编程题

1.实现一个裸机应用程序A，能打印调用栈。

以 rCore tutorial ch2 代码为例，在编译选项中我们已经让编译器对所有函数调用都保存栈指针（参考 `os/.cargo/config` ），因此我们可以直接从 fp 寄存器追溯调用栈：

```
os/src/stack_trace.rs
use core::{arch::asm, ptr};

pub unsafe fn print_stack_trace() -> () {
    let mut fp: *const usize;
    asm!("mv {}, fp", out(reg) fp);

    println!("== Begin stack trace ==");
    while fp != ptr::null() {
        let saved_ra = *fp.sub(1);
        let saved_fp = *fp.sub(2);

        println!("0x{:016x}, fp = 0x{:016x}", saved_ra, saved_fp);

        fp = saved_fp as *const usize;
    }
    println!("== End stack trace ==");
}
```

导入了 `core crate` 的 `asm` 和 `ptr` 模块，用于使用汇编语言和指针进行操作。定义了一个公共的、不安全的函数 `print_stack_trace`。使用汇编语言获取当前帧指针（`Frame Pointer, FP`），并将其存储在变量 `fp` 中。`mv` 指令是将 FP 寄存器的值移动到 `fp` 变量中。`out(reg) fp` 表示将 `fp` 变量作为输出寄存器来使用。开始打印调用栈。只要帧指针不是空指针，就打印当前帧的返回地址（`Return Address, RA`）和帧指针（FP），然后将帧指针指向上一帧。结束打印调用栈。

之后我们将其加入 `main.rs` 作为一个子模块：

加入 `os/src/main.rs`

```
// ...
mod syscall;
mod trap;
mod stack_trace;
// ...

注释掉这两行 否则会发生冲突
//pub mod syscall;
//pub mod trap;
```

声明一个名为 `syscall` （系统调用）的模块，它可以包含 Rust 程序中有关系统调用的函数、常量、结构体等内容。

声明一个名为 `trap` （中断和异常）的模块，它可以包含 Rust 程序中有关中断和异常处理的函数、常量、结构体等内容。

声明一个名为 `stack_trace` （调用栈跟踪）的模块，它可以包含 Rust 程序中有关调用栈的函数、常量、结构体等内容。

将打印调用栈的代码加入 `panic handler` 中，在每次 panic 的时候打印调用栈：

```
os/src/batch.rs
//...
use crate::stack_trace::print_stack_trace;
//...
unsafe { print_stack_trace(); }
//...
```

现在，panic 的时候输入的信息变成了这样：

```
== Begin stack trace ==
0x0000000080200770, fp = 0x0000000080206e60
0x0000000080200ff4, fp = 0x0000000080206ef0
0x0000000080200de4, fp = 0x0000000080208ff0
0x0000000080400032, fp = 0x0000000080209000
0x0000000000000000, fp = 0x0000000000000000
== End stack trace ==
```

