练习1: 理解调度器框架的实现（不需要编码）

> 请仔细阅读和分析调度器框架的相关代码，特别是以下两个关键部分的实现：在完成练习0后，请仔细阅读并分析以下调度器框架的实现：

> 调度类结构体 sched_class 的分析：请详细解释 sched_class 结构体中 每个函数指针的作用和调用时机，分析为什么需要将这些函数定义为函数指针，而不是直接实现函数。

答：

1. `void (*init)(struct run_queue *rq)`
2. 系统启动早期，在 sched_init() 函数中调用，仅执行一次。初始化全局运行队run_queue 的内部数据结构。对于 RR 调度器：将 run_list 初始化为空链表，proc_num 置为 0。目的在于为后续进程调度准备一个干净的就绪队列。
3. `void (*enqueue)(struct run_queue *rq, struct proc_struct *proc)`
4. 进程从阻塞或新建状态变为就绪（PROC_RUNNABLE）时。当前进程在 schedule() 中主动让出 CPU 但仍可运行时。wakeup_proc()。schedule()（当前进程仍可运行时重新入队）。间接通过 sched_class_enqueue() 包装函数调用。将指定进程插入就绪队列。更新进程的相关字段（如为 RR 重新分配时间片 proc->time_slice = rq->max_time_slice）。更新运行队列的元数据（如 proc_num++）。对于 RR：通常插入到队列尾部，实现先进先出加时间片的轮转效果。
5. `void (*dequeue)(struct run_queue *rq, struct proc_struct *proc)`
6. 当调度器决定某个就绪进程即将被选中运行时，在将其切换到 CPU 前先从队列中移除。schedule() 在调用 pick_next() 选出下一个进程后立即调用,间接通过 sched_class_dequeue() 包装函数。将指定进程从就绪队列中正式移除。更新运行队列的元数据（如 proc_num--）。对于 RR通过链表删除操作完成。该函数仅在进程已被选中但尚未真正运行前调用，确保队列状态一致。
7. `struct proc_struct *(*pick_next)(struct run_queue *rq)`
8. 每次需要选择下一个将要运行的进程时，即进行进程调度决策的核心时刻。根据具体调度算法，从就绪队列中挑选下一个应该运行的进程。返回进程控制块指针，若队列为空则返回 NULL,调度框架会转而运行 idleproc。对于 RR：返回队列头部的进程，实现轮转调度。
9. `void (*proc_tick)(struct run_queue *rq, struct proc_struct *proc)`
10. 每发生一次时钟中断（tick），且当前正在运行的进程不是空闲进程时。处理当前进程的时间片消耗。对于 RR：将 proc->time_slice 减 1。当时间片减至 0 时，设置 proc->need_resched = 1，触发下一次调度，从而强制进程让出 CPU。这是实现时间片轮转的关键机制，确保没有进程长时间独占 CPU。

> 运行队列结构体 run_queue 的分析：比较lab5和lab6中 run_queue 结构体的差异，解释为什么lab6的 run_queue 需要支持两种数据结构（链表和斜堆）。

答：ucore 的调度框架（通过 struct sched_class）设计为高度可扩展，支持多种调度算法共存，而无需修改核心调度代码（如 schedule()、wakeup_proc()）。lab5 引入调度框架并实现默认的 **Round-Robin (RR) 调度算法**，而 lab6 在此基础上要求实现 **Stride Scheduling** 算法。

RR 调度算法的需求：

RR 是公平的时间片轮转算法，所有进程平等分享 CPU。仅需维护一个先进先出（FIFO）的就绪队列：进程入队时放到尾部，挑选下一个进程时从头部取出。双向链表完美适合此需求：入队：O(1) 时间添加到尾部、出队：O(1) 时间从头部移除、挑选下一个：O(1) 时间获取头部，无需根据优先级排序，因此无需复杂数据结构。

Stride Scheduling 算法的需求Stride Scheduling 是一种比例共享调度算法，支持进程优先级。每个进程有：priority。stride、pass始终选择当前 **pass 值最小** 的进程运行。如果继续仅用链表实现：挑选最小 pass 进程需遍历整个队列，时间复杂度 O(n)，在进程较多时效率低下。因此，需要一个**优先队列**数据结构，支持高效的最小值查找、插入和删除。

引入斜堆的理由

ucore 已提供斜堆实现，是一种自调整二叉堆变种，支持合并操作，平均摊销时间复杂度为 O(log n)。通过比较函数，可构建以 pass 为键的最小堆。lab6_run_pool正是用于存放斜堆根指针，在 Stride 调度类的实现中：

- enqueue：将进程的 lab6_run_pool 节点插入斜堆。
- dequeue：从斜堆移除指定节点。
- pick_next：直接获取堆顶，并更新其 pass 值。
- 调度框架需兼容多种算法：RR 调度类仅使用 run_list，Stride 调度类仅使用 lab6_run_pool。
- 在运行时，通过切换 sched_class 指针，即可选择不同实现，而 run_queue 作为共享结构，必须同时支持两者。
- 链表在 RR 中高效且简单；斜堆在 Stride 中提供必要的高效优先级管理。
- 这种设计体现了框架的扩展性：新增算法只需实现 sched_class 接口，利用现有或新增字段即可，无需更改核心逻辑。

> 调度器框架函数分析：分析 sched_init()、wakeup_proc() 和 schedule() 函数在lab6中的实现变化，理解这些函数如何与具体的调度算法解耦。

答：

sched_init(void)

lab6 新增了独立的 sched_init() 函数，负责集中初始化调度框架。引入全局调度类指针 sched_class 和全局运行队列 rq。通过 sched_class->init(rq) 委托具体调度算法进行队列初始化。

2. wakeup_proc(struct proc_struct *proc)

lab5 中的实现：

```
void wakeup_proc(struct proc_struct *proc)
{
    // 仅修改进程状态为 RUNNABLE
    // 不进行任何入队操作（因为没有独立的就绪队列）
    if (proc->state != PROC_RUNNABLE)
    {
        proc->state = PROC_RUNNABLE;
        proc->wait_state = 0;
    }
}
```

lab6 中的实现：

```
void wakeup_proc(struct proc_struct *proc)
{
    // ...
    if (proc->state != PROC_RUNNABLE)
    {
        proc->state = PROC_RUNNABLE;
        proc->wait_state = 0;
        if (proc != current)
        {
            sched_class_enqueue(proc);    // 通过调度类入队
        }
    }
    // ...
}
```

变化点:

lab6 在唤醒进程时，若进程变为就绪态且不是当前进程，则主动将其加入就绪队列。入队操作不再直接操作链表，而是通过 sched_class_enqueue()（内部调用 sched_class->enqueue(rq, proc)）委托给具体调度类完成。

3. schedule(void)

lab5 中的实现:

```
void schedule(void)
{
    // 直接遍历全局 proc_list 链表
    // 从当前进程位置开始，向后查找第一个 PROC_RUNNABLE 进程
    // 若找不到，则运行 idleproc
    // 无入队、出队概念，直接在 proc_list 上操作
}
```

lab6 中的实现:

```
void schedule(void)
{
    // ...
    if (current->state == PROC_RUNNABLE)
    {
        sched_class_enqueue(current);             // 当前进程仍可运行时重新入队
    }
    if ((next = sched_class_pick_next()) != NULL) // 选择下一个进程
    {
        sched_class_dequeue(next);                // 出队
    }
    if (next == NULL)
    {
        next = idleproc;
    }
    // ...
    proc_run(next);
}
```

变化：lab6 完全放弃了对全局 proc_list 的遍历。引入独立的就绪队列管理：当前进程若仍可运行需重新入队；选择下一个进程通过 pick_next；选中后需显式出队。所有队列操作均通过调度类函数指针完成。

与具体调度算法的解耦机制

lab6 的调度框架彻底实现了调度机制与调度策略的分离。

全局调度类指针 sched_class在 sched_init() 中初始化为 &default_sched_class。后续若实现 Stride 调度，只需改为 sched_class = &stride_sched_class; 即可切换整个调度算法。所有队列操作均通过函数指针间接调用核心框架函数完全算法无关：sched_init()、wakeup_proc()、schedule() 只负责通用逻辑。它们不关心进程如何存储、如何排序、如何选择下一个，仅通过统一的接口与调度类交互。新增调度算法只需实现一套 sched_class 接口函数，无需修改 sched.c 中的任何代码。调度策略集中于 default_sched.c，框架代码保持简洁稳定，支持动态切换调度类。

