OSlan3操作系统中断与异常处理

一、 实验目的

本次实验旨在加深对操作系统中断和异常处理机制的理解。通过编程实践，掌握在RISC-V架构下处理时钟中断、非法指令异常和断点异常的具体流程，并深入理解中断上下文的保存与恢复、特权级切换以及中断处理流程等核心概念。

二、实验内容

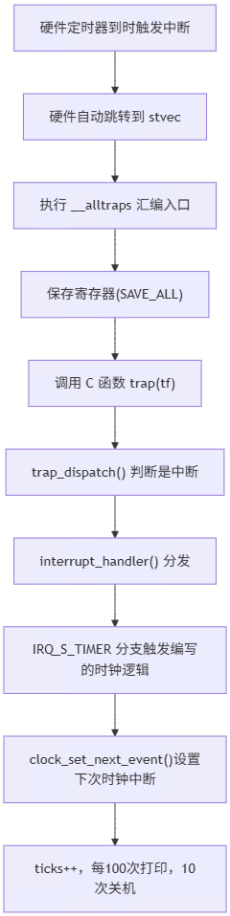
练习1：完善中断处理

定时器中断中断处理的流程

1. 在init.c，内核初始化函数中，调用了 `idt_init()` 和函数 `clock_init()` 和 `intr_enable()`
   a. `idt_init()` 中，设置**sscratch**寄存器的值为0，即在内核里出触发trap，不需要存储内核的地址，设置**stvec**寄存器，指向异常处理入口函数__alltraps，此处为直接位置，不需要映射
   b. `clock_init()` 函数，设置 `sie` 寄存器，**控制 S 模式下允许响应时钟中断**，调用封装的opensbi系统调用函数，设置第一次时钟中断。由于每一次只能设置设计一个时间中断，在时钟初始化的时候设置第一次时钟中断，之后的时间中断都在trap处理函数中设置。
   c. `intr_enable()` 调用是sstatus寄存器设置函数，允许内核响应中。
2. 由于在 `clock_init()` 函数中设置了第一次时钟中断（注意：时钟频率是10MHz，每秒触发100次时钟中断，则时钟中断间隔是100000Hz，大约0.01s）,第一次时钟中断发生。当 CPU 发生 中断 / 异常 时，硬件会自动做三件事：
   a. 把当前指令地址（出错位置）存入 `sepc` ；
   b. 把 trap 原因存入 `scause` ；
   c. 跳转到 `stvec` （异常入口地址，也就是 `__alltraps` ）。
3. 但CPU 不会自动保存通用寄存器，如果不马上保存寄存器，它们可能会被 trap 处理代码覆盖掉。
4. 时钟中断发生，通过 `stvec` 跳转到 `__alltraps` ，执行该部分汇编代码，其主要功能如下
   a. 保存寄存器上下文，包括32个通用寄存器和4个CSR寄存器，保存当前的栈帧sp寄存器的值，封装到结构体中，便于C语言程序访问。
   b. 跳转到trap函数位置，通过 `scause` 的值，时钟中断高位 `scause` 为1，调用中断处理函数，再通过switch分支跳转到具体的时钟中断发生的处理程序。

c. 在`IRQ_S_TIMER`分支里，设置下次时钟中断，更新外部变量`ticks`（在clock初始化时设置的时钟中断计数器），当`ticks`达到指定次数1000时，触发opensbi的系统调用函数`sbi_shutdown()`关机。

大致的流程如下



值得注意的是：S模式有访问CPU时钟计数器的权限，但是无法设置时钟中断，时钟中断只能在M模式下进行设置。设置时钟中断时调用了`clock_set_next_event();`，该函数封装了`sbi_set_timer()`，`sbi_set_timer()`中封装了`sbi_call()`,通过内联汇编和ecall指令调用opensbi提供的服务（在M模式）。

时间中断是在M模式触发，但时间中断的处理在S模式下进行。默认情况下，RISC-V遵循最保守的安全设计：所有中断与异常都会首先陷入最高特权级—— M 模式。为了减少性能开销和提高灵活性，RISC-V引入了**中断委托机制**。在riscv.h中，设置`CSR_MIDELEG 0x303`，M 模式将时钟中断委托给S模式处理。

