K210 移植与多核支持总结报告

小问题

在撰写本报告的时候，我又想到了某些实现有一点小问题，先记录下来：

1. CPU 核数、频率以及内存大小都可以通过设备树查询，目前都是硬编码；
2. K210 串口中断处理利用 mstatus.mprv 可以做到更好；
3. 将 k210-sdk-stuff/k210-hal/k210-pac 移出项目，不然代码行数太夸张了；
4. 如何处理 icache 刷新；
5. 所有线程退出后 panic 的机制；
6. k210 忘记在时钟中断里面 run_current_thread_later 为何也能跑？

实验结果

目前在代码可以在这里找到，其中的 README.md 提到了如何运行 tutorial。Qemu 平台与原先保持不变，K210 平台则需要进行如下步骤：

1. 准备好 microSD 卡用于放置文件系统（可以在这里购买，质量不过关的 SD 卡会引起很多不必要的麻烦）

   将 microSD 卡放入读卡器并插入 PC，随后（目前只支持 Ubuntu 或虚拟机）

   ```
   cd user && make sdcard
   ```

2. 将 microSD 卡插入 K210 开发板
3. 切换到主目录，并 `make run BOARD=k210`。

在启动之后会进入用户终端，键入用户程序的名称即可运行该程序。目前支持三个用户程序：

1. hello_world
2. notebook[特别地，通过 Ctrl + C 可以退出]
3. fantastic_text

在每个用户程序运行结束后，会给出其主线程的运行统计信息，即其在每个核的用户态和内核态各运行了多长时间。单位是 CPU 的时钟周期。如：

```
thread 4 exit with code 0
on Core #0, user time = 11915, kernel time = 13772
on Core #1, user time = 10892, kernel time = 29879
total user = 22807, kernel = 43651, sum = 66458
Process 3 exited
```

若要退出整个 OS，Qemu 平台可以通过 Ctrl A + X，K210 平台则可以通过 Ctrl +]。

目前的多核仅支持到原 master 分支，即还不支持 lab3+ 的虚拟存储。但是单核情况下，已经能在 K210 上跑 lab3+ 了，参见这里。

下面，我们分别介绍一下相关的改动：

将 OpenSBI 换成 RustSBI

RustSBI 是在 2020 年 OS 夏令营活动中完善的一个项目。它基于已有的 Rust 嵌入式生态用 Rust 语言重写 OpenSBI，目前我们所用到的 OpenSBI 中的功能 RustSBI 都已经实现了，还有一些相比 OpenSBI 更为强大的功能（如软中断、向后兼容 S 特权级软件等），且更加轻量级。至少在 Qemu 和 K210 平台上基于 RustSBI 跑我们的 Tutorial 已经不成问题，于是我们全面将 Tutorial 中的 OpenSBI 换成 RustSBI。

我们将预编译好的镜像 `rustsbi-qemu.bin` 和 `rustsbi-k210.bin` 保存在项目的 `bootloader` 目录下，在启动时使用。

多平台架构

为了能够同时支持 Qemu/K210 两个平台，新增 `board` 目录保存两个平台硬件上不同的地方：

```
board/
    - k210/
        - mod.rs
        - config.rs
        - entry.asm
        - interrupt.rs
        - linker.ld
    - qemu/
        - mod.rs
        - config.rs
        - entry.asm
        - interrupt.rs
        - linker.ld
    - mod.rs
```

下面我们分别来看看它们的用途：

`config.rs` 中的内容（以 K210 平台）为例：

```
// 硬编码的可用物理地址结尾
pub const BOARD_MEMORY_END_ADDRESS: usize = 0x8060_0000;
// 适应当前平台的内核堆大小
pub const BOARD_KERNEL_HEAP_SIZE: usize = 0x20_0000;
// 适应当前平台的运行栈大小
pub const BOARD_STACK_SIZE: usize = 0x8000;
// 适应当前平台的内核栈大小
pub const BOARD_KERNEL_STACK_SIZE: usize = 0x8000;
// 所有需要用到的外设 MMIO 地址区间
// 值得一提的是，我们会在 MemorySet::new_kernel 的时候插入相应的映射
pub const MMIO_INTERVALS: &[(usize, usize)] = &[
    (0x0200_0000, 0x1000), /* CLINT      */
    (0x0C00_0000, 0x1000), /* PLIC       */
    ...
    (0x5400_0000, 0x1000), /* SPI2       */
];
// CPU 实现的 RISC-V 规范版本
// 目前唯一用到的地方是 1.9 和 1.10 一个显著的差异
// 即 sstatus.sum 和 sstatus.pum 意义完全相反
// 需要根据版本来判断是否设置这一位
pub const RISCV_SPEC_MAJOR: usize = 1;
pub const RISCV_SPEC_MINOR: usize = 9;
pub const RISCV_SPEC_PATCH: usize = 1;
// 核心数，用于多核
```

```
pub const CPU_NUM: usize = 2;
// CPU 频率，目前直接硬编码
pub const CPU_FREQUENCY: usize = 390000000;
```

`entry.asm` 和 `linker.ld` 则会在构建的时候将当前构建平台的版本复制到 `os/src/entry.asm` 和 `os/src/linker.ld` 用于实际的构建。`entry.asm` 中只有为每个核设置的启动栈大小不同，以及启动栈的数量（等于核数）不同；而 `linker.ld` 中则是内核的起始虚拟地址 `BASE_ADDRESS` 不同。

`interrupt.rs` 分别保存了两个平台上不同的中断服务例程。在 K210 平台上，由于该平台奇怪的属性，`supervisor_external` 的功能被移交给 `supervisor_soft`。除此之外二者没有不同。