这里打印的两个数字，第一个是栈帧上保存的返回地址，第二个是保存的上一个 frame pointer。

```
[rustsbi] RustSBI version 0.3.1, adapting to RISC-V SBI v1.0.0
.______       __    __      _______.___________.  _______..______   __
|   _  \     |  |  |  |    /       |           | /       ||   _  \ |  |
|  |_)  |    |  |  |  |   |   (----`---|  |----`|   (----`|  |_)  ||  |
|      /     |  |  |  |    \   \       |  |      \   \    |   _  < |  |
|  |\  \----.|  `--'  |.----)   |      |  |  .----)   |   |  |_)  ||  |
| _| `._____| \______/ |_______/       |__|  |_______/    |______/ |__|
[rustsbi] Implementation     : RustSBI-QEMU Version 0.2.0-alpha.2
[rustsbi] Platform Name      : riscv-virtio,qemu
[rustsbi] Platform SMP       : 1
[rustsbi] Platform Memory    : 0x80000000..0x88000000
[rustsbi] Boot HART          : 0
[rustsbi] Device Tree Region : 0x87000000..0x87000ef2
[rustsbi] Firmware Address   : 0x80000000
[rustsbi] Supervisor Address : 0x80200000
[rustsbi] pmp01: 0x00000000..0x80000000 (-wr)
[rustsbi] pmp02: 0x80000000..0x80200000 (---)
[rustsbi] pmp03: 0x80200000..0x88000000 (xwr)
[rustsbi] pmp04: 0x88000000..0x00000000 (-wr)
[kernel] Hello, world!
[kernel] num_app = 5
[kernel] app_0 [0x8020c038, 0x8020d4e8)
[kernel] app_1 [0x8020d4e8, 0x8020ea40)
[kernel] app_2 [0x8020ea40, 0x80210158)
[kernel] app_3 [0x80210158, 0x80211698)
[kernel] app_4 [0x80211698, 0x80212bd0)
[kernel] Loading app_0
== Begin stack trace ==
0x0000000080200770, fp = 0x0000000080222f60
0x000000008020015e, fp = 0x0000000080223000
```

```
0x0000000080200010, fp = 0x0000000000000000
== End stack trace ==
Hello, world!
[kernel] Application exited with code 0
[kernel] Loading app_1
== Begin stack trace ==
0x0000000080200770, fp = 0x0000000080206e00
0x000000008020088c, fp = 0x0000000080206e60
0x0000000080200f3a, fp = 0x0000000080206ef0
0x0000000080200de4, fp = 0x0000000080209000
0x0000000000000000, fp = 0x0000000000000000
== End stack trace ==
Into Test store_fault, we will insert an invalid store operation...
Kernel should kill this application!
[kernel] PageFault in application, kernel killed it.
[kernel] Loading app_2
== Begin stack trace ==
0x0000000080200770, fp = 0x0000000080206e60
0x0000000080200ff4, fp = 0x0000000080206ef0
0x0000000080200de4, fp = 0x0000000080208ff0
0x0000000080400032, fp = 0x0000000080209000
0x0000000000000000, fp = 0x0000000000000000
== End stack trace ==
3^10000=5079(MOD 10007)
3^20000=8202(MOD 10007)
3^30000=8824(MOD 10007)
3^40000=5750(MOD 10007)
3^50000=3824(MOD 10007)
3^60000=8516(MOD 10007)
3^70000=2510(MOD 10007)
3^80000=9379(MOD 10007)
3^90000=2621(MOD 10007)
3^100000=2749(MOD 10007)
Test power OK!
[kernel] Application exited with code 0
[kernel] Loading app_3
== Begin stack trace ==
0x0000000080200770, fp = 0x0000000080206e00
0x000000008020088c, fp = 0x0000000080206e60
0x0000000080200f3a, fp = 0x0000000080206ef0
0x0000000080200de4, fp = 0x0000000080209000
0x0000000000000000, fp = 0x0000000000000000
== End stack trace ==
Try to execute privileged instruction in U Mode
Kernel should kill this application!
[kernel] IllegalInstruction in application, kernel killed it.
[kernel] Loading app_4
== Begin stack trace ==
0x0000000080200770, fp = 0x0000000080206e60
0x0000000080200ff4, fp = 0x0000000080206ef0
0x0000000080200de4, fp = 0x0000000080208ff0
0x0000000080400032, fp = 0x0000000080209000
0x0000000000000000, fp = 0x0000000000000000
== End stack trace ==
Try to access privileged CSR in U Mode
```

```
Kernel should kill this application!
[kernel] IllegalInstruction in application, kernel killed it.
All applications completed!
```

