目录的 Makefile，然后把所有 .o 文件编译成一个整体的可执行文件。编译链接是什么请搜索 C 编译链接详解。实验不做要求，但实验对 Makefile 有要求。

请补充完整 `./arch/riscv/kernel/Makefile` 文件使得整个项目能够顺利编译，最后，将你的代码补充在下方的代码框中。

你需要确保 `make test` 指令和 `make clean` 指令均可正常运行（在 `**lab1**` 目录下），如果运行成功会显示绿色的 Success 字样提示。

```
# TODO: 将本文件夹下的所有.S与.c文件编译成.o文件
# 1. 使用wildcard获取所有.S与.c文件（TODO）
ASM_SRC= $(sort $(wildcard *.S))
C_SRC= $(sort $(wildcard *.c))

# 2. 使用patsubst将.S与.c文件转换成.o文件（TODO）
ASM_OBJ= $(patsubst %.S, %.o, $(ASM_SRC))
C_OBJ= $(patsubst %.c, %.o, $(C_SRC))

# 3. 定义目标文件all与依赖关系。执行make命令时可以指定编译目标，比如make all，make clean，然后make会寻找该编译目标，并根据make的运行机制进行编译。（TODO）
all:$(ASM_OBJ) $(C_OBJ)

# 4. 使用%.o:%.S与%.o:%.c定义依赖关系与命令（Done）
%.o:%.S
    ${CC}  ${CFLAG}  -c $<
%.o:%.c
    ${CC}  ${CFLAG}  -c $<

# 请自行查询makefile相关文档，包括makefile函数、变量、通配符，make的运行机制等
```

```
clean:
    $(shell rm *.o 2>/dev/null)
```

3.2.2 解释 Makefile 命令

请解释 **`lab1/Makefile` 中的下列命令：

```
line 25: ${MAKE} -C arch/riscv all
```

> all 不是全部的意思，而是makefile文件里定义的目标

含义：

- `$(MAKE)` 是一个变量，通常用于指代系统中的"make"工具;
- `-C arch/riscv` 是一个选项， 展开为`-directory=arch/riscv` ，它告诉"make"工具进入指定的目录以执行构建操作。在这里，`arch/riscv` 是要进入的目录路径;
- `all` 是一个目标，在"Makefile"中定义。它指示"make"工具执行"all"这个目标所定义的构建任务;

因此，这整行的意思是使用"make"工具进入到指定的目录 `arch/riscv` 并执行该目录中定义的"all"目标所关联的构建任务。

请解释** `lab1/arch/riscv/kernel/Makefile` 中的下列命令：**

```
line 16: %.o:%.c
    ${CC}  ${CFLAG}  -c $<
```

含义：

- `%.o:%.c`表示所有的.o文件的依赖条件都是同名的.c文件;
- `${CC}`表示指定编译器的变量，`${CFLAG}`表示指定的编译参数变量，`-c`表示只编译不链接;
- `$<`是一个自动化变量，表示了所有依赖目标的挨个值；
- 整行命令的意思是使用`${CC}`指定的编译器编译所有.o文件时，都自动找到同名的.c文件作为依赖并按照`${CFLAG}`中的参数进行编译。

3.3 学习 RISC-V 相关知识及特权架构（5%）

后续实验中将持续使用 RISC-V 指令集相关的内容，请参考【附录B.RISC-V指令集】了解相关知识，**下载并阅读 RISC-V 手册，掌握基础知识、基本命令及特权架构相关内容。**

3.3.1 基础命令掌握情况

请按照下面提供的例子，补充完整各指令含义。

```
# 加载立即数 0x40000 到 t0 寄存器中
li t0,0x40000

# 将寄存器t0中的值写入名为“satp”的控制状态寄存器中
csrw satp, t0

# 将t0-t1的值写入t0中
sub t0, t0, t1
```

```
# 将x1的值存入内存地址为sp+8的位置
sd x1, 8(sp)

# 将stack_top这个标签对应的地址存到sp
la sp, stack_top
```

3.3.2 对risc-v特权模式的理解

请解释risc-v特权模式存在的意义：

1. 安全性：RISC-V特权模式允许不同级别的特权级别，有助于分离软件与硬件设施，包括隔离用户应用程序和操作系统内核，以防止恶意软件访问关键资源；
2. 操作系统支持：不同的特权级别有助于操作系统的设计和实现，可以确保操作系统能够有效地管理系统资源；
3. 虚拟化：RISC-V特权模式支持虚拟化，使多个虚拟机可以在同一硬件上并行运行，而互相不受干扰。这对云计算和服务器领域特别重要，因为它可以提高硬件资源的利用率和隔离不同虚拟机之间的资源。
4. 低级控制：机器模式是RISC-V的最高特权级别，允许对底层硬件进行更高程度的控制。这对于嵌入式系统、固件开发和操作系统内核的实现非常有用，因为它们需要直接访问底层硬件资源。

假设我们有一个实现了U态和M态的risc-v架构CPU，并在其上运行os与用户程序，**请解释CPU如何在U态与M态之间进行切换（言之有理即可）：** CPU从U态切换到M态通常是由软件触发的，通常用户程序需要执行一个特权指令，例如“ecall”来请求切换到机器模式，然后硬件会捕捉到这个请求并引发中断；CPU要从M态切换回U态同样需要执行一种特权指令，如“mret”，然后硬件会还原用户的各种控制状态寄存器来切换回U态（提示：从U态到M态是“谁操作”的，是硬件还是软件，从M态到U态呢？）

3.4 通过 OpenSBI 接口实现字符串打印函数（30%）

3.4.1 程序执行流介绍

对于本次实验，我们选择使用 OpenSBI 作为 bios，来进行机器启动时的硬件初始化与寄存器设置（此时机器处于M态），并使用 OpenSBI 所提供的接口完成诸如字符打印等操作。

请参考【附录C.OpenSBI介绍】了解 OpenSBI 平台的功能及启动方式，参考【附录E. Linux Basic】了解 `vmlinux.lds`、`vmlinux` 的作用，理解执行 `make run` 命令时程序的执行过程。

```
# make run 依赖 vmlinux
# 因此，他首先会编译目标 vmlinux 然后执行 lab1/Makefile 中的该行命令
@qemu-system-riscv64 -nographic --machine virt -bios default -device
loader,file=vmlinux,addr=0x80200000 -D log
```

QEMU 模拟器完成从 ZSBL 到 OpenSBI 阶段的工作，本行指令使用 -bios default 选项将 OpenSBI 代码加载到 0x80000000 起始处。QEMU完成一部分初始化工作后（例如Power-On Self-Test 和 Initialization and Boot ROM），将会跳转到 0x80000000 处开始执行。在 OpenSBI 初始化完成后，将跳转到 0x80200000 处继续执行。因此，我们还需要将自己编译出的 `vmlinux` 程序加载至地址 0x80200000 处。

vmlinux.lds 链接脚本就可以帮助我们完成这件事情。它指定了程序的内存布局，最先加载的 `.text.init` 段代码为 `head.S` 文件的内容，该部分代码会执行调用 `main()` 函数。`main()` 函数调用了打印函数，打印函数通过 `sbi_call()` 向 OpenSBI 发起调用，完成字符的打印。