`mod.rs` 实现了两份 `device_init` 函数，因为两个平台在这一阶段要做的事情并不相同。

`board/mod.rs` 则会将这些设置根据不同的目标平台 export 出来给 Tutorial 的其他模块使用。

```
#[cfg(feature = "board_k210")]
pub use k210::{config, device_init, interrupt};
#[cfg(feature = "board_qemu")]
pub use qemu::{config, device_init, interrupt};
```

比如，当目标平台是 K210 的时候，`crate::board::config::*` 就会是 K210 的配置，反之则会是 Qemu 的配置。

要让用于区分的两个 feature `board_k210, board_qemu` 发挥作用，首先我们要在 `Cargo.toml` 中进行声明：

```
[features]
board_qemu = []
board_k210 = []
```

接下来 `Makefile` 中也要加以改动：

```
# 编译 kernel 的时候复制 linker.ld 和 entry.asm 并加上相应 feature
kernel:
    @cp src/board/$(BOARD)/linker.ld src/linker.ld
    @cp src/board/$(BOARD)/entry.asm src/entry.asm
    @cargo build --features "board_$(BOARD)"

run: build run-inner
run-inner: build
# 判断当前的目标平台
ifeq ($(BOARD),qemu)
    @qemu-system-riscv64 \
            -machine virt \
            -nographic \
            # 这里换成 RustSBI
            -bios $(BOOTLOADER) \
            -device loader,file=$(BIN_FILE),addr=0x80200000 \
            -drive file=$(IMG_FILE),format=qcow2,id=sfs \
            -device virtio-blk-device,drive=sfs \
            # 开启多核
            -smp 4
else
    # K210 一键运行，后面再说
endif
```

由此，即可 `make run` 跑 Qemu，`make run BOARD=k210` 跑 K210。

K210 移植

感谢 @freheit889 的工作，让很多坑得以提前被解决！

下述的移植代码，虽然是为了能正常在 K210 上跑起来，但是在 Qemu 上也是用的同样的代码。

本节的内容截止到能单核在 K210 上跑 lab1~lab6 和 lab3+ 为止，多核还会有一些变化。

启动流程

还是从 `os/Makefile` 的那个地方来看：

```
SBI         ?= rustsbi
BOOTLOADER  := ../bootloader/$(SBI)-$(BOARD).bin
# 目前仅支持 Ubuntu，默认为 /dev/ttyUSB0
K210-SERIALPORT = /dev/ttyUSB0
K210-BURNER     = ../tools/kflash.py

run-inner: build
ifeq ($(BOARD),qemu)
    # 运行 Qemu
else
    # 将 RustSBI 镜像与 kernel 拼接在一起
    @cp $(BOOTLOADER) $(BOOTLOADER).copy
    @dd if=$(BIN_FILE) of=$(BOOTLOADER).copy bs=128K seek=1
    @mv $(BOOTLOADER).copy $(BIN_FILE)
    # 给 k210 的串口设备必要的权限
    @sudo chmod 777 $(K210-SERIALPORT)
    # 通过工具 kflash.py 将内核镜像烧写到 k210 的 flash 内，并启动串口终端
    # 注意，此工具依赖于 pyserial
    python3 $(K210-BURNER) -p $(K210-SERIALPORT) -b 1500000 -t $(BIN_FILE)
endif
```

烧写的速度大概只有 30~40KiB 每秒，因此我们将 K210 平台上的 `BASE_ADDRESS` 调整为 `0xFFFF_FFFF_8002_0000` 正是为了缩小最后的内核镜像大小并提高烧写速度。

清空 .bss section

在裸机平台上，清空 .bss 段是一项很重要的工作，我们要在访问任何零初始化的全局变量之前完成它。

```
// memory/mod.rs
fn clear_bss() {
    extern "C" {
        fn sbss_clear();
        fn ebss_clear();
    }
    let bss_start = sbss_clear as usize;
    let bss_end = ebss_clear as usize;
    let bss_aligned = bss_end - bss_end % 8;
    // clear bss section
    (bss_start..bss_end).step_by(8).for_each(|p| {
        unsafe { (p as *mut u64).write_volatile(0) }
    });
    if bss_aligned < bss_end {
        (bss_aligned..bss_end).step_by(1).for_each(|p| {
```

```
            unsafe { (p as *mut u8).write_volatile(0) }
        });
    }
}
```

我们也需要给出要初始化的部分的符号定义：

```
# linker.ld
bss_start = .;
    .bss : {
        *(.bss.stack)
        sbss_clear = .;
        *(.sbss .bss .bss.*)
        ebss_clear = .;
    }
```

注意我们不能将整个 .bss 段清空，那样会错误的将启动栈清空。

MMU 相关

初始内核映射中新增了以下大页：

```
// entry.asm
boot_page_table:
+    .8byte (0x00000 << 10) | 0xcf
+    .8byte (0x40000 << 10) | 0xcf
     .8byte (0x80000 << 10) | 0xcf
     .zero 505 * 8
     .8byte (0x00000 << 10) | 0xcf
+    .8byte (0x40000 << 10) | 0xcf
     .8byte (0x80000 << 10) | 0xcf
     .8byte 0
```

这是为了无论用虚拟地址还是物理地址都能访问到所有外设的设备寄存器（K210 上某些外设的 MMIO 地址在 0x4000_0000 之后）。

确定了 K210 平台所有能用到的设备 MMIO 区间：

```
// board/k210/config.rs
pub const MMIO_INTERVALS: &[(usize, usize)] = &[
    (0x0200_0000, 0x1000), /* CLINT      */
    (0x0C00_0000, 0x1000), /* PLIC       */
    ...
    (0x5400_0000, 0x1000), /* SPI2       */
];
```