实现过程

1. 了解代码框架，明确时钟中断的处理流程
2. 到指定位置完善时钟中断的具体响应操作
3. 根据要求，设置下一次时钟中断点`clock_set_next_event()`，更新ticks这个外部变量，在clock初始化时初始化，用于纪律时钟中断的次数。时钟中断每经过100次时，打印信息。当中断次数达到1000时，调用封装的关机函数`sbi_shutdown()`。

Challenge 1：描述与理解中断流程

回答：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。

我们来分类详细讨论在 U/S/M 三种特权级下，中断异常是如何产生和处理的。

在讨论之前，我们要先了解委托机制。RISC-V 架构规定，所有中断和异常默认都由最高权限的M-mode 捕获。但是，如果让 M-mode 处理所有事情，效率会很低，并且 OS 内核（运行在 S-mode）将无法直接管理硬件。因此，M-mode 可以通过设置 medeleg 和 mideleg两个寄存器，将特定类型的中断和异常“委托”给 S-mode 处理。其中，medeleg按位指定哪些同步异常 （如缺页、非法指令、来自 U/S 态的 ecall ）可以直接由 S-mode 处理，无需 M-mode 干预。mideleg按位指定哪些异步中断（如S态定时器中断、外部设备中断）可以直接由 S-mode 处理。

在我们的 uCore 实验中，M-mode 的引导程序 OpenSBI已经配置好了这两个寄存器，将绝大部分与 OS 相关的中断和异常都委托给了 S-mode。

1. U-mode (用户态)

这是应用程序运行的模式，权限最低。

阶段一：系统初始化（设置阶段）

在操作系统启动的早期阶段，内核会进行一次性的设置，为后续所有中断/异常处理铺好路。

4. idt_init() 函数： 在 kern/trap/trap.c 中定义，在init.c中被调用。
5. 设置 stvec： idt_init 的核心工作是设置 stvec寄存器。它将 stvec 的值设置为 __alltraps 的入口地址。
    - stvec 告诉 CPU，当在 S-mode（或从 U-mode/S-mode 陷入 S-mode）发生任何中断或异常时，应该无条件地跳转到哪个地址去执行。
    - __alltraps： 定义在 kern/trap/trapentry.S 中的一段汇编代码，是所有中断/异常处理的统一入口点。

至此，硬件知道了处理程序的地址。

阶段二：事件发生（触发阶段）

CPU 正在 U-mode 下执行用户程序代码。突然，一个需要内核介入的事件发生了。

- 中断：异步事件。
    - 时钟中断： CPU 内部的计时器到期，通知内核调度新的进程。
    - 外部中断： 硬盘读写完成、网卡收到数据包等。
- 异常： 同步事件，由当前执行的指令直接导致。
    - ecall from U-mode： 用户程序请求操作系统服务（即系统调用 ）。

- 缺页异常（Page Fault）：用户程序访问了一个尚未映射或无权访问的内存地址。
- 非法指令：CPU 无法识别当前指令。

- 以ecall from User Mode (系统调用)为例： CPU 检测到 ecall 指令,查询 medeleg 寄存器，发现来自 U-mode 的 ecall (异常码 8) 已被委托。

阶段三：硬件原子操作（陷入阶段）

这一步由 CPU 硬件自动、原子地完成，软件无法干预，也无法观察到中间状态。当上述事件发生时，CPU 会 立即暂停当前的用户程序，并在同一个原子操作中完成以下所有事情：

a. 保存现场到 CSR 寄存器：
- sepc ← 当前 pc (即被中断指令的地址)。
- scause ← 陷阱原因的编码（指明是时钟中断、缺页还是 ecall 等）。
- stval ← 导致异常的相关地址（例如，缺页异常时的目标地址）。

b. 更新状态并切换特权级：
- sstatus.SPP ← 保存当前的特权级（即 U-mode）。
- sstatus.SPIE ← 保存 sstatus.SIE 的值（即陷入前 S-mode 中断是否使能）。
- sstatus.SIE ← 清零 ，以在内核处理期间 禁用 S-mode 中断 ，防止嵌套。
- CPU 内部的特权级从 U-mode 切换到 S-mode 。

