# Lab7 Timer and Scheduler

## 0. 前言

### 实验概述

本实验将学习FIFO调度器、定时器Timer与RR调度器。通过运行代码观察FIFO与RR调度器下进程的运行差异。

### 实验代码

拉取实验代码：

```
$ git clone https://github.com/sdww0/sustech-os-lab.git && cd sustech-os-lab/lab7-timer-sched
```

或更新实验代码：

```
# In sustech-os-lab
$ git pull && cd lab7-timer-sched
```

本次代码新增系统调用：

1. SYS_GETTIME: 用于用户态程序读取当前系统时间戳

本次代码新增功能：

1. FIFO调度器
2. RR调度器
3. RR的测试代码，用于查看不同调度器下的时间戳区别

## 1. 定时器与调度策略

定时器（Timer）是操作系统的重要基础设施，为整个系统提供了基础的时间感知能力。系统的许多功能，如确认系统已运行时间、在指定时间后触发系统事件等，都依赖于定时器。在RISC-V架构中，定时器更是作为核心功能嵌入到架构内部。

定时器以固定频率（可调节但不超过基础频率）向系统发送中断，通知系统已过去指定时间片（如1ms）。本节课不涉及定时器的多种模式（如单次触发、重复触发、下沿触发等），通用操作系统通常使用固定频率重复触发的定时器。

> **RISC-V的中断**
>
> 在RISC-V中，**中断** 是一种由**外部异步事件**（如定时器到期、外部设备I/O请求）引发的异常，其目的是通知CPU暂停当前执行流，转去处理特定事件，处理完毕后返回继续执行。RISC-V的中断大致可以分为三类：
>
> 1. 软件中断：由一个CPU核心向另一个核心发送的中断。
> 2. Timer中断：由内部的定时器触发，即定时器到期。
> 3. 外部中断：来自核心外部的设备中断，例如网卡，硬盘等。

操作系统的调度策略分为抢占式调度和非抢占式调度。抢占式调度是最常用的策略，它在运行的进程耗尽时间片后打断运行并切换到下一个进程，从而避免因进程卡死导致的系统未响应。这一功能依赖于定时器实现。

系统内部会维护一个全局计数器，每次定时器到期触发时钟中断时会将计数器加1，调度器判断计数器是否达到指定值。若条件满足，则进行调度切换进程。本实验的RR（Round Robin）调度器就依赖定时器实现抢占式调度。

## 2. 从FIFO调度器开始

之前在使用`Task::run`运行进程时，我们未深入其中代码，实际上`Task::run`不会直接运行指定线程，而是将其加入调度器，在调度过程中运行线程。OSTD为方便开发提供了默认的FIFO调度器（位置：`ostd/src/task/scheduler/fifo_scheduler.rs`，非抢占式调度，FIFO算法）。当任务开始运行且系统未注册调度器时，OSTD会自动注入FIFO调度器，这就是之前无需关注调度器实现的原因。