并在创建进程地址空间的时候映射过去：

```
// memory/mapping/memory_set.rs MemorySet::new_kernel
for mmio_pair in MMIO_INTERVALS.iter() {
    segments.push(
        Segment {
            map_type: MapType::Device,
            range:
Range::from(VirtualAddress(mmio_pair.0)..VirtualAddress(mmio_pair.0 +
mmio_pair.1)),
            flags: Flags::READABLE | Flags::WRITABLE,
        }
    );
}
```

这里的 `MapType::Device` 是一种新增的映射类型，可看作是一种偏移量为 0 的 `MapType::Linear`。提供这种映射的原因是 K210 的设备驱动库直接访问物理地址而非虚拟地址。

```
// memory/mapping/mapping.rs Mapping::map
MapType::Device => {
    for vpn in segment.page_range().iter() {
        self.map_one(vpn, Some(PhysicalPageNumber(vpn.0)), segment.flags |
Flags::VALID)?;
    }
}
```

根据 RISC-V 规范版本决定要不要设置 status.sum 位。

```
// memory/mod.rs
if RISCV_SPEC_MINOR >= 10 {
    println!("riscv spec version >= 1.10!");
    unsafe { riscv::register::sstatus::set_sum() };
}
```

由于 K210 上的 1.9.1 版本不区分 LoadPageFault 和 LoadFault 两个异常，而是只有一种 LoadFault 异常（Store/Instruction 同理）。我们需要在 `handle_interrupt` 中加入相应分支处理缺页异常：

```
// interrupt/handler.rs
        | Trap::Exception(Exception::LoadFault)
        | Trap::Exception(Exception::StoreFault)
        | Trap::Exception(Exception::InstructionFault) => page_fault(context,
scause, stval),
```

刷新 icache

在 `main.rs` 即将通过 `__restore` 切换到第一个线程之前刷新了一下 icache：

```
llvm_asm!("fence.i" :::: "volatile");
```

理论上来说在内存中写入可能会被取指的部分之后确实应该刷新一下 icache，而且曾经确实遇到过由于没有刷新 icache 导致取到错误指令的情况。但是现在好像把它删除掉也没问题。麻烦的事情是，如果想自圆其说，在每个应该刷新的地方都刷新一下是相当复杂的。所以这个家伙到底如何处理还需要讨论一下。

串口中断

之前提到过，K210 的 S 特权级软中断和外部中断处理正好调换了一下。这是由于 K210 的硬件缺陷，S 特权级外部中断不存在。因此我们只能在 RustSBI 的 M 特权级外部中断中进行处理并转发给 S 特权级软中断。这使得 K210 的中断处理看起来非常奇怪。

```
// board/k210/interrupt.rs
// just return
pub fn supervisor_external(context: &mut Context) -> *mut Context {
    context
}
pub fn supervisor_soft(context: &mut Context) -> *mut Context {
    // RustSBI: read character from serial and save it into stval CSR
    let mut c = stval::read();
    // just like supervisor_external on Qemu platform
    if c <= 255 {
        if c == '\r' as usize {
            c = '\n' as usize;
        }
        STDIN.push(c as u8);
    }
    // clear SSIP
    unsafe {
        let mut _sip: usize = 0;
        llvm_asm!("csrci sip, 1 << 1" : "=r"(_sip) ::: "volatile");
    }
    context
}
```

为了和 Qemu 一样只在 hart0 上进行串口中断的接收与处理，我们要在初始化的时候做如下工作：

```
// board/k210/mod.rs
pub fn device_init(_: PhysicalAddress) {
    // after RustSBI, txen = rxen = 1, rxie = 1, rxcnt = 0
    // start UART interrupt configuration
    // disable external interrupt on hart1 by setting threshold
    let hart0_m_threshold: *mut u32 = 0x0c20_0000 as *mut u32;
    let hart1_m_threshold: *mut u32 = 0x0c20_2000 as *mut u32;
    unsafe {
        hart0_m_threshold.write_volatile(0u32);
        hart1_m_threshold.write_volatile(1u32);
    }
    // now using UARTHS whose IRQID = 33
    // assure that its priority equals 1
    let uarths_irq_priority: *mut u32 = (0x0c00_0000 + 33 * 4) as *mut u32;
    assert_eq!(unsafe{ uarths_irq_priority.read_volatile() }, 1);
    // open interrupt enable register on PLIC
    let hart0_m_int_enable_hi: *mut u32 = 0x0c00_2004 as *mut u32;
    unsafe {
        hart0_m_int_enable_hi.write_volatile(1 << 0x1);
    }
```

```
    // now, we can receive UARTHS interrupt on hart0!
    // 接下来要对 sd 卡进行初始化，会修改某些 PLL 的频率进而串口的频率会受到影响
    // 我们需要等待一段时间确保串口缓冲内的字符全部输出，不然会出现乱码
    crate::drivers::soc::sleep::usleep(1000000);
}
```

SDCard 驱动

主要来自以下项目：k210-sdk-stuff。新增的 `drivers/soc` 目录以及 `drivers/block/sdcard.rs` 都修改自于该项目。此外，k210-sdk-stuff 还依赖于 k210-pac 和 k210-hal 两个库。由于它们都需要修改，所以目前是把这两个库直接丢进 Tutorial 项目里面了。可以在项目根目录的 `tools` 目录下找到它们。

不谈硬件的细节，`sdcard.rs` 提供了一个 `Sd_card` 类并实现了 `BlockDevice` 接口。因而在 `fs/mod.rs` 中，我们这样来初始化文件系统：

```
#[cfg(feature = "board_k210")]
lazy_static! {
    /// 根文件系统的根目录的 INode
    pub static ref ROOT_INODE: Arc<dyn INode> = {
        let device = Arc::new(Sd_card::new());
        let sfs = SimpleFileSystem::open(device).expect("failed to open SFS");
        sfs.root_inode()
    };
}
```

