lab3

编程题

1.扩展内核，能够显示操作系统切换任务的过程。

切换任务在 os/src/task/mod.rs 中的 TASK_MANAGER 中完成的，主要靠 mark_current_suspended() 和 mark_current_exited() 标记任务运行结束，任务开始运行，在其中插入输出即可显示切换任务的过程（下面代码中的加入的三行:

run_next_task 标

println! ）

03sleep.rs 测例会导致频繁切换任务，使得输出过多。可以修改测例的等待时间来减少输出。

```
/// 将当前正在运行的任务的状态更改为“就绪”。
fn mark_current_suspended(&self) {
 let mut inner = self.inner.exclusive_access(); // 获取内部数据的独占访问权
let current = inner.current_task; // 获取当前任务的 ID
 println!("task {} suspended", current); // 打印日志，表明当前任务被暂停
inner.tasks[current].task_status = TaskStatus::Ready; // 将当前任务的状态设置
为“就绪”
 }
 /// 将当前正在运行的任务的状态更改为“已退出”。
fn mark_current_exited(&self) {
 let mut inner = self.inner.exclusive_access(); // 获取内部数据的独占访问权
let current = inner.current_task; // 获取当前任务的 ID
 println!("task {} exited", current); // 打印日志，表明当前任务已经退出
inner.tasks[current].task_status = TaskStatus::Exited; // 将当前任务的状态设置
为“已退出”
 }
 /// 切换当前正在运行的任务到我们找到的下一个任务，
/// 或者如果没有“就绪”任务，则退出并完成所有应用程序。
fn run_next_task(&self) {
 if let Some(next) = self.find_next_task() { // 查找下一个“就绪”任务
let mut inner = self.inner.exclusive_access(); // 获取内部数据的独占访问权
let current = inner.current_task; // 获取当前任务的 ID
 println!("task {} start", current); // 打印日志，表明当前任务已经开始
inner.tasks[next].task_status = TaskStatus::Running; // 将下一个任务的状态设
置为“正在运行”
 inner.current_task = next; // 更新当前任务的 ID 为下一个任务的 ID
 // 获取当前任务和下一个任务的上下文指针
let current_task_cx_ptr = &mut inner.tasks[current].task_cx as *mut
TaskContext;
 let next_task_cx_ptr = &inner.tasks[next].task_cx as *const TaskContext;
 drop(inner); // 释放内部数据的独占访问权
// 在切换任务之前，应该手动释放必须手动释放的本地变量
// 执行不安全的操作，切换到下一个任务的上下文中
unsafe {
 __switch(current_task_cx_ptr, next_task_cx_ptr);
```

```
 }
 // 回到用户模式
} else {
// 如果没有找到“就绪”任务，则退出并完成所有应用程序
println!("All applications completed!");
 use crate::board::QEMUExit;
 crate::board::QEMU_EXIT_HANDLE.exit_success();
 }
 }
```

修改测例的等待时间

gedit user/src/bin/03sleep.rs ：

将等待时间改为 3

```
 let wait_for = current_timer + 3;
```

修改完之后我们运行程序，就能得到结果：

```
5^140000 = 386471875(MOD 998244353)
Test power_5 OK!
[kernel] Application exited with code 0
task 1 exited
task 1 start
task 2 suspended
task 2 start
task 3 suspended
task 3 start
power_3 [90000/200000]
power_3 [100000/200000]
power_3 [110000/200000]
power_3 [120000/200000]
power_3 [130000/200000]
power_3 [140000/200000]
power_3 [150000/200000]
```

可以看到此时已经能看到程序的切换过程了。

2.编写应用程序或扩展内核，能够统计任务切换的大致开销。

所有任务切换都通过 task/mod.rs 中的 __switch ，可以包装一下这个函数，统计它运行的开销。首先删除 use switch::__switch ，然后加入以下函数来代替

```
/// 切换的开始时间
static mut SWITCH_TIME_START: usize = 0;
 /// 切换的总时间
static mut SWITCH_TIME_COUNT: usize = 0;
```