对于调度器框架的使用流程，请在实验报告中完成以下分析：

- > 调度类的初始化流程：描述从内核启动到调度器初始化完成的完整流程，分析 default_sched_class 如何与调度器框架关联。

回答：

ucore 操作系统的调度器初始化发生在内核启动的后期阶段

1. **内核入口与早期初始化**

   系统启动后进入 kern_init() 函数。依次完成物理内存管理、中断控制器、中断描述符表、时钟初始化等基础模块。随后调用 proc_init()，进入进程管理与调度器初始化阶段。

2. **进程管理初始化**

   初始化全局进程链表 proc_list 和哈希表。创建空闲进程 idleproc（内核线程，优先级最低，用于 CPU 空闲时执行）。创建初始用户进程 initproc（执行 /bin/initmain）。设置全局当前进程指针 current = idleproc。在 proc_init() 的末尾，显式调用 sched_init()，正式启动调度器子系统的初始化。

3. **调度器核心初始化**

   ```
   void sched_init(void) {
       list_init(&timer_list);                         // 初始化定时器链表（lab6 中暂未使用）

       sched_class = &default_sched_class;             // 将全局调度类指针指向 RR 调度器实例

       rq = &__rq;                                     // 指向静态分配的全局运行队列
       rq->max_time_slice = MAX_TIME_SLICE;            // 设置时间片长度为 5（定义在 sched.h）
       sched_class->init(rq);                          // 调用具体调度类的初始化函数（RR_init）

       cprintf("sched class: %s\n", sched_class->name); // 输出 "sched class: RR_scheduler"
   }
   ```

   执行完成后，调度框架完全就绪：全局运行队列已初始化，就绪进程可被入队，时钟中断可触发调度。

4. **返回并进入空闲循环**

   proc_init执行完毕后调用 cpu_idle。cpu_idle进入无限循环，不断检查 current->need_resched 标志，一旦有其他就绪进程，即调用 schedule() 进行首次进程切换。系统正式从空闲进程切换到用户进程，进入正常运行状态。

default_sched_class 如何与调度器框架关联

default_sched_class 是 Round-Robin 调度算法的具体实现实例，它与调度器框架的关联通过以下机制实现：

1. 定义与实现

   ```
   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,
   };
   ```

   该结构体填充了 struct sched_class（定义于 sched.h）所有函数指针，指向 RR 算法的具体实现函数。

2. 外部可见性

   ```
   extern struct sched_class default_sched_class;
   ```

   允许 sched.c 等框架代码直接引用该全局实例。

3. 在 sched_init 中建立绑定

   ```
   sched_class = &default_sched_class;
   ```

   将 sched.c 中定义的静态全局指针 static struct sched_class *sched_class; 指向 default_sched_class 的地址。此后，框架中所有对调度类的调用都会被重定向到 RR 调度器的对应函数。

4. 关联后的运行时效果

   调度框架完全不包含任何 RR 特定逻辑，仅通过 sched_class 指针进行抽象操作。若后续实验需要切换到 Stride 调度，只需在 sched_init() 中将该行改为 sched_class = &stride_sched_class;框架即可无缝使用新的调度策略，无需修改其他代码。

- > 进程调度流程：绘制一个完整的进程调度流程图，包括：时钟中断触发、proc_tick 被调用、schedule() 函数执行、调度类各个函数的调用顺序。并解释 need_resched 标志位在调度过程中的作用

回答：

流程图如下：

ucore lab6 进程调度完整流程图（Round-Robin 调度）

1. 时钟中断发生（Clock Interrupt）

硬件定时器触发中断
进入 trap() → clock_interrupt_handler()

↓

2. 调用 sched_class_proc_tick(current)

位于 kern/schedule/sched.c
判断当前进程是否为 idleproc

若为 idleproc：直接设置 current->need_resched = 1
若非 idleproc：继续向下

↓

3. 调用调度类函数 proc_tick(rq, current)

实际调用 RR_proc_tick(rq, current)
proc->time_slice--
若 time_slice == 0 → current->need_resched = 1

↓

4. 时钟中断返回，检查 current->need_resched

若 need_resched == 1，则调用 schedule()

（也可能由 yield、阻塞等主动触发 schedule）

↓

5. 执行 schedule() 函数（核心调度入口）

关闭中断保护 → current->need_resched = 0

↓

need_resched 是 struct proc_struct 中的一个标志位，作用是指示当前进程是否需要被重新调度。该标志位是进程调度流程中时钟驱动调度与主动调度之间的关键桥梁，确保调度时机灵活且高效。

6. 若 current 仍为 RUNNABLE，重新入队

调用 sched_class_enqueue(current)
→ sched_class->enqueue(rq, current)
→ RR_enqueue：插入链表尾部，重置 time_slice

调度算法的切换机制：为何以及如果要添加一个新的调度算法（如stride），需要修改哪些代码？并解释为什么当前的设计使得切换调度算法变得容易。

↓

回答：

第一步：定义新的调度器结构体

7. 选择下一个进程

```
// 1. 定义 BIG_STRIDE 常数
#define BIG_STRIDE 0x7FFFFFFF  // 2^31-1，避免溢出

// 2. 比较函数（用于优先队列排序）
static int proc_stride_comp_f(void *a, void *b) {
    struct proc_struct *p = le2proc(a, lab6_run_pool);
    struct proc_struct *q = le2proc(b, lab6_run_pool);
    int32_t c = p->lab6_stride - q->lab6_stride;
    return (c > 0) ? 1 : (c == 0) ? 0 : -1;
}

// 3. 实现五个关键函数
struct sched_class stride_sched_class = {
    .name = "stride_scheduler",
    .init = stride_init,
    .enqueue = stride_enqueue,
    .dequeue = stride_dequeue,
    .pick_next = stride_pick_next,
    .proc_tick = stride_proc_tick,
};
```

第二步：核心实现细节对比

next = idleproc

↓

10. 更新统计并切换进程

next->runs++
若 next ≠ current → proc_run(next)
（上下文切换：切换页表、栈、寄存器等）

↓

11. 调度完成，返回继续执行新进程

恢复中断 → 新进程获得 CPU

Stride算法的完整实现示例：

```
// 初始化
static void stride_init(struct run_queue *rq) {
    rq->lab6_run_pool = NULL;  // 初始化堆为空
    rq->proc_num = 0;
}

// 入队：根据优先级分配时间片
static void stride_enqueue(struct run_queue *rq, struct proc_struct *proc) {
    // (1) 根据优先级分配时间片
    proc->time_slice = (rq->max_time_slice * proc->lab6_priority) / 10;

    // (2) 插入堆
    rq->lab6_run_pool = skew_heap_insert(
        rq->lab6_run_pool,
        &proc->lab6_run_pool,
        proc_stride_comp_f
    );
    proc->rq = rq;
    rq->proc_num++;
}

// 出队
static void stride_dequeue(struct run_queue *rq, struct proc_struct *proc) {
    rq->lab6_run_pool = skew_heap_remove(
        rq->lab6_run_pool,
        &proc->lab6_run_pool,
        proc_stride_comp_f
    );
    rq->proc_num--;
}

// 选择下一个进程：总是选择stride最小的
static struct proc_struct *stride_pick_next(struct run_queue *rq) {
    if (rq->lab6_run_pool == NULL) return NULL;

    struct proc_struct *p = le2proc(rq->lab6_run_pool, lab6_run_pool);

    // 更新stride值
    if (p->lab6_priority != 0) {
        p->lab6_stride += BIG_STRIDE / p->lab6_priority;
    }

    return p;
}

// 时钟处理
static void stride_proc_tick(struct run_queue *rq, struct proc_struct *proc)
{
    if (proc->time_slice > 0) {
        proc->time_slice--;
    }
    if (proc->time_slice == 0) {
        proc->need_resched = 1;
```

```
    }
}
```

