lab3

李佳：2312668　朱晨瑞：2312674　马溴怡：2311061

练习1：完善中断处理

1. 时钟初始化函数 `clock_init()`

在 `kern/driver/clock.c` 中完成：

```
void clock_init(void) {
  // 开启时钟中断
  set_csr(sie, MIP_STIP);
  // 在 S 模式下打开全局中断开关
  set_csr(sstatus, SSTATUS_SIE);
  // 对于 QEMU，timebase 取 sbi_timebase()/100
  timebase = sbi_timebase() / 100;
  // 设置下次时钟中断
  clock_set_next_event();
  // 初始化计数器
  ticks = 0;
  cprintf("++ setup timer interrupts\n");
}
```

注：实际编译运行后发现打印 "100 ticks" 较快，要实现大约每一秒打印一个 "100 ticks"，需要设置 `timebase = sbi_timebase() / 100;` 为 `timebase = sbi_timebase() / 10;`

2. 中断处理函数 `interrupt_handler()`

在 `trap.c` 中添加 `IRQ_S_TIMER` 的中断处理：

```
case IRQ_S_TIMER:
    clock_set_next_event(); //调用 sbi_set_timer() 再次设置下次中断时间，实现周期性触发
    {
        static int tick_count = 0; // 到达的时钟中断计数（0 .. TICK_NUM-1）
        static int print_num = 0;  // 已经打印 "100 ticks" 的次数

        //每次时钟中断到来就递增计数
        tick_count++;

        // 到达 TICK_NUM（100）次时触发打印并重置计数
        if (tick_count >= TICK_NUM) {
            tick_count = 0;  //重新计数//
            print_ticks();   //打印 "100 ticks"//
            print_num++;
            //打印达到 10 次后调用关机//
            if (print_num >= 10) {
                sbi_shutdown();
            }
        }
    }
    break;
```

说明：

- 每次中断执行一次 `clock_set_next_event()`，保证时钟持续触发；
- 每 100 次中断打印一次 `"100 ticks"`；
- 打印 10 次后自动调用 `sbi_shutdown()` 关机。

3. SBI 接口函数`sbi_timebase()`

由于原始的 `sbi.c` 文件中没有提供 `sbi_timebase()` 的实现，编译会出现链接错误：

```
undefined reference to `sbi_timebase'
```

在 `libs/sbi.c` 中增加下面函数

```
/*
 返回当前机器时间计数器 (time CSR) 的值。
 在 RV64 平台上，time CSR 是 XLEN 宽度（64-bit），
 因此使用 unsigned long 与寄存器宽度一致。
*/
unsigned long sbi_timebase(void) {
    unsigned long result;
    __asm__ volatile("csrr %0, time" : "=r"(result));
    return result;
}
```

说明：

- `time` 是 RISC-V 的 CSR（Control and Status Register），用于记录时间计数器。
- `csrr` 指令读取 CSR 内容至寄存器。
- `asm volatile` 告诉编译器该指令不可优化。
- 此函数返回当前机器时间（通常以 CPU 周期为单位）。

该函数的返回值用于设定下次定时器触发时间，从而控制中断周期。

4.定时器中断处理流程

step1 时钟初始化

由kern/driver/clock.c 下的 clock_init() 实现：

- `sbi_timebase()`：通过读取 `time` CSR 寄存器获取当前时钟计数。
- `sbi_set_timer()`：通过 SBI 调用设置下一次定时器中断触发时间。
- `set_csr(sstatus, SSTATUS_SIE)`：开启 **S 模式全局中断使能**，允许内核响应中断。
- 输出 `"++ setup timer interrupts"` 表示时钟初始化完成。

