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（其存储在变量 fp 中。mv 指令是将 FP 寄存器的值移动到

FramePointer,FP)，并将 fp 变量中。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（out(reg)

fp 表示将 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：

```
// ...
 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(); }
//...
```

然后我们再启动，可以看到此时已经能打印调用栈了。

```
[kernel] Loading app_3
== Begin stack trace ==
0x00000000802008f2, fp = 0x0000000080206d30
0x0000000080201028, fp = 0x0000000080206d90
0x0000000080201102, fp = 0x0000000080206e00
0x0000000080200ac2, fp = 0x0000000080206ef0
0x000000008020006c, 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 ==
0x00000000802008f2, fp = 0x0000000080206e00
0x0000000080200bd8, fp = 0x0000000080206ef0
0x000000008020006c, fp = 0x0000000080208fc0
0x0000000080400018, fp = 0x0000000080209000
0x0000000000000000, fp = 0x0000000000000000
== End stack trace ==
```

打开(O)

stack_trace.rs
~/lab2/rCore-Tutorial-v3/os/src

stack_trace.rs

```rust
use core::{arch::asm, ptr};

pub unsafe fn print_stack_trace() {
    let mut fp: *const usize;
    unsafe {
        asm!("mv {}, fp", out(reg) fp);
    }
    println!("== Begin stack trace ==");
    while fp != ptr::null() {
        unsafe {
            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 ==");
}
```

编程题 2:

** 扩展内核，实现新系统调用 get_taskinfo，能显示当前 task 的 id 和 task name；实现一个裸机应用程序 B，能访问 get_taskinfo 系统调用

```rust
#![no_std]
#![no_main]

#[macro_use]
extern crate user_lib;

#[unsafe(no_mangle)]

pub fn print_app_info(num_app: usize, app_start: &[usize]) {
    println!("[kernel] num_app = {}", num_app);
    for i in 0..num_app {
        println!(
            "[kernel] app_{} [{:#x}, {:#x})",
            i,
            app_start[i],
            app_start[i + 1]
        );
    }
}
```

编写程序 B，使其能够打印出当前 task 的 id 和 task name。

```
[kernel] num_app = 6
[kernel] app_0 [0x8020b040, 0x8020c048)
[kernel] app_1 [0x8020c048, 0x8020d0f0)
[kernel] app_2 [0x8020d0f0, 0x8020e4a8)
[kernel] app_3 [0x8020e4a8, 0x8020f538)
[kernel] app_4 [0x8020f538, 0x802105c8)
[kernel] app_5 [0x802105c8, 0x80211598)
[kernel] Loading app_0
```

编程题 3：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：

```
void usertrap()
{
    set_kerneltrap();
    struct trapframe *trapframe = curr_proc()->trapframe;

    if ((r_sstatus() & SSTATUS_SPP) != 0)
            panic("usertrap: not from user mode");

    uint64 cause = r_scause();
    if (cause & (1ULL << 63)) {
            cause &= ~(1ULL << 63);
            switch (cause) {
            case SupervisorTimer:
                    tracef("time interrupt!\n");
                    set_next_timer();
                    yield();
                    break;
            default:
                    unknown_trap();
                    break;
            }
    } else {
            switch (cause) {
            case UserEnvCall:
                    trapframe->epc += 4;
                    syscall();
                    break;
            case StoreMisaligned:
            case StorePageFault:
            case InstructionMisaligned:
            case InstructionPageFault:
            case LoadMisaligned:
            case LoadPageFault:
                    printf("%d in application, bad addr = %p, bad instruction = %p, "
                                    "core dumped.\n",
                                    cause, r_stval(), trapframe->epc);
                    exit(-2);
                    break;
            case IllegalInstruction:
                    printf("IllegalInstruction in application, core dumped.\n");
                    exit(-3);
                    break;
            default:
                    unknown_trap();
```

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。

```
                // 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();
 }
```

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。
三 实践作业：
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
例，并得到预期输出（详见测例注释）。
sys_write 安全检查的实现，正确执行目标用户测
注意：如果设置默认 log 等级，从 lab2 开始关闭所有 log 输出。

```
[kernel] Loading app_2
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
Try to execute privileged instruction in U Mode
Kernel should kill this application!
[kernel] IllegalInstruction in application, kernel killed it.
[kernel] Loading app_4
Try to access privileged CSR in U Mode
Kernel should kill this application!
[kernel] IllegalInstruction in application, kernel killed it.
```

可以看到程序 2 成功输出了，但是后面的程序 3，4，5 都没有能够安全测试的检查，被内核所消灭了这些进程。

实践作业：
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
例，并得到预期输出（详见测例注释）。
sys_write 安全检查的实现，正确执行目标用户测
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] 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!
```

```
[kernel] Application exited with code 0
[kernel] Loading app_3
Try to execute privileged instruction in U Mode
Kernel should kill this application!
[kernel] IllegalInstruction in application, kernel killed it.
[kernel] Loading app_4
Try to access privileged CSR in U Mode
Kernel should kill this application!
[kernel] IllegalInstruction in application, kernel killed it.
[kernel] Loading app_5
out of range!
out of range!
out of range!
Test write0 OK!
[kernel] Application exited with code 0
[kernel] Loading app_6
string from data section
strinstring from stack section
strin
Test write1 OK!
```