其他部分和 Qemu 均保持一致。

用户态终端

首先要新增一个系统调用 `sys_exec` 表示启动一个新进程并**永久阻塞当前进程**（这个后面还会修改）

```
// kernel/process.rs
pub (super) fn sys_exec(path:*const u8,context:Context)->SyscallResult{

    let name=unsafe{from_cstr(path)};
    let app = ROOT_INODE.find(name);
    match app{
        Ok(inode)=>{
            let data = inode.readall().unwrap();
            let elf = ElfFile::new(data.as_slice()).unwrap();
            let process = Process::from_elf(&elf, true).unwrap();
            let thread=Thread::new(process, elf.header.pt2.entry_point() as
usize, None).unwrap();
            PROCESSOR.lock().add_thread(thread);
            PROCESSOR.lock().sleep_current_thread();
            PROCESSOR.lock().park_current_thread(&context);
            PROCESSOR.lock().prepare_next_thread();
        },
        Err(_)=>{
            println!("");
            println!("command not found");
        }
    }
    SyscallResult::Proceed(0)
}
```

```
pub unsafe fn from_cstr(s:*const u8)->&'static str{
    use core::{slice,str};
    let len=(0usize..).find(|&i| *s.add(i)==0).unwrap();
    str::from_utf8(slice::from_raw_parts(s,len)).unwrap()
}
```

其他地方也要对应修改：

```
// kernel/syscall.rs
pub const SYS_EXEC: usize = 221;

SYS_EXEC => sys_exec(args[0] as *const u8, *context),
```

用户态的改动则比较简单，可自行去了解。

值得一提的是，在 K210 平台上内存资源宝贵，因此在构建用户程序的时候使用 release 模式并扔掉调试相关的段：

```
build: dependency
ifeq ($(MODE),release)
    @cargo build --release
else
    @cargo build
endif
    $(foreach bin_file,$(BIN_FILES),riscv64-unknown-elf-objcopy --strip-debug
$(bin_file) $(bin_file);)
```

多核支持

架构微调

将 `condvar.rs` 放在一个新的名为 `sync` 的模块中。

并将原来名为 `kernel` 的模块重命名为 `syscall`，专注于系统调用。

设置启动栈

```
# board/k210/entry.asm
_start:
    # RustSBI 会将当前的核 id 保存在 a0 寄存器中
    # 每个核启动栈栈顶地址计算公式：栈区域开头地址+(核ID+1)*启动栈大小
    add t0, a0, 1
    # K210 上每个启动栈大小为 64KiB
    slli t0, t0, 16
    lui sp, %hi(boot_stack)
    add sp, sp, t0
```

获取当前核 ID

将核 ID 保存在 tp 寄存器中，且在整个运行过程中不应该变化。

```
# board/k210/entry.asm
_start:
    mv tp, a0
    ...
```

之后我们可以读取 tp 寄存器来获取当前的核 ID：

```
// process/processor.rs
pub fn hart_id() -> usize {
    let mut hartid: usize = 0;
    unsafe {
        llvm_asm!("mv $0, tp" : "=r"(hartid) ::: "volatile");
    }
    hartid
}
```

但是需要注意的是在中断上下文保存/恢复的时候 tp(x4) 不能被覆盖，为方便实现，我们将 gp(x3) 也一并忽略。

```
# interrupt/interrupt.asm

SAVE    x1, 1
# 将本来的栈地址 sp（即 x2）保存
csrr    x1, sscratch
SAVE    x1, 2
# 保存 x5 至 x31，而忽略 x3 和 x4
.set    n, 5
.rept   27
SAVE_N  %n
.set    n, n + 1
.endr
# 恢复的时候同理
```

多核启动

和 RustSBI 的作者讨论之后，决定用如下这种方式进行多核启动：

hart0 最先从 RustSBI 正常跳转到内核所在位置开始执行内核代码，而其他核被卡在 RustSBI 里面等待一个软中断。当 hart0 完成内核的初始化工作之后，通过向其他 hart 发送软中断唤醒那些 hart，于是它们可以通过 RustSBI 来进入内核。

> 之前在内核内使用一个全局的原子变量来在 hart0 初始化完成之前卡住其他 hart，但是它是放在 .bss 段的，在裸机环境下，其他 hart 会在 hart0 将 .bss 段清零之前就访问这个原子变量，产生错误的结果。

也就是说，内核的 `rust_main` 大概长成这个样子：

```
// just pseudocode
if hartid == 0 {
    global_initialization();
    for i in 1..CPU_NUM {
        send_ipi(i);
    }
}
per_hart_initialization();
...
```

其中 `send_ipi` 是 RustSBI 提供的一个 SBI 接口，传入的参数是一个 mask 所在的虚拟地址，而 mask 从低位到高位每一位代表一个 hart，最低位对应 hart0，次低位对应 hart1，以此类推。若 mask 某一位是 1 的话，则代表给对应的 hart 发一个 M 态软中断。

因此，在 Tutorial 中 hart0 是这样唤醒其他 hart 的：

```
// main.rs
for i in 1..CPU_NUM {
    let mask: usize = 1 << i;
    sbi::send_ipi(&mask as *const _ as usize);
}
```

接下来介绍全局初始化和局部初始化分别做哪些事情。

全局初始化 `global_initialization` 如下：

```
// main.rs
// 内存全局初始化
memory::bsp_init();
// 设备初始化
crate::board::device_init(dtb_pa);
// 初始化文件系统
fs::init();
// 将用户终端线程放入线程池
THREAD_POOL
    .lock()
    .add_thread(create_user_process("user_shell"));
```

局部初始化 `local_initialization` 如下：

```
// 内存局部初始化
memory::thread_local_init();
// 中断初始化
interrupt::init();
// 获取 idle 线程中断上下文，这里后面会讲
let context = processor_main();
unsafe {
    llvm_asm!("fence.i" :::: "volatile");
    // 切换到 idle 线程
    __restore(context as usize);
}
```