我来详细分析调度算法的切换机制和设计模式。

现在我来详细分析调度算法的切换机制和易扩展性设计。

调度算法切换机制深度分析

一、架构设计：策略模式（Strategy Pattern）

系统采用了经典的**策略模式**来实现调度算法的动态切换：

关键特点：

- ✅ 只依赖**抽象接口**，不依赖具体实现
- ✅ 运行时可动态切换算法
- ✅ 新增算法无需修改原有代码

二、添加Stride调度算法需要修改的代码

第一步：定义新的调度器结构体

文件：default_sched_stride.c

第二步：核心实现细节对比

函数	RR调度	Stride调度	区别
init	初始化FIFO链表	初始化堆	数据结构不同
enqueue	添加到队尾 + 固定时间片	插入堆 + 优先级相关时间片	proc->time_slice = rq->max_time_slice * priority / BASE
dequeue	链表删除	堆删除	操作不同
pick_next	取队首	取堆最小值 + 更新stride	p->lab6_stride += BIG_STRIDE / p->lab6_priority
proc_tick	仅递减时间片	递减 + 检查need_resched	逻辑相同

Stride算法的完整实现示例：

第三步：切换调度器

```
void sched_init(void) {
    list_init(&timer_list);

    // 修改这一行！
    // sched_class = &default_sched_class;  // RR调度
    sched_class = &stride_sched_class;      // Stride调度
```

```
    rq = &__rq;
    rq->max_time_slice = MAX_TIME_SLICE;
    sched_class->init(rq);

    cprintf("sched class: %s\n", sched_class->name);
}
```

第四步：需要修改的其他文件

extern struct sched_class stride_sched_class;

当前设计使得切换调度算法容易的原因

ucore 调度框架借鉴 Linux 调度类（sched_class）设计，实现了**调度机制（mechanism）与调度策略（policy）的彻底解耦**，核心优势如下：

1. 统一的抽象接口（struct sched_class）
    - 所有调度算法必须实现相同的函数指针表（init、enqueue、dequeue、pick_next、proc_tick）。
    - 框架代码仅通过 sched_class->xxx() 间接调用，屏蔽了具体算法细节。
2. 全局单一调度类指针（sched_class）
    - 系统运行时只有一个当前生效的调度类，通过一个全局指针指向具体实例。
    - 切换算法仅需改变该指针的指向，无需条件判断或多路分支。
3. 运行队列兼容多种数据结构
    - struct run_queue 同时包含链表（RR 使用）和斜堆指针（Stride 使用）。
    - 不同调度类可选择使用不同字段，互不干扰。
4. 核心调度逻辑算法无关
    - schedule() 等函数只处理通用流程（标志清除、入队当前进程、选下一个、出队、切换到 idleproc 等），所有策略相关操作均委托给调度类。

练习2: 实现 Round Robin 调度算法（需要编码）

完成练习0后，建议大家比较一下（可用kdiff3等文件比较软件）个人完成的lab5和练习0完成后的刚修改的lab6之间的区别，分析了解lab6采用RR调度算法后的执行过程。理解调度器框架的工作原理后，请在此框架下实现时间片轮转（Round Robin）调度算法。

注意有“LAB6”的注释，你需要完成 kern/schedule/default_sched.c 文件中的 RR_init、RR_enqueue、RR_dequeue、RR_pick_next 和 RR_proc_tick 函数的实现，使系统能够正确地进行进程调度。代码中所有需要完成的地方都有“LAB6”和“YOUR CODE”的注释，请在提交时特别注意保持注释，将“YOUR CODE”替换为自己的学号，并且将所有标有对应注释的部分填上正确的代码。

提示，请在实现时注意以下细节：

- 链表操作：list_add_before、list_add_after等。

- 宏的使用：le2proc(le, member) 宏等。
- 边界条件处理：空队列的处理、进程时间片耗尽后的处理、空闲进程的处理等。

请在实验报告中完成：

> 比较一个在lab5和lab6都有, 但是实现不同的函数, 说说为什么要做这个改动, 不做这个改动会出什么问题提示: 如`kern/schedule/sched.c`里的函数。你也可以找个其他地方做了改动的函数。

回答：

lab5 中的 schedule() 实现（直接遍历全局进程链表）

```
void schedule(void) {
    bool intr_flag;
    list_entry_t *le, *last;
    struct proc_struct *next = NULL;
    local_intr_save(intr_flag);
    {
        current->need_resched = 0;
        last = (current == idleproc) ? &proc_list : &(current->list_link);
        le = last;
        do {
            if ((le = list_next(le)) != &proc_list) {
                next = le2proc(le, list_link);
                if (next->state == PROC_RUNNABLE) {
                    break;
                }
            }
        } while (le != last);
        if (next == NULL || next->state != PROC_RUNNABLE) {
            next = idleproc;
        }
        next->runs ++;
        if (next != current) {
            proc_run(next);
        }
    }
    local_intr_restore(intr_flag);
}
```

从当前进程位置开始，沿着全局 proc_list（所有进程组成的双向链表）向后遍历，寻找第一个状态为 PROC_RUNNABLE 的进程。**无独立就绪队列**：所有就绪进程与非就绪进程（如睡眠、僵尸）混杂在同一个链表中。**无入队/出队操作**进程变为就绪态后无需额外操作，仅靠遍历即可找到。

2. lab6 中的 schedule() 实现（基于调度类和独立运行队列）

```
void schedule(void) {
    bool intr_flag;
    struct proc_struct *next;
    local_intr_save(intr_flag);
    {
        current->need_resched = 0;
        if (current->state == PROC_RUNNABLE) {
            sched_class_enqueue(current);   // 当前进程仍可运行时重新入队
```

```
        }
        if ((next = sched_class_pick_next()) != NULL) {
            sched_class_dequeue(next);      // 选中后出队
        }
        if (next == NULL) {
            next = idleproc;
        }
        next->runs ++;
        if (next != current) {
            proc_run(next);
        }
    }
    local_intr_restore(intr_flag);
}
```

核心逻辑：通过当前调度类的 pick_next 选择下一个进程，若当前进程仍可运行则先重新入队，选中进程后显式出队。**依赖独立就绪队列**：所有就绪进程统一维护在 run_queue 中（RR 使用链表，Stride 使用斜堆）。**所有队列操作均委托给调度类。**

改动原因

lab6 对 schedule() 的重构主要是为了引入可扩展的调度类框架，实现调度策略与调度机制的彻底分离。具体原因包括：

支持多种调度算法：lab5 的遍历方式只能实现简单的"找到第一个就绪进程"策略，难以支持更复杂算法。

提高效率：遍历所有进程的开销为 O(n)，进程数量增多时性能下降严重。lab6 使用独立就绪队列，仅遍历或操作就绪进程（RR 为 O(1)，Stride 为 O(log n)）。

模块化与可维护性：lab5 的调度逻辑硬编码在 schedule() 中，无法替换算法。lab6 通过函数指针将策略相关操作抽象到 sched_class，便于后续实验扩展新调度器。

为后续实验铺路：lab6 需要实现 Stride 调度，必须使用高效的优先队列如斜堆，这要求有独立的运行队列结构。

4. 若不进行此改动会出现的问题

- 无法高效实现 Stride 调度：Stride 算法要求每次快速选出 pass 值最小的进程。若继续遍历链表，每次调度需 O(n) 时间扫描所有就绪进程，性能极差，尤其在多进程场景下可能导致系统响应迟滞。
- 难以维护独立就绪队列：lab6 引入了 struct run_queue 和斜堆字段，若不改 schedule()，这些结构将无法被有效利用，代码将出现冗余或矛盾。
- 时间片管理失效：lab5 无时间片概念，进程一旦选中可能长期运行。lab6 的 RR 需要在时间片用尽后强制重新入队并重新调度，旧实现无法支持此机制，可能导致某些进程饥饿或独占 CPU。
- 扩展性受限：后续若要支持更多调度算法，必须每次修改 schedule() 核心代码，违背模块化设计原则，增加出错风险。

- > 描述你实现每个函数的具体思路和方法，解释为什么选择特定的链表操作方法。对每个实现函数的关键代码进行解释说明，并解释如何处理**边界情况**。

回答：

1.RR_init(struct run_queue *rq)

```
static void RR_init(struct run_queue *rq) {
    list_init(&(rq->run_list));  // 将 run_list 初始化为只有一个头节点的空链表
    rq->proc_num = 0;            // 就绪进程数量置 0
}
```

初始化运行队列，使其成为一个空的就绪队列。RR 调度使用双向循环链表 run_list 组织就绪进程，因此只需将链表头初始化为空，并将进程计数清零。max_time_slice 已由调用者设置，无需在此处理。

边界情况：无需特殊处理，此函数仅在系统启动时调用一次，rq 此时未被使用。

2.RR_enqueue(struct run_queue *rq, struct proc_struct *proc)