step2 中断触发

当硬件计数寄存器 `mtime` 的值达到 `mtimecmp`（由 `sbi_set_timer()` 设定的值）时：

- 硬件向内核发出 STI （由时钟触发、发给运行在 S 模式的内核的中断信号。）

- CPU 保存当前状态，进入中断入口

此过程由硬件和 OpenSBI 自动完成，OS无需干预。

补充：RISC-V 定义了不同的中断类型

中断类型	缩写	触发来源	目标特权级
Machine Timer Interrupt	MTI	硬件计时器	M 模式
Supervisor Timer Interrupt	STI	硬件计时器（经 OpenSBI 转发）	S 模式
Machine External Interrupt	MEI	外设输入	M 模式
Supervisor External Interrupt	SEI	外设输入（经 OpenSBI 转发）	S 模式
Software Interrupt	SSI	软件触发	S模式

step3 内核陷入

由kern/trap/trap.c 下的 trap() 实现：

```
void trap(struct trapframe *tf) {
    if ((tf->cause & SCAUSE_IRQ_FLAG) && ((tf->cause & SCAUSE_CODE_MASK) ==
IRQ_S_TIMER)) {
        interrupt_handler(tf);
    } else {
        exception_handler(tf);
    }
}
```

中断发生时，CPU 跳转至 `trap()` 函数。`trap()` 根据 `tf->cause` 判断是否为时钟中断。若是，则调用 `interrupt_handler()`。

step4 中断处理

由kern/trap/trap.c 下的 interrupt_handler()实现：

- `clock_set_next_event()`
  调用 `sbi_set_timer()` 再次设置下次中断时间，实现周期性触发。

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

- `ticks` 全局变量，用于记录系统节拍数，每发生一次时钟中断自增一次。
- `printk("%d ticks\n", ticks)` 每累计 100 次打印一次，形成控制台输出的 “100 ticks”

5.编译运行

执行`make`后编译通过，执行`make qemu`

```
PMP0: 0x0000000080000000-0x000000008001ffff (A)
PMP1: 0x0000000000000000-0xffffffffffffffff (A,R,W,X)
DTB Init
HartID: 0
DTB Address: 0x82200000
Physical Memory from DTB:
  Base: 0x0000000080000000
  Size: 0x0000000008000000 (128 MB)
  End:  0x0000000087ffffff
DTB init completed
(THU.CST) os is loading ...
Special kernel symbols:
  entry  0xffffffffc0200054 (virtual)
  etext  0xffffffffc0201fae (virtual)
  edata  0xffffffffc0207030 (virtual)
  end    0xffffffffc02074a8 (virtual)
Kernel executable memory footprint: 30KB
memory management: default_pmm_manager
physcial memory map:
  memory: 0x0000000008000000, [0x0000000080000000, 0x0000000087ffffff].
check_alloc_page() succeeded!
satp virtual address: 0xffffffffc0206000
satp physical address: 0x0000000080206000
++ setup timer interrupts
100 ticks
100 ticks
100 ticks
100 ticks
100 ticks
100 ticks
100 ticks
100 ticks
100 ticks
100 ticks
```

对内存进行分析

```
PMP0: 0x0000000080000000-0x000000008001ffff (A)
PMP1: 0x0000000000000000-0xffffffffffffffff (A,R,W,X)
DTB Init
HartID: 0
DTB Address: 0x82200000
Physical Memory from DTB:
Base: 0x0000000080000000
Size: 0x0000000008000000 (128 MB)
End: 0x0000000087ffffff
DTB init completed
```

对应内存示意图如下

```
物理地址空间 (64-bit)

┌──────────────────────────────┐ 0x87FFFFFF
|        可用 DRAM (128MB)     | ← ucore 可使用
|    [0x80020000, 0x87FFFFFF]|
├──────────────────────────────┤
|  OpenSBI 固件区 (128KB)      | ← 受 PMP0 保护
|    [0x80000000, 0x8001FFFF]|
├──────────────────────────────┤
|  低地址区（I/O, ROM等）        | ← 一般未映射
|    [0x00000000, 0x7FFFFFFF]|
└──────────────────────────────┘ 0x0
```