其中：

```
// 内存全局初始化
```

```rust
pub fn bsp_init() {
    global_init();
    thread_local_init();
}
// 清零 .bss 段以及初始化内核堆
fn global_init() {
    clear_bss();
    heap::init();
    println!("mod memory initialized");
}
// 内存局部初始化
pub fn thread_local_init() {
    // 允许内核读写用户态内存
    // 取决于 CPU 的 RISC-V 规范版本就行处理
    if RISCV_SPEC_MINOR >= 10 {
        unsafe { riscv::register::sstatus::set_sum(); }
    }
}
```

多核调度

目前的设计是：多个核在初始化之后运行 idle 线程，它们共享一个底层的调度队列。当调度队列中有线程可用时，哪个核能够最先获取到锁，该线程就会在哪个核上运行。

原先的 `Processor` 设计上既包括了单核目前的运行状态，也将调度队列包括了进去。因此我们要将调度队列的部分拆分出去变成一个新的类 `ThreadPool`：

```rust
// process/thread_pool.rs
pub struct ThreadPool {
    pub scheduler: SchedulerImpl<Arc<Thread>>,
    pub sleeping_threads: HashSet<Arc<Thread>>,
}
// 全局共享的调度队列
lazy_static! {
    pub static ref THREAD_POOL: Mutex<ThreadPool> =
Mutex::new(ThreadPool::default());
}
```

它要实现的功能只涉及到线程在就绪和休眠队列之间的转移、或是单纯的移入移出。接口如下：

```rust
// process/thread_pool.rs
impl ThreadPool {
    // 新增线程，就绪队列+
    pub fn add_thread(&mut self, thread: Arc<Thread>) {
        self.scheduler.add_thread(thread);
    }
    // 线程结束，就绪队列-
    pub fn kill_thread(&mut self, thread: Arc<Thread>) {
        self.scheduler.remove_thread(&thread);
    }
    // 唤醒线程，休眠队列->就绪队列
    pub fn wake_thread(&mut self, thread: Arc<Thread>) {
        thread.inner().sleeping = false;
        self.sleeping_threads.remove(&thread);
        self.scheduler.add_thread(thread);
    }
```

```
    // 休眠线程，就绪队列->休眠队列
    pub fn sleep_thread(&mut self, thread: Arc<Thread>) {
        thread.inner().sleeping = true;
        self.sleeping_threads.insert(thread);
    }
}
```

注意到，`sleep_thread` 里面并没有和 `wake_thread` 一样对称的把线程从就绪队列中移除。这是因为此时该线程已经从就绪队列中移除了。事实上，我们需要做到对于每个线程，它只能出现在某个核的 `current_thread`、就绪队列、休眠队列三者其中之一。也就是说，当一个线程即将被换入执行的时候，在 `prepare_next_thread` 里面它已经被从就绪队列转移到某个核的 `current_thread` 了，之后在就绪队列中你将不会找到它。

故此，调度器（就绪队列）也不能在 `get_next` 的时候将取出的线程再放回队尾了。因为这样同样会导致线程同时出现在某个核的 `current_thread` 以及就绪队列中。后面我们会提到这种情况下该线程如何回到就绪队列。另外，由于先前的某些 bug，将 `FifoScheduler` 内部的数据结构从 `LinkedList` 换成了 `VecDeque`，不过事后证明应该不是数据结构选用的问题。应该可以换回来。

> 在多核情况下之前的实现会导致，一个线程已经在某个核上执行了，却又能被其他核抢到，就是因为该线程仍能在就绪队列中找到。

将调度队列的内容分离出去之后，现在 `Processor` 仅仅描述某个核当前的运行状态：

```
// process/processor.rs
pub struct Processor {
    /// 当前正在执行的线程
    current_thread: Option<Arc<Thread>>,
    /// idle 线程，每个核独立
    idle_thread: Arc<Thread>,
}
```

