操作系统lab3实验报告

小组成员：

陈秋彤（2311815）徐盈蕊（2311344）杨欣瑞（2312246）

练习1：完善中断处理 （需要编程）

我们对 interrupt_handler 函数中的 IRQ_S_TIMER 情况进行了实现，当时钟中断发生时，程序会进入这个case分支执行以下操作：

首先，立即调用 `clock_set_next_event()` 函数，这个函数的作用是设置下一次时钟中断的触发时间，确保在未来中断能够周期性地发生。如果不进行这个调用，中断将只会发生一次。我们可以查看在 `clock.c` 中有关这个函数的具体实现，函数首先获取当前时钟周期数，然后加上一个固定间隔，从而确定下一次中断的时刻。

设置好下次中断后，开始处理本次中断的逻辑：

```
ticks++;
if (ticks % TICK_NUM == 0) {
    print_ticks();
    print_times++;
}
if (print_times >= 10) {
    extern void sbi_shutdown(void);
    sbi_shutdown();
}
```

这里使用了两个全局变量 `ticks` 和 `print_times` 来维护系统的计时状态。

`ticks` 变量是一个外部引入的全局变量，来自 `clock.c` 文件，它是一个基础计数器，负责对时钟中断进行精确计数。每次进入时钟中断处理程序时，都会执行 `ticks++` 操作，记录系统自启动以来发生的时钟中断次数。接下来，只需要判断 `ticks % TICK_NUM == 0` 是否为真，就能确定是否达到了100次中断，如果到达，就需要触发打印操作。

`print_times` 是我们自己声明的一个全局变量，用于记录打印 `100 ticks` 的次数。每当完成100次中断的计数后，这个变量就会加1。通过检查 `print_times >= 10` ，就够控制系统在打印10次 `100 ticks` 后通过调用 `sbi_shutdown()` 函数来关闭系统。

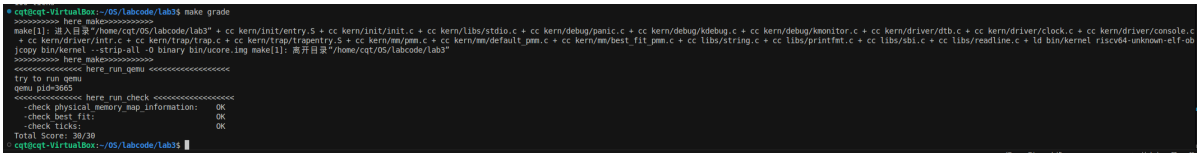
测试结果

```
cqt@cqt-VirtualBox:~/OS/labcode/lab3$ make qemu

OpenSBI v0.4 (Jul  2 2019 11:53:53)
   ____                    ____  ____  _____
  / __ \                  / ___||  _ \|_   _|
 | |  | |_ __   ___ _ __ | (___ | |_) || |
 | |  | | '_ \ / _ \ '_ \ \___ \|  _ < | |
 | |__| | |_) |  __/ | | |____) | |_) || |_
  \____/| .__/ \___|_| |_|_____/|____/_____|
        | |
        |_|

Platform Name          : QEMU Virt Machine
Platform HART Features : RV64ACDFIMSU
Platform Max HARTs     : 8
Current Hart           : 0
Firmware Base          : 0x80000000
Firmware Size          : 112 KB
Runtime SBI Version    : 0.1

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  0xffffffffc0201ee2 (virtual)
  edata  0xffffffffc0206028 (virtual)
  end    0xffffffffc02064a0 (virtual)
Kernel executable memory footprint: 26KB
memory management: best_fit_pmm_manager
physcial memory map:
  memory: 0x0000000008000000, [0x0000000080000000, 0x0000000087ffffff].
check_alloc_page() succeeded!
satp virtual address: 0xffffffffc0205000
satp physical address: 0x0000000080205000
++ setup timer interrupts
100 ticks
100 ticks
100 ticks
100 ticks
100 ticks
100 ticks
100 ticks
100 ticks
100 ticks
100 ticks
100 ticks
cqt@cqt-VirtualBox:~/OS/labcode/lab3$
```



扩展练习 Challenge1：描述与理解中断流程

问题一：处理中断异常的流程?