系统启动后打印如下信息，对其进行分析

```
Special kernel symbols:
entry 0xffffffffc0200054 (virtual)
etext 0xffffffffc0201fae (virtual)
edata 0xffffffffc0207030 (virtual)
end 0xffffffffc02074a8 (virtual)
Kernel executable memory footprint: 30KB
```

符号名	虚拟地址（virtual）	含义	作用说明
entry	`0xffffffffc0200054`	内核入口地址	内核从此处开始执行，即 `kern_entry` 所在位置，通常对应启动汇编（entry.S）的第一条有效指令。
etext	`0xffffffffc0201fae`	代码段结束地址	表示内核代码段（.text）的结束位置，位于最后一条函数指令之后。该段内存区域为**只读可执行。**
edata	`0xffffffffc0207030`	数据段结束地址	表示内核已初始化数据段（.data）的结束位置。位于全局变量、静态变量等初始化数据之后。
end	`0xffffffffc02074a8`	内核镜像结束地址	表示整个内核映像文件的结束位置，含代码、数据、BSS等所有段。内核启动时，内存分配将从此地址之后开始。
footprint	end - entry 30KB	内核镜像总大小	end - entry

内核内存管理初始化阶段打印的信息，对其进行分析

```
memory management: default_pmm_manager  //表示系统正在使用名为 default_pmm_manager 的物理页管理模块
physcial memory map:
memory: 0x0000000008000000,
[0x0000000080000000, 0x0000000087ffffff]. //说明可用物理内存从 0x80000000 开始，大小为 0x08000000 = 128MB。这与设备树中声明的内存范围一致。内核加载和页管理均基于该物理区域。
check_alloc_page() succeeded!
satp virtual address: 0xffffffffc0206000  //这是根页表的虚拟地址
satp physical address: 0x0000000080206000 //根页表的物理地址
```

扩展练习一：描述与理解中断流程

1.中断异常处理流程

（1）异常触发

当 CPU 执行过程中发生中断（如时钟中断）或异常（如非法指令）时，硬件自动完成以下操作：

- 确定异常原因（存储在scause寄存器）
- 记录异常发生的地址（存储在sepc寄存器，即被打断的指令地址）
- 根据stvec寄存器（异常向量表基地址）跳转到预设的异常入口（即_alltraps)

（2）保存上下文

进入__alltraps后，首先执行SAVE_ALL宏，将当前 CPU 状态（寄存器、状态信息）保存到栈中，形成struct trapframe结构。具体包括：所有通用寄存器、状态寄存器（sstatus）、异常地址寄存器（sepc）、错误地址寄存器（sbadaddr）、原因寄存器（scause）。

（3）调用中断处理函数

通过mov a0, sp将栈指针（此时指向struct trapframe的起始地址）作为参数传递给trap函数，进入处理逻辑。

（4）中断/异常分发

trap函数调用trap_dispatch，根据scause寄存器的值区分中断（cause < 0）和异常（cause >= 0），分别调用interrupt_handler或exception_handler。

（5）具体处理

若为时钟中断（IRQ_S_TIMER），更新计数器，每 100 次打印100 ticks，累计 10 次后关机；若为其他中断 / 异常，执行对应处理（如非法指令异常打印信息并跳过错误指令）。

（6）恢复上下文与返回

处理完成后，执行RESTORE_ALL宏从栈中恢复所有寄存器和状态信息，最后通过sret指令返回到sepc记录的地址，继续执行被打断的程序。

2.mov a0，sp

函数调用的第一个参数通过寄存器`a0`传递。`trap`函数的原型为`void trap(struct trapframe *tf)`，需要接收一个指向`struct trapframe`的指针作为参数。`SAVE_ALL`宏执行后，栈指针`sp`恰好指向栈中保存的`struct trapframe`结构体的起始地址（该结构体包含所有寄存器和状态信息）。因此，`mov a0, sp`的作用是将`struct trapframe`的地址通过`a0`寄存器传递给`trap`函数，使`trap`函数能访问并操作中断 / 异常发生时的上下文信息。