2.扩展内核，统计执行异常的程序的异常情况（主要是各种特权级涉及的异常），能够打印异常程序的出错的地址和指令等信息。

在trap.c中添加相关异常情况的处理：

`os/trap.c`

```
void usertrap()
{
    // 进入内核态
    set_kerneltrap();
    // 获取当前进程的 trapframe
    struct trapframe *trapframe = curr_proc()->trapframe;

    // 判断是否来自用户态
    if ((r_sstatus() & SSTATUS_SPP) != 0)
            panic("usertrap: not from user mode");

    // 获取异常原因
    uint64 cause = r_scause();
    // 判断是否是外部中断
    if (cause & (1ULL << 63)) {
            // 清除外部中断标志位
            cause &= ~(1ULL << 63);
            switch (cause) {
            // 时间中断
            case SupervisorTimer:
                    // 打印日志
                    tracef("time interrupt!\n");
                    // 设置下一个时钟中断
                    set_next_timer();
                    // 切换到其他进程
                    yield();
                    break;
            default:
                    // 未知的异常，处理函数中止进程
                    unknown_trap();
                    break;
            }
    } else {
            switch (cause) {
            // 系统调用
            case UserEnvCall:
                    // 调整程序计数器，执行系统调用
                    trapframe->epc += 4;
                    syscall();
                    break;
            // 存储地址未对齐
            case StoreMisaligned:
```

```
            // 存储页错误
            case StorePageFault:
            // 指令地址未对齐
            case InstructionMisaligned:
            // 指令页错误
            case InstructionPageFault:
            // 载入地址未对齐
            case LoadMisaligned:
            // 载入页错误
            case LoadPageFault:
                  // 打印日志并终止进程
                  printf("%d in application, bad addr = %p, bad instruction =
%p, "
                           "core dumped.\n",
                           cause, r_stval(), trapframe->epc);
                  exit(-2);
                  break;
            // 非法指令
            case IllegalInstruction:
                  // 打印日志并终止进程
                  printf("IllegalInstruction in application, core dumped.\n");
                  exit(-3);
                  break;
            default:
                  // 未知的异常，处理函数中止进程
                  unknown_trap();
                  break;
            }
   }
   // 返回用户态
   usertrapret();
}
```

进入 `usertrap` 函数，首先调用 `set_kerneltrap` 函数来设置内核陷入帧。定义一个名为 `trapframe` 的指向当前进程 `curr_proc()` 的陷阱帧结构体的指针。检查当前特权级是否为用户模式，如果不是，就会触发 panic。检查陷入原因（trap cause），如果是一个异常，如时钟中断（SupervisorTimer），则设置下一次时钟中断并调用 `yield` 函数进行进程切换。如果不是一个异常，而是一个系统调用（UserEnvCall），则将陷阱帧结构体中的 `epc` 加上 4，并调用 `syscall` 函数。如果是一个访问异常（如存储访问违例、存储页错误、加载违例等），则打印错误信息，并调用 `exit` 函数结束进程。如果是一个非法指令异常（IllegalInstruction），则打印错误信息，并调用 `exit` 函数结束进程。如果是其它未知的陷入原因，则调用 `unknown_trap` 函数打印错误信息。最后，调用 `usertrapret` 函数将控制权返回到用户模式。

问答题

1.函数调用与系统调用有何区别?

- 函数调用用普通的控制流指令，不涉及特权级的切换；系统调用使用专门的指令（如 RISC-V 上的 ecall），会切换到内核特权级。
- 函数调用可以随意指定调用目标；系统调用只能将控制流切换给调用操作系统内核给定的目标。

2.为了方便操作系统处理，M态软件会将 S 态异常/中断委托给 S 态软件，请指出有哪些寄存器记录了委托信息，rustsbi 委托了哪些异常/中断？（也可以直接给出寄存器的值）

- 两个寄存器记录了委托信息：`mideleg`（中断委托）和`medeleg`（异常委托）
- 参考 RustSBI 输出

  ```
  [rustsbi] mideleg: ssoft, stimer, sext (0x222)
  [rustsbi] medeleg: ima, ia, bkpt, la, sa, uecall, ipage, lpage, spage
  (0xb1ab)
  ```

  可知委托了中断：

  - `ssoft`：S-mode 软件中断
  - `stimer`：S-mode 时钟中断
  - `sext`：S-mode 外部中断

  委托了异常：

  - `ima`：指令未对齐
  - `ia`：取指访问异常
  - `bkpt`：断点
  - `la`：读异常
  - `sa`：写异常
  - `uecall`：U-mode 系统调用
  - `ipage`：取指 page fault
  - `lpage`：读 page fault
  - `spage`：写 page fault

3.如果操作系统以应用程序库的形式存在，应用程序可以通过哪些方式破坏操作系统?