c. 跳转到处理程序：
- pc ← stvec 寄存器的值（即 __alltraps 的地址）。

陷入阶段完成后，CPU 已经处于 S-mode，并且下一条要执行的指令是 __alltraps 的第一条指令。

阶段四：软件处理（处理阶段）

现在，控制权交给了我们的内核代码。

1. 汇编入口 (__alltraps)：
   - SAVE_ALL：执行宏，将 所有通用寄存器 (x0-x31) 压入当前进程的内核栈。这在栈上形成了一个 struct trapframe 的结构。
   - move a0, sp：将栈顶指针 sp（它现在正指向 trapframe 的起始位置）放入 a0 寄存器。这是为了给即将调用的 C 函数传递第一个参数,也就是trapframe 的地址。
   - jal trap：跳转并链接到 C 函数 trap()。
2. C 语言分发 (trap() 和 trap_dispatch())：
   - trap() 函数接收 a0 传来的 trapframe 指针（参数名为 tf）。
   - 它首先将 sepc, scause 等 CSR 寄存器的值，保存到 tf 结构体对应的字段中（tf->epc, tf->cause），至此，完整的现场信息都集中到了 trapframe 中。

- trap() 调用 trap_dispatch(tf)。
- trap_dispatch() 检查 tf->cause 的最高位：
  - 如果为 1 (< 0)：是中断，调用 interrupt_handler(tf)。
  - 如果为 0 (>= 0)：是异常，调用 exception_handler(tf)。

3. 具体处理：
   - interrupt_handler 或 exception_handler 会进一步根据 tf->cause 的具体值，调用最终的处理函数。
   - 例如，如果是系统调用，会执行 syscall()；如果是时钟中断，会执行 run_timer_list()；如果是缺页，会执行 pgfault_handler()。
   - 这些函数可能会读取或修改 trapframe 中的内容。例如，syscall() 会将返回值写入 tf->gpr.a0，这样当用户程序恢复执行时，就能在 a0 寄存器中拿到返回值。

阶段五：返回用户态（返回阶段）

处理完成，需要将控制权安全地交还给用户程序。

1. 返回到汇编 (__trapret)：
   - 从具体处理函数层层返回，最终 trap() 函数执行完毕，返回到 trapentry.S 中 jal trap 的下一条指令。
   - RESTORE_ALL：执行宏，从内核栈上的 trapframe 中，将所有通用寄存器的值恢复到 CPU 的物理寄存器中。恢复的顺序和当时保存的顺序反过来，先加载两个CSR, 再加载通用寄存器。
2. sret 指令：
   - 这是另一条特殊的原子硬件指令，用于从陷阱中返回。它执行与陷入时相反的操作：
     - pc ← sepc 的值（回到用户程序被中断的地方）。
     - CPU 特权级 ← sstatus.SPP 中保存的特权级（即恢复到 U-mode）。
     - sstatus.SIE ← sstatus.SPIE 的值（恢复陷入前的 S-mode 中断使能状态）。

sret 执行完毕后，CPU 的控制权就神不知鬼不觉地回到了用户程序中， 从被中断指令的下一条指令继续执行，仿佛什么都没有发生过一样。

2. S-mode (内核态)

这是操作系统内核运行的模式，权限较高，可以访问 I/O 和管理内存。

内核自身在运行时也可能触发异常。例如：

2.1 内核代码试图访问一个不存在的内核地址，发生Page Fault in Kernel (内核缺页)。

- 流程:

a. MMU 发现地址翻译错误。

b. 由于事件发生在 S-mode，特权级不发生变化，仍然是 S-mode。

c. 硬件保存 pc 到 sepc，出错地址到 stval，原因到 scause，禁用中断，然后跳转到 stvec (__alltraps)。

d. trap 函数会发现这是一个来自内核的异常（通过 trap_in_kernel 判断），通常会打印错误信息并 panic（宕机），因为内核无法从自己的严重错误中恢复。

2.2 ecall from Supervisor Mode (SBI 调用):

- 场景: 内核需要 M-mode 的固件（OpenSBI）提供服务，例如关闭计算机、给其他 CPU 核发送核间中断 (IPI)、或设置 M-mode 的定时器。
- 流程:

a. 内核执行 ecall 指令。

b. 硬件检测到 ecall 来自 S-mode，这没有被委托。

c. 硬件自动切换：S-mode → M-mode。

d. 硬件跳转到 mtvec 指向的 M-mode 陷阱处理入口（由 OpenSBI 设置）。

e. OpenSBI 处理该 SBI 调用。

f. OpenSBI 执行 mret 指令，硬件自动切换回 S-mode，返回到 ecall 的下一条指令继续执行。

3. M-mode (机器态)

这是最高权限的模式，通常由底层固件（如 OpenSBI）使用，负责最底层的硬件管理。例如，未被委托给 S-mode 的中断或异常发生（例如机器错误 Machine Error ）或S-mode 执行 ecall 请求 SBI 服务。