3.SAVE_ALL中寄存器保存在栈中的位置的确定依据

寄存器在栈中的存储位置由`struct trapframe`的结构体定义决定。`struct trapframe`的布局需与`SAVE_ALL`的存储顺序严格对应：

- 前 32 个位置存储通用寄存器（`x0-x31`），对应`struct pushregs`成员；
- 后续位置依次存储`status`（`sstatus`）、`epc`（`sepc`）、`badvaddr`（`sbadaddr`）、`cause`（`scause`）。

这种对应关系确保 代码能通过`tf`指针正确访问栈中保存的寄存器和状态信息。

4.对于任何中断，__alltraps中是否需要保存所有寄存器?

需要保存所有寄存器。

中断 / 异常是异步发生的，中断处理程序会修改寄存器的值。若不保存所有寄存器，处理完成后返回原程序时，原程序依赖的寄存器状态已被破坏（如临时变量、循环计数器等），将导致程序执行错误。

因此，为了保证中断处理完成后原程序能准确恢复执行，`__alltraps`必须通过`SAVE_ALL`保存所有通用寄存器及状态寄存器（`sstatus`、`sepc`等），并在返回前通过`RESTORE_ALL`完整恢复。

扩展练习二：理解上下文切换机制

1.指令操作与目的

`csrw sscratch,sp`: 把用户栈指针（进入异常瞬间的 sp）写入sscratch，为后续在异常处理程序里找回用户栈提供保险。

`csrrw s0, sscratch, x0`: 把 sscratch 清零并同时把旧值（用户 sp）读回 s0。 拿到用户 sp，方便 SAVE_ALL 把它压栈；把 sscratch 清零，给嵌套异常打标记，若再次异常时 sscratch==0，内核就知道“已经在内核态”，避免重复拿错 sp。

2.保存不恢复的原因

（1）stval、scause等 CSR 是陷阱发生时的关键状态信息：

- `scause`：记录陷阱类型（如中断、页错误、非法指令等），是操作系统判断如何处理陷阱的核心依据。
- `stval`：记录导致陷阱的错误地址（如页错误时的访问地址），用于定位问题（如填充页表、修复访问）。

（2）恢复上下文的目的是让程序从陷阱处正确继续执行，而stval、scause等 CSR 的作用仅限于本次陷阱处理：

- 它们是“一次性”信息，处理完成后无需保留（下次陷阱会产生新的信息）。

- 恢复时仅需还原sstatus（状态寄存器，如中断使能位）和sepc（陷阱发生的指令地址，确保sret正确返回），其他 CSR 不影响程序后续执行，因此无需还原。

（3）意义在于：

- 内核日志打印 “SegFault at 0x..., cause=0xd” 需要这两个值。
- POSIX 信号（如 SIGSEGV）需要把出错地址、原因打包给用户态。
- gdb 可以通过 /proc//siginfo 拿到当时的 scause/stval。
- 若在处理异常时又触发异常，内核可对比前后两次 scause/stval 判断是否为“双重故障”。

扩展练习三：完善异常中断

1.实现步骤

（1）在init.c中新增`test_exceptions()`函数，通过内联汇编主动触发两种异常，具体逻辑如下：

- **触发非法指令异常**：使用`.word 0x00000000`定义无效指令，后续添加`nop`防止指令流水重叠导致的执行异常。
- **触发断点异常**：使用 RISC-V 架构专用断点指令`ebreak`，该指令会直接触发 CAUSE_BREAKPOINT 类型异常，同样用`nop`保证指令完整性。
- 函数调用时机：在`kern_init()`中完成时钟、中断、内存管理初始化后，调用`test_exceptions()`执行异常测试。

```
void test_exceptions(void){
    cprintf("\nstarting tests\n\n");

    //触发非法指令异常
    cprintf("test CAUSE_ILLEGAL_INSTRUCTION\n");
    __asm__ volatile (
        ".word 0x00000000 \n"
        "nop                "
    );
    cprintf("illegal instruction test completed\n\n");

    //触发断点异常
    cprintf("test CAUSE_BREAKPOINT\n");
    __asm__ volatile (
        "ebreak \n"
        "nop      "
    );
    cprintf("breakpoint test completed\n\n");
}
```