它的接口主要与当前正在执行的线程有关：

```
// process/processor.rs
impl Processor {
    /// 获取一个当前线程的 `Arc` 引用
    /// 保持不变
    pub fn current_thread(&self) -> Arc<Thread> {
        let thread = self.current_thread.as_ref().unwrap().clone();
        thread
    }
    /// 新增，表示将一个线程控制块内的中断上下文取出并放到当前核的内核栈上
    /// 返回操作后内核栈的栈顶
    /// 这里只是作为一个 shortcut
    pub fn prepare_thread(&mut self, thread: Arc<Thread>) -> *mut Context {
        unsafe {
            self.current_thread = Some(thread.clone());
            // 这里我们修改了 Thread 的接口让它看起来在语义上更加明确：
            // store_context(&self, context: Context) 表示将中断上下文
            // 保存在该线程的 TCB 中
            // retrieve_context(&self) -> Context 表示将中断上下文
            // 从该线程的 TCB 中取出，不再直接将中断上下文放到内核栈上
            let raw_context = thread.retrieve_context();
            // 我们在这里将取出的中断上下文放到内核栈上
            // 每个核有自己的内核栈，可以在 process/kernel_stack.rs 中找到
            let context = KERNEL_STACK[hart_id()].push_context(raw_context);
```

```
            context
        }
    }
    // 新增，表示准备好执行自己的 idle 线程
    pub fn processor_main(&mut self) -> *mut Context {
        self.prepare_thread(self.idle_thread.clone())
    }
    // 新增，对应于上面提到的调度器 get_next 的问题。
    // 我们在适当的时候调用此函数将当前即将被换出的线程重新放回就绪队列，
    // 而非在调度器的 get_next 中进行。
    pub fn run_current_thread_later(&mut self) {
        if **self.current_thread.as_ref().unwrap() != *self.idle_thread {
            THREAD_POOL.lock()
                .scheduler
                .add_thread(self.current_thread());
        }
    }
    /// 激活下一个线程的 `Context`
    /// 修改
    pub fn prepare_next_thread(&mut self) -> *mut Context {
        // 向调度器询问下一个线程
        let mut thread_pool = THREAD_POOL.lock();
        if let Some(next_thread) = thread_pool.scheduler.get_next() {
            // 前面有一段统计相关代码，暂且略过
            // 调用 prepare_thread 准备好内核栈，中断返回后运行新线程
            self.prepare_thread(next_thread.clone())
        } else {
            /*
            // 没有活跃线程
            if thread_pool.sleeping_threads.is_empty() {
                // 也没有休眠线程，则退出
                panic!("all threads terminated, shutting down");
            } else {
                //println!("prepare IDLE_THREAD!");
                // 有休眠线程，则等待中断
                let context = self.prepare_thread(self.idle_thread.clone());
                assert!(self.idle_thread.clone().inner().context.is_none());
                context
            }
             */
            // 之前的实现是发现休眠队列和就绪队列都没有线程直接 panic
            // 但在多核情况下，若线程总数少于核数就会直接 panic
            // TODO: 我们需要在其他地方实现该机制
            let context = self.prepare_thread(self.idle_thread.clone());
            assert!(self.idle_thread.clone().inner().context.is_none());
            context
        }
    }
    /// 以下内容基本保持不变
    /// 保存当前线程的 `Context`
    pub fn park_current_thread(&mut self, context: &Context) {
        self.current_thread().store_context(*context);
    }
    /// 令当前线程进入休眠
    pub fn sleep_current_thread(&mut self) {
        let current_thread = self.current_thread();
        THREAD_POOL.lock().sleep_thread(current_thread);
    }
```

```
    /// 终止当前的线程
    pub fn kill_current_thread(&mut self) {
        // 从调度器中移除
        // 注意由于某些回收机制，下面的两行不能调换

 THREAD_POOL.lock().kill_thread(self.current_thread.as_ref().unwrap().clone());
        self.current_thread.take();
    }
}
```

接下来，由于每个核都要有一个 `Processor`，我们创建一个 `PROCESSORS` 数组：

```
// process/processor.rs
lazy_static! {
    pub static ref PROCESSORS: Vec<Lock<Processor>> = {
        let mut processors = Vec::new();
        for i in 0..CPU_NUM {
            processors.push(Lock::new(
                Processor {
                    current_thread: None,
                    idle_thread: Thread::new(
                        KERNEL_PROCESS.clone(),
                        busy_loop as usize,
                        None,
                    ).unwrap(),
                }
            ));
        }
        processors
    };
}
```

在多核情况下，idle 线程的内容设置为死循环就好了，反正他们本来也没事做，都是一样在消耗电能。不过想想其实 wfi 也行？总之死循环总没错。

```
// process/processor.rs
fn busy_loop() -> ! {
    loop {}
}
```

每个核的 idle 线程都是内核线程，它们都挂在同一个内核进程下面，我们不希望搞多个内核进程，于是只搞一个，把它预先弄出来：

```
// process/process.rs
lazy_static! {
    pub static ref KERNEL_PROCESS: Arc<Process> =
Process::new_kernel().unwrap();
}
```

有了 `PROCESSORS` 数组，我们可以将 `Processor` 的各项功能重新封装为**当前核**上的版本：

```
// process/processor.rs
pub fn current_thread() -> Arc<Thread> {
    PROCESSORS[hart_id()].lock().current_thread()
}
```

```rust
pub fn prepare_next_thread() -> *mut Context {
    PROCESSORS[hart_id()].lock().prepare_next_thread()
}
pub fn park_current_thread(context: &Context) {
    PROCESSORS[hart_id()].lock().park_current_thread(context)
}
pub fn run_current_thread_later() {
    PROCESSORS[hart_id()].lock().run_current_thread_later();
}
pub fn sleep_current_thread() {
    PROCESSORS[hart_id()].lock().sleep_current_thread()
}
pub fn kill_current_thread() {
    PROCESSORS[hart_id()].lock().kill_current_thread()
}
pub fn processor_main() -> *mut Context {
    let mut processor = PROCESSORS[hart_id()].lock();
    processor.processor_main()
}
```

那么，在中断处理里面我们就可以用这些函数了！

例如，时钟中断的处理就可重写为：

```rust
// board/k210/interrupt.rs
pub fn supervisor_timer(context: &mut Context) -> *mut Context {
    timer::tick();
    park_current_thread(context);
    // 在这里当前线程即将被换出，如果不是 idle 线程的话，我们将其放回到就绪队列
    run_current_thread_later();
    prepare_next_thread()
}
```

结束线程：

```rust
// syscall/syscall.rs
SyscallResult::Kill => {
    // 一段统计代码
    // 终止，跳转到 PROCESSOR 调度的下一个线程
    kill_current_thread();
    prepare_next_thread()
}
```

休眠线程：

```rust
// 目前只支持在条件变量只能够进行休眠
// sync/condvar.rs
pub fn wait(&self) {
    self.watchers
    .lock()
    .push_back(current_thread());
    sleep_current_thread();
}
// 随后在 syscall 返回的时候
// syscall/syscall.rs
SyscallResult::Park(ret) => {
```

```
    // 将返回值放入 context 中
    context.x[10] = ret as usize;
    // 保存 context，准备下一个线程
    park_current_thread(context);
    prepare_next_thread()
}
```

唤醒线程：

```
// sync/condvar.rs
// 与 Processor 无关，而是与底层的共享调度队列 THREAD_POOL 有关
pub fn notify_one(&self) {
    if let Some(thread) = self.watchers.lock().pop_front() {
        THREAD_POOL.lock().wake_thread(thread);
    }
}
```