去除多核等无关内容后，FIFO调度器的关键代码如下（位置：`src/sched/fifo.rs`）：

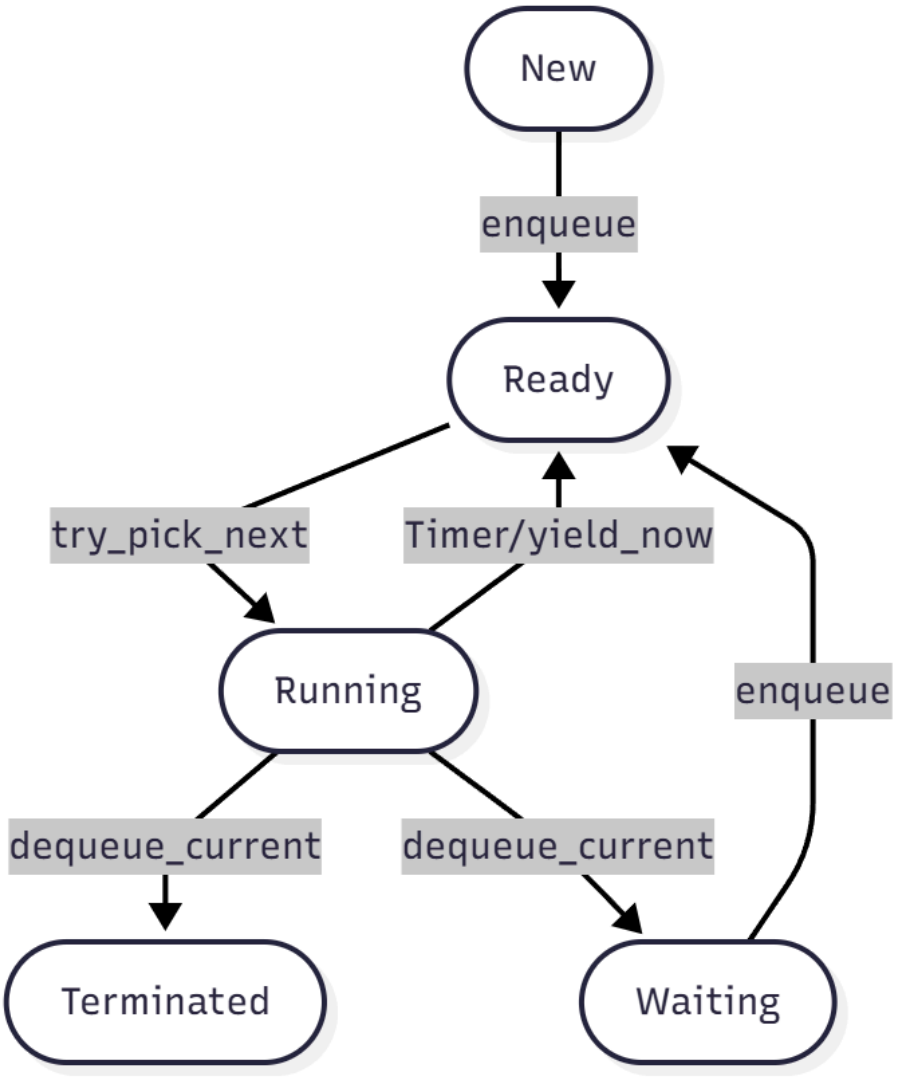
```
pub struct FifoScheduler {
    run_queue: SpinLock<FifoRunQueue>,
}

impl Scheduler for FifoScheduler {
    fn enqueue(&self, runnable: Arc<Task>, _flags: EnqueueFlags) -> Option<CpuId>
{
        let mut run_queue = self.run_queue.disable_irq().lock();
        run_queue.queue.push_back(runnable);
        None
    }

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

    fn mut_local_rq_with(&self, f: &mut dyn FnMut(&mut dyn LocalRunQueue<Task>))
{
        let _guard = disable_preempt();
        let local_rq: &mut FifoRunQueue = &mut
self.run_queue.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 dequeue_current(&mut self) -> Option<Arc<Task>> {
```

```
        self.current.take()
    }

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

        self.current.as_ref()
    }

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

包含两种结构体：`FifoScheduler`与`FifoRunQueue`，功能如下：

- `FifoScheduler`：调度器实现，里面会有两类接口：（1）将Task加入到运行队列中；（2）获取**当前CPU运行队列**的不可变或可变引用。
- `FifoRunQueue`：**单个CPU的运行队列**，包含当前运行Task和等待运行Task队列，实现以下方法：
  - `current`：获取当前运行线程
  - `dequeue_current`：移除当前运行线程，用于两种场景：（1）当前Task退出，防止退出的任务再次运行；（2）当前Task进入等待状态，唤醒且条件满足后才重新加入等待队列
  - `try_pick_next`：进行一次调度，FIFO算法中将当前Task放入队尾，运行队首Task
  - `update_current`：启用事件通知机制，当时钟中断（**Tick**）或其他Task事件（**Wait、Yield、Exit**）发生时调用此方法。通过返回值控制OSTD是否进行调度。FIFO无抢占式调度，除Tick外的事件都会触发调度

了解这两个结构的API后，我们来看调度器如何参与操作系统进程生命周期切换，生命周期图如下：




其中较为特殊的为Running到Ready的切换，会包含两种情况：抢占式调度（FIFO调度暂无此功能）或调用`Task::yield_now`主动放弃运行：

- 抢占式调度：每次时钟中断发生时，OSTD会调用`update_current`并传入`UpdateFlags::Tick`，运行队列会根据Tick的触发次数判断是否要进行调度
- `Task::yield_now`：调用`update_current`判断是否切换到下一个可运行Task，若否直接返回；若是则调用`try_pick_next`进行切换

## 3. RR调度器

RR调度器结合了FIFO与抢占式调度，其为每个进程分配不超过一个时间片的CPU运行时间。具体实现在`src/sched/rr.rs`中，与FIFO调度器实现差异不大，去除未变动太多的实现后代码如下：

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

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