（2）异常处理函数完善

在trap.c的`exception_handler()`函数中添加两种异常的分支处理，读取异常上下文信息并输出指定格式。

```
case CAUSE_ILLEGAL_INSTRUCTION:
            /* 扩展练习 Challenge3 输出 */
            cprintf("Illegal instruction caught at 0x%016lx\n", tf->epc);
            cprintf("Exception type:Illegal instruction\n");
            tf->epc += 4;                   // 跳过 4 字节非法指令
            break;

case CAUSE_BREAKPOINT:
            /* 扩展练习 Challenge3 输出 */
            cprintf("ebreak caught at 0x%016lx\n", tf->epc);
            cprintf("Exception type: breakpoint\n");
            tf->epc += 4;                   // 跳过 ebreak
            break;
```

2. 异常处理流程分析

（1）**异常触发**：`test_exceptions()` 中通过内联汇编执行非法指令（0x00000000）或断点指令（ebreak），硬件自动检测异常。

（2）**上下文保存**：CPU 将当前寄存器状态、异常原因（scause）、触发地址（sepc）等保存到 `struct trapframe`，并跳转到异常入口 `__alltraps`。

（3）**异常分发**：`trap()` 函数根据 `tf->cause` 判断为异常（cause >= 0），调用 `exception_handler()`。

（4）**具体处理**：`exception_handler()` 根据异常类型分支执行，输出指定信息并调整 `tf->epc` 跳过异常指令。

（5）**恢复执行**：通过 `RESTORE_ALL` 宏恢复寄存器状态，`sret` 指令返回到 `tf->epc` 指向的后续指令（nop），程序继续执行。

3.运行结果验证

输入make进行编译，再输入make qemu运行。

```
starting tests

test CAUSE_ILLEGAL_INSTRUCTION
Illegal instruction caught at 0xffffffffc02000b4
Exception type:Illegal instruction
illegal instruction test completed

test CAUSE_BREAKPOINT
ebreak caught at 0xffffffffc02000d2
Exception type: breakpoint
breakpoint test completed

100 ticks
100 ticks
100 ticks
100 ticks
100 ticks
100 ticks
100 ticks
100 ticks
100 ticks
100 ticks
ubuntu@Ubuntu-lj:~/Desktop/lab3/lab3$
```

知识点

一. 实验中重要知识点

实验中的重要知识点	对应的 OS 原理知识点	理解
时钟中断处理机制（`clock_init()` 初始化、`interrupt_handler()` 处理时钟中断、`clock_set_next_event()` 设置周期性中断）	操作系统中断机制（中断响应、中断服务程序、中断优先级、中断嵌套）	**含义**：实验中通过初始化时钟中断使能、设置中断周期、累计中断次数并触发打印 / 关机，实现了周期性时钟中断的完整流程；原理中中断机制是 OS 响应外部事件（如时钟、外设）的核心，包括中断请求、CPU 响应、现场保护、中断处理、现场恢复等抽象流程。**关系**：实验是原理在 RISC-V 架构下的具体实现，时钟中断是原理中 "中断" 的典型实例（用于时间片调度、定时器等）。**差异**：实验聚焦于时钟中断这一特定类型，依赖 RISC-V 的 CSR 寄存器（如 `sie`、`sstatus`）和 SBI 接口；原理中的中断机制是通用概念，适用于所有硬件架构，涵盖更多中断类型。