我们可以概括一下目前的改动：

- `Thread` 的接口被改成 `store_context` 和 `retrieve_context`，与内核栈无关，只负责将中断上下文存入/取出 TCB。
- 原 `Processor` 与调度有关的部分被分离成一个新的类 `ThreadPool`，表示所有核共享的底层的调度队列，与每个核当前的执行状态无关，负责线程的插入、删除，以及线程在就绪和休眠状态间的装换。
- 被剥离了调度相关内容的 `Processor` 目前只负责保存每个核当前的运行状态，每个核的 `Processor` 都保存一个独占的 idle 内核线程，目前只是一个死循环。从线程转移的角度看，它负责与 `current_thread` 有关的所有转移。

支持多程序运行的用户终端

我们需要对 `sys_exec` 系统调用进行改进，让它可以在新线程启动后被阻塞，并在进程退出后唤醒终端线程。

首先，我们给进程 `Process` 加上一个进程 ID `pid`：

```
// process/process.rs
pub struct Process {
    pub pid: usize,
    ...
}
```

随后在新建进程的时候使用进程 ID 分配器给它分配一个 ID：

```
// process/process.rs
// 原理非常简单的 pid 分配器
pub struct PidAllocator {
    max_id: usize,
    recycled: Vec<usize>,
}
impl PidAllocator {
    pub fn new() -> Self {
        Self {
            max_id: 0,
            recycled: Vec::new(),
        }
```

```
    }
    // 分配一个 pid
    // 优先从回收的部分分配，如果没有的话就弄一个新的
    pub fn alloc(&mut self) -> usize {
        if let Some(pid) = self.recycled.pop() {
            pid
        } else {
            self.max_id += 1;
            self.max_id
        }
    }
    // 回收一个 pid
    pub fn dealloc(&mut self, pid: usize) {
        assert!(pid <= self.max_id);
        assert!(
            self.recycled.iter().filter(|p| **p == pid).next().is_none()
        );
        self.recycled.push(pid);
    }
}
lazy_static! {
    // 全局 pid 分配器
    pub static ref PID_ALLOCATOR: Mutex<PidAllocator> =
Mutex::new(PidAllocator::new());
}
```

这样，我们就可以通过进程 ID 来区分一个进程。

维护一个 HashMap，键为进程 ID ，值为一个线程的 Arc 引用，意为：当键对应的进程退出的时候，我们应唤醒值对应的线程。

```
// process/process.rs
lazy_static! {
    pub static ref WAIT_MAP: Mutex<HashMap<usize, Weak<Thread>>> =
Mutex::new(HashMap::new());
}
```

这样，当我们在 `sys_exec` 的时候：

```
// syscall/process.rs sys_exec
Ok(inode) => {
    let data = inode.readall().unwrap();
    let elf = ElfFile::new(data.as_slice()).unwrap();
    // 创建用户进程
    let process = Process::from_elf(&elf, true).unwrap();
    // 创建用户进程的主线程
    let thread=Thread::new(process, elf.header.pt2.entry_point() as usize,
None).unwrap();
    // 保存用户进程的 pid
    let pid = thread.process.pid as isize;
    // 将用户线程插入调度队列
    THREAD_POOL.lock().add_thread(thread);
    // 将键值对插入 WAIT_MAP，用户进程退出之后将唤醒终端线程
    WAIT_MAP.lock().insert(pid as usize, Arc::downgrade(&current_thread()));
    // 这里要 sleep，然后和 syscall 返回之前的 park/prepare_next 组成 combo
    sleep_current_thread();
```

```
    SyscallResult::Park(0)
},
```

当进程退出的时候唤醒等待的终端线程：

```
// process/process.rs
impl Drop for Process {
    fn drop(&mut self) {
        println!("Process {} exited", self.pid);
        // 回收 pid
        PID_ALLOCATOR.lock().dealloc(self.pid);
        // 若有终端线程在等待，则唤醒它
        if let Some(thread) = WAIT_MAP.lock().get(&self.pid) {
            THREAD_POOL.lock()
                .wake_thread(thread.upgrade().unwrap());
        }
    }
}
```

至此，我们的终端终于能运行多个程序了。

线程计时功能

最早设计这个功能是为了能更加直观的看出程序确实是在多核上跑。但仔细想来，这个功能或许还能有更多的用处，比如比较各模块、算法的性能，计算吞吐量等等。

当一个线程退出的时候，我们希望能输出这些统计信息：

```
<notebook>
hello, world!
thread 4 exit with code 0
on Core #0, user time = 13616, kernel time = 15507
on Core #1, user time = 10078, kernel time = 28238
total user = 23694, kernel = 43745, sum = 67439
Process 3 exited
```

即在每个核上，用户和内核时间各是多少，单位为 CPU 周期。注意，线程在就绪队列中等待被调度以及在休眠队列中等待被唤醒的时间是不统计的。今后可能增加 `gettimeofday` 的系统调用，我们可以得到线程从开始到结束的总时间，减去上面给出 `sum` 就是所有的等待时间。

为了做到这一点，我们要在线程控制块内保存一个新的数据结构 `ThreadTrace`：

```
// process/thread.rs
pub struct ThreadInner {
    ...
    pub thread_trace: ThreadTrace,
}

pub struct ThreadTrace {
    // 键为核 ID，值为 (用户时间，内核时间)
    hart_time: HashMap<usize, (usize, usize)>,
    // 当前核
    current_hart: usize,
    // 当前时间戳
    time_clock: usize,
}
```

一个线程的执行可以分成若干段，从它被调度到某个核上运行，到它被放回调度队列/等待队列为止。这期间可能还会经历若干次串口中断、软中断或系统调用，但不能是时钟中断，因为一旦遇到时钟中断就会将当前的线程换出。