- 流程:

a. 任何未被委托的事件都会导致硬件陷入 M-mode。

b. CPU 跳转到 mtvec 指向的地址。

c. M-mode 的固件（OpenSBI）接管控制权。

d. 它会处理该事件。如果是 SBI 调用，就提供相应服务；如果是严重硬件错误，可能会重置系统。

e. 处理完毕后，通过 mret 指令返回到之前的特权级（通常是 S-mode）。

清晰地梳理了一遍流程，我们完全可以解答全部的问题：

1. mov a0,sp的目的是将栈顶指针 sp（它现在正指向 trapframe 的起始位置）放入 a0 寄存器。这是为了给即将调用的 C 函数传递第一个参数,也就是trapframe 的地址。
2. 是由 C 语言中的 struct trapframe 结构体定义和汇编代码 SAVE_ALL 宏之间的一个 硬编码约定 确定的。

C 语言的视角 (struct trapframe)：在 C 代码（如 trap.c ）中，当中断处理程序需要访问被中断程序的寄存器时，它不是直接去操作物理寄存器，而是通过一个指向 struct trapframe 结构体的指针来访问。这个结构体定义了所有通用寄存器以及 sstatus , sepc 等状态寄存器在内存中应该如何排列。

汇编语言的视角 (SAVE_ALL)：在汇编代码 trapentry.S 中， SAVE_ALL 宏的作用就是在栈上创建一个与 struct trapframe 内存布局一模一样的结构。它通过一系列的 sd (store doubleword) 指令，按照预先约定好的顺序，将每个物理寄存器的值存放到栈上的特定偏移位置。

这个顺序必须严格匹配，如果 struct trapframe 中第一个字段是 x1 (ra)，那么 SAVE_ALL 宏的第一条 sd 指令就必须是保存 ra 寄存器。如果 C 结构体和汇编宏的顺序或大小不一致，C 代码就会从错误的内存地址读取寄存器的值，导致灾难性的后果。

3. 是的，必须全部保存。

这看起来似乎有些低效，特别是对于一个简单的时钟中断，它可能只需要修改一两个寄存器。但保存所有寄存器是出于以下几个至关重要的原因：

透明性：中断/异常对于被中断的程序来说应该是“透明的”。当程序恢复执行时，它不应该感知到自己曾经被暂停过。这意味着它的所有寄存器状态都必须和中断发生前一模一样 。内核无法预知被中断的程序正在用哪些寄存器，所以最安全、最简单的做法就是全部保存，全部恢复。

未知上下文： __alltraps 是一个通用的入口点。它不知道即将要处理的是一个简单的时钟中断，还是一个复杂的缺页异常，更不知道即将调用的 C 函数 (trap 以及后续的函数) 会用到哪些寄存器。

C 函数调用约定：当中断处理流程从汇编进入 C 函数 (trap) 时，必须遵守 C 语言的调用约定。调用者（在这里是 __alltraps ）有责任保护那些“调用者保存”（caller-saved）的寄存器。而被调用者 (trap 函数) 则会假设某些“被调用者保存”（callee-saved）的寄存器在函数返回后保持不变。为了满足这个复杂的约定，并且不破坏被中断程序的上下文，最简单的方法就是不加区分地保存所有寄存器。

Challenge 2：理解上下文切换机制

回答：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？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？

问题一：

sscratch 寄存器的核心作用是在内核态和用户态之间安全切换栈空间 。

a. csrw sscratch, sp

这个指令将当前的栈指针 sp 写入 sscratch 寄存器。当陷阱（中断/异常）发生时，CPU 从用户态（U-mode）切换到内核态（S-mode）。此时 sp 仍然指向用户态的栈，这是不安全的，可能被用户篡改，内核需要切换到自己的内核栈。所以这一个操作就是要保存用户态栈指针到 sscratch ，为后续切换内核栈做准备。

b. csrrw s0, sscratch, x0

- 操作： 这是一个 原子交换操作,读取 sscratch 的值（即刚才保存的用户态 sp ）到 s0 寄存器,同时将 x0（0）写入 sscratch 。通过 s0 暂存用户态栈指针（后续会将其保存到 trapframe 中）,将 sscratch 清零，避免内核态使用时被用户态数据干扰。

