0. 前言

实验概述

FIFO与RR调度器。运行实验代码，了解FIFO与RR调度器下进程运行规则

实验内容

1. 拉取实验所需环境与代码
2. 了解实验代码的新架构
3. 通过FIFO调度器，了解OSTD提供的调度器API
4. 了解时钟中断与RR调度器实现
5. 运行实验代码，验证RR调度器

1. 实验环境及代码

拉取实验所需环境以及代码：

```
podman pull glcr.cra.ac.cn/operating-systems/asterinas_labs/images/lab4:0.1.0
mkdir os-lab
podman run -it -v ./os-lab:/root/os-lab glcr.cra.ac.cn/operating-systems/asterinas_labs/images/lab4:0.1.0

git clone -b lab7 https://github.com/sdww0/sustech-os-lab.git
cd sustech-os-lab
```

运行实验代码:

```
cargo osdk run --scheme riscv --target-arch=riscv64
```

环境与代码须知：

1. 本次实验课环境需要更新OSDK的工具链版本，在容器中运行命令以升级工具链：`cargo install cargo-osdk --version 0.9.4`
2. 本节课开始，默认运行的程序从`hello_world`变成了`shell`，以方便大家在系统中运行自定义的程序。

2. 从FIFO调度器开始

在之前的实验课我们提到，OSTD的`Task`抽象与调度器进行了深度绑定，同时，为了方便开发基于OSTD进行开发提供了一个默认的FIFO调度器（位置为:`ostd/src/task/scheduler/fifo_scheduler.rs`）。当任务开始运行，且系统未注册调度器的时候，OSTD便会将FIFO调度器注入到系统内，这也就是为什么之前我们都不需要关注有关调度器的实现。

下面我们来看FIFO调度策略是怎么适配到OSTD的调度器API上的，下面的代码经过了简化，去除了用来支持SMP（多核）以及其它无关的内容（位置：`src/sched/fifo.rs`：

```
// SPDX-License-Identifier: MPL-2.0
```

```rust
// Ref: asterinas: ostd/src/task/scheduler/fifo_scheduler.rs

pub fn init() {
    let fifo_scheduler = Box::new(FifoScheduler {
        rq: SpinLock::new(FifoRunQueue::new()),
    });
    inject_scheduler(Box::leak(fifo_scheduler));
}

/// A simple FIFO (First-In-First-Out) task scheduler.
struct FifoScheduler {
    rq: SpinLock<FifoRunQueue>,
}

impl Scheduler for FifoScheduler {
    fn enqueue(&self, runnable: Arc<Task>, _flags: EnqueueFlags) ->
Option<CpuId> {
        let mut rq = self.rq.disable_irq().lock();
        rq.queue.push_back(runnable);
        Some(CpuId::bsp())
    }

    fn local_rq_with(&self, f: &mut dyn FnMut(&dyn LocalRunQueue<Task>)) {
        let _preempt_guard = disable_preempt();
        let local_rq: &FifoRunQueue = &self.rq.disable_irq().lock();
        f(local_rq);
    }

    fn local_mut_rq_with(&self, f: &mut dyn FnMut(&mut dyn
LocalRunQueue<Task>)) {
        let _preempt_guard = disable_preempt();
        let local_rq: &mut FifoRunQueue = &mut self.rq.disable_irq().lock();
        f(local_rq);
    }
}

struct FifoRunQueue {
    current: Option<Arc<Task>>,
    queue: VecDeque<Arc<Task>>,
}

impl LocalRunQueue for FifoRunQueue {
    fn current(&self) -> Option<&Arc<Task>> {
        self.current.as_ref()
    }

    fn update_current(&mut self, flags: UpdateFlags) -> bool {
        !matches!(flags, UpdateFlags::Tick)
    }

    fn pick_next_current(&mut self) -> Option<&Arc<Task>> {
        let next_task = self.queue.pop_front()?;
        if let Some(prev_task) = self.current.replace(next_task) {
            self.queue.push_back(prev_task);
        }
```

```
55
56              self.current.as_ref()
57          }
58
59          fn dequeue_current(&mut self) -> Option<Arc<Task>> {
60              self.current.take()
61          }
62      }
63
```