__switch ：

```
// 安全的切换函数
unsafe fn __switch(current_task_cx_ptr: *mut TaskContext, next_task_cx_ptr:
*const TaskContext) {
 SWITCH_TIME_START = get_time_us(); // 获取切换的开始时间
switch::__switch(current_task_cx_ptr, next_task_cx_ptr); // 调用底层切换函数
SWITCH_TIME_COUNT += get_time_us() - SWITCH_TIME_START; // 累加切换的总时间
}
 // 获取切换的总时间
fn get_switch_time_count() -> usize {
 unsafe { SWITCH_TIME_COUNT } // 使用 unsafe 代码块读取全局变量
}
```

小心

这里统计时间使用了一个

get_time_us ，即计算当前的微秒数。这是因为任务切换的时间比较短，不好用毫秒来计数。对应的实现在

timer.rs 中：

```
const USEC_PER_SEC: usize = 1000000; // 常量：一秒钟的微秒数
/// 获取当前时间（微秒）函数
pub fn get_time_us() -> usize {
 time::rad() / (CLOCK_FREQ / USEC_PER_SEC) // 获取时间戳并转换成微秒
}
```

```rust
//use switch::__switch;
use task::{TaskControlBlock, TaskStatus};

/// 切换的开始时间
static mut SWITCH_TIME_START: usize = 0;
 /// 切换的总时间
static mut SWITCH_TIME_COUNT: usize = 0;
 __switch：
// 安全的切换函数
unsafe fn __switch(current_task_cx_ptr: *mut TaskContext, next_task_cx_ptr:
*const TaskContext) {
 SWITCH_TIME_START = get_time_us(); // 获取切换的开始时间
switch::__switch(current_task_cx_ptr, next_task_cx_ptr); // 调用底层切换函数
SWITCH_TIME_COUNT += get_time_us() - SWITCH_TIME_START; // 累加切换的总时间
}
 // 获取切换的总时间
fn get_switch_time_count() -> usize {
 unsafe { SWITCH_TIME_COUNT } // 使用unsafe代码块读取全局变量
}
```

需要注意的是对于 Rust 2024 之后，已经声明了函数为 unsafe fn，但是函数体内的危险操作（如对 mutable static 变量的读写和调用不安全函数）仍然需要显式包裹在 unsafe { … } 块中。所以我们还需要在 switch_time_start 前面套上一层 unsafe：

```
unsafe fn __switch(current_task_cx_ptr: *mut TaskContext, next_task_cx_ptr
 unsafe {
        SWITCH_TIME_START = get_time_us();
    }

    // 调用不安全函数需要使用 unsafe 块
    unsafe {
        switch::__switch(current_task_cx_ptr, next_task_cx_ptr);
    }

    // 再次使用 unsafe 块对 mutable static 进行操作
    unsafe {
        SWITCH_TIME_COUNT += get_time_us() - SWITCH_TIME_START;
    }
}
fn get_switch_time_count() -> usize {
    unsafe { SWITCH_TIME_COUNT }
}
```

最后，在 run_next_task 中所有程序退出后，增加一条输出语句即可：

```
            // go back to user mode
        } else {
            println!("All applications completed!");
            println!("task switch time: {} us", get_switch_time_count());
            shutdown(false);
```

我们可以 run 一下看看结果：

```
task 3 start
Test sleep OK!
[kernel] Application exited with code 0
task 3 exited
All applications completed!
task switch time: 11 us
```

