

Lab6 调度器框架实验报告

小组成员：陈忠镇 姚文广 田子煊

实验目的

理解操作系统的调度管理机制 熟悉 ucore 的系统调度器框架，实现缺省的Round-Robin 调度算法 基于调度器框架实现一个(Stride Scheduling)调度算法来替换缺省的调度算法

练习1: 理解调度器框架的实现

1. [sched_class]结构体解析

[sched_class]结构体定义了调度算法的接口规范，包含了一系列函数指针：

- `char *name`：调度类的名称，在调度器初始化时用于打印调试信息，便于区分不同的调度算法。
- `(*init)(struct run_queue *rq)`：初始化函数，在调度器启动时被调用，用于初始化特定调度算法的数据结构（如RR需要初始化空链表，Stride需要初始化空斜堆）。
- `(*enqueue)(struct run_queue *rq, struct proc_struct *proc)`：将进程加入运行队列，负责把进程挂入运行队列并更新相关的调度数据。
- `(*dequeue)(struct run_queue *rq, struct proc_struct *proc)`：将进程从运行队列中移除，保证调度算法能正确维护其数据结构。
- `proc_struct *(*pick_next)(struct run_queue *rq)`：核心调度函数，选择下一个要运行的进程。
- `(*proc_tick)(struct run_queue *rq, struct proc_struct *proc)`：时间片处理函数，每次时钟中断调用，决定是否需要设置 [need_resched]标志。

将这些函数定义为函数指针的原因是为了实现调度算法与框架的解耦，通过更换 [sched_class]指针，可以在不修改核心调度框架的情况下切换不同的调度算法。

2. [run_queue]结构差异

- **Lab5**: 没有独立的 [run_queue]结构，调度器直接扫描全局 [proc_list]，策略和进程管理耦合在一起。
- **Lab6**: [run_queue]同时包含 [run_list]（链表）和 [lab6_run_pool]（斜堆），可以同时支持 FIFO（RR 使用）和优先队列（Stride 使用）。这样的设计使得切换调度器时无需修改其他模块。

Lab6的 [run_queue] 需要支持两种数据结构（链表和斜堆）是因为不同的调度算法需要不同的数据结构来高效地实现其调度策略。RR算法使用链表来维护先进先出的顺序，而Stride算法使用斜堆来快速找到stride值最小的进程。

3. Lab6 中的调度框架函数

- [sched_init()]: 初始化全局 [run_queue]，绑定默认的 [sched_class]，并调用调度类的 [init] 回调完成数据结构准备。
- [wakeup_proc()]: 除了将进程状态改为 [PROC_RUNNABLE] 外，还调用 `sched_class->enqueue()` 将进程加入运行队列。
- [schedule()]: 不再手动遍历 [proc_list]，而是调用当前调度类的 [pick_next]函数选择下一个要运行的进程，实现了调度策略与框架的解耦。

4. 调度器初始化流程

1. `kern_init()` 初始化基础子系统后调用 `[sched_init()]`。
2. `[sched_init()]`将 `[sched_class]`绑定到 `[default_sched_class]`，初始化 `[run_queue]`。
3. 调用 `[proc_init()]`创建初始进程并将其加入运行队列。

5. 进程调度流程图

```

时钟中断
  ↓
sched_class_proc_tick(current)
  ↓ (如果时间片用完)
设置 current->need_resched = 1
  ↓
trap() 返回用户态前检查 need_resched
  ↓
schedule()
  ↓ (如果当前进程仍可运行)
sched_class_enqueue 将当前进程放回队列
  ↓
next = sched_class_pick_next()
  ↓
sched_class_dequeue(next)
  ↓
if (next != current) proc_run(next)
  ↓
继续执行