这里以U模式陷入到S模式为例，运行在U模式的程序通过ecall指令会触发中断，定时器也会在指定时间出发时钟中断，程序异常也会导致系统陷入内核态，此时，硬件需要保存相关信息，sepc保存被中断指令的虚拟地址或者异常发生的指令地址，scause保存中断或者异常的具体原因，stavl提供与异常相关的附加信息，紧接着，保存并修改中断使能状态。将当前的中断使能状态sstatus.SIE保存到sstatus.SPIE中，并且会将sstatus.SIE清零，从而禁用 S 模式下的中断。这是为了保证在处理中断时不会被其他中断打断。然后，保存当前的特权级信息。将当前特权级（即 U 模式，值为 0）保存到sstatus.SPP中，并将当前特权级切换到 S 模式。此时，系统已经进入 S 模式，准备跳转到中断处理程序。将pc设置为stvec寄存器中的值，并跳转到中断处理程序的入口。

进入到中断处理程序trapentry.S之后，首先 SAVE_ALL 保存上下文，之后把上下文（也就是寄存器及相关信息）包装成结构体，传递给真正的中断处理函数trap那里去，进行中断处理，处理完成之后会回到这里向下继续执行寄存器恢复操作，首先，从trapframe中恢复用户程序的寄存器值（这由RESTORE_ALL宏完成），使得用户程序能够继续运行。接着，根据中断或者异常的类型重新设置sepc，确保程序能够从正确的地址继续执行。然后，将sstatus.SPP设置为 0，表示要返回到 U 模式。

当准备工作完成后，会执行sret指令，根据sstatus.SPP的值（此时为 0）切换回 U 模式。随后，恢复中断使能状态，将sstatus.SIE恢复为sstatus.SPIE的值。由于在 U 模式下总是使能中断，因此中断会重新开启。接着，更新sstatus，将sstatus.SPIE设置为 1,sstatus.SPP设置为 0，为下一次中断做准备。最后，将sepc的值赋给pc，并跳转回用户程序（sepc指向的地址）继续执行。此时，系统已经安全地从 S 模式返回到 U 模式，用户程序继续执行。

问题二：mov a0，sp的目的是什么?

在这段RISC-V汇编代码中，move a0, sp的核心目的是将“保存了完整上下文的栈帧地址”作为参数，传递给后续调用的C语言trap函数。在执行move a0, sp之前，代码先调用了SAVE_ALL宏， SAVE_ALL的核心作用是将当前所有通用寄存器、关键CSR寄存器（如sstatus、sepc等）的值，按固定顺序保存到当前栈（sp指向的栈空间）中。这些保存的数据共同构成了一个“中断上下文结构体”，其内存布局与栈中保存的寄存器顺序严格对应。

RISC-V的函数调用规范（即“调用约定”）明确规定：函数的第一个参数通过a0寄存器传递，第二个参数通过a1传递，以此类推（a0~a7用于传递最多8个参数）。而这里的sp寄存器，在执行完SAVE_ALL后，恰好指向“中断上下文结构体（trapframe）在栈中的起始地址”。因此，move a0, sp本质是：
把“中断上下文结构体的起始地址”存入a0寄存器，作为给后续jal trap调用的C函数 trap 的第一个参数。

我们以trap.c文件中的tarp_dispatch函数为例：

```
static inline void trap_dispatch(struct trapframe *tf) {
    //scause的最高位是1，说明trap是由中断引起的
    if ((intptr_t)tf->cause < 0) {
        // interrupts
        interrupt_handler(tf);
    } else {
        // exceptions
        exception_handler(tf);
    }
}
```

可以看到:C函数 trap 接收的struct trapframe *tf参数，正是汇编中通过a0传递的“上下文地址”。通过这个指针， trap_dispatch 函数能读取tf->scause（中断原因）.

问题三：SAVE_ALL中寄寄存器保存在栈中的位置是什么确定的?

在 RISC-V 中断 / 异常处理的 SAVE_ALL 宏中，寄存器在栈中的保存位置由预先定义的 “上下文结构体（trapframe）布局” 确定—— 本质是将所有需要保存的寄存器（通用寄存器、关键 CSR）按照固定顺序 “映射” 到栈空间中，形成一个结构化的内存块（即 trapframe），后续 RESTORE_ALL 会严格按照相同顺序恢复，确保寄存器值不混乱。

问题四：对于任何中断，__alltraps 中都需要保存所有寄存器吗?

对任何中断，__alltraps 都需要保存所有通用寄存器和关键 CSR（控制状态寄存器）。

中断的核心逻辑是 “暂停当前正在执行的程序（用户态 / 内核态）→ 执行中断处理程序 → 回到被打断的位置继续执行”。被中断程序的执行状态，完全由 寄存器集合 决定（包括通用寄存器的数值、CSR 中的特权级 / 中断使能状态等）。如果中断处理前不保存这些寄存器，中断处理程序在运行时会覆盖它们的值 —— 导致被中断程序恢复时，寄存器状态已被破坏，无法回到中断前的正确执行流程（比如变量值丢失、指令地址错误）。

但是并非所有 CSR 都需要保存，仅需保存关键状态
，“需要保存的寄存器” 主要指 通用寄存器（x0-x31） 和 与中断 / 特权级相关的关键 CSR（如 sstatus、sepc、scause 等），而非所有 CSR。代码中 RESTORE_ALL 仅恢复 sstatus（特权级和中断使能状态）和 sepc（中断返回地址），因为其他 CSR（如 sscratch、stvec）在中断处理中可能未被修改，或其状态对被中断程序的恢复无影响。

扩增练习 Challenge2：理解上下文切换机制

问题一：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?