3.4.2 编写 sbi_call() 函数（10%）

当系统处于 m 模式时，对指定地址进行写操作便可实现字符的输出。但我们编写的内核运行在 s 模式**（因为我们使用了 OpenSBI 帮助我们初始化，OpenSBI会在执行内核代码之前先进入S态）**，需要使用OpenSBI 提供的接口，让运行在 m 模式的 OpenSBI 帮助我们实现输出。即运行在 s 模式的内核通过调用 `ecall` 指令（汇编级指令）发起 sbi 调用请求，触发中断，接下来 RISC-V CPU 会从 s 态跳转到 m 态的 OpenSBI 固件中。

执行 `ecall` 时需要指定 sbi 调用的编号，传递的参数。一般而言：

- `a6` 寄存器存放 SBI 调用 `Function ID` 编号
- `a7` 寄存器存放 SBI 调用 `Extension ID` 编号
- `a0`、`a1`、`a2`、`a3`、`a4`、`a5` 寄存器存放 SBI 的调用参数，不同的函数对于传递参数要求也不同。

简单来讲，你可以认为我们需要填好 `a0` 到 `a7` 这些寄存器的值，调用 `ecall` 后，OpenSBI 会根据这些值做相应的处理。以下是一些常用的函数表。

Function Name	Function ID	Extension ID
sbi_set_timer （设置时钟相关寄存器）	0	0x00
sbi_console_putchar （打印字符）	0	0x01
sbi_console_getchar （接收字符）	0	0x02
sbi_shutdown （关机）	0	0x08

你需要编写内联汇编语句以使用 OpenSBI 接口，本实验给出的函数定义如下：（注意：本实验是 64 位 riscv 程序，这意味着我们使用的寄存器都是 64 位寄存器）

```
typedef unsigned long long uint64_t;
struct sbiret {
  uint64_t error;
  uint64_t value;
};

struct sbiret sbi_call(uint64_t ext, uint64_t fid, uint64_t arg0, uint64_t arg1,
                       uint64_t arg2, uint64_t arg3, uint64_t arg4,
                       uint64_t arg5);
```

在该函数中，你需要完成以下内容：

- 将 ext (Extension ID) 放入寄存器 a7 中，fid (Function ID) 放入寄存器 a6 中，将 arg0 ~ arg5 放入寄存器 a0 ~ a5 中。
- 使用 ecall 指令。ecall 之后系统会进入 M 模式，之后 OpenSBI 会完成相关操作。
- OpenSBI 的返回结果会存放在寄存器 a0 ， a1 中，其中 a0 为 error code， a1 为返回值， 我们用 sbiret 结构来接受这两个返回值。

请参考【附录C.内联汇编】相关知识，以内联汇编形式实现 `lab1/arch/riscv/kernel/sbi.c` 中的 `sbi_call()` 函数。

注意：如果你在内联汇编中直接用到了某寄存器（比如本函数必然要直接使用 a0~a7 寄存器），那么你需要在内联汇编中指出，本段代码会影响该寄存器，如何指出请参考**[【附录D】](#8dc79ae2)**，如果不加声明，编译器可能会将你声明的要放到寄存器里的变量，放到你直接使用的寄存器内，可能引发意想不到的错误**。

最后，请将你编写好的 `sbi_call` 函数复制到下面代码框内。

```
// lab1/arch/riscv/kernel/sbi.c

#include "defs.h"

struct sbiret sbi_call(uint64_t ext, uint64_t fid, uint64_t arg0, uint64_t arg1,
                       uint64_t arg2, uint64_t arg3, uint64_t arg4,
                       uint64_t arg5) {
  struct sbiret ret;
   __asm__ volatile(
       // Your code
    "mv a7 %[ext]\n"
    "mv a6 %[fid]\n"
    "mv a0 %[arg0]\n"
    "mv a1 %[arg1]\n"
    "mv a2 %[arg2]\n"
    "mv a3 %[arg3]\n"
    "mv a4 %[arg4]\n"
    "mv a5 %[arg5]\n"
    "ecall\n"
    "mv %[error] a0\n"
    "mv %[value] a1"
    :[error] "=r" (ret.error), [value] "=r" (ret.value)
    :[arg0] "r" (arg0), [arg1] "r" (arg1), [arg2] "r" (arg2), [arg3] "r" (arg3),
[arg4] "r" (arg4), [arg5] "r" (arg5), [fid] "r" (fid), [ext] "r" (ext)
    :"memory", "a0", "a1", "a2", "a3", "a4", "a5", "a6", "a7"
   );
  return ret;
}
```

> 可以自己在makefile中添加命令把sbi.c编译成汇编代码（使用gcc的-S选项），看一下sbi_call在汇编层面是如何运行的，有没有优化空间。

####

3.4.3 编写字符串打印函数（20%）

现在你已经有了一个 C 语言层面的 `sbi_call` 接口函数，因此，后面的代码中，你只需要调用这个接口函数即可，并不需要再写汇编代码。

本节，你需要在 `./arch/riscv/kernel/print.c` 文件中通过调用 `sbi_call()` 实现字符串打印函数 `int puts(char* str)` 及数字打印函数 `int put_num(uint64_t n)`，后者可将数字转换为字符串后调用前者执行。（注意处理边界 n = 0 的情况）

提示：上节已经给出了你一个 OpenSBI 调用函数表，具体使用方法可参考OpenSBI 文档。为了利用 OpenSBI 接口打印字符，我们需要向 `sbi_call()` 函数传入 `ext=1, fid=0` 以调用 `sbi_console_putchar(int ch)` 函数，之后，第一个参数 `arg0` 需要传入待打印字符的 ASCII 码，其余没有用到的参数可直接设为0。

最后，请将你编写好的函数复制到下面代码框内。

```
// ./arch/riscv/libs/print.c

#include "defs.h"
extern struct sbiret sbi_call(uint64_t ext, uint64_t fid, uint64_t arg0,
                              uint64_t arg1, uint64_t arg2, uint64_t arg3,
                              uint64_t arg4, uint64_t arg5);

int puts(char *str) {
  // TODO
  while(*str){
    sbi_call(1, 0, *str, 0, 0, 0, 0, 0);
    str ++;
  }
  return 0;
}

int put_num(uint64_t n) {
  // TODO
  int index = PUT_NUM_LENGTH - 1;
  char str[PUT_NUM_LENGTH] = "";
  str[PUT_NUM_LENGTH - 1] = 0;
  if(n == 0){
    str[--index] = '0';
  }else{
    while (n > 0){
      index--;
      str[index] = n % 10 + '0';
      n = n / 10;
    }
  }
  char *newStr = str + index;
  puts(newStr);
  return 0;
}
```