```

`[need_resched]`标志位是调度策略与调度框架之间的通信桥梁，当进程时间片用完时，调度算法会设置此标志，通知调度框架需要进行调度。

6. 调度算法切换机制

要添加新的调度算法（如 Stride），只需要：

1. 实现新的调度类结构体，包含相应的函数指针实现
2. 在 `[sched_init()]`中更改 `[sched_class]`的绑定即可

这种设计使得切换调度算法非常容易，因为所有调度相关操作都通过函数指针调用，调度框架本身不依赖于具体调度算法的实现。

练习2: 实现 Round Robin 调度算法

1. Lab5 与 Lab6 函数对比

比较 `[schedule()]`和 `[wakeup_proc()]`函数的实现差异：

- **Lab5:** `[schedule()]`直接遍历全局 `[proc_list]`寻找下一个可运行的进程，调度策略与框架耦合在一起。

- **Lab6:** [schedule()]通过 `sched_class->pick_next()` 调用具体调度算法的实现，实现了策略与框架的分离。
- **Lab5:** [wakeup_proc()]不调用队列操作函数，只是简单地改变进程状态。
- **Lab6:** [wakeup_proc()]调用 `sched_class->enqueue()` 将进程加入运行队列。

如果不做这些改动，调度框架将无法支持多种调度算法，每种算法都需要修改调度框架代码，违反了开闭原则。此外，进程状态改变后无法正确加入运行队列，可能导致某些进程永远得不到调度。

2. RR 算法实现

以下是在 [kern/schedule/default_sched.c]中实现的代码：

```
#include <defs.h>
#include <list.h>
#include <proc.h>
#include <assert.h>
#include <default_sched.h>

static void
RR_init(struct run_queue *rq) {
    list_init(&(rq->run_list)); // 初始化运行队列为一个空链表
    rq->proc_num = 0;          // 初始化进程数量为0
}

static void
RR_enqueue(struct run_queue *rq, struct proc_struct *proc) {
    assert(list_empty(&(proc->run_link))); // 确保进程不在任何队列中
    // 将进程添加到运行队列的末尾，使用list_add_before将新节点添加到哨兵节点前面
    list_add_before(&(rq->run_list), &(proc->run_link));
    // 重置时间片，如果进程时间片为0或者大于最大时间片，则设置为最大时间片
    if (proc->time_slice == 0 || proc->time_slice > rq->max_time_slice) {
        proc->time_slice = rq->max_time_slice;
    }
    proc->rq = rq;
    rq->proc_num++; // 增加队列中进程数量
}

static void
RR_dequeue(struct run_queue *rq, struct proc_struct *proc) {
    // 从运行队列中移除进程
    list_del_init(&(proc->run_link));
    rq->proc_num--; // 减少队列中进程数量
}

static struct proc_struct *
RR_pick_next(struct run_queue *rq) {
    // 选择队列中的第一个进程 (FIFO)
    list_entry_t *le = list_next(&(rq->run_list));
    if (le != &(rq->run_list)) { // 如果队列不为空
        return le2proc(le, run_link);
    }
}
```

```

    }
    return NULL;           // 如果队列为空，返回NULL
}

static void
RR_proc_tick(struct run_queue *rq, struct proc_struct *proc) {
    // 递减时间片
    if (proc->time_slice > 0) {
        proc->time_slice--;
    }
    // 如果时间片用完，设置调度标志
    if (proc->time_slice == 0) {
        proc->need_resched = 1;
    }
}

struct sched_class default_sched_class = {
    .name = "RR_scheduler",
    .init = RR_init,
    .enqueue = RR_enqueue,
    .dequeue = RR_dequeue,
    .pick_next = RR_pick_next,
    .proc_tick = RR_proc_tick,
};

```

实现思路说明：

- **RR_init**: 初始化运行队列为一个空链表，设置进程数量为0
- **RR_enqueue**: 将进程添加到链表尾部（使用 [list_add_before] 将新节点添加到 [run_list] 前面，相当于添加到队列尾部），并重置时间片
- **RR_dequeue**: 从链表中移除进程节点
- **RR_pick_next**: 选择链表的第一个节点，实现先进先出的调度策略
- **RR_proc_tick**: 递减时间片，当时间片用完时设置 [need_resched] 标志