这两行代码配合起来实现了用户栈到内核栈的安全切换 。简单来说,陷阱发生前,sp 指向用户栈,保存到 sscratch .陷阱处理时，内核需要使用自己的栈,通过交换操作取出用户栈指针并切换到内核栈；最终，用户栈指针会被保存到 trapframe 中，以便陷阱处理完成后恢复。

问题二

SAVE_ALL 会保存 sstatus 、 sepc 、 stval 、 scause 等 CSR 寄存器。

- sstatus： 记录陷阱发生前的特权级和中断使能状态；
- sepc： 记录陷阱发生前的程序计数器（PC）；
- stval： 记录陷阱相关的地址（如缺页异常的访问地址）；
- scause： 记录陷阱的原因（如时钟中断、系统调用）。

那么， 为什么不还原这些 CSR 寄存器呢?

因为这些 CSR 寄存器是“一次性”的临时信息，仅用于当前陷阱处理，不需要也不能还原。

当陷阱处理完成后，CPU 会通过 sret 指令返回用户态。此时，这些 CSR 寄存器（如 scause 、stval ）的值已经被内核处理过，不再需要。下一次陷阱发生时，硬件会自动重新设置这些寄存器。如果手动还原，反而会覆盖新的陷阱信息，导致错误。

那么spec和sstatus是怎么处理的呢？sret 指令会自动将 sepc 的值恢复到 PC 中，无需手动还原。另外，sret 还会自动恢复 sstatus 中的特权级（ SPP 字段）和中断使能状态（ SPIE 字段），无需手动修改。

此外，还原这些寄存器没有意义。被中断的用户程序不依赖这些 CSR 寄存器的值 。用户程序只关心自己的通用寄存器（如 x0-x31 ）和栈指针，而 scause 和 stval 等是内核处理陷阱的临时数据，与用户程序无关

问题三

在__alltraps中，store是用于高效地记录中断的进程的上下文，从而形成一个标准的struct trapframe，把它传给C函数，对症下药。C函数拿到这个trapframe以后，会先通过trap_dispatch()，根据 tf->cause判断要把中断包给interrupt_handler()处理还是exception_handler()处理。exception_handler() 需要进一步看 tf->cause，如果是缺页异常，它就要调用页错误处理函数 pgfault_handler()。pgfault_handler() 需要读取 tf->badvaddr 才能知道是哪个地址访问出错了，然后才能去分配物理页、建立映射。syscall() 函数处理完系统调用后，需要修改 tf->epc，让它指向 ecall 的下一条指令，否则程序恢复后会无限循环系统调用。同时，它还要把返回值写入 tf->gpr.a0。如果没有把这些 CSR 存到 trapframe 里，C 代码每需要一个信息，就得内联一段汇编去读一次 CSR 寄存器，这将使得 C 代码混乱不堪且难以维护。

当系统崩溃时，内核做的最后一件事通常就是打印 trapframe 里的所有信息（通过 print_trapframe 函数）。这就相当于飞机的黑匣子，记录了坠毁前一刻的所有关键数据。如果 trapframe 里没有保存 scause, sepc, stval，那么我们面对一个崩溃的系统，将完全不知道它因何而死，排错将无从下手。

总之，store 这些 CSR 寄存器的操作，是为了让后续的 C 代码能够方便安全地访问到此次陷阱事件的所有上下文信息，从而进行正确的处理、决策和调试。它是连接底层汇编和高层 C 代码的桥梁。

Challenge 3：完善异常中断

编程完善在触发一条非法指令异常 mret和，在 kern/trap/trap.c的异常处理函数中捕获，并对其进行处理，简单输出异常类型和异常指令触发地址，即“Illegal instruction caught at 0x(地址)”，“ebreak caught at 0x（地址）”与“Exception type:Illegal instruction"，“Exception type: breakpoint”。

基本原理

在 RISC-V 体系结构中，处理器在执行过程中遇到非法操作（如执行不存在的指令、断点指令、越界访存等），会触发 **异常（Exception）**。
异常触发后，CPU 自动完成以下步骤：