按照时间点，我们可将每一段看成这样的模式：

```
    |--------|--------|--------|--------|--------|...|--------|--------|
   out      in       out      in       out      in  out      in       out
(prologue)
```

其中 out 代表即将离开内核态；in 代表刚刚进入内核态。当在 in 时间点的时候，我们知道该线程此前在用户态/内核态（取决于它是个用户线程还是内核线程）执行了一段时间；当在 out 时间点的时候，我们知道该线程刚刚的一段时间在内核态进行 Trap 处理。因此，我们只需要在正确的时间点将它与上一个时间点的差值加到用户时间或内核时间即可。

当一个线程被调度到某个核上运行的时候，我们需要运行：

```
impl ThreadTrace {
    // prologue 是一个特殊的 out，因为它不需要统计差值并累加
    pub fn prologue(&mut self, hart: usize, time_clock: usize) {
        self.current_hart = hart;
        self.time_clock = time_clock;
    }
}
```

当在 in 时间点，也即刚刚进入内核态的时候，我们需要调用：

```
impl ThreadTrace {
    pub fn into_kernel(&mut self, hart: usize, time_clock: usize, is_user: bool)
{
        let delta_time = time_clock - self.time_clock;
        // 判断是否已经在当前 hart 上跑过了
        if let Some(time_pair) = self.hart_time.get(&hart) {
            if is_user {
                self.hart_time.insert(hart, (time_pair.0 + delta_time,
time_pair.1));
            } else {
                self.hart_time.insert(hart, (time_pair.0, time_pair.1 +
delta_time));
            }
        } else {
            if is_user {
                self.hart_time.insert(hart, (delta_time, 0));
            } else {
                self.hart_time.insert(hart, (0, delta_time));
            }
        }
        self.time_clock = time_clock;
    }
}
```

当在 out 时间点，也即准备离开内核态的时候，我们需要调用：

```
impl ThreadTrace {
    pub fn exit_kernel(&mut self, hart: usize, time_clock: usize) {
        let delta_time = time_clock - self.time_clock;
        if let Some(time_pair) = self.hart_time.get(&hart) {
            self.hart_time.insert(hart, (time_pair.0, time_pair.1 +
delta_time));
        } else {
            panic!("had not executed on current hart before!");
        }
        self.time_clock = time_clock;
    }
}
```

那么在代码中哪里是 prologue, in, out 各自合适的时间点呢?

prologue 是在一个线程即将被从调度队列中取出到某个核上执行的时候，所以它应该被放在:

```
// process/processor.rs
impl Processor {
    pub fn prepare_next_thread(&mut self) -> *mut Context {
        let mut thread_pool = THREAD_POOL.lock();
        if let Some(next_thread) = thread_pool.scheduler.get_next() {
            next_thread.as_ref().inner().thread_trace.prologue(hart_id(),
read_time());
            ...
        }
        ...
    }
}
```

in 表示刚刚进入内核态，因此应该被放在进入 trap handler 的时候:

```
// interrupt/handler.rs
pub fn handle_interrupt(context: &mut Context, scause: Scause, stval: usize) ->
*mut Context {
    let start_thread = current_thread().clone();
    let is_user = start_thread.process.is_user;
    start_thread.as_ref()
        .inner()
        .thread_trace
        .into_kernel(hart_id(), read_time(), is_user);
    ...
}
```

out 的情况则相对比较复杂，要分成以下几种情况:

第一种情况：它被换出了;

```
// process/processor.rs
impl Processor {
    pub fn prepare_next_thread(&mut self) -> *mut Context {
        let mut thread_pool = THREAD_POOL.lock();
        if let Some(next_thread) = thread_pool.scheduler.get_next() {
            ...
            if self.current_thread.is_some() {
                self.current_thread()
```

```
                    .as_ref()
                    .inner()
                    .thread_trace
                    .exit_kernel(hart_id(), read_time());
            }
        }
        ...
    }
}
```

第二种情况：它被阻塞或者退出了；

```
// syscall/syscall.rs
SyscallResult::Park(ret) => {
    context.x[10] = ret as usize;
    current_thread().as_ref().inner().thread_trace.exit_kernel(hart_id(),
read_time());
    ...
}
SyscallResult::Kill => {
    let current_thread = current_thread();
    current_thread.as_ref().inner().thread_trace.exit_kernel(hart_id(),
read_time());
    // 注意，这里线程退出了，直接打印 trace 信息
    current_thread.as_ref().inner().thread_trace.print_trace();
    ...
}
```

第三种情况，当前线程到内核态走了一下没有被换出又回到了内核/用户态：

```
// interrupt/handler.rs
pub fn handle_interrupt(context: &mut Context, scause: Scause, stval: usize) ->
*mut Context {
    ...

    let context = ...;
    // 我们进来的时候保存了一下 start_thread，如果要离开 trap 的时候没有发生变化
    // 那么这也是一个 out 时间点，需要 exit_kernel
    if *current_thread() == *start_thread {
        start_thread.as_ref().inner().thread_trace.exit_kernel(hart_id(),
read_time());
    }

    context
}
```

这个 `start_thread` 属于一个线程在 `current_thread`、就绪队列、休眠队列三者之外另一个可能有引用计数的地方。但是这样实现应该并没有啥问题。因为它仅在 trap 处理期间持有该引用计数。

下一步

功能增强（Optional）

实现 greenthread 或者类似 pthread 的系统调用，或者多种多核调度算法。

直接调 rcore-fs 的库不是很爽，自己搞一个简化版的？

更多的 I/O 设备或总线方面的支持。

面向 book 编程

重新构想整本书的架构与布局，重新设计层次分明的实验，然后再完善一下框架。

最后，整理仓库及代码，写文档！