3.4.4 修改链接脚本文件

裸机程序从 `.text` 段起始位置执行，所以需要利用 `vmlinux.lds` 中 `.text` 段的定义来确保 `head.S` 中的 `.text` 段被放置在其他 `.text` 段之前。这可以通过重命名来解决。

首先将 `head.S` 中的 `.text` 命名为 `.text.init`：

```
<<<<< before
.section .text
============
.section .text.init
>>>>> after
```

接下来将 `entry.S` 中的 `.text` 命名为 `.text.entry`：

```
<<<<< before
.section .text
============
.section .text.entry
>>>>> after
```

然后修改 `vmlinux.lds` 文件中的 `.text` 展开方式：

```
<<<<< before
.text : {
        *(.text)
        *(.text.*)
    }
============
.text : {
        *(.text.init)
        *(.text.entry)
        *(.text)
        *(.text.*)
    }
>>>>> after
```

如果你没有理解这段代码为什么这样修改，请阅读【附录 E: Linux Basic】部分的说明。

3.4.5 编译运行

在 `lab1` 目录下执行 `make print_only && make run`，如果输出 `"2023 Hello Oslab"` 则实验成功。

3.5 实现时钟中断（55%）

3.5.1 实现逻辑

本实验的目标是**定时触发时钟中断并在相应的中断处理函数中输出相关信息。**

代码实现逻辑如下：

- 在初始化阶段，设置 CSR 寄存器以允许 S 模式的时钟中断发生，利用 SBI 调用触发第一次时钟中断。
- SBI 调用触发时钟中断后，OpenSBI 平台自动完成 M 模式时钟中断处理，并触发 S 模式下的时钟中断，接下来会进入程序设置好的（汇编级）中断函数中。中断函数保存寄存器现场后会调用（C 语言级）中断处理函数。
- 在中断处理函数中打印相关信息，并设置下一次时钟中断，从而实现定时（每隔一秒执行一次）触发时钟中断并打印相关信息的效果。函数返回后恢复寄存器现场，调用 S 模式异常返回指令 sret 回到发生中断的指令。

对应到程序流程中：

1. 各类 init 函数：允许中断，开启时钟中断，设置第一次时钟中断，设置中断处理函数的地址。
2. trap_s 函数：保存寄存器，进入 handler_s 函数处理中断
3. handler_s 函数：判断是否是时钟中断，是就设置下一次时钟中断并输出信息。
4. 下一次时钟中断触发，再次回到 2

为了完成实验，需要同学们在 `init.c` 中设置 CSR 寄存器允许时钟中断发生，在 `clock.c` 中写设置时钟中断开启和下一次时钟中断发生时间的函数，最后在 `entry.S` 及 `trap.c` 中编写中断处理函数。

3.5.2 编写 init.c 中的相关函数（15%）

在qemu完成初始化并进入os时，默认处于S态，接下来我们将在S态实现时钟中断。

首先，我们需要开启 S 模式下的中断总开关，需要对以下寄存器进行设置：

1. 设置 `stvec` 寄存器。`stvec` 寄存器中存储着 S 模式下发生中断时跳转的地址，我们需要编写相关的中断处理函数，并将地址存入 `stvec` 中。
2. 将 `sstatus` 寄存器中的 `sie` 位打开。`sstatus[sie]` 位为 S 模式下的中断总开关，这一位为 1 时，才能响应中断。

编写 `intr_enable()` / `intr_disable()`：

这两个函数的作用如下注释。你需要使用 CSR 命令设置 `sstatus[sie]` 的值。本实验中在 `riscv.h` 文件中为你提供了一些宏定义，可以方便的使用 CSR 指令。当然，你也可以自行通过内联汇编实现。

> 提示：你需要根据 RISC-V中文手册**中的【第十章 RV32/64 特权架构】**中的内容确定 sstatus 寄存器的 sie 位是第几位，从而为该位赋 1。

> **另一个教程：CSR 寄存器各位含义见** 特权架构。

> `write_csr(a, 2)` 相当于 a = 2
> `set_csr(a, 2)` 相当于 a |= 2
> `clear_csr(a, 2)` 相当于 a &= (~2)

请在下方代码框中补充完整你的代码：

```
void intr_enable(void) {
  // 设置 sstatus[sie] = 1, 打开 s 模式的中断开关
  // TODO
  set_csr(sstatus, 1 << 1);
}

void intr_disable(void) {
  // 设置 sstatus[sie] = 0, 关闭 s 模式的中断开关
  // TODO
  set_csr(sstatus, 0);
}
```

请对你的代码做简要解释：

答：

- 根据注释的说明，打开S模式的中断使能开关需要将`sstatus`表示`sie`的一位置为1，然后我查询指导中提供的RISCV手册查询可知`sie`是`sstatus`寄存器的第一位，所以这里使用`riscv.h`中提供的函数`set_csr(reg, bit)`将`sstatus[sie]`置为1；
- 同理，用相同方法将`sstatus[sie]`置为0，关闭S模式的中断开关；

编写 `idt_init()`：

该函数需要你向 `stvec` 寄存器中写入中断处理后跳转函数的地址，在本实验中，我们的中断处理函数是 `trap_s` 这个函数。

> 提示：C 语言中，可以使用取地址符和函数名，获取函数的地址。

请在下方代码框中补充完整你的代码：

```
void idt_init(void) {
  extern void trap_s(void);
  // 向 stvec 寄存器中写入中断处理后跳转函数的地址
  // TODO
  write_csr(stvec, &trap_s);
}
```

3.5.3 编写 `clock.c` 中的相关函数（20%）

我们的时钟中断需要利用 OpenSBI 提供的 `sbi_set_timer()` 接口触发，**向该函数传入一个时刻，OpenSBI 在那个时刻将会触发一次时钟中断。**

我们需要“**每隔若干时间就发生一次时钟中断**”，但是 OpenSBI 提供的接口一次只能设置一个时钟中断事件。本实验采用的方式是：一开始设置一个时钟中断，之后每次发生时钟中断的时候，在相应的中断处理函数中设置下一次的时钟中断。这样就达到了每隔一段时间发生一次时钟中断的目的。

对于代码而言，在文件 `clock.c` 中：

- `clock_init()` 函数将会启用时钟中断并设置第一个时钟中断
- `clock_set_next_event()` 用于调用 OpenSBI 函数 `set_sbi_timer()` 设置下一次的时钟中断时间。
- `get_cycles()` 函数是已经为你提供好的函数。其通过 `rdtime` 伪指令读取一个叫做 `mtime` 的 CSR 寄存器数值，表示 CPU 启动之后经过的真实时间。