在 RISC-V 汇编代码中，`csrw sscratch, sp; csrrw s0, sscratch, x0` 这两条指令的作用是**将栈指针（`sp`）保存到 `sscratch` 寄存器，再将 `sscratch` 的值（即原 `sp`）转移到 `s0` 寄存器**，其核心目的是**在中断/异常入口（trap entry）时安全地保存栈指针，并为后续上下文保存做准备。**

具体操作分析：

1. `csrw sscratch, sp` - `csrw` 是“控制状态寄存器写”指令，用于将通用寄存器的值写入控制状态寄存器（CSR）。 - 这条指令将当前栈指针 `sp` 的值写入 `sscratch` 寄存器（`sscratch` 是一个特殊的 CSR，通常用于中断/异常处理时临时存储数据）。
2. `csrrw s0, sscratch, x0` - `csrrw` 是“控制状态寄存器读写交换”指令，功能是：将 CSR 的值读到第一个通用寄存器，同时将第二个通用寄存器的值写入该 CSR。 - 这里的操作是：将 `sscratch` 中保存的原 `sp` 值读取到 `s0` 寄存器，同时将 `x0`（零寄存器，恒为 0）写入 `sscratch`。

目的：

在中断/异常触发时，处理器需要保存当前上下文（如通用寄存器、栈指针等），但此时 `sp` 可能指向用户栈或内核栈，直接操作可能冲突。这两条指令的核心作用是：

1.先通过 `sscratch` 临时缓存当前 `sp` 的值，再将其转移到 `s0` 寄存器（`s0` 是被调用者保存寄存器，在中断处理中可安全使用），确保栈指针不丢失。

2.最后将 `sscratch` 清零，为后续处理（如切换到内核栈）做准备（`sscratch` 常被用作“是否在特权态”的标志，0 通常表示当前需要使用内核栈）。 简言之，这是中断入口处保存栈指针并初始化 `sscratch` 的标准操作，为后续上下文保存和栈切换奠定基础。

问题二：`save all`里面保存了`stval scause`这些`csr`，而在`restore all`里面却不还原它们?

因为这些寄存器是**只读的陷阱信息寄存器**，或者它们的值在返回时**已经没有必要恢复**。

1. `stval` (Supervisor Trap Value)

- **作用**：硬件在发生特定异常（如缺页、访问错误、非法指令）时，自动向 `stval` 写入与异常相关的附加信息（例如，缺页的虚拟地址）。
- **性质**：对软件来说是**只读**的。它记录了**一次性的、特定于本次陷阱的事件信息。**
- **为何不恢复**：当从陷阱返回时，导致这次异常的条件可能已经解决（例如，缺页处理程序已经分配了物理页），或者该异常是致命的（进程将被终止）。无论如何，**上一次异常的那个 `stval` 值对于未来的执行已经没有意义。**下一次再发生异常时，硬件会自动用新的值**覆盖**它。恢复一个即将被覆盖的旧值是徒劳的。

2. `scause` (Supervisor Cause Register)

- **作用**：硬件在陷入时自动写入，记录陷入的原因（中断还是异常，以及具体类型）。
- **性质**：同样是**只读**的，并且是**一次性的陷阱原因记录。**
- **为何不恢复**：和 `stval` 类似，`scause` 的价值仅在于**告知本次陷阱的处理程序发生了什么。**处理完毕返回后，这个原因就成为历史。下一次陷入时，硬件会再次写入新的原因代码。恢复它没有任何益处。

3. `sstatus` (Supervisor Status Register)

`sstatus` 是需要在 `RESTORE_ALL` 中恢复的!

- **作用**：它包含全局状态位，如中断使能位 `SIE`、之前的特权级 `SPP`、之前的中断使能 `SPIE` 等。
- **为何必须恢复**：这些状态（尤其是 `SIE/SPIE` 和 `SPP`）决定了 CPU 在 `sret` 指令执行后的行为：
  - `sret` 会使用 `sstatus.SPP` 来决定返回到用户态还是内核态。
  - `sret` 会恢复中断使能状态（将 `SIE` 设为 `SPIE`）。
  - 如果不恢复 `sstatus`，CPU 将无法正确地返回并恢复到陷阱发生前的机器状态。

问题三：保存 `stval`、`scause` 的意义何在?