如果操作系统以应用程序库的形式存在，那么编译器在链接OS库时会把应用程序跟OS库链接成一个可执行文件，两者处于同一地址空间，这也是LibOS（Unikernel）架构，此时存在如下几个破坏操作系统的方式：

- 缓冲区溢出：应用程序可以覆盖写其合法内存边界之外的部分，这可能会危及 OS；
- 整数溢出：当对整数值的运算产生的值超出整数数据类型可以表示的范围时，就会发生整数溢出，这可能会导致OS出现意外行为和安全漏洞。 例如，如果允许应用程序分配大量内存，攻击者可能会在内存分配例程中触发整数溢出，从而可能导致缓冲区溢出或其他安全漏洞；
- 系统调用拦截：应用程序可能会拦截或重定向系统调用，从而可能损害OS的行为。例如，攻击者可能会拦截读取敏感文件的系统调用并将其重定向到他们选择的文件，从而可能危及 unikernel 的安全性。
- 资源耗尽：应用程序可能会消耗内存或网络带宽等资源，可能导致拒绝服务或其他安全漏洞。

4.编译器/操作系统/处理器如何合作，可采用哪些方法来保护操作系统不受应用程序的破坏?

硬件操作系统运行在一个硬件保护的安全执行环境中，不受到应用程序的破坏；应用程序运行在另外一个无法破坏操作系统的受限执行环境中。 现代CPU提供了很多硬件机制来保护操作系统免受恶意应用程序的破坏，包括如下几个：

- 特权级模式：处理器能够设置不同安全等级的执行环境，即用户态执行环境和内核态特权级的执行环境。处理器在执行指令前会进行特权级安全检查，如果在用户态执行环境中执行内核态特权级指令，会产生异常阻止当前非法指令的执行。
- TEE（可信执行环境）：CPU的TEE能够构建一个可信的执行环境，用于抵御恶意软件或攻击，能够确保处理敏感数据的应用程序（例如移动银行和支付应用程序）的安全。
- ASLR（地址空间布局随机化）：ASLR 是CPU的一种随机化进程地址空间布局的安全功能，其能够随机生成进程地址空间，例如栈、共享库等关键部分的起始地址，使攻击者预测特定数据或代码的位置。

5.操作系统在完成用户态<–>内核态双向切换中的一般处理过程是什么?

当 CPU 在用户态特权级（ RISC-V 的 U 模式）运行应用程序，执行到 Trap，切换到内核态特权级（ RISC-V的S 模式），批处理操作系统的对应代码响应 Trap，并执行系统调用服务，处理完毕后，从内核态返回到用户态应用程序继续执行后续指令。

RISC-V处理器的S态特权指令有哪些，其大致含义是什么，有啥作用?

RISC-V处理器的S态特权指令有两类：指令本身属于高特权级的指令，如 sret 指令（表示从 S 模式返回到 U 模式）。指令访问了S模式特权级下才能访问的寄存器或内存，如表示S模式系统状态的 控制状态寄存器 sstatus 等。如下所示：

- sret：从 S 模式返回 U 模式。如可以让位于S模式的驱动程序返回U模式。
- wfi：让CPU在空闲时进入等待状态，以降低CPU功耗。
- sfence.vma：刷新 TLB 缓存，在U模式下执行会尝试非法指令异常。
- 访问 S 模式 CSR 的指令：通过访问spce/stvec/scause/sscartch/stval/sstatus/satp等CSR来改变系统状态。

6.RISC-V处理器在用户态执行特权指令后的硬件层面的处理过程是什么?

CPU 执行完一条指令（如 ecall ）并准备从用户特权级 陷入（ Trap ）到 S 特权级的时候，硬件会自动完成如下这些事情：

- sstatus 的 SPP 字段会被修改为 CPU 当前的特权级（U/S）。
- sepc 会被修改为 Trap 处理完成后默认会执行的下一条指令的地址。
- scause/stval 分别会被修改成这次 Trap 的原因以及相关的附加信息。
- cpu 会跳转到 stvec 所设置的 Trap 处理入口地址，并将当前特权级设置为 S ，然后从Trap 处理入口地址处开始执行。

CPU 完成 Trap 处理准备返回的时候，需要通过一条 S 特权级的特权指令 sret 来完成，这一条指令具体完成以下功能： * CPU 会将当前的特权级按照 sstatus 的 SPP 字段设置为 U 或者 S ； * CPU 会跳转到 sepc 寄存器指向的那条指令，然后继续执行。

7.在哪些情况下会出现特权级切换：用户态–>内核态，以及内核态–>用户态?