边界情况处理：

- 空队列处理：在 [RR_pick_next] 中检查队列是否为空
- 进程时间片耗尽：在 [RR_proc_tick] 中检查时间片是否用完并设置 [need_resched]
- 空闲进程处理：当没有可运行进程时，[RR_pick_next] 返回 NULL，调度器会选择 [idleproc]

3. make grade 输出结果

```

priority:                (3.1s)
  -check result:          OK
  -check output:          OK
Total Score: 50/50

```

4. QEMU 观察到的调度现象

在 QEMU 中可以看到：

- 调度器显示使用的是 RR 算法："sched class: RR_scheduler"
- 多个进程轮流执行，体现了时间片轮转的特性
- 每个进程运行一定时间后会让出 CPU 给其他进程
- 所有检查均通过，RR 调度器能够正确运行 [priority] 用户程序并完成内核自检

5. Round Robin 算法分析

优点：

- 公平性好，每个进程都能得到相等的 CPU 时间
- 响应时间有保证，适合交互式应用
- 实现简单，易于理解和实现

缺点：

- 上下文切换开销较大
- 时间片设置困难：太大会退化为 FCFS，太小会增加切换开销
- 不考虑进程的紧急程度

时间片调整：

- 对于交互式应用，应使用较小时间片以提高响应速度
- 对于批处理应用，可使用较长时间片以减少切换开销

need_resched 标志的重要性： 在 [RR_proc_tick] 中设置 [need_resched] 是必要的，它通知调度器当前进程的时间片已经用完，需要选择新的进程运行。

6. 拓展思考

优先级 RR 调度： 需要修改数据结构，为每个优先级维护一个独立的运行队列，高优先级队列空时才调度低优先级队列。

多核调度支持： 当前实现只支持单核，要支持多核需要为每个 CPU 维护独立的运行队列，实现负载均衡机制。

Challenge 1: 实现 Stride Scheduling 调度算法

1. 多级反馈队列 (MLFQ) 设计概要

多级反馈队列 (MLFQ) 的设计要点：

- 维护多个优先级队列，每个队列具有不同的时间片长度
- 新进程进入最高优先级队列
- 时间片用完的进程降级到下一优先级队列
- 阻塞后唤醒的进程回到高优先级队列
- 设置老化机制防止饥饿

2. Stride 算法公平性说明

Stride 算法中，每个进程 i 的步长为 $pass_i = BIG_STRIDE / share_i$ ，每次运行后其 $[lab6_stride]$ 增加 $[pass_i]$ 。经过足够多的时间片后，被调度次数 $[t_i]$ 与 $[share_i]$ 成正比，即 $[t_i] / [t_j] \approx share_i / share_j$ 。

3. Stride 实现过程

Stride 算法使用斜堆 (skew heap) 作为数据结构维护按 $[lab6_stride]$ 排序的进程列表，每次选择 $[lab6_stride]$ 最小的进程运行，运行后增加其 $[lab6_stride]$ 值。

重要知识点总结

1. 本实验中的重要知识点

- **调度器框架设计**：通过函数指针实现调度算法与调度框架的解耦
- **运行队列**：支持多种数据结构的统一接口
- **时间片管理**：RR 算法的时间片轮转机制
- **进程调度流程**：从时钟中断到进程切换的完整流程

2. OS 原理中对应的知识点

- **调度算法**：FCFS、RR、优先级调度、多级反馈队列、Stride 等
- **调度层次**：长程调度、中程调度、短程调度
- **调度指标**：周转时间、等待时间、响应时间、吞吐量等

3. 实验中未覆盖的 OS 原理知识点

- 死锁预防和避免算法
- 内存管理中的页面置换算法
- I/O 子系统的缓冲和调度策略
- 实时系统的调度算法 (如 EDF、RM)
- 多处理器调度和负载均衡

总结

本实验通过实现调度器框架，展示了如何在操作系统中实现调度算法与调度框架的解耦，使得切换调度算法变得更加容易。通过实现 RR 算法和 Stride 算法，加深了对调度算法原理的理解。实验不仅提高了编程能力，更重要的是理解了操作系统调度的本质和设计思想。