- 保存当前执行现场（寄存器、`pc` 等）到内核栈；
- 写入异常原因到 `scause`；
- 写入异常触发指令地址到 `sepc`；
- 跳转到内核 Trap 入口；
- 最终调用 C 语言层的异常处理函数 `exception_handler(struct trapframe *tf)`。

完善触发异常的代码

1、在 `trap.c` 中，`trap_dispatch()` 会根据 `scause` 的最高位判断是 **中断** 还是 **异常**；然后进入 `exception_handler()` 进行具体类型的处理。

2、在 `exception_handler(struct trapframe *tf)` 函数中添加对 **非法指令异常** 和 **断点异常** 的处理分支。

代码块

```
        case CAUSE_ILLEGAL_INSTRUCTION:
             // 非法指令异常处理
             /* LAB3 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("Illegal instruction caught at 0x%016lx\n", (unsigned
long)tf->epc);
            cprintf("Exception type: Illegal instruction\n");
            tf->epc += 4;  // 跳过非法指令（RISC-V 32-bit 指令）
            break;
        case CAUSE_BREAKPOINT:
            //断点异常处理
            /* LAB3 CHALLLENGE3   YOUR CODE :  */
            /*(1)输出指令异常类型（ breakpoint）
             *(2)输出异常指令地址
             *(3)更新 tf->epc寄存器
            */
            cprintf("ebreak caught at 0x%016lx\n", (unsigned long)tf->epc);
            cprintf("Exception type: breakpoint\n");
            tf->epc += 2;  // 跳过 ebreak 指令
            break;
```