修改时钟中断间隔：

QEMU 中外设晶振的频率为 10mhz，即每秒钟 `time` 的值将会增大 10^7。我们可以据此来计算每次 `time` 的增加量，以控制时钟中断的间隔。

编写 `clock_init()`： （10%）

请根据注释在下方代码框中补充完整你的代码：

```
void clock_init(void) {
  puts("ZJU OS LAB       Student_ID:3210102037\n");

  // 对 sie 寄存器中的时钟中断位设置（ sie[stie] = 1 ）以启用时钟中断
  // 设置第一个时钟中断
  // TODO
  set_csr(sie, 1 << 5);
  sbi_call(0, 0, get_cycles(), 0, 0, 0, 0, 0);
  ticks = 0;
}
```

编写 `clock_set_next_event()`： （10%）

> 提示：你需要调用 OpenSBI 提供的接口 `sbi_set_timer()` ，你需要通过 Lab 1 中编写好的 sbi_call 函数调用他。该函数对应的 Function ID 为 0，Extension ID 也为 0，接收一个参数 (arg0)，表示触发时钟中断的时间点。

请根据注释在下方代码框中补充完整你的代码：

```
void clock_set_next_event(void) {
  // 获取当前 cpu cycles 数并计算下一个时钟中断的发生时刻
  // 通过调用 OpenSBI 提供的函数触发时钟中断
  // TODO
  sbi_call(0, 0, get_cycles() + timebase, 0, 0, 0, 0, 0);
  ticks++;
}
```

3.5.5 编写并调用中断处理函数（20%）

在 `entry.S` 中编写中断处理函数：（10%）

在【3.5.3】中，我们向 `stvec` 寄存器存入了中断处理函数的地址，中断发生后将自动进行硬件状态转换，程序将读取 `stvec` 的地址并进行跳转，运行 `trap_s` 函数。该函数需要在栈中保存 `caller saved register` 及 `sepc` 寄存器，读取 `scause` 这个 CSR 寄存器并作为参数传递给 `handle_s` 函数进行中断处理，调用返回后需要恢复寄存器并使用 `sret` 命令回到发生中断的指令。

> **提示**：你可以参考 RISC-V中文手册 3.2 节相关内容完成实验；本实验中寄存器大小为 8 字节；需要使用 CSR 命令操作 CSR 寄存器；若不清楚 `caller saved register`，也可将寄存器全都保存；对汇编语言不是特别了解的建议把中文手册读一遍，或在网上自行学习汇编语言基本的函数调用约定知识。

> 调用 handler_s 函数，如何传参数？ 给 `a0` 寄存器存入 scause 的值即可。如果你不知道为什么 `a0` 寄存器存储的是这个参数的话，请参考 RISC-V中文手册 第 3.2 节。（简单来说，一般规定 a 开头寄存器用来做参数传递，编译的时候也遵守了这个规则）

请根据注释在下方代码框中补充完整你的代码：

```
trap_s:
    # TODO: save the caller saved registers and sepc
    sd ra, 0(sp)
    sd t0, 8(sp)
    sd t1, 16(sp)
    sd t2, 24(sp)
    sd a0, 32(sp)
    sd a1, 40(sp)
    sd a2, 48(sp)
    sd a3, 56(sp)
    sd a4, 64(sp)
    sd a5, 72(sp)
    sd a6, 80(sp)
    sd a7, 88(sp)
    sd t3, 96(sp)
    sd t4, 104(sp)
    sd t5, 112(sp)
    sd t6, 120(sp)
    csrr t0, sepc
    sd t0, 128(sp)
    addi sp, sp, 128

    # TODO: call handler_s(scause)
    csrr a0, scause
    call handler_s
```

```
    # TODO: load sepc and caller saved registers
    ld ra, 0(sp)
    ld t0, 8(sp)
    ld t1, 16(sp)
    ld t2, 24(sp)
    ld a0, 32(sp)
    ld a1, 40(sp)
    ld a2, 48(sp)
    ld a3, 56(sp)
    ld a4, 64(sp)
    ld a5, 72(sp)
    ld a6, 80(sp)
    ld a7, 88(sp)
    ld t3, 96(sp)
    ld t4, 104(sp)
    ld t5, 112(sp)
    ld t6, 120(sp)
    ld t0, 128(sp)
    csrw sepc, t0

    sret
```

为什么需要保存 sepc 寄存器：

答：

sepc寄存器保存了调用trap_s中断处理函数的指令地址，当处于S模式的trap_s函数中断处理结束后，需要通过读取sepc中的值来返回调用trap_s的处于最外层的函数，进而结束S模式的中断处理回到正常运行中。

3.4.2 在 `trap.c` 中编写中断处理函数（10%）

正常情况下，异常处理函数需要根据 `[m|s]cause` 寄存器的值判断异常的种类后分别处理不同类型的异常，但在本次实验中简化为只判断并处理时钟中断。

【3.5.4】中提到，为了实现"定时触发中断"，我们需要在该函数中继续设置下一次的时钟中断。此外，为了进行测试，中断处理函数中还需要打印时钟中断发生的次数，你需要在 `clock.c` 中利用 `ticks` 变量进行统计，请更新【3.5.4】中相关代码，每设置一次时钟中断，便给 `ticks` 变量加一。

本函数的流程如下：

- 判断是否是中断（可能是中断，可能是异常）
- 判断是否是时钟中断 **(注意 C 语言中的运算符优先级，若使用位运算请加括号)**
    - 如果是
        - 设置下一次时钟中断的时间点
        - 打印已经触发过的中断次数

触发时钟中断时， `scause` 寄存器的值是？据此填写代码中的条件判断语句：

答：scause寄存器的值是 `0x8000000000000005`
请根据注释在下方代码框中补充完整你的代码：

```
void handler_s(uint64_t cause) {
  // TODO
  // interrupt
  if (cause >> 63) {
```

```
    // supervisor timer interrupt
    if (((cause << 1) >> 1) == 5) {
      // 设置下一个时钟中断，打印当前的中断数目
      // TODO
      clock_set_next_event();
      puts("Supervisor timer interrupt times = ");
      put_num(ticks);
      puts("\n");
    }
  }
}
```

【同步异常与中断的区别】当处理同步异常时应该在退出前给 `epc` 寄存器+4（一条指令的长度），当处理中断时则不需要，请解释为什么要这样做。（请阅读附录内容）

答:

同步异常是在指令执行期间产生，如访问了无效的存储器地址或执行了具有无效操作码的指令。当同步异常发生时，处理器需要将异常处理程序开始地址设置为`pc`，以便在异常处理完毕后可以从异常指令之后的指令继续执行。为了实现这一点，`epc`寄存器需要存储的是当前`pc`的值，而不是异常指令之前的地址。因此，在退出同步异常处理程序时，需要将`epc`寄存器加上4，以跳过异常指令，进入下一条指令执行.