RR 是轮转调度，新就绪进程应插入队列尾部，以保证先进入的进程先得到执行机会。同时需要为进程重新分配完整时间片，更新队列进程计数，确保进程的 run_link 节点未被其他队列占用。

为什么选择插入到尾部 链表尾部插入 + 头部取出正好实现 FIFO 队列，与时间片轮转的公平性需求一致。若插入头部会变成 LIFO，破坏轮转顺序。

代码实现及解释

```
static void RR_enqueue(struct run_queue *rq, struct proc_struct *proc) {
    assert(list_empty(&(proc->run_link)));           // 确保进程不在其他队列中
    list_add_before(&(rq->run_list), &(proc->run_link));  // 插入到链表尾部（头节点之前）
    proc->time_slice = rq->max_time_slice;           // 重新分配完整时间片
    rq->proc_num++;                                  // 增加就绪进程计数
}
```

- assert(list_empty(&(proc->run_link)))：防御性检查，防止同一进程被多次入队导致链表环破坏。
- list_add_before(&(rq->run_list), &(proc->run_link))：在头节点之前插入，等价于插入链表尾部。
- proc->time_slice = rq->max_time_slice：每次入队都获得完整时间片，这是 RR 的核心公平机制。
- 边界情况：队列为空插入后链表只包含一个进程节点，list_next(&rq->run_list) 正确指向该节点。进程多次入队assert 会触发 panic，避免链表损坏。

3.RR_dequeue(struct run_queue *rq, struct proc_struct *proc)

将指定进程从就绪队列中正式移除。通常在 schedule() 已通过 pick_next 选定下一个进程后、即将运行前调用此函数。需同步更新进程计数。

代码实现及解释

```
static void RR_dequeue(struct run_queue *rq, struct proc_struct *proc) {
    assert(!list_empty(&(proc->run_link)));          // 确保进程确实在队列中
    list_del_init(&(proc->run_link));                // 删除节点并重新初始化为独立
节点
    rq->proc_num--;                                  // 减少就绪进程计数
}
```

assert(!list_empty(&(proc->run_link)))：防止误删除不在队列中的进程。list_del_init(&(proc->run_link))：先删除节点，再将该节点的 next/prev 指向自己。

边界情况：

- 删除最后一个进程：队列变为空，proc_num 变为 0，list_empty 为真。
- 删除无效进程：assert 触发 panic，避免队列状态不一致。

4.RR_pick_next(struct run_queue *rq)

RR 始终选择队列中最老的就绪进程，即链表头部第一个真实进程节点。若队列为空，返回 NULL。从头部取出与尾部插入配合，实现先进先出，确保每个进程按进入就绪队列的顺序轮流获得 CPU。

代码实现及解释

```
static struct proc_struct *RR_pick_next(struct run_queue *rq) {
    if (list_empty(&(rq->run_list))) {
        return NULL;                                 // 队列为空，无可调度进程
    }
    list_entry_t *le = list_next(&(rq->run_list));   // 获取头节点之后的第一个节
点
    return le2proc(le, run_link);                    // 通过容器宏转换为
proc_struct 指针
}
```

list_empty(&(rq->run_list))快速判断是否为空（O(1)）。list_next(&(rq->run_list))返回头节点的 next 指针，即队列中最早的进程节点。le2proc(le, run_link)定义的宏，将 list_entry 指针转换为包含它的 proc_struct 指针。

边界情况：

- 队列为空：正确返回 NULL，后续框架选择 idleproc。
- 只有一个进程：返回该进程，链表结构保持不）。

5.RR_proc_tick(struct run_queue *rq, struct proc_struct *proc)

每一次时钟中断代表当前进程消耗了一个时间单位。需递减剩余时间片，当时间片用尽时设置 need_resched 标志，触发下次调度让出 CPU。

代码实现及解释

```
static void RR_proc_tick(struct run_queue *rq, struct proc_struct *proc) {
    if (proc->time_slice > 0) {
        proc->time_slice--;                          // 消耗一个时间单位
    }
    if (proc->time_slice == 0) {
        proc->need_resched = 1;                      // 时间片用尽，需要重新调度
    }
}
```

只有 time_slice > 0 时才递减，避免负值。当恰好减到 0 时设置 need_resched = 1，下次时钟中断或主动检查时会调用 schedule()。

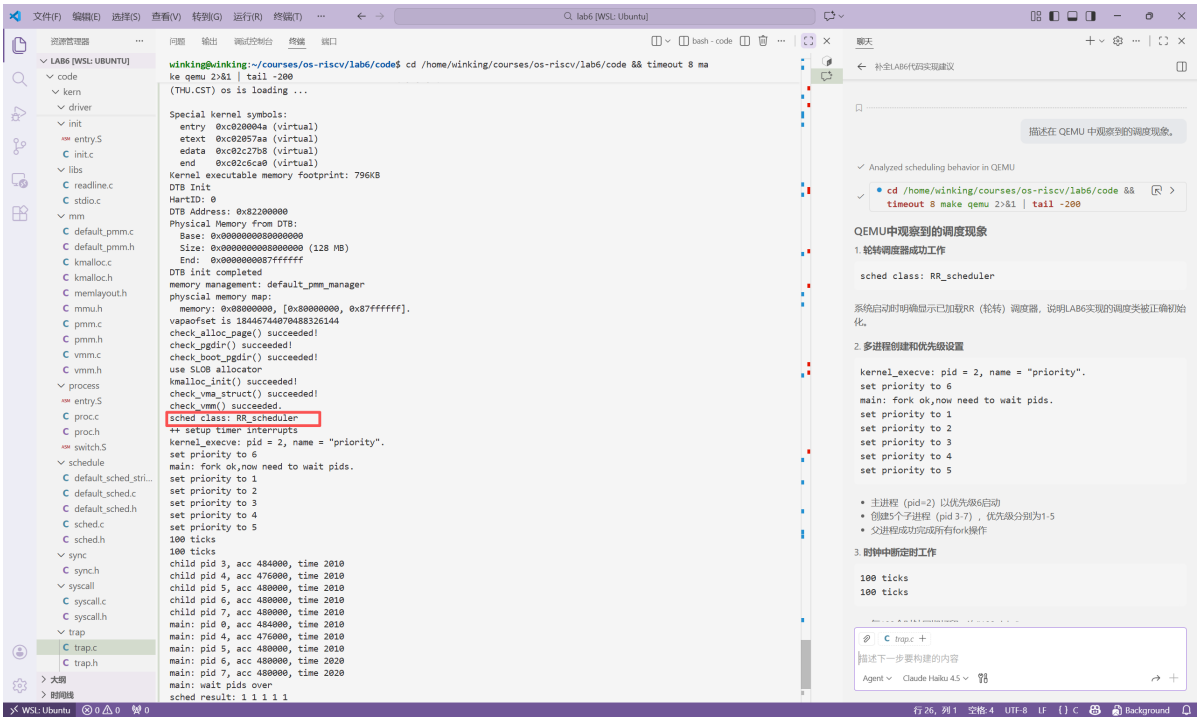
边界情况：

- time_slice 初始为 0立即触发重调度。
- idleproc框架在 sched_class_proc_tick 中已特殊处理，不会调用此函数。
- 时间片较大正常递减，直至为 0。