- 用户态->内核态：应用程序发起系统调用；应用程序执行出错，需要到批处理操作系统中杀死该应用并加载运行下一个应用；应用程序执行结束，需要到批处理操作系统中加载运行下一个应用。
- 内核态->用户态：启动应用程序需要初始化应用程序的用户态上下文时；应用程序发起的系统调用执行完毕返回应用程序时。

8.Trap上下文的含义是啥？在本章的操作系统中，Trap上下文的具体内容是啥？如果不进行Trap上下文的保存于恢复，会出现什么情况?

> Trap上下文的主要有两部分含义：
>
> - 在触发 Trap 之前 CPU 运行在哪个特权级；
> - CPU 需要切换到哪个特权级来处理该 Trap ，并在处理完成之后返回原特权级。在本章的实际操作系统中，Trap上下文的具体内容主要包括通用寄存器和栈两部分。如果不进行Trap的上下文保存与恢复，CPU就无法在处理完成之后，返回原特权级。

实验练习

sys_write 安全检查

- 实现分支: ch2-lab
- 目录要求不变
- 为 sys_write 增加安全检查

  在 os 目录下执行 `make run TEST=1` 测试 `sys_write` 安全检查的实现，正确执行目标用户测例，并得到预期输出（详见测例注释）。

  注意：如果设置默认 log 等级，从 lab2 开始关闭所有 log 输出。

```
cd rCore-Tutorial-v3
cd os
git checkout -f ch2-lab
make run TEST=1
```

  输出结果为：

  如图，实现了安全检查，由于app0,1,2运行时出现了异常，所以为了保护内核受到破坏，我们停止了app0,1,2直接切换到了app3的运行，结果中只显示了有两个应用程序运行成功。

```
[rustsbi] RustSBI version 0.2.0-alpha.6
.______       __    __      _______.___________.  _______..______   __
|   _  \     |  |  |  |    /       |           | /       ||   _  \ |  |
|  |_)  |    |  |  |  |   |   (----`---|  |----`|   (----`|  |_)  ||  |
|      /     |  |  |  |    \   \       |  |      \   \    |   _  < |  |
|  |\  \----.|  `--'  |.----)   |      |  |  .----)   |   |  |_)  ||  |
| _| `._____| \______/ |_______/       |__|  |_______/    |______/ |__|

[rustsbi] Implementation: RustSBI-QEMU Version 0.0.2
[rustsbi-dtb] Hart count: cluster0 with 1 cores
[rustsbi] misa: RV64ACDFHIMSU
[rustsbi] mideleg: ssoft, stimer, sext (0x1666)
[rustsbi] medeleg: ima, ia, bkpt, la, sa, uecall, ipage, lpage, spage
(0xb1ab)
```

```
[rustsbi] pmp0: 0x10000000 ..= 0x10001fff (rwx)
[rustsbi] pmp1: 0x80000000 ..= 0x8fffffff (rwx)
[rustsbi] pmp2: 0x0 ..= 0xffffffffffffff (---)
[rustsbi] enter supervisor 0x80200000
[kernel] Hello, world!
[kernel] num_app = 2
[kernel] app_0 [0x8020b020, 0x80210578)
[kernel] app_1 [0x80210578, 0x80215ba0)
[kernel] Loading app_0
[kernel] Panicked at src/trap/mod.rs:45 Unsupported trap
Exception(LoadFault), stval = 0x0!
```

问答作业：

1.正确进入 U 态后，程序的特征还应有：使用 S 态特权指令，访问 S 态寄存器后会报错。请自行测试这些内容 (运行 Rust 三个 bad 测例) ，描述程序出错行为，注明你使用的 sbi 及其版本。

2.请结合用例理解 trap.S 中两个函数 alltraps 和 restore 的作用，并回答如下几个问题:

答：alltraps 和 restore 是 RISC-V 操作系统中用于处理中断和异常的两个重要函数。它们一起协作来保证操作系统的正确性和可靠性。

alltraps 函数是用于处理所有的中断和异常的入口点。当中断或异常发生时，处理器会跳转到 alltraps 函数。在 __alltraps 函数中，操作系统会根据中断或异常的类型，保存一些寄存器的值，然后进行一些必要的处理，例如更新页表、保存上下文等，最后调用相应的中断或异常处理程序来处理中断或异常。

__restore 函数则是用于在中断或异常处理完成后，将处理器的控制权返回到用户态的函数。它会将中断或异常处理过程中保存的寄存器的值恢复回来，并将栈指针恢复到用户态的栈，最后通过 sret 指令返回到用户态的指令位置继续执行。

1. L40：刚进入 restore 时，a0 代表了什么值。请指出 __restore 的两种使用情景。

   答：刚进入 __restore 函数时，a0 寄存器中保存的是内核栈的栈顶指针，即在处理中断或异常时被保存在内核栈上的 TrapContext 结构体的地址。

   __restore 函数有两种使用情景：

   - 在处理完中断或异常后返回用户态：当中断或异常处理完成后，操作系统需要将处理器的控制权恢复回用户态。此时，操作系统会调用 restore 函数，并将内核栈的栈顶指针传入 a0 寄存器。在 restore 函数中，会将保存在内核栈上的 TrapContext 结构体中保存的寄存器的值恢复回来，并将栈指针恢复到用户态的栈，最后通过 sret 指令返回到用户态的指令位置继续执行。
   - 启动应用程序：在操作系统启动应用程序时，会通过 restore 函数来将处理器的控制权转移到应用程序的入口点。此时，操作系统会将应用程序的入口地址、用户态栈的栈顶地址和一些其他参数保存在 TrapContext 结构体中，并将内核栈的栈顶指针传入 a0 寄存器。在 restore 函数中，会将保存在 TrapContext 结构体中的参数恢复回来，并将栈指针设置为用户态栈的栈顶指针，最后通过 sret 指令跳转到应用程序的入口点开始执行。

2. L46-L51：这几行汇编代码特殊处理了哪些寄存器？这些寄存器的的值对于进入用户态有何意义？请分别解释。

```
ld t0, 32*8(sp)
ld t1, 33*8(sp)
ld t2, 2*8(sp)
csrw sstatus, t0
csrw sepc, t1
csrw sscratch, t2
```

答：这几行汇编代码特殊处理了以下寄存器：

- sstatus 寄存器：这个寄存器用于控制 CPU 的运行模式和中断使能等。在 restore 函数中，通过从内核栈中恢复的值来恢复 sstatus 的值，以确保在返回用户态后，CPU 的运行模式和中断使能状态与之前一致。
- sepc 寄存器：这个寄存器用于记录产生中断或异常的指令地址。在 restore 函数中，通过从内核栈中恢复的值来恢复 sepc 的值，以确保在返回用户态后，CPU 能够从产生中断或异常的指令地址继续执行。
- sscratch 寄存器：这个寄存器用于保存一个指向当前线程内核栈顶部的指针。在 restore 函数中，通过从内核栈中恢复的值来恢复 sscratch 的值，以确保在返回用户态后，当前线程的内核栈顶部指针正确。
- 通用寄存器（除 sp 和 tp 之外）：这些寄存器用于保存执行中断或异常前的状态。在 restore 函数中，通过从内核栈中恢复的值来恢复这些寄存器的值，以确保在返回用户态后，执行状态与之前一致。

这些寄存器的值对于进入用户态有以下意义：

1. 恢复 sstatus 和 sepc 寄存器的值，确保 CPU 运行模式和中断使能状态与中断或异常处理前一致，以及能够从中断或异常产生的指令地址继续执行。
2. 恢复 sscratch 寄存器的值，确保当前线程内核栈顶部指针正确。
3. 恢复通用寄存器的值，确保执行状态与中断或异常处理前一致，以便能够正确地继续执行。

3. L53-L59：为何跳过了 x2 和 x4?

```
ld x1, 1*8(sp)
ld x3, 3*8(sp)
.set n, 5
.rept 27
   LOAD_GP %n
   .set n, n+1
.endr
```

答：在 restore 函数中，x2 和 x4 寄存器被跳过是因为它们是保留寄存器，不能用于保存通用数据。在 RISC-V 的规范中，x2 和 x4 寄存器分别是 hardwired zero 和 thread pointer，有特定的用途，不能被用户代码使用。因此，在 restore 函数中，不需要对它们进行恢复操作，直接跳过即可。