中断是由外部设备或时钟触发的事件，与当前指令的执行无关。中断处理程序通常需要保存和恢复处理器的状态，但不需要跳回到中断点之后的指令，因为中断是异步事件，不涉及当前指令。因此，在处理中断时，`epc`寄存器保留了触发中断时的地址，不需要加上4，因为中断处理程序不需要跳回到原来的指令。

3.5 编译及测试

请修改 `clock.c` 中的 `ID:123456` 为自己的学号，在项目最外层输入 `make run` 命令调用 Makefile 文件完成整个工程的编译及执行。

如果编译失败，及时使用`make clean`命令清理文件后再重新编译。

默认的 Makefile 为你提供了 `****make debug****` **命令，你可以用此命令以 debug 模式启动程序，此时程序会在入口停下，你可以参照 Lab 0 的方式使用 gdb + target remote :1234 的方式连接并进行调试。**

预期的实验结果如下:

开始时会打印 OSLAB 和 学号，之后每隔一秒触发一次时钟中断，打印一次时钟中断发生的次数。

【注意：由于代码最后有死循环，所以输出完成后整个进程不会自动退出，你需要手动 Ctrl+a，x 来退出 QEMU 模拟器】

请在此附上你的实验结果截图。

```
Make vmlinux Success!

OpenSBI v0.6
   ____                    _____ ____ _____
  / __ \                  / ____|  _ \_   _|
 | |  | |_ __   ___ _ __ | (___ | |_) || |
 | |  | | '_ \ / _ \ '_ \ \___ \|  _ < | |
 | |__| | |_) |  __/ | | |____) | |_) || |_
  \____/| .__/ \___|_| |_|_____/|____/_____|
        | |
        |_|

Platform Name           : QEMU Virt Machine
Platform HART Features  : RV64ACDFIMSU
Platform Max HARTs      : 8
Current Hart            : 0
Firmware Base           : 0x80000000
Firmware Size           : 120 KB
Runtime SBI Version     : 0.2

MIDELEG : 0x0000000000000222
MEDELEG : 0x000000000000b109
PMP0    : 0x0000000080000000-0x000000008001ffff (A)
PMP1    : 0x0000000000000000-0xffffffffffffffff (A,R,W,X)
ZJU OS LAB       Student_ID:3210102037
Supervisor timer interrupt times = 1
Supervisor timer interrupt times = 2
Supervisor timer interrupt times = 3
Supervisor timer interrupt times = 4
Supervisor timer interrupt times = 5
Supervisor timer interrupt times = 6
QEMU: Terminated
root@58414b7bd670:/home/oslab/os_experiment/lab1#
```

4 讨论和心得

请在此处填写实验过程中遇到的问题及相应的解决方式。

这次实验确实遇到了许多问题：

1. 首先是编写Makefile文件时有Error报错，经过检查发现是在调用定义过的变量名时没有按照`$(xxx)`的格式调用，以及在些`all`的依赖文件时漏写了`ASM_OBJ`导致的；
2. 编写字符串打印函数时，第一次写完`make print_only && make run`后发现输出的语句是顺序颠倒的，检查后发现是因为数字在取余时总是先取出最后一位数字，如果按正常顺序只会倒着输出，因此改为倒着放即可；
3. 整个程序写完后，发现时钟中断无法输出，最后经过检查发现，是由于`trap.c`中的`handle_s`函数的第二个`if`判断条件有误，发生中断时`mcause`寄存器的`Exception Code`应该是值为5，而并非是2^5，修改判断条件即可；

实验可能存在不足之处，欢迎同学们对本实验提出建议。

附录

A. Makefile 介绍

Makefile 是一种描述整个工程编译规则的文件，通过自动化编译极大的提高了软件开发效率。Makefile 定义了一系列规则来指定文件之间的编译依赖关系，也可以手动指定每一个依赖如何处理。

简单的说，我们编写的项目往往由多个文件组成，文件和文件之间可能存在依赖关系，比如 code.exe 依赖 hello.c 文件，hello.c 文件依赖 hello.h 文件。Makefile 可以帮助我们书写项目中的依赖关系，并指定当依赖变化的时候，进行什么样的操作。比如 hello.h 文件变化的时候，需要重新编译 hello.c 文件，当 hello.c 文件变化的时候，需要重新编译 code.exe 文件等等。

但是 Makefile 的规则和符号还是比较多的，请阅读 Makefile介绍 来进行详细的学习，你也可以根据工程文件夹里 Makefile 的代码与注释边解释边学习。

B. RISC-V 指令集与异常处理

汇编知识较为繁多复杂，小小的附录承载不完，但在 ZJU 的教学中，大二学习的计组应该教学过，对于没学过的同学，可以查阅一些资料学习基本的汇编知识，**然后请下载并认真阅读 RISC-V中文手册，**掌握基础知识、基本命令及特权架构相关内容。

实验中涉及了特权相关的重要寄存器，在中文手册中进行了简要介绍，其具体布局以及详细介绍可以参考 The RISC-V Instruction Set Manual Volume II: Privileged Architecture Version 1.9.1。

RISC-V 汇编指令

常用的汇编指令包括 `la`、`li`、`j`、`ld` 等，可以自行在 RISC-V中文手册 附录中了解其用法。

RISC-V 指令集中有一类**特殊寄存器 CSR (Control and Status Registers)**，这类寄存器存储了 CPU 的相关信息，只有特定的控制状态寄存器指令 (csrrc、csrrs、csrrw、csrrci、csrrsi、csrrwi等)才能够读写 CSR。例如，保存 `sepc` 的值至内存时需要先使用相应的 CSR 指令将其读入寄存器，再通过寄存器保存该值，写入 sepc 时同理。

```
csrr t0, sepc
sd t0, 0(sp)
```

RISC-V 特权模式

RISC-V 有三个特权模式：U（user）模式、S（supervisor）模式和 M（machine）模式。它通过设置不同的特权级别模式来管理系统资源的使用。其中 M 模式是最高级别，该模式下的操作被认为是安全可信的，主要为对硬件的操作；U 模式是最低级别，该模式主要执行用户程序，操作系统中对应于用户态；S 模式介于 M 模式和 U 模式之间，操作系统中对应于内核态，当用户需要内核资源时，向内核申请，并切换到内核态进行处理。

本实验主要在 S 模式运行，通过调用运行在 M 模式的 OpenSBI 提供的接口操纵硬件。

Level	Encoding	Name	Abbreviation
0	00	User/Application	U
1	01	Supervisor	S
2	10	Reserved	
3	11	Machine	M

RISC-V 中的异常

异常 (trap) 是指是不寻常的运行时事件，由硬件或软件产生，当异常产生时控制权将会转移至异常处理程序。异常是操作系统最基础的概念，一个没有异常的操作系统无法进行正常交互。

RISC-V 将异常分为两类。一类是硬件中断 (interrupt)，它是与指令流异步的外部事件，比如鼠标的单击。另外一类是同步异常 (exception)，这类异常在指令执行期间产生，如访问了无效的存储器地址或执行了具有无效操作码的指令。

这里我们用异常 (trap) 作为硬件中断 (interrupt) 和同步异常 (exception) 的集合，另外 trap 指的是发生硬件中断或者同步异常时控制权转移到 handler 的过程。