编写触发异常的测试代码

文件路径：`kern/init/init.c`

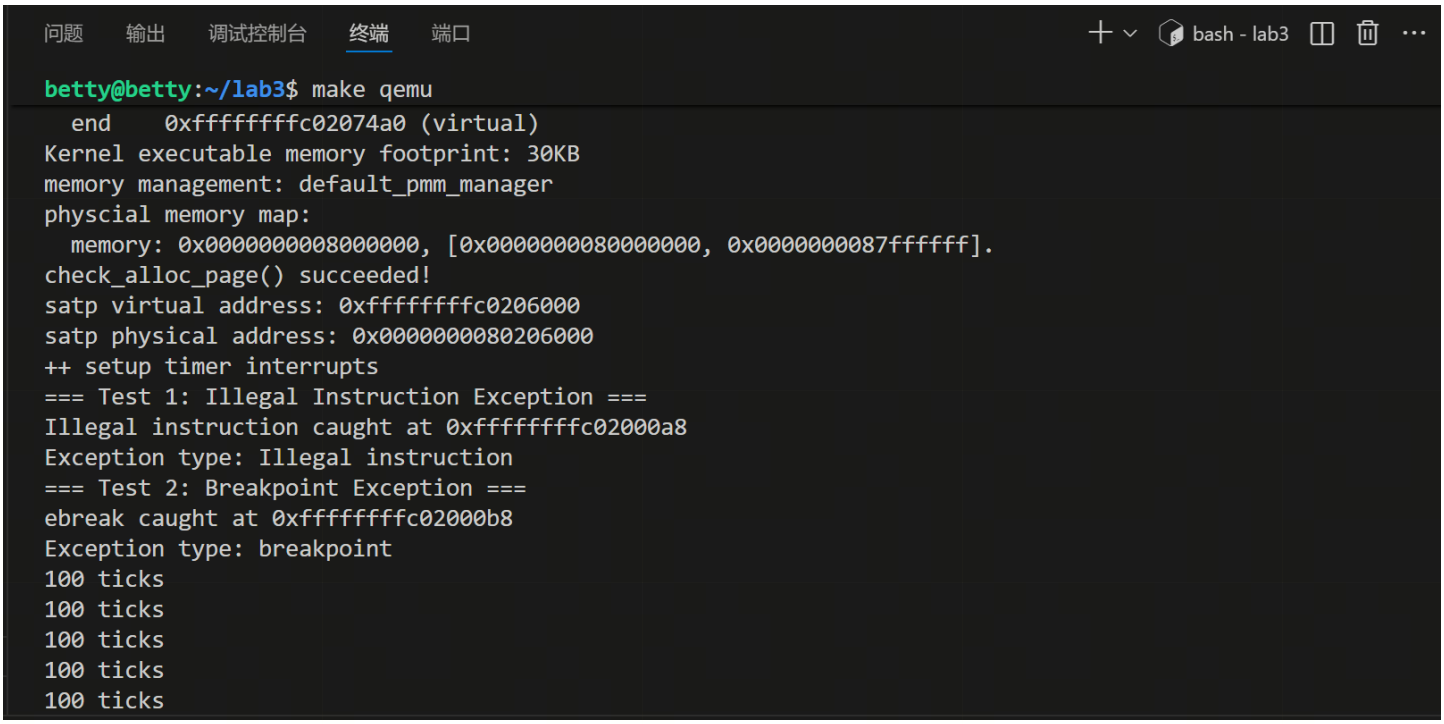
在 `kern_init()` 初始化结束后手动插入触发代码：

代码块

```
int kern_init(void) {
    ...
    cprintf("=== Test 1: Illegal Instruction Exception ===\n");
    asm volatile(".word 0x00000000");  // 插入一条非法指令

    cprintf("=== Test 2: Breakpoint Exception ===\n");
    asm volatile("ebreak");  // 执行断点指令

    while (1);
}

```

输出示例

在终端中运行内核后输出如下：



可以看到本次成功实现了 RISC-V 平台上的非法指令和断点异常的捕获与处理。通过在 `trap.c` 中完善 `exception_handler()`，内核能够正确输出异常类型和触发地址，并在更新 EPC 后恢复执行。

其他代码层面重要内容：

kern/sync/sync.h：为确保内存管理修改相关数据时不被中断打断，提供两个功能，一个是保存sstatus寄存器中的中断使能位(SIE)信息并屏蔽中断的功能，另一个是根据保存的中断使能位信息来使能中断的功能

代码块

```
//思考：这里宏定义的 do{}while(0)起什么作用?#define local_intr_save(x) \
    do {                    \
        x = __intr_save(); \
    } while (0)#define local_intr_restore(x) __intr_restore(x);
```

`do { ... } while (0)` 是宏定义里的**安全打包壳**：它让多条语句宏在任何地方（特别是 if/else 结构里）都能安全使用，执行时只会执行一次，不真的循环。

、比如：

代码块

```
#define local_intr_save(x)  x = __intr_save(); printf("saved\n");
```

展开后是：

```
代码块
1   if (a > 0)
2       x = __intr_save();    // 只有这一句被 if 控制
3   printf("saved\n");        // 无条件执行！！
4   else
5       do_something();       // ❌ 报错：else 没有对应的 if!
```

问题：宏只是“文本替换”，没有语法意识。**if 只控制第一条语句**，后面的就乱套了。

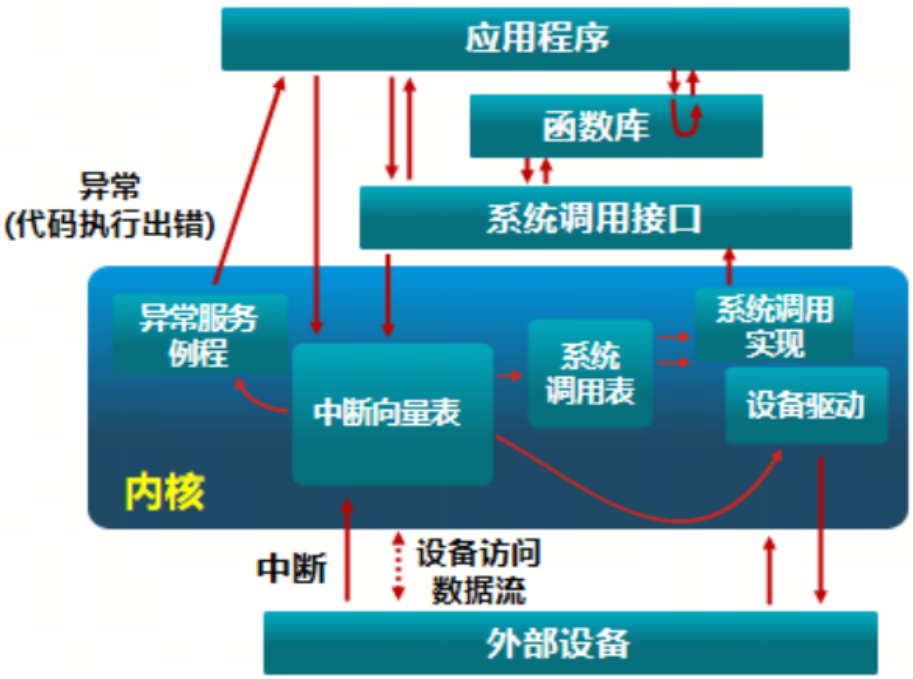
核心原因：**是为了未来的可扩展性和一致性。**虽然现在只有一条语句，但写内核、驱动、底层库的人通常会考虑“宏以后可能会变复杂”。比如将来可能要加打印、日志、调试检查之类的代码：