最先看到的是`FifoScheduler`，会实现OSTD中`Scheduler` trait的接口，第一个接口是为了将新加入的任务加入到运行队列（`RunQueue`）中，对于后两个接口主要是为了获取运行队列的不可变引用和可变引用。运行队列中存放着两个数据，可运行任务列表与当前运行任务：可运行任务在任务状态表中代表的是Ready状态，当调度器需要进行调度时，就会从运行队列中获取下一个可运行的任务，并转为Running状态；当前运行任务即为Running状态的任务，当前执行的任务。

`RunQueue`中的四个接口分别代表着（1）获取当前运行任务；（2）当事件发生时，更新运行队列状态；（3）从可运行任务中选择其一，替代当前运行任务（Ready <-> Running）；（4）取消当前运行的任务。

FIFO调度算法的实现在两个关键接口：有任务进入与获取下一任务，其对应的接口为`FifoScheduler::enqueue`与`FifoRunQueue::pick_next_current`，里面会使用到`push_back`与`pop_front`接口来实现FIFO算法。

3. RR实现

RR算法基于时间片来进行抢占式调度，为每一个进程分配不超过一个时间片的CPU运行时间。

3.1 时钟中断

时钟中断可以理解为一个定时器，当定时器达到了我们设定的时间点后，便会向系统发送一个中断。RR的算法依赖于时钟中断的实现，有了时钟中断我们才可以在进程运行时打断执行，并检查是否超出时间片，以决定是否要暂停当前进程，运行下一个进程。

> 操作系统配置时钟中断一般需要涉及到二到三个步骤，取决于时钟是单次触发还是多次间隔触发，如为单次触发则需要在中断处理函数中重复步骤二：
>
> 1. 获取时钟基本频率
> 2. 根据期望的系统时钟频率（如1ms触发一次时钟中断）配置寄存器，告知时钟触发中断时间

RISC-V对于时钟中断有很好的支持，基本与架构进行了绑定，可以通过使用SBI定义的接口来实现时钟中断。我们基于单次触发实现了RISCV平台上的时钟中断，文件路径为：`ostd/src/arch/riscv/timer/mod.rs`，包含三个功能：（1）`sie::set_stimer()`，使能时钟中断；（2）`set_next_tick`，获取当前时间点并设置下一个触发时钟中断的时间点；（3）`timer_callback`，时钟中断处理函数，会进一步调用注册的回调函数。

3.2 RR调度器

有了时钟中断作为支持，OSTD便会拥有抢占式调度的功能来支持我们的RR调度器了，其具体实现在`src/sched/rr.rs`中，观察可以发现其和FIFO调度器的实现没有太多区别，以下代码去除了未变动太多的实现：

```
1   pub struct RrScheduler {
2       run_queue: SpinLock<RrRunQueue>,
```

```rust
}

impl Scheduler for RrScheduler {
    fn enqueue(&self, runnable: Arc<Task>, _flags: EnqueueFlags) -> Option<CpuId> {
        let _irq = disable_local();
        let mut rq = self.run_queue.lock();
        rq.entities.push_back(Entity {
            task: runnable,
            time_slice: TimeSlice::default(),
        });
        Some(CpuId::bsp())
    }
    // ...
}

struct RrRunQueue {
    current: Option<Entity>,
    entities: VecDeque<Entity>,
}

impl LocalRunQueue for RrRunQueue {
    // ...
    fn update_current(&mut self, flags: ostd::task::scheduler::UpdateFlags) -> bool {
        match flags {
            ostd::task::scheduler::UpdateFlags::Tick => {
                let Some(entity) = self.current.as_mut() else {
                    return false;
                };
                entity.time_slice.elapse()
            }
            _ => true,
        }
    }
    // ...
}

struct Entity {
    task: Arc<Task>,
    time_slice: TimeSlice,
}

#[derive(Default)]
struct TimeSlice {
    tick: usize,
}

impl TimeSlice {
    const PROCESS_TIME_SLICE: usize = 100;

    fn elapse(&mut self) -> bool {
        self.tick = (self.tick + 1) % Self::PROCESS_TIME_SLICE;

        self.tick == 0
```

```
    }
}