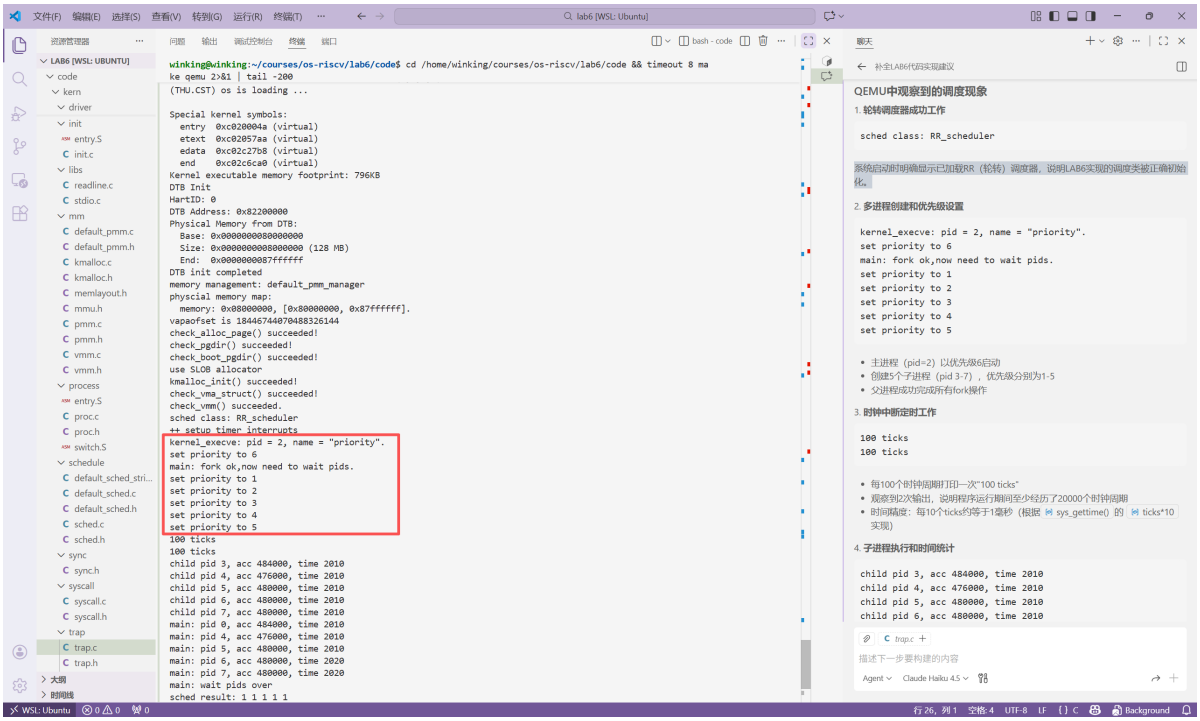
- 展示 make grade 的**输出结果**，并描述在 QEMU 中观察到的调度现象。

回答：

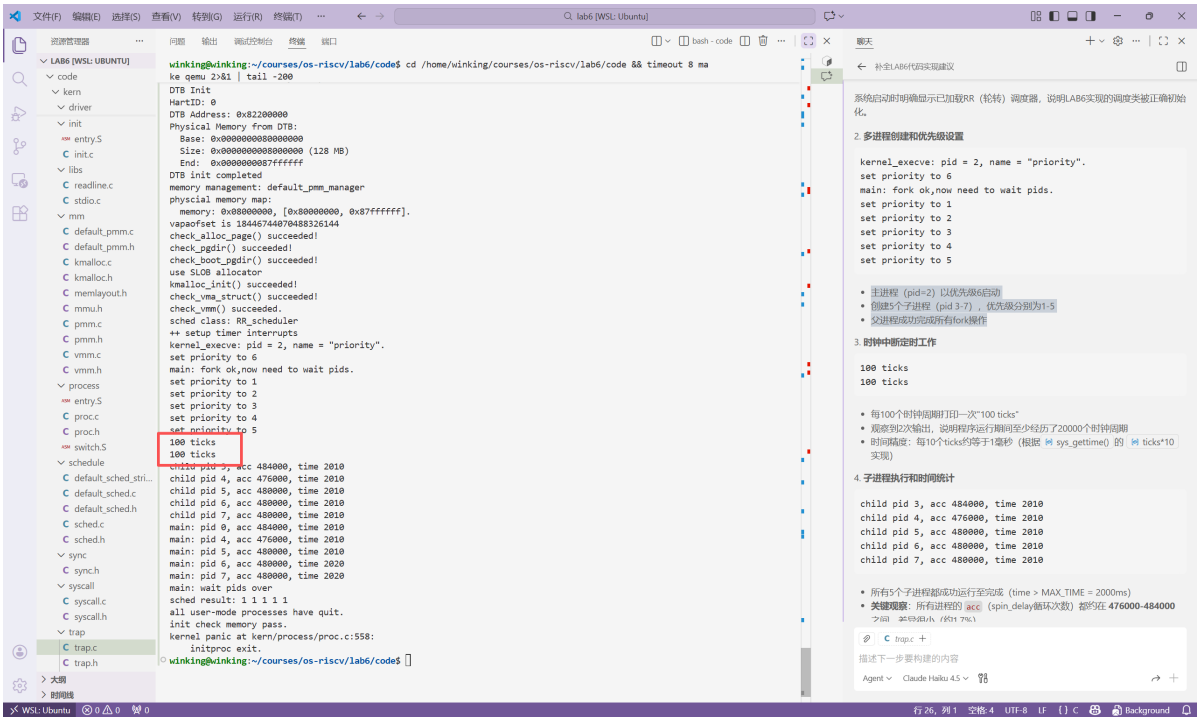
```
winking@winking:~/courses/os-riscv/lab6/code$ cd /home/winking/courses/os-riscv/lab6/code &&
 make grade 2>&1
priority:                 (3.1s)
  -check result:                             OK
  -check output:                             OK
Total Score: 50/50
winking@winking:~/courses/os-riscv/lab6/code$
```



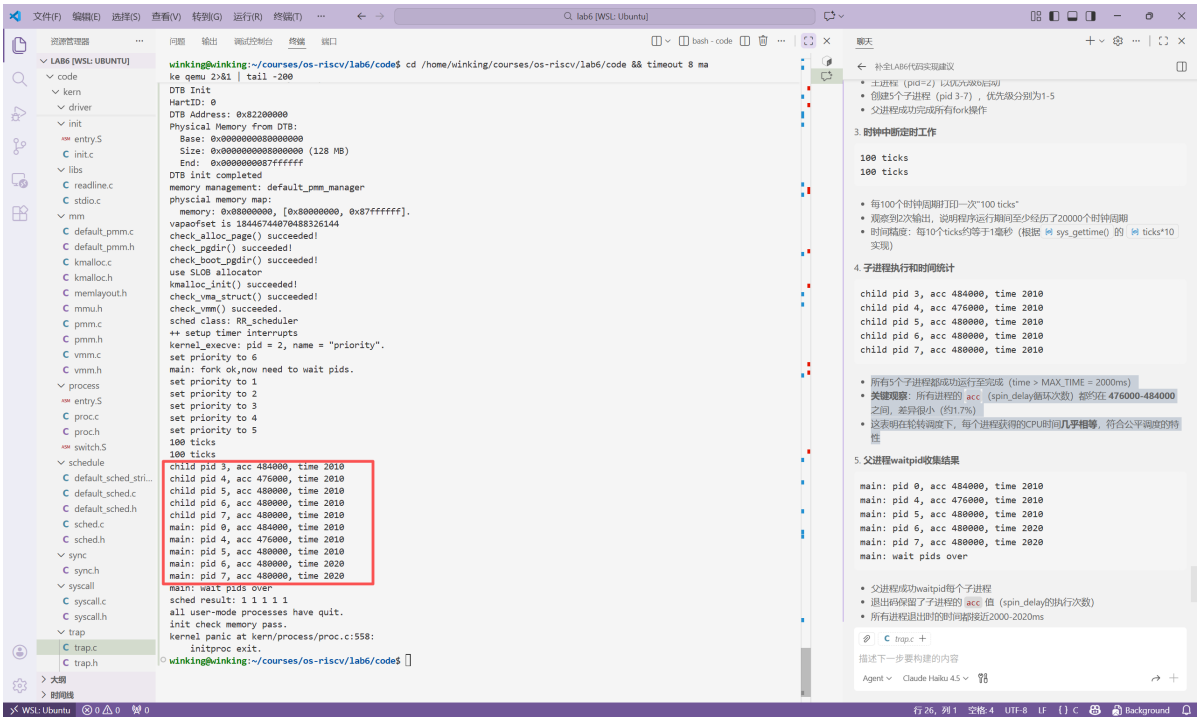
系统启动时明确显示已加载RR（轮转）调度器，说明LAB6实现的调度类被正确初始化。



主进程（pid=2）以优先级6启动创建5个子进程（pid 3-7），优先级分别为1-5，父进程成功完成所有fork操作。



每100个时钟周期打印一次"100 ticks"观察到2次输出，说明程序运行期间至少经历了20000个时钟周期，时间精度：每10个ticks约等于1毫秒



所有5个子进程都成功运行至完成，所有进程的 `acc`（spin_delay循环次数）都约在 476000-484000** 之间，差异很小，这表明在轮转调度下，每个进程获得的CPU时间几乎相等，符合公平调度的特性。