4. L63：该指令之后，sp 和 sscratch 中的值分别有什么意义？

```
csrrw sp, sscratch, sp
```

答：在 __restore 函数中，该指令用于交换 sp 和 sscratch 寄存器的值。在指令执行后，sp 寄存器中保存的是用户态栈的栈顶指针，sscratch 寄存器中保存的是内核栈的栈顶指针。这个操作的意义在于，将栈指针切换到用户态的栈上，使得返回到用户态后，CPU 能够正确地访问用户态的栈空间。这个操作通常在返回用户态之前执行。

5. __restore：中发生状态切换在哪一条指令？为何该指令执行之后会进入用户态？

   答：在 restore 函数中，发生状态切换的指令是 sret 指令。在执行 sret 指令之前，restore 函数完成了恢复寄存器的操作，将栈指针切换到了用户态的栈上，恢复了中断和异常处理前的 CPU 运行状态。

   当 sret 指令被执行时，它会将 sstatus 寄存器中的 SPP（Previous Privilege Mode）位设置为用户态，将 sstatus 中的 SPIE（Previous Interrupt Enable）位设置为之前的中断使能状态，然后跳转到 sepc 寄存器中保存的用户态程序指令地址开始执行。这样就完成了从内核态到用户态的状态切换，并开始执行用户态程序。

6. L13：该指令之后，sp 和 sscratch 中的值分别有什么意义？

   ```
   csrrw sp, sscratch, sp
   ```

   答：在 __alltraps 函数中，该指令用于交换 sp 和 sscratch 寄存器的值。在指令执行后，sp 寄存器中保存的是当前线程内核栈的栈顶指针，sscratch 寄存器中保存的是之前保存的用户态栈的栈顶指针。

   这个操作的意义在于，将栈指针切换到内核栈上，以确保在处理中断或异常时，能够使用内核栈保存上下文信息，避免栈溢出等问题。同时，也需要将用户态栈的栈顶指针保存下来，以便在返回用户态时能够恢复用户态栈的栈顶指针。

7. 从 U 态进入 S 态是哪一条指令发生的？

   答：从用户态进入核心态的指令是 ecall 指令。在 RISC-V 中，用户程序可以通过调用系统调用来进入核心态，具体方式是通过将系统调用号放入 a7 寄存器中，然后执行 ecall 指令。当 ecall 指令被执行时，处理器会将当前的 PC、寄存器和状态保存到相应的寄存器中，并跳转到固定的内核入口地址开始执行相应的中断或异常处理程序，从而完成了从用户态到核心态的状态切换。

3.对于任何中断，alltraps 中都需要保存所有寄存器吗？你有没有想到一些加速 alltraps 的方法？简单描述你的想法。

答：对于所有中断或异常，__alltraps 中并不一定需要保存所有寄存器。因为在中断或异常的处理过程中，不是所有寄存器都被使用到，有些寄存器可能是无用的。因此，可以根据不同的中断或异常类型，选择需要保存的寄存器，减少保存寄存器的开销，提高处理效率。

一些加速 __alltraps 的方法包括：

1. 只保存被使用的寄存器：在处理中断或异常的过程中，只保存被使用到的寄存器，而不是所有的寄存器。这样可以减少保存寄存器的开销，提高处理效率。
2. 使用硬件压缩寄存器：一些 RISC-V 处理器支持硬件压缩寄存器，即将 32 位寄存器压缩成 16 位，或者将 64 位寄存器压缩成 32 位。在处理中断或异常的过程中，可以使用硬件压缩寄存器来减少寄存器的保存开销，提高处理效率。

3. 使用异步中断处理：一些 RISC-V 处理器支持异步中断处理，即将中断处理过程分为两个阶段，第一阶段是快速响应，只保存必要的寄存器；第二阶段是完整处理，将所有寄存器保存下来，进行完整的中断处理。这样可以在保证中断处理能够及时响应的同时，减少保存寄存器的开销。