问答题

1.协作式调度与抢占式调度的区别是什么？

协作式调度中，进程主动放弃 (yield) 执行资源，暂停运行，将占用的资源让给其它进程；抢占式调度中，进程会被强制打断暂停，释放资源让给别的进程。

2.中断、异常和系统调用有何异同之处？

相同点
都会从通常的控制流中跳出，进入 trap handler 进行处理。
不同点
中断的来源是异步的外部事件，由外设、时钟、别的 hart 等外部来源，与 CPU 正在做什么没关系。异常是 CPU 正在执行的指令遇到问题无法正常进行而产生的。系统调用是程序有意想让操作系统帮忙执行一些操作，用专门的指令（如 ecall ）触发的。

3.RISC-V 支持哪些中断/异常？

见下图

Interrupt	Exception Code	Description
1	0	User software interrupt
1	1	Supervisor software interrupt
1	2	*Reserved for future standard use*
1	3	Machine software interrupt
1	4	User timer interrupt
1	5	Supervisor timer interrupt
1	6	*Reserved for future standard use*
1	7	Machine timer interrupt
1	8	User external interrupt
1	9	Supervisor external interrupt
1	10	*Reserved for future standard use*
1	11	Machine external interrupt
1	12–15	*Reserved for future standard use*
1	≥16	*Reserved for platform use*
0	0	Instruction address misaligned
0	1	Instruction access fault
0	2	Illegal instruction
0	3	Breakpoint
0	4	Load address misaligned
0	5	Load access fault
0	6	Store/AMO address misaligned
0	7	Store/AMO access fault
0	8	Environment call from U-mode
0	9	Environment call from S-mode
0	10	*Reserved*
0	11	Environment call from M-mode
0	12	Instruction page fault
0	13	Load page fault
0	14	*Reserved for future standard use*
0	15	Store/AMO page fault
0	16–23	*Reserved for future standard use*
0	24–31	*Reserved for custom use*
0	32–47	*Reserved for future standard use*
0	48–63	*Reserved for custom use*
0	≥64	*Reserved for future standard use*

4.如何判断进入操作系统内核的起因是由于中断还是异常？

检查 mcause 寄存器的最高位，1 表示中断，0 表示异常。
当然在 Rust 中也可以直接利用 riscv 库提供的接口判断:

```
let scause = scause::read();
    if scause.is_interrupt() {
        do_something
    }
    if scause.is_exception() {
        do_something
    }
```

又或者，可以按照 trap/mod.rs:trap_handler() 中的写法，用 match scause.cause() 来判断。

5.在 RISC-V 中断机制中，PLIC 和 CLINT 各起到了什么作用?

CLINT 处理时钟中断（MTI）和核间的软件中断（MSI）；PLIC 处理外部来源的中断（PLIC 的规范文档： https://github.com/riscv/riscv-plic-spec

6.基于 RISC-V 的操作系统支持中断嵌套？请给出进一步的解释说明。

RISC-V 原生不支持中断嵌套。(在 S 态的内核中)只有 sie 寄存器控制哪些中断可以触发。触发中断时， sstatus 的 SIE 位为 1 时，才会开启中断，再由 sstatus.SIE 置为 0；当执行 1。这意味着触发中断时，因为 sstatus.SPIE 置为 sret 时， sstatus.SIE ，而 sstatus.SIE 置为 sstatus.SPIE ，而 sstatus.SPIE 置为 sstatus.SIE 为 0，所以无法再次触发中断。

7.本章提出的任务的概念与前面提到的进程的概念之间有何区别与联系?

联系：任务和进程都有自己独立的栈、上下文信息，任务是进程的“原始版本”，在第五章会将目前的用户程序从任务升级为进程。
区别：任务之间没有地址空间隔离，实际上是能相互访问到的；进程之间有地址空间隔离，一个进程无法访问到另一个进程的地址。