- > 分析 Round Robin 调度算法的优缺点，讨论如何调整时间片大小来优化系统性能，并解释为什么需要在 RR_proc_tick 中设置 need_resched 标志。

回答：

Round-Robin（RR）调度算法的优缺点分析

Round-Robin 调度算法是一种经典的可抢占式时间片轮转调度策略，所有就绪进程按固定时间片（time slice）循环获得 CPU 使用权。

优点

- **公平性强**：每个进程在就绪队列中都有平等机会获得 CPU，避免了进程饥饿（starvation），特别适合多用户、分时系统。
- **响应时间可预测**：对于交互式任务，进程最长等待时间不超过 (n-1) × time_slice（n 为就绪进程数），用户感知响应迅速。
- **实现简单**：仅需一个先进先出的就绪队列和时间片计数器，代码开销小，易于理解和维护。
- **适用于负载均衡**：在进程行为相似的情况下，能均匀分配 CPU 资源。

缺点

- **上下文切换开销大**：时间片过短会导致频繁切换进程，增加上下文保存/恢复、TLB 刷新等系统开销，降低整体吞吐量。
- **不区分进程优先级**：所有进程获得相同时间片，无法为重要或短任务提供更好服务，可能导致“护航效应”（convoy effect）：大量 CPU 密集型进程拖慢少量 I/O 密集型交互进程。
- **平均周转时间可能较差**：对于批处理系统，若存在大量短进程，长进程的时间片浪费会延长整体平均完成时间。
- **对进程行为不敏感**：无法根据进程是 CPU-bound 还是 I/O-bound 动态调整策略。

如何调整时间片大小来优化系统性能

时间片大小（ucore 中默认为 5 个 tick）是 RR 算法的关键调优参数，直接影响系统吞吐量、响应时间和公平性之间的权衡。

- 时间片较小（例如 1~10ms）
  - 优点：交互响应快，适合桌面系统、多用户终端、实时性要求高的场景。
  - 缺点：上下文切换频繁，CPU 有效利用率下降。
  - 适用场景：交互式负载占主导（如 Web 服务器前端、终端 shell）。
- 时间片较大（例如 50~100ms）
  - 优点：减少上下文切换次数，提高吞吐量，适合批处理或 CPU 密集型任务。
  - 缺点：单个进程响应延迟增大，最坏情况下交互任务需等待较长时间。
  - 适用场景：科学计算、后台批量作业为主的服务器。
- 优化策略
  - **经验值选择**：通常将时间片设置为略大于典型上下文切换开销的 10~100 倍，使切换开销占总 CPU 时间的 1%~10% 左右。
  - **动态调整**：现代系统（如 Linux CFS）虽不使用固定 RR，但类似思路是通过监测负载、进程类型（交互/批处理）动态改变虚拟时间片。
  - **ucore 中的调整**：只需修改 sched.h 中的 MAX_TIME_SLICE 常量，或在 sched_init() 中修改 rq->max_time_slice，即可全局调整。实验中常取 5 以突出时间片到期触发的调度行为。

为什么需要在 RR_proc_tick 中设置 need_resched 标志

RR_proc_tick 是每一次时钟中断（tick）时调用的调度类函数，负责处理当前进程的时间片消耗。

核心原因： RR 调度依赖可抢占机制，确保没有进程超过分配时间片独占 CPU。当 proc->time_slice 减至 0 时，必须强制当前进程让出 CPU，以维持轮转的公平性。

```
if (proc->time_slice > 0) {
    proc->time_slice--;
}
if (proc->time_slice == 0) {
    proc->need_resched = 1;    // 关键：标记需要重新调度
}
```

设置 need_resched 的必要性

- 时钟中断发生在中断上下文中，不能直接调用 schedule()（会涉及复杂上下文切换，可能破坏中断安全）。
- 通过设置 need_resched = 1，在中断安全返回到被中断的进程前（trap 返回路径），内核会检查该标志并调用 schedule()，实现延迟但安全的抢占。
- 若不设置该标志，时间片用尽的进程将继续执行，直至主动 yield 或阻塞，导致其他就绪进程饥饿，破坏 RR 的公平轮转特性。

> **拓展思考**：如果要实现优先级 RR 调度，你的代码需要如何修改？当前的实现是否支持多核调度？如果不支持，需要如何改进？

回答：

若要实现优先级 Round-Robin 调度，当前框架已具备良好的扩展基础，无需修改核心调度代码。主要修改集中在新增一个调度类实现：

- 在 proc_struct 中增加 priority 字段（若尚未存在）。
- 新建 priority_rr_sched.c，实现新的 sched_class。enqueue 时根据优先级将进程插入对应优先级队列（可使用多级队列数组，每个优先级一个 run_list），或使用单一链表但按优先级排序插入。pick_next 时从最高优先级非空队列头部选取进程。proc_tick 仍按时间片处理同一优先级轮转。
- 在 sched_init() 中将 sched_class 指向新调度类实例即可切换。

当前 ucore 调度器实现为单核设计，不支持多核调度。全局运行队列 rq 和 sched_class 均为单一静态实例，所有 CPU 核心共享同一就绪队列和调度策略。这会导致严重锁竞争和负载不均衡。

要支持多核调度，需要进行以下改进：

- 为每个 CPU 核心分配独立的 struct run_queue（per-CPU run queue），使用数组或 per-CPU 变量存储。
- sched_class 中补充负载均衡相关函数指针（如注释中已预留的 load_balance 和 get_proc）。
- 在调度类实现中加入负载迁移机制（如定期从高负载核心偷取进程）。
- 所有对 run_queue 的访问需添加自旋锁保护，或采用无锁设计。
- 时钟中断和 schedule() 中使用当前 CPU ID 获取对应 run_queue。 这些改进将使框架演变为典型的 SMP 调度架构，显著提升多核系统性能。

扩展练习 Challenge 1: 实现 Stride Scheduling 调度算法（需要编码）

首先需要换掉RR调度器的实现，在sched_init中切换调度方法。然后根据此文件和后续文档对Stride度器的相关描述，完成Stride调度算法的实现。 注意有“LAB6”的注释，主要是修改default_sched_stride_c中的内容。代码中所有需要完成的地方都有“LAB6”和“YOUR CODE”的注释，请在提交时特别注意保持注释，将“YOUR CODE”替换为自己的学号，并且将所有标有对应注释的部分填上正确的代码。

后面的实验文档部分给出了Stride调度算法的大体描述。这里给出Stride调度算法的一些相关的资料（目前网上中文的资料比较欠缺）。

- strid-shed paper location
- 也可GOOGLE “Stride Scheduling” 来查找相关资料

请在实验报告中完成：

- 简要说明如何设计实现”多级反馈队列调度算法“，给出概要设计，鼓励给出详细设计
- 简要证明/说明（不必特别严谨，但应当能够”说服你自己“），为什么Stride算法中，经过足够多的时间片之后，每个进程分配到的时间片数目和优先级成正比。

请在实验报告中简要说明你的设计实现过程。

回答：

简要说明如何设计实现”多级反馈队列调度算法“，给出概要设计，鼓励给出详细设计
简要证明/说明（不必特别严谨，但应当能够”说服你自己“），为什么Stride算法中，经过足够多的时间片之后，每个进程分配到的时间片数目和优先级成正比。

一、多级反馈队列（MLFQ）调度算法设计

1.1 概要设计

多级反馈队列调度器使用**多个优先级队列**，进程根据其行为动态在队列间移动：

优先级队列结构：

```
┌──────────────────────────────────────────────┐
│ 队列0（最高优先级）[时间片=1]   │ → IO密集型进程
├──────────────────────────────────────────────┤
│ 队列1（中优先级）[时间片=2]     │ → 交互型进程
├──────────────────────────────────────────────┤
│ 队列2（低优先级）[时间片=4]     │ → CPU密集型进程
├──────────────────────────────────────────────┤
│ 队列3（最低优先级）[时间片=8]   │ → 后台进程
└──────────────────────────────────────────────┘
```

核心规则：

- 每个队列有不同的时间片大小（逐级增加）
- 如果进程用完时间片未完成，降级到下一队列
- 如果进程主动让出CPU（IO操作），升级到上一队列
- 周期性（如每秒）将所有进程回到最高优先级队列