> 本实验统一用异常指代 trap，中断/硬件中断指代 interrupt，同步异常指代 exception。

M 模式下的异常

硬件中断的处理（以时钟中断为例）：

简单地来说，中断处理经过了三个流程：中断触发、判断处理还是忽略、可处理时调用处理函数。

- 中断触发：时钟中断的触发条件是 hart（硬件线程）的时间比较器 `mtimecmp` 小于实数计数器 `mtime`。
- 判断是否可处理：
  - 当时钟中断触发时，并不一定会响应中断信号。M 模式只有在全局中断使能位 `mstatus[mie]` 置位时才会产生中断，如果在 S 模式下触发了 M 模式的中断，此时无视 `mstatus[mie]` 直接响应，即运行在低权限模式下，高权限模式的全局中断使能位一直是 enable 状态。
  - 此外，每个中断在控制状态寄存器 `mie` 中都有自己的使能位，对于特定中断来说，需要考虑自己对应的使能位，而控制状态寄存器 `mip` 指示目前待处理的中断。
  - **以时钟中断为例，只有当 `mstatus[mie]` = 1，`mie[mtie]` = 1，且 `mip[mtip]` = 1 时，才可以处理机器的时钟中断。**其中 `mstatus[mie]` 以及 `mie[mtie]` 需要我们自己设置，而 `mip[mtip]` 在中断触发时会被硬件自动置位。
- 调用处理函数：
  - 当满足对应中断的处理条件时，硬件首先会发生一些状态转换，并跳转到对应的异常处理函数中，在异常处理函数中我们可以通过分析异常产生的原因判断具体为哪一种，然后执行对应的处理。
  - 为了处理异常结束后不影响 hart 正常的运行状态，我们首先需要保存当前的状态【即上下文切换】。我们可以先用栈上的一段空间来把**全部**寄存器保存，保存完之后执行到我们编写的异常处理函数主体，结束后退出。

M 模式下的异常相关寄存器：

M 模式异常需要使用的寄存器有提到的 `mstatus`，`mie`，`mtvec` 寄存器，这些寄存器需要我们操作；剩下还有 `mip`，`mepc`，`mcause` 寄存器，这些寄存器在异常发生时**硬件会自动置位**，它们的功能如下：

- `mepc`：存放着中断或者异常发生时的指令地址，当我们的代码没有按照预期运行时，可以查看这个寄存器中存储的地址了解异常处的代码。通常指向异常处理后应该恢复执行的位置。
- `mcause`：存储了异常发生的原因。
- `mstatus`：Machine Status Register，其中 m 代表 M 模式。此寄存器中保持跟踪以及控制 hart (hardware thread) 的运算状态。通过对 `mstatus` 进行位运算，可以实现对不同 bit 位的设置，从而控制不同运算状态。
- `mie`、`mip`：`mie` 以及 `mip` 寄存器是 Machine Interrup Registers，用来保存中断相关的一些信息，通过 `mstatus` 上 mie 以及 mip 位的设置，以及 `mie` 和 `mip` 本身两个寄存器的设置可以实现对硬件中断的控制。注意 mip 位和 `mip` 寄存器并不相同。

- `mtvec`：Machine Trap-Vector Base-Address Register，主要保存M模式下的 trap vector（可理解为中断向量）的设置，包含一个基地址以及一个 mode。

与时钟中断相关的还有 `mtime` 和 `mtimecmp` 寄存器，它们的功能如下：

- `mtime`：Machine Time Register。保存时钟计数，这个值会由硬件自增。
- `mtimecmp`：Machine Time Compare Register。保存需要比较的时钟计数，当 `mtime` 的值大于或等于`mtimecmp` 的值时，触发时钟中断。

需要注意的是，`mtime` 和 `mtimecmp` 寄存器需要用 MMIO 的方式即使用内存访问指令（sd，ld等）的方式交互，可以将它们理解为 M 模式下的一个外设。

事实上，异常还与 `mideleg` 和 `medeleg` 两个寄存器密切相关，它们的功能将在 **S 模式下的异常**部分讲解，主要用于将 M 模式的一些异常处理委托给S模式。

同步异常的处理：

同步异常的触发条件是当前指令执行了未经定义的行为，例如：

- Illegal instruction：跳过判断可以处理还是忽略的步骤，硬件会直接经历一些状态转换，然后跳到对应的异常处理函数。
- 环境调用同步异常 ecall：主要在低权限的 mode 需要高权限的 mode 的相关操作时使用的，比如系统调用时 U-mode call S-mode ，在 S-mode 需要操作某些硬件时 S-mode call M-mode。

需要注意的是，不管是中断还是同步异常，都会经历相似的硬件状态转换，并跳到**同一个**异常处理地址（由`mtvec` / `stvec` 寄存器指定），异常处理函数根据 `mcause` 寄存器的值判断异常出现原因，针对不同的异常进行不同的处理。

S 模式下的异常

由于 hart 位于 S 模式，我们需要在 S 模式下处理异常。这时首先要提到委托（delegation）机制。

委托机制：

*RISC-V 架构所有 mode 的异常在默认情况下都跳转到 M 模式处理**。为了提高性能，RISC-V支持将低权限mode 产生的异常委托给对应 mode 处理，该过程涉及了 `mideleg` 和 `medeleg` 这两个寄存器。

- `mideleg`：Machine Interrupt Delegation。该寄存器控制将哪些中断委托给 S 模式处理，它的结构可以参考 `mip` 寄存器，如 `mideleg[5]` 对应于 S 模式的时钟中断，如果把它置位， S 模式的时钟中断将会移交 S 模式的异常处理程序，而不是 M 模式的异常处理程序。
- `medeleg`：Machine Exception Delegation。该寄存器控制将哪些同步异常委托给对应 mode 处理，它的各个位对应 `mcause` 寄存器的返回值。

S 模式下时钟中断处理流程：

事实上，即使在 `mideleg` 中设置了将 S 模式产生的时钟中断委托给 S 模式，委托仍未完成，因为硬件产生的时钟中断仍会发到 M 模式（`mtime` 寄存器是 M 模式的设备），所以我们需要**手动触发S模式下的时钟中断。**

此前，假设设置好 `[m|s]status` 以及 `[m|s]ie`，即我们已经满足了时钟中断在两种 mode 下触发的使能条件。接下来一个时钟中断的委托流程如下：

1. 当 `mtimecmp` 小于 `mtime` 时，**触发 M 模式时钟中断**，硬件**自动**置位 `mip[mtip]`。
2. 此时 `mstatus[mie]` = 1，`mie[mtie]` = 1，且`mip[mtip]` = 1 表示可以处理 M 模式的时钟中断。
3. 此时 hart 发生了异常，硬件会自动经历状态转换，其中 `pc` 被设置为 `mtvec` 的值，即程序将跳转到我们设置好的 M 模式处理函数入口。
4. （注：`pc` 寄存器是用来存储指向下一条指令的地址，即下一步要执行的指令代码。）