三、 知识点梳理

本实验重要知识点与OS原理的对应关系

[此处列出本次实验直接涉及的知识点，以及可能与OS课堂对应的部分]

系统调用的实现：



- **RISC-V 内核路径**:
  a. `__alltraps()`：总入口，保存现场。
  b. `trap() -> trap_dispatch`：分析原因。它检查 `tf->cause == CAUSE_USER_ECALL`，发现“哦，这是一个来自用户的 `ecall` 请求”。
  c. `syscall()`：转到系统调用总分派函数。
  d. `tf->tf_regs.a0 == SYS_read`：它检查 `a0` 寄存器（在第1步中设置的）的值，发现是 `SYS_read` (102)。

e. `sys_read()`：内核从保存的寄存器（`tf->sp`）中取出 `fd`，`buf`，`length` 等参数，执行真正的 `sys_read` 内核函数。

f. `sysfile_read()`：调用文件系统函数读取文件。

g. `trapret()`：恢复现场，降权返回用户态。

中断委托机制

默认情况下，RISC-V遵循最保守的安全设计：所有中断与异常都会首先陷入最高特权级—— M 模式。让所有中断（例如来自用户态的系统调用或时钟中断）都经由 M 模式处理，会带来不必要的性能开销和灵活性限制。为解决此问题，RISC-V引入了中断委托机制。

`mideleg` 负责委托中断，`medeleg` 负责委托异常。

在此次实验中，时间中断是在M模式下进行设置的，在riscv.h文件中设置 `#define CSR_MIDELEG 0x303` M模式将时钟中断委托给S模式处理。

OS原理中重要但在实验中未对应的知识点

[此处列出你认为重要的、但本次实验没有直接涉及的OS原理知识点]

为了系统调用请求，用户做了什么？

```
int
ys_read(int64_t fd, void *base, size_t len) {
    return syscall(SYS_read, fd, base, len);
}
```

```
syscall(int num, ...) {
...
    asm volatile (
                "lw a0, %1\n"
    "lw a1, %2\n "
    "lw a2, %3\n"
    "lw a3, %4\n"
    "lw a4, %5\n"
    "lw a5, %6\n"
    "ecall\n "//这一句会个产生一个" 中断"
    "sw a0, %0"
    : "=m" (ret)
    : "m" (num), //这一句的作用是把102放进了a0
      "m" (a[0]),
      "m" (a[1]),
      "m" (a[2]),
      "m" (a[3]),
      "m" (a[4])
    : "memory"
  ); return ret;
```

```
/* syscall number */
#define SYS_exit            1
#define SYS_fork            2
#define SYS_wait            3
#define SYS_exec            4
#define SYS_clone           5
#define SYS_yield           10
#define SYS_sleep           11
#define SYS_kill            12
#define SYS_gettime         17
#define SYS_getpid          18
#define SYS_mmap            20
#define SYS_munmap          21
#define SYS_shmem           22
#define SYS_putc            30
#define SYS_pgdir           31
#define SYS_open            100
#define SYS_close           101
#define SYS_read            102
#define SYS_write           103
#define SYS_seek            104
```

系统调用事件有一个固定中断号，同时借助一个寄存器，传递了系统调用中具体哪个调用（系统调用号）

C代码到汇编代码：`sys_read(...)` 会调用 `syscall(...)`。它把系统调用号（`SYS_read` 对应 102）放入 `a0` 寄存器，把其他参数（`fd`，`base`，`len`）放入 `a1`，`a2`，`a3` 等寄存器。接着，执行 `ecall` 指令。这条指令在RISC-V上的作用与X86的 `int 80H` 完全相同：触发一个异常，使CPU提升权限到内核态，并跳转到内核的异常处理程序。