8.简单描述一下任务的地址空间中有哪些类型的数据和代码。

可参照
user/src/linker.ld ：

.text ： 任务的代码段，其中开头打印的字符串存在这里
.text.entry 段包含任务的入口地址
.rodata ： 只读数据，包含字符串常量
.data ： 需要初始化的全局变量
.bss ： 未初始化或初始为 0 的全局变量。
在之后第四章的 user/src/bin/00power_3.rs 中，会把第三章中在用户栈上定义的数组移到全局变量中 static mut S: [u64; LEN] = [0u64; LEN];
在第五章的 user/lib.rs 中，会在 bss 段构造一个用户堆 static mut HEAP_SPACE: [u8; USER_HEAP_SIZE] = [0; USER_HEAP_SIZE];
os/src/loader.rs:USER_STACK 也属于各自任务的地除此之外，在内核中为每个任务构造的用户栈 址。

9.任务控制块保存哪些内容?

在本章中，任务控制块即 os/src/task/task.rs:TaskControlBlock 保存任务目前的执行状态 task_status 和任务上下文 task_cx 。

10.任务上下文切换需要保存与恢复哪些内容?

需要保存通用寄存器的值，PC; 恢复的时候除了保存的内容以外还要恢复特权级到用户态。

11.特权级上下文和任务上下文有何异同?

相同点：特权级上下文和任务上下文都保留了一组寄存器，都代表一个“执行流”
不同点：
特权级上下文切换可以发生在中断异常时，所以它不符合函数调用约定，需要保存所有通用寄存器。同时它又涉及特权级切换，所以还额外保留了一些 CSR，在切换时还会涉及更多的 CSR。任务上下文由内核手动触发，它包装在 os/src/task/switch.rs:__switch() 里，所以除了“返回函数与调用函数不同”之外，它符合函数调用约定，只需要保存通用寄存器中 callee 类型的寄存器。为了满足切换执行流时“返回函数与调用函数不同”的要求，它还额外保存
ra 。

12.上下文切换为什么需要用汇编语言实现?

上下文切换过程中，需要我们直接控制所有的寄存器。C 和 Rust 编译器在编译代码的时候都会“自作主张”使用通用寄存器，以及我们不知道的情况下访问栈，这是我们需要避免的。切换到内核的时候，保存好用户态状态之后，我们将栈指针指向内核栈，相当于构建好一个高级语言可以正常运行的环境，这时候就可以由高级语言接管了。

13.有哪些可能的时机导致任务切换?

系统调用（包括进程结束执行）、时钟中断。

14.在设计任务控制块时，为何采用分离的内核栈和用户栈，而不用一个栈?

用户程序可以任意修改栈指针，将其指向任意位置，而内核在运行的时候总希望在某一个合法的栈上，所以需要用分开的两个栈。
此外，利用后面的章节的知识可以保护内核和用户栈，让用户无法读写内核栈上的内容，保证安全。

实践作业：

获取任务信息

ch3 中，我们的系统已经能够支持多个任务分时轮流运行，我们希望引入一个新的系统调用 sys_task_info 以获取任务的信息，定义如下：

fn sys_task_info(id: usize, ts: *mut TaskInfo) -> isize

syscall ID: 410

根据任务 ID 查询任务信息，任务信息包括任务 ID、任务控制块相关信息（任务状态）、任务使用

的系统调用及调用次数、任务总运行时长。

```
struct TaskInfo {
 id: usize,
 status: TaskStatus,
 call: [SyscallInfo; MAX_SYSCALL_NUM],
 time: usize
 }
```

系统调用信息采用数组形式对每个系统调用的次数进行统计，相关结构定义如下：

```
struct SyscallInfo {
 id: usize,
 times: usize
 }
```

参数：

id: 待查询任务 idts: 待查询任务信息

返回值：执行成功返回 0，错误返回-1.

运行结果：

```
[kernel] Application exited with code 0
task 0 exited
task 0 start
power_7 [130000/160000]
power_7 [140000/160000]
power_7 [150000/160000]
power_7 [160000/160000]
7^160000 = 667897727(MOD 998244353task 2 suspended
task 2 start
)
Test power_4 OK!
 id: 136,
    times: 7
```