意义在于：**为 C 语言的陷阱分发函数 `trap()` 提供完整的、用于决策的上下文信息。**

当 `SAVE_ALL` 将这些 CSR 的值保存到栈上的 `trapframe` 结构后：

1. **C 代码可访问**：`trap(struct trapframe *tf)` 函数可以通过 `tf->scause` 来判断陷阱类型，通过 `tf->stval` 来获取详细信息（如缺页地址）。
2. **形成完整快照**：`trapframe` 是陷阱发生瞬间的**完整架构状态快照。**这对于调试、核心转储（core dump）或高级异常处理（如向信号传递详细信息）至关重要。
3. **历史记录**：即使硬件寄存器被覆盖，保存在内存栈上的这个“记录”依然存在，可供后续分析。

扩展练习Challenge3：完善异常中断

```
case CAUSE_ILLEGAL_INSTRUCTION:
    cprintf("Exception type: Illegal instruction\n");
    cprintf("Illegal instruction caught at 0x%08x\n", tf->epc);
    tf->epc += 4;
    break;
```

我们首先通过 `cprintf` 输出异常类型信息`"Exception type: Illegal instruction"`，接下来我们需要输出异常指令的具体地址，为了找到这个地址，我们首先要知道传入的陷阱帧 `tf` 中保存了哪些信息，我们先查看`trapframe`结构体：

```
struct trapframe {
    struct pushregs gpr;
    uintptr_t status;
    uintptr_t epc;
    uintptr_t badvaddr;
    uintptr_t cause;
};
```

可以看到，`trapframe`结构体在异常或中断发生时，保存了以下内容：

- gpr：保存所有通用寄存器（x0-x31）的值，用于异常处理后恢复现场
- status：保存处理器状态信息，包括当前特权级和中断使能位
- epc：异常程序计数器，保存引发异常的指令地址，也就是我们输出异常地址的来源
- badvaddr：在某些异常中保存出错的虚拟地址
- cause：异常原因码，标识具体的异常或中断类型

这样，我们就可以通过`tf->epc`来找到发生异常的地址并输出。

最后一步是 `tf->epc += 4` 。这一步需要调整程序计数器，通过将其加4（指令长度为4字节），就可以使异常返回后能够跳过当前有问题的指令，继续执行下一条指令。这种处理方式防止了系统在遇到非法指令时陷入死循环。

对于断点异常（CAUSE_BREAKPOINT），我们采用了类似的实现方法：

```
case CAUSE_BREAKPOINT:
    cprintf("Exception type: breakpoint\n");
    cprintf("ebreak caught at 0x%08x\n", tf->epc);
    tf->epc += 4;
    break;
```

断点异常通常由 `ebreak` 指令触发。按照要求，我们首先输出`"Exception type: breakpoint"`，然后通过`cprintf("ebreak caught at 0x%08x\n", tf->epc);`显示具体的断点位置。同样地，最后通过 `tf->epc += 4` 使程序能够跳过断点指令继续执行。

本组认为本实验中重要的知识点

1.**中断处理流程**：包括中断触发、上下文保存、中断分发、具体处理及上下文恢复的全过程。

2.**时钟中断的实现**：通过 `clock_set_next_event()` 设置下一次中断，并利用 `ticks` 和 `print_times` 控制打印与关机逻辑。

3.**上下文保存与恢复机制**：通过 `SAVE_ALL` 和 `RESTORE_ALL` 宏实现寄存器状态的保存与恢复，确保程序正确返回。

4.**异常处理扩展**：如非法指令和断点异常的处理，包括输出异常信息、定位异常地址并调整 `epc` 以跳过异常指令。

5. `sscratch` 寄存器的作用：在中断入口处用于暂存栈指针，辅助栈切换与上下文保存。

对应的OS原理中的知识点

1. **中断与异常机制**：包括中断的分类、触发方式、处理流程及其在保护模式下的实现。
2. **上下文切换**：进程或线程切换时保存和恢复CPU状态的过程，确保执行流的正确切换。
3. **系统调用与陷入机制**：用户程序通过陷入指令（如 `ecall`）进入内核态执行特权操作。
4. **时钟管理与调度**：利用时钟中断实现时间片轮转、定时任务等调度机制。
5. **异常处理与错误恢复**：操作系统对程序运行中出现的异常进行捕获、处理和恢复的策略。

二者的含义、关系、差异等方面的理解

含义：实验通过具体代码实现了OS原理中关于中断、异常、上下文切换等核心机制。

关系：实验是原理的具体化，例如 `SAVE_ALL` 对应上下文保存，`trap_dispatch` 对应中断分发逻辑。

差异：

- 实验中关注的是RISC-V架构下的具体实现，如CSR寄存器的使用、汇编与C的交互。
- 原理中更强调通用概念与设计思想，实验则聚焦于具体硬件平台和代码细节。
- 实验中未涉及多进程/线程调度、虚拟内存等高级主题，仅聚焦于中断和异常的基础处理。

思考问题二

本组认为OS原理中很重要，但在实验中没有对应上的知识点

实验目前已经涵盖多方面知识点