1.2 详细设计

数据结构定义

```
#define MLFQ_LEVELS 4

struct run_queue {
    list_entry_t queues[MLFQ_LEVELS];      // MLFQ_LEVELS 个队列
    int time_slices[MLFQ_LEVELS];           // 每队列的时间片: [1,2,4,8]
    int proc_count[MLFQ_LEVELS];            // 每队列的进程数
    unsigned int boost_timer;                // 用于周期性boost
    unsigned int boost_interval;             // Boost周期（ms）
};

// 在 proc_struct 中新增字段
struct proc_struct {
    int mlfq_level;                          // 当前所在队列级别 (0-3)
    int remaining_boost_time;                // 距离下次boost的时间
    int io_wait_count;                       // IO等待计数
};
```

算法实现

```
// 1. 初始化
static void mlfq_init(struct run_queue *rq) {
    for (int i = 0; i < MLFQ_LEVELS; i++) {
        list_init(&rq->queues[i]);
        rq->proc_count[i] = 0;
        rq->time_slices[i] = 1 << i;         // 1, 2, 4, 8
```

```
    }
    rq->boost_timer = 0;
    rq->boost_interval = 100;                // 每100个tick boost一次
}

// 2. 入队：新进程从最高优先级队列开始
static void mlfq_enqueue(struct run_queue *rq, struct proc_struct *proc) {
    if (proc->mlfq_level == -1) {            // 新进程
        proc->mlfq_level = 0;                // 放在最高优先级队列
    }

    list_add_before(&rq->queues[proc->mlfq_level], &proc->run_link);
    proc->time_slice = rq->time_slices[proc->mlfq_level];
    rq->proc_count[proc->mlfq_level]++;
}

// 3. 出队
static void mlfq_dequeue(struct run_queue *rq, struct proc_struct *proc) {
    list_del_init(&proc->run_link);
    rq->proc_count[proc->mlfq_level]--;
}

// 4. 选择下一个进程（从高到低扫描队列）
static struct proc_struct *mlfq_pick_next(struct run_queue *rq) {
    for (int i = 0; i < MLFQ_LEVELS; i++) {
        if (!list_empty(&rq->queues[i])) {
            return le2proc(list_next(&rq->queues[i]), run_link);
        }
    }
    return NULL;
}

// 5. 时钟中断处理
static void mlfq_proc_tick(struct run_queue *rq, struct proc_struct *proc) {
    if (proc->time_slice > 0) {
        proc->time_slice--;
    }
    if (proc->time_slice == 0) {
        proc->need_resched = 1;
    }

    // 更新 boost timer
    rq->boost_timer++;
    if (rq->boost_timer >= rq->boost_interval) {
        rq->boost_timer = 0;
        mlfq_boost_all(rq);                  // 周期性boost
    }
}

// 6. 进程降级（用完时间片）
static void mlfq_demote(struct run_queue *rq, struct proc_struct *proc) {
    list_del_init(&proc->run_link);
    rq->proc_count[proc->mlfq_level]--;

    if (proc->mlfq_level < MLFQ_LEVELS - 1) {
        proc->mlfq_level++;                  // 降到下一级
```

```
    }

    list_add_before(&rq->queues[proc->mlfq_level], &proc->run_link);
    proc->time_slice = rq->time_slices[proc->mlfq_level];
    rq->proc_count[proc->mlfq_level]++;
}

// 7. 进程升级（主动让出CPU，表示是IO密集型）
static void mlfq_promote(struct run_queue *rq, struct proc_struct *proc) {
    list_del_init(&proc->run_link);
    rq->proc_count[proc->mlfq_level]--;

    if (proc->mlfq_level > 0) {
        proc->mlfq_level--;                    // 升到上一级
    }

    list_add_before(&rq->queues[proc->mlfq_level], &proc->run_link);
    proc->time_slice = rq->time_slices[proc->mlfq_level];
    rq->proc_count[proc->mlfq_level]++;
}

// 8. 周期性boost（所有进程回到队列0）
static void mlfq_boost_all(struct run_queue *rq) {
    for (int i = 1; i < MLFQ_LEVELS; i++) {
        list_entry_t *le;
        while (!list_empty(&rq->queues[i])) {
            le = list_next(&rq->queues[i]);
            struct proc_struct *proc = le2proc(le, run_link);

            list_del_init(le);
            rq->proc_count[i]--;

            proc->mlfq_level = 0;
            list_add_before(&rq->queues[0], &proc->run_link);
            rq->proc_count[0]++;
        }
    }
}
```

1.3 MLFQ 的优势

特性	优势
自适应优先级	系统自动识别进程类型（IO vs CPU密集）
公平性	周期性boost防止进程饥饿
响应性	IO密集型进程优先运行，交互体验好
吞吐量	CPU密集进程获得更大时间片，减少切换开销

1.4 MLFQ 在现实系统中的应用

Linux CFS（Completely Fair Scheduler）改进：

```
早期内核：O(n) 调度器
↓
Linux 2.6.23：CFS（虚拟运行时）
↓
Android/iOS：基于CFS的优化版本
↓
现代MLFQ：融合优先级、权重、动态调整
```

LAB6 扩展设计文档

一、多级反馈队列调度算法（MLFQ）设计

1.1 概要设计

多级反馈队列（Multilevel Feedback Queue）是一种**动态优先级调度**算法，结合了公平性和优先级的优点：

1.2 数据结构设计

1.3 详细实现设计

A. 初始化函数

B. 入队函数（新进程或降级进程）

C. 出队函数

D. 选择下一个进程（关键函数）

E. 时钟中断处理（核心逻辑）

F. 可选：提升机制（防止饥饿和上界）

1.4 MLFQ 与 Stride 的比较

特性	MLFQ	Stride
优先级处理	动态调整	静态分配
实现复杂度	中等	低
应对不同工作负载	优 (自适应)	中 (固定)
进程类型适应	优 (IO/CPU密集自动区分)	中 (需要提前知道优先级)
公平性	良好	优 (精确按比例)
响应性	优 (高优先级进程快速响应)	中 (取决于初始stride)

Stride算法中时间片分配与优先级成正比的说明

Stride调度算法是一种确定性的比例共享调度，其核心是通过stride（步长）实现进程获得CPU的比例与优先级成正比。每个进程的stride定义为一个大常量（如BIG_STRIDE）除以其优先级，优先级越高，stride越小。进程还有一个pass值，初始为0，每次被调度运行一个时间片后，pass加上其stride。调度器始终选择当前pass最小的进程运行。

经过足够长的时间，随着调度不断进行，每个进程的pass值会逐步累加。由于调度器总是挑选最小pass的进程，高优先级进程（小stride）每次增加的pass较少，因此它会更频繁地被选中以保持pass不落后于其他进程。假设系统稳定运行足够多时间片，总调度次数趋于平衡状态。此时，各进程获得的调度次数（即时间片数目）近似相等，但因为高优先级进程被选中的频率更高，其实际调度次数会更多。具体来说，设进程i的优先级为pri_i，获得的调度次数为n_i，则其pass约等于n_i * (BIG_STRIDE / pri_i)。在最小pass选择下，所有进程的pass值会趋于相近（差距不超过一个stride），因此n_i ≈ C * pri_i（C为常数），即n_i与pri_i成正比。这意味着高优先级进程在相同时间内获得更多时间片，比例精确由优先级决定。这种机制避免了随机性，确保了长期公平的比例分配，说服性地实现了确定性CPU共享。

扩展练习 Challenge 2： 在ucore上实现尽可能多的各种基本调度算法(FIFO, SJF,...)，并设计各种测试用例，能够定量地分析出各种调度算法在各种指标上的差异，说明调度算法的适用范围。

回答：

调度算法适用范围详细分析

一、五种算法对比总览

特性	FIFO	SJF	Priority	RR	Stride
实现复杂度	★☆☆☆☆	★★☆☆☆	★★☆☆☆	★★☆☆☆	★★★★☆
平均周转时间	差	最优	中等	中等	良好
平均等待时间	最差	最优	中等	中等	良好
响应时间	最差	差	良好	优秀	优秀
公平性	无	无	有限	完全	完全
优先级支持	✗	✗	✓	✗	✓
可抢占性	✗	✗	✗	✓	✓
饥荒风险	无	无	✓高	无	无