```

这段代码中，我们引入了`TimeSlice`和`Entity`两个结构，`TimeSlice`结构代表的是RR算法中的时间片概念，当调用`elapse`时自动进行递增，并根据`PROCESS_TIME_SLICE`返回当前时间片是否耗尽。`Entity`则是`Arc<Task>`的一层包装，里面会将进程与时间片进行绑定。

继续看代码可以发现RR算法中的`RunQueue`和`Scheduler`相比FIFO算法会多出一段对于Tick的创建与处理，在`RrRunQueue::update_current`中会进行时间+1并检查是否时间片耗尽的逻辑，如果耗尽则会返回`true`，告诉OSTD需要进行抢占式调度。OSTD收到后，会调用`yield_now`，放弃当前进程的执行，间接调用到`RrRunQueue::pick_next_current`。

4. RR验证

我们实现了FIFO和RR两个调度算法，接下来将运行用户态程序来观察这两个调度算法的不同，本实验课多出了一个`round_robin`用户态程序，我们可以通过选择不同的调度算法观察到完全不同的输出。`round_robin`程序中将会克隆出10个进程并运行，每个进程均会进行些操作，并在操作过后将`count`加1，且每个进程固定在用户程序启动10s后全部退出，并将`count`存放到退出码中。总的来说，`count`代表的就是fork出的进程在10s中占据到的CPU时间。

当我们使用FIFO调度算法时，运行`round_robin`用户态程序可以得到以下输出：

```
~ # round_robin
 Running command: round_robin
main: fork ok, now need to wait pids.
main: pid 3, count 3305768
main: pid 4, count 1
main: pid 5, count 1
main: pid 6, count 1
main: pid 7, count 1
main: pid 8, count 1
main: pid 9, count 1
main: pid 10, count 1
main: pid 11, count 1
main: pid 12, count 1
main: wait pids over
```

可以看到，最先fork出的进程占据了所有的时间片，只有当其退出时才会允许其它进程执行。其他进程进行了一次操作后检测到超出10s后直接就退出了。

修改`lib.rs`中调度器的初始化，切换到RR调度算法后，运行`round_robin`用户态程序可以得到以下输出：

```
~ # round_robin
 Running command: round_robin
main: fork ok, now need to wait pids.
main: pid 3, count 406136
main: pid 4, count 384359
main: pid 5, count 355059
main: pid 6, count 337204
main: pid 7, count 299709
main: pid 8, count 275136
```

```
main: pid 9, count 241144
main: pid 10, count 197379
main: pid 11, count 154189
main: pid 12, count 117215
main: wait pids over
```

可以看到，克隆出的其它进程也得到了CPU的执行，而不是由最先克隆出的进程独享CPU执行的10s。

但在这里也能看到后面的`count`并不是相近的，最后克隆出的进程得到的时间片会小于最先克隆的进程，这是因为我们将单个进程所占据的时间片设为间隔较大的100ms，并且主线程fork之后也会进入到RR中的FIFO调度器，需要等待其它运行队列中的进程执行完所属时间片后才会再fork下一个进程。如果我们希望这个数值接近，可以依靠减小时间间隔来达到目的。

修改`PROCESS_TIME_SLICE`的值为10，再次运行代码我们可以看到以下输出：

```
~ # round_robin
 Running command: round_robin
main: fork ok, now need to wait pids.
main: pid 3, count 299300
main: pid 4, count 301242
main: pid 5, count 307394
main: pid 6, count 296233
main: pid 7, count 295679
main: pid 8, count 291899
main: pid 9, count 285620
main: pid 10, count 270232
main: pid 11, count 285524
main: pid 12, count 275725
main: wait pids over
```

可以看到此时数值几乎相等。

5. 上手练习

基于RR调度器，给予不同的进程以不同的时间片，比如可以根据进程的PID来设置时间片大小，进程PID为1的进程所占据的时间片比PID为10的进程所占据的时间片小，观察输出结果。

Hint: 将进程的时间片存放到PCB中，调度器中可以参考`current_process()`函数的逻辑获取到`Task`对应的`Process`