```
        None
    }
    // ...
}

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
    }
}
```

代码引入了`TimeSlice`和`Entity`结构：`TimeSlice`代表RR算法中的时间片概念，调用`elapse`时自动递增，根据`PROCESS_TIME_SLICE`返回当前时间片是否耗尽；`Entity`是`Arc<Task>`的包装层，将进程与时间片绑定。

RR算法的`RunQueue`和`Scheduler`相比FIFO算法增加了Tick的创建与处理。
`RrRunQueue::update_current`中进行时间+1并检查时间片是否耗尽，若耗尽则返回`true`，通知OSTD需要进行调度。

# 4. 从用户态程序观察FIFO & RR差异

我们实现了FIFO和RR两种调度算法，接下来通过运行用户态程序观察它们的差异。本实验新增`rr_test`用户态程序，选择不同调度算法可观察到不同输出。`rr_test`程序会克隆10个进程运行，每个进程进行耗时操作后将`count`加1，所有进程在启动10秒后退出，并将`count`存放到退出码中。`count`代表fork出的进程在10秒内获得的CPU时间。

使用FIFO调度算法运行`rr_test`程序，输出如下：

```
~ # rr_test
 Running command: rr_test
main: fork ok, now need to wait pids.
main: pid 4, count 293325
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: pid 13, 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后便直接退出。

修改`sched/mod.rs::USE_RR_SCHEDULER`切换到RR调度算法，运行`rr_test`程序输出如下：

```
~ # rr_test
 Running command: rr_test
main: fork ok, now need to wait pids.
main: pid 4, count 50860
main: pid 5, count 46566
main: pid 6, count 40205
main: pid 7, count 48144
main: pid 8, count 33301
main: pid 9, count 27350
main: pid 10, count 25190
main: pid 11, count 19247
main: pid 12, count 12493
main: pid 13, count 9069
main: wait pids over
```

可以看到，克隆出的所有进程都获得了CPU执行时间，而非由最先克隆的进程独享CPU。

但后面的`count`值并不相近，最后克隆的进程获得的时间片少于最先克隆的进程。这是因为我们将单个进程的时间片设为较大的100ms，且主线程fork后也会进入RR中的FIFO调度器，需要等待运行队列中其他进程执行完时间片才能fork下一个进程。要使得数值接近，可以减少时间间隔。修改`PROCESS_TIME_SLICE`的值为10，再次运行程序输出如下：

```
~ # rr_test
 Running command: rr_test
main: fork ok, now need to wait pids.
```

```
main: pid 4, count 47637
main: pid 5, count 42869
main: pid 6, count 50362
main: pid 7, count 38142
main: pid 9, count 32044
main: pid 10, count 20263
main: pid 11, count 27766
main: pid 12, count 29246
main: pid 13, count 23782
main: pid 8, count 31316
main: wait pids over
```

此时数值基本相等。

> **Rust中的Release与Debug模式**
>
> Rust有两种主要的编译配置模式：Debug模式和Release模式，它们对程序性能和调试能力有显著影响：
>
> - **Debug模式**：默认配置，编译时不进行优化，保留完整的调试信息，编译速度快，便于开发和调试。
> - **Release模式**：通过`--release`标志启用，编译时进行大量优化，生成代码运行速度快但编译时间更长。此模式会移除调试信息，并可能禁用某些运行时检查以提高性能。
>
> 在操作系统开发中，通常使用Debug模式进行开发和调试，而Release模式用于生成最终的性能版本。运行`cargo osdk run --target-arch=riscv64 --release`的输出如下：
>
> ```
> ~ # rr_test
>  Running command: rr_test
> main: fork ok, now need to wait pids.
> main: pid 4, count 311659
> main: pid 5, count 338335
> main: pid 6, count 351708
> main: pid 7, count 367376
> main: pid 8, count 314491
> main: pid 9, count 326456
> main: pid 10, count 366834
> main: pid 11, count 350444
> main: pid 12, count 334183
> main: pid 13, count 348868
> main: wait pids over
> ```
>
> 与Debug模式相比，Release模式下的`count`值明显更高。

# 5. 上手练习

基于RR调度器，根据PID来设置不同进程的时间片大小，PID越大，其时间片越大，运行rr_test以展示你的结果（预期结果：count值最大的pid为8~10之间，而不是之前的4~6之间）。推荐设置最大时间片为PID * 10，并用release模式运行系统：`cargo osdk run --target-arch=riscv64 --release`。

**Hint:** 参考`current_process()`来将`Arc<Task>`转为`Arc<Process>`