5. M 模式处理函数将分析异常原因，判断为时钟中断，为了将时钟中断委托给 S 模式，于是将 `mip[stip]` 置位，并且为了防止在 S 模式处理时钟中断时继续触发 M 模式时钟中断，于是同时将 `mie[mtie]` 清零。
6. M 模式处理函数处理完成并退出，此时 `sstatus[sie]` = 1，`sie[stie]` = 1，且 `sip[stip]` = 1 (由于 sip 是 mip 的子集，所以第4步中令 `mip[stip]` 置位等同于将 `sip[stip]` 置位)，于是**触发 S 模式的时钟中断。**
7. 此时 hart 发生了异常，硬件自动经历状态转换，其中 `pc` 被设置为stvec，即跳转到我们设置好的 S 模式处理函数入口。
8. S 模式处理函数分析异常原因，判断为时钟中断，于是进行相应的操作，然后利用 `ecall` 触发异常，跳转到 M 模式的异常处理函数进行最后的收尾。
9. M 模式异常处理函数分析异常原因，发现为ecall from S-mode，于是设置 `mtimecmp` += 100000，将`mip[stip]` 清零，表示 S 模式时钟中断处理完毕，并且设置 `mie[mtie]` 恢复M模式的中断使能，保证下一次时钟中断可以触发。
10. 函数逐级返回，整个委托的时钟中断处理完毕。

中断前后硬件的自动转换：

当 `mtime` 寄存器中的的值大于 `mtimecmp` 时，`sip[stip]` 会被置位。此时，如果 `sstatus[sie]` 与 `sie[stie]` 也都是1，硬件会自动经历以下的状态转换（这里只列出 S 模式下的变化）：

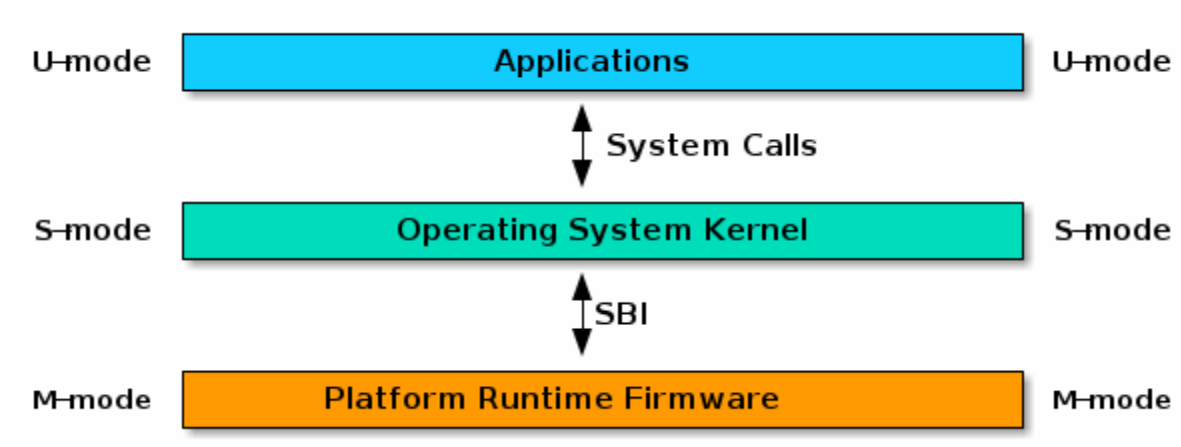
- 发生异常的时 `pc` 的值被存入 `sepc`，且 `pc` 被设置为 `stvec`。
- `scause` 根据异常类型进行设置，`stval` 被设置成出错的地址或者其它特定异常的信息字。
- `sstatus` 中的 SIE 位置零，屏蔽中断，且中断发生前的 `sstatus[sie]` 会被存入 `sstatus[spie]`。
- 发生异常时的权限模式被保存在 `sstatus[spp]`，然后设置当前模式为 S 模式。

在我们处理完中断或异常，并将寄存器现场恢复为之前的状态后，**我们需要用 `sret` 指令回到之前的任务中。**`sret` 指令会做以下事情：

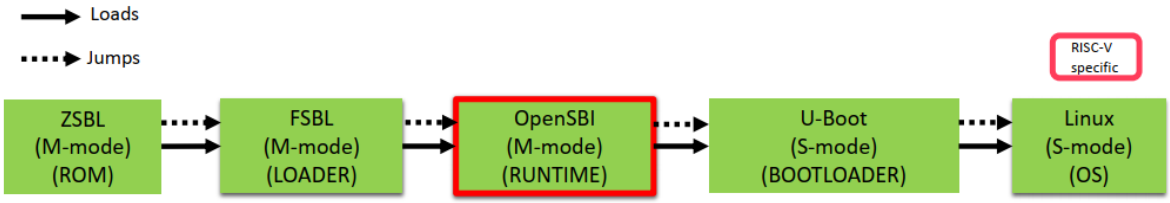
- 将 `pc` 设置为 `sepc`。
- 通过将 `sstatus` 的 SPIE 域复制到 `sstatus[sie]` 来恢复之前的中断使能设置。
- 并将权限模式设置为 `sstatus[spp]`。

C. OpenSBI 介绍

SBI (Supervisor Binary Interface) 是 S-Mode 的 kernel 和 M-Mode 执行环境之间的标准接口，而 OpenSBI 项目的目标是为在 M 模式下执行的平台特定固件提供 RISC-V SBI 规范的开源参考实现。为了使操作系统内核可以适配不同硬件，OpenSBI 提出了一系列规范对 m-mode 下的硬件进行了抽象，运行在 s-mode 下的内核可以按照标准对这些硬件进行操作。



RISC-V启动过程



上图是RISC-V架构计算机的启动过程：

- ZSBL (Zeroth Stage Boot Loader)：片上 ROM 程序，烧录在硬件上，是芯片上电后最先运行的代码。它的作用是加载 FSBL 到指定位置并运行。
- FSBL (First Stage Boot Loader) ：启动 PLLs 和初始化 DDR 内存，对硬件进行初始化，加载下一阶段的bootloader。
- OpenSBI：运行在 m 模式下的一套软件，提供接口给操作系统内核调用，以操作硬件，实现字符输出及时钟设定等工作。OpenSBI 就是一个开源的 RISC-V 虚拟化二进制接口的通用的规范。
- Bootloader：OpenSBI 初始化结束后会通过 mret 指令将系统特权级切换到 s 模式，并跳转到操作系统内核的初始化代码。这一阶段，将会完成中断地址设置等一系列操作。之后便进入了操作系统。

更多内容可参考 An Introduction to RISC-V Boot Flow。

为简化实验，从 ZSBL 到 OpenSBI 运行这一阶段的工作已通过 QEMU 模拟器完成。运行 QEMU 时，我们使用 -bios default 选项将 OpenSBI 代码加载到 0x80000000 起始处。OpenSBI 初始化完成后，会跳转到0x80200000 处。因此，我们所编译的代码需要放到 0x80200000 处。