实验中的重要知识点	对应的 OS 原理知识点	理解
异常处理（非法指令、断点异常的触发与处理，`exception_handler()` 分支逻辑，`tf->epc` 调整）	操作系统异常处理机制（故障、陷阱、终止，异常向量表）	**含义**：实验中通过内联汇编触发非法指令和断点异常，在处理函数中输出信息并调整 `epc` 跳过异常指令，实现了异常的捕获与恢复；原理中异常处理是 OS 应对程序执行错误的机制，需区分异常类型并采取对应策略。 **关系**：实验是原理中 "陷阱"（如断点 `ebreak`）和 "故障"（如非法指令）处理的具体实现，验证了异常处理的核心流程（触发→保存上下文→处理→恢复执行）。**差异**：实验仅处理两种简单异常，且通过调整 `epc` 直接恢复执行；原理中异常类型更多（如页错误、算术溢出），处理更复杂（如页错误需动态分配内存），部分异常无法恢复（如终止类异常）。
上下文保存与恢复（`SAVE_ALL` / `RESTORE_ALL` 宏、`struct trapframe` 结构、`sscratch` 寄存器使用）	进程上下文切换（现场保护与恢复）	**含义**：实验中通过宏指令保存 / 恢复所有通用寄存器和状态寄存器（`sstatus`、`sepc` 等），确保中断 / 异常处理后程序能继续执行；原理中上下文切换是 OS 在调度进程时，保存当前进程状态、加载目标进程状态的过程，是多任务并发的基础。 **关系**：实验中的上下文保存是原理中 "现场保护" 的子集，聚焦于中断 / 异常场景下的临时状态保存；原理中的上下文切换更通用，涵盖进程切换、模式切换（用户态→内核态）等场景。 **差异**：实验中上下文保存在栈上，依赖硬件寄存器辅助；原理中上下文切换可保存在内存（如进程控制块 PCB），且需处理更多状态。
SBI 接口使用（`sbi_timebase()` 读取时间寄存器、`sbi_set_timer()` 设置定时器、`sbi_shutdown()` 关机）	操作系统与硬件交互接口（特权指令、系统调用、固件接口）	**含义**：实验中通过 SBI实现与底层硬件的交互；原理中 OS 需通过硬件接口（如特权指令、固件调用）操作硬件，避免用户程序直接访问硬件。 **关系**：SBI 是原理中 "硬件交互接口" 在 RISC-V 架构下的具体实现，类似 x86 的 BIOS 或 UEFI，提供了 OS 与硬件之间的抽象层。 **差异**：SBI 是 RISC-V 特有的分层接口（用户态→内核态→SBI→硬件），而原理中的硬件接口是通用概念，不同架构实现不同。

实验中的重要知识点	对应的 OS 原理知识点	理解
内存管理初始化（物理内存映射、根页表地址`satp`、内存布局）	物理内存管理与虚拟内存机制	**含义**：实验中展示了物理内存范围（128MB）、内核镜像布局（`entry`/`etext`等符号）、根页表的物理 / 虚拟地址；原理中内存管理负责物理内存分配、虚拟地址到物理地址的转换，是 OS 管理内存资源的核心。 **关系**：实验中的内存布局和页表设置是原理中 "物理内存管理" 和 "分页机制" 的基础实现，验证了内存地址空间的划分与映射。 **差异**：实验仅展示内存布局和根页表地址，未涉及内存分配算法（如伙伴系统）、虚拟内存的页置换（如 LRU）等原理中关键机制；原理中的内存管理更强调动态分配与高效利用。

二、OS 原理中重要但实验未涉及的知识点

1. **文件系统**：文件系统负责管理外存数据的组织与访问，是 OS 与用户交互的重要接口。实验未涉及任何文件操作或外存管理。
2. **设备管理（除时钟外）**：设备管理需处理各类外设（如磁盘、网卡、键盘）的中断与驱动，包括设备独立性、缓冲技术等。实验仅涉及时钟设备，未涉及其他外设的驱动与中断处理。
3. **死锁处理**：死锁是多进程竞争资源时可能出现的问题，涉及死锁预防、避免、检测与恢复。实验无多进程 / 资源竞争场景，因此未涉及。
4. **用户态与内核态的权限隔离**：用户态程序不可直接访问内核资源，需通过系统调用陷入内核态。实验中未涉及用户态程序，所有代码运行在内核态，未体现权限隔离机制。