D. 内联汇编

我们在用 C 语言写代码的时候会遇到直接对寄存器进行操作的情况，这时候就需要用的内联汇编了。内联汇编的详细使用方法可参考 **GCC内联汇编**。

在此给出一个简单示例。其中，三个`:`将汇编部分分成了四部分。

第一部分是汇编指令，指令末尾需要添加'\n'。指令中以 `%[name]` 形式与输入输出操作数中的同名操作符 `[name]` 绑定，也可以通过 `%数字` 的方式进行隐含指定。假设输出操作数列表中有1个操作数，“输入操作数”列表中有2个操作数，则汇编指令中 `%0` 表示第一个输出操作数，`%1` 表示第一个输入操作数，`%2` 表示第二个输入操作数。

第二部分是输出操作数，第三部分是输入操作数。输入或者输出操作符遵循 `[name] "CONSTRAINT" (variable)` 格式，由三部分组成：

- `[name]`：符号名：就是汇编里面如何使用这个操作数，相当于给这个操作数起了一个名字。用的时候就 %[name] 就可以使用。
- `"constraint"`：限制字符串，用于约束此操作数变量的属性。字母 `r` 代表使用编译器自动分配的寄存器来存储该操作数变量，字母 `m` 代表使用内存地址来存储该操作数变量，字母 `i` 代表立即数。
- 对于输出操作数而言，等号"="代表输出变量用作输出，原来的值会被新值替换，而且是只写的；加号"+"代表输出变量不仅作为输出，还作为输入。此约束不适用于输入操作数。
- `(variable)`：C/C++ 变量名或者表达式。

示例中，输出操作符 `[ret_val] "=r" (ret_val)` 代表将汇编指令中 `%[ret_val]` 的值更新到变量 ret_val 中，输入操作符 `[type] "r" (type)` 代表着将 `()` 中的变量 `type` 放入寄存器中。

第四部分是可能影响的寄存器或存储器，用于告知编译器当前内联汇编语句可能会对某些寄存器或内存进行修改，使得编译器在优化时将其因素考虑进去。

```
unsigned long long s_example(unsigned long long type,unsigned long long arg0) {
    unsigned long long ret_val;
    __asm__ volatile (
        #汇编指令
        "mv x10, %[type]\n"
        "mv x11,%[arg0]\n"
        "mv %[ret_val], x12"
        #输出操作数
        : [ret_val] "=r" (ret_val)
        #输入操作数
        : [type] "r" (type), [arg0] "r" (arg0)
        #可能影响的寄存器或存储器
        : "memory", "x10", "x11", "x12"
    );
    return ret_val;
}
```

示例二定义了一个宏，其中 `%0` 代表着输入输出部分的第一个符号，即 `val`。其中 `#reg` 是 C 语言的一个特殊宏定义语法，相当于将 `reg` 进行宏替换并用双引号包裹起来。例如 `write_csr(sstatus,val)` 宏展开会得到：
`({asm volatile ("csrw " "sstatus" ", %0" :: "r"(val)); })`

```
#define write_csr(reg, val) ({
    asm volatile ("csrw " #reg ", %0" :: "r"(val)); })
```

E. Linux Basic

vmlinux

vmlinux 通常指 Linux Kernel 编译出的可执行文件（Executable and Linkable Format， ELF），特点是未压缩的，带调试信息和符号表的。在本实验中，vmlinux 通常指将你的代码进行编译，链接后生成的可供 QEMU 运行的 RISC-V 64-bit 架构程序。如果对 vmlinux 使用 `file` 命令，你将看到如下信息：

```
$ file vmlinux
vmlinux: ELF 64-bit LSB executable, UCB RISC-V, version 1 (SYSV), statically
linked, not stripped
```

System.map

System.map 是内核符号表（Kernel Symbol Table）文件，是存储了所有内核符号及其地址的一个列表。使用System.map 可以方便地读出函数或变量的地址，为 Debug 提供了方便。"符号"通常指的是函数名，全局变量名等等。使用 `nm vmlinux` 命令即可打印vmlinux的符号表，符号表的样例如下：

```
0000000080200000 A BASE_ADDR
0000000080208306 R _end
0000000080200000 T _start
0000000080200306 R bss_end
0000000080200306 R bss_start
0000000080200010 T main
000000008020019c T put_num
0000000080200128 T puts
0000000080200054 T sbi_call
0000000080208306 R stack_top
0000000080200110 T test
```

vmlinux.lds

GNU ld 即链接器，用于将 *.o 文件（和库文件）链接成可执行文件。在操作系统开发中，为了指定程序的内存布局，ld 使用链接脚本（Linker Script）来控制，在 Linux Kernel 中链接脚本被命名为 vmlinux.lds。更多关于ld 的介绍可以使用 `man ld` 命令查看。

下面给出一个 vmlinux.lds 的例子：

```
/* 目标架构 */
OUTPUT_ARCH( "riscv" )
/* 程序入口 */
ENTRY( _start )
/* 程序起始地址 */
BASE_ADDR = 0x80000000;
SECTIONS
{
  /* . 代表当前地址 */
  . = BASE_ADDR;
  /* code 段 */
  .text : { *(.text) }
  /* data 段 */
  .rodata : { *(.rodata) }
  .data : { *(.data) }
  .bss : { *(.bss) }
  . += 0x8000;
  /* 栈顶 */
  stack_top = .;
  /* 程序结束地址 */
  _end = .;
}
```

首先我们使用 OUTPUT_ARCH 指定了架构为RISC-V，之后使用 ENTRY 指定程序入口点为 `_start` 函数，程序入口点即程序启动时运行的函数，经过这样的指定后在 head.S 中需要编写 `_start` 函数，程序才能正常运行。

链接脚本中有 `.` `*`两个重要的符号。单独的 `.` 在链接脚本代表当前地址，它有赋值、被赋值、自增等操作。而`*` 有两种用法，其一是 `*()` 在大括号中表示将所有文件中符合括号内要求的段放置在当前位置，其二是作为通配符。

链接脚本的主体是 SECTIONS 部分，在这里链接脚本的工作是将程序的各个段按顺序放在各个地址上，在例子中就是从 0x80000000 地址开始放置了.text，.rodata，.data 和 .bss 段。各个段的作用可以简要概括成：

段名	主要作用
.text	通常存放程序执行代码
.rodata	通常存放常量等只读数据
.data	通常存放已初始化的全局变量、静态变量
.bss	通常存放未初始化的全局变量、静态变量

在链接脚本中可以自定义符号，例如 stack_top 与 _end 都是我们自己定义的，其中 stack_top 与程序调用栈有关。

更多有关链接脚本语法可以参考这里。

Image

在 Linux Kernel 开发中，为了对 vmlinux 进行精简，通常使用 objcopy 丢弃调试信息与符号表并生成二进制文件，这就是 Image。Lab0 中 QEMU 正是使用了 Image 而不是 vmlinux 运行。

```
$ objcopy -O binary vmlinux Image --strip-all
```

此时再对 Image 使用 `file` 命令时：

```
$ file Image
image: data
```