二、详细适用场景分析

1. FIFO (First In First Out) - 先进先出调度

特点：

- **算法原理**：按到达顺序执行，每个进程运行到完成
- **时间复杂度**：O(1) 插入/删除
- **上下文切换**：最少（仅在 I/O 等待时）

适用场景：

✓ 完全适合的场景：

1. **单道批处理系统**
   - 传统的大型机/超级计算机
   - 无需用户交互的离线处理
   - 示例：夜间数据处理、科学计算任务
2. **简单的嵌入式系统**
   - 固定任务流程
   - 任务优先级固定且不变
   - 示例：工业控制设备、传统 PLC
3. **后台批处理队列**
   - 文件处理系统
   - 图片批量转换
   - 日志处理系统

性能指标：

假设有 3 个任务，运行时间分别为 3, 6, 9 时间单位：

```
FIFO 执行顺序：
P1(3) → P2(6) → P3(9)
├─ P1: 完成时间=3,  等待时间=0,  周转时间=3
├─ P2: 完成时间=9,  等待时间=3,  周转时间=9
└─ P3: 完成时间=18, 等待时间=9,  周转时间=18

平均周转时间 = (3+9+18)/3 = 10
平均等待时间 = (0+3+9)/3 = 4
```

⚠️ 不适合的场景：

- ✗ 交互式系统（用户需要快速响应）
- ✗ 长短任务混合（长任务导致其他任务等待久）
- ✗ 需要优先级区分的系统
- ✗ 多用户共享系统

缺点分析：

```
Convoy Effect (车队效应)：
假设有 1 个 CPU 密集任务(100ms) + 4 个 I/O 密集任务(5ms)

FIFO:
[====CPU任务====][IO1][IO2][IO3][IO4]
                  ↑
         所有 IO 任务被阻塞，等待 CPU 任务完成
         总时间 = 120ms，资源利用率低
```

---

2. SJF (Shortest Job First) - 最短作业优先

特点：

- **算法原理**：优先运行最短的作业，最小化平均周转时间
- **最优性**：在非抢占情况下，具有最小平均周转时间
- **预测难度**：需要预知或估计任务长度

适用场景：

✓ 完全适合的场景：

1. **批处理系统（可知任务长度）**

   ```
   例：数据库维护任务
   - 日志压缩(1min)
   - 索引优化(5min)
   - 备份(30min)
   - 统计分析(2min)

   SJF 顺序：统计→日志→索引→备份
   最小化总等待时间
   ```

2. **网络服务中的短连接处理**
   - Web 服务器（短查询优先）
   - DNS 服务（短请求优先）
   - 负载均衡器
3. **分布式任务调度系统**
   - MapReduce 任务调度
   - Spark 任务分发
   - 优先处理短 Task

性能指标：

同样的 3 个任务（3, 6, 9）：

```
SJF 执行顺序：
P1(3) → P2(6) → P3(9)  （正好是最优排序）
```

```
├─ P1: 完成时间=3,  等待时间=0,  周转时间=3
├─ P2: 完成时间=9,  等待时间=3,  周转时间=9
└─ P3: 完成时间=18, 等待时间=9,  周转时间=18

平均周转时间 = (3+9+18)/3 = 10
平均等待时间 = (0+3+9)/3 = 4

如果顺序不同 (9,6,3):
P1(9) → P2(6) → P3(3)

├─ P1: 完成时间=9,  等待时间=0,  周转时间=9
├─ P2: 完成时间=15, 等待时间=9,  周转时间=15
└─ P3: 完成时间=18, 等待时间=15, 周转时间=18

平均周转时间 = (9+15+18)/3 = 14  ← 更差
平均等待时间 = (0+9+15)/3 = 8    ← 更差
```

实现策略：

```
// 在我们的实现中，使用 time_slice 作为任务长度估计
// 策略：
// 1. 第一次运行：time_slice = MAX_TIME_SLICE (表示未知)
// 2. 后续运行：根据历史运行时间调整（指数加权移动平均）
//
// 预测公式：τ_{n+1} = α·t_n + (1-α)·τ_n
// 其中：t_n = 第n次实际运行时间
//       τ_n = 第n次预测时间
//       α   = 平滑系数（通常 0.5）
```

⚠️ 不适合的场景：

- ✗ 交互式系统（短任务也要等待长任务）
- ✗ 无法预知任务长度的系统
- ✗ 长任务频繁到达（导致长任务饥荒）
- ✗ 需要响应时间保证的系统

问题示例：

```
Starvation of Long Jobs（长任务饥荒）：

时间轴：
0-1:   Job A(长, 100ms) 运行
1:     Job B(短, 5ms) 到达 → 立即被调度
6:     Job C(短, 5ms) 到达 → 立即被调度
11:    Job D(短, 5ms) 到达 → 立即被调度
...
如果短任务不断到达，长任务 A 可能永不被执行！
```

---

3. Priority (优先级) - 静态优先级调度

特点：

- **算法原理**：优先级高的进程优先执行，同级 FIFO
- **优先级来源**：用户指定（静态）
- **无抢占**：当前进程运行至完成

适用场景：

✓ 完全适合的场景：

1. **实时系统（非硬实时）**

```
系统分层：
P1(优先级5)：紧急故障处理
P2(优先级4)：安全监控
P3(优先级3)：常规业务
P4(优先级2)：后台维护
P5(优先级1)：低优先级辅助
```

2. **操作系统内核**
   - 系统任务（P=高）
   - 用户任务（P=低）
   - 中断处理（P=最高）
3. **多级应用**
   - 多媒体系统（播放>录制>编码）
   - 数据库（查询>更新>维护）
   - 日志系统（错误>警告>信息>调试）

实现示例：

```
// 优先级分配策略
enum priority_level {
    CRITICAL = 5,    // 紧急，系统级操作
    HIGH = 4,        // 重要，用户可见
    NORMAL = 3,      // 正常
    LOW = 2,         // 低，后台任务
    IDLE = 1,        // 最低，空闲时执行
};

// 队列结构：多个 FIFO 队列，按优先级排列
struct priority_rq {
    list_entry_t queues[PRIORITY_LEVELS];  // 5 个独立队列
    int proc_count[PRIORITY_LEVELS];       // 每个队列的进程数
};

// 调度逻辑：从最高优先级队列取进程
struct proc_struct *pick_next() {
    for (int p = CRITICAL; p >= IDLE; p--) {
        if (rq->queues[p] 不为空)
```

```
            return 从 rq->queues[p] 取第一个进程;
    }
    return NULL;
}
```

性能指标：

```
场景：3 个任务，运行时间 (3,6,9)，优先级 (3,2,1)

执行顺序：P1(高优先级) → P2(中) → P3(低)

├─ P1: 完成时间=3,  等待时间=0,  周转时间=3
├─ P2: 完成时间=9,  等待时间=3,  周转时间=9
└─ P3: 完成时间=18, 等待时间=9,  周转时间=18

平均周转时间 = 10
平均等待时间 = 4

优先级效果：高优先级任务得到优先服务
响应时间：取决于高优先级任务数量
```

⚠️ 严重问题：

饥荒 (Starvation) 问题：

```
P1(优先级高) 不断到达
├─ P1_1 运行
├─ P1_2 到达 → 等待
├─ P1_3 到达 → 等待
└─ P1_4 到达 → 等待
...
↓
P2(优先级低) 永远得不到运行机会!
```

解决方案（老化/Aging）：

```c
// 每次调度周期，低优先级进程的优先级加 1
void priority_aging() {
    for (int p = LOW; p < CRITICAL; p++) {
        for (每个在队列p中的进程) {
            proc->priority++;  // 优先级逐渐升高
            if (proc->priority >= CRITICAL)
                proc->priority = CRITICAL;
        }
    }
}

// 周期性调用（如每 100ms）
```

⚠️ 不适合的场景：

- ✗ 所有任务同等重要
- ✗ 需要公平性保证
- ✗ 无法分配合理优先级

---

4. RR (Round Robin) - 轮转调度

特点：

- **算法原理**：每个进程分配相等的时间片，轮流执行
- **时间片**：通常 10-100ms（我们实现中 = 5）
- **可抢占**：时间片耗尽强制切换
- **完全公平**：所有进程获得相等的 CPU 时间

适用场景：

✓ 最适合的场景：

1. **分时系统（Time-sharing）**
   - 经典 Unix/Linux 多用户系统
   - 多任务操作系统（Windows, macOS）

   ```
   场景：多个用户在同一系统工作
   - 用户 A：文本编辑
   - 用户 B：编译代码
   - 用户 C：网络浏览

   RR 保证每个用户都能快速响应
   ```

2. **现代服务器系统**
   - Web 服务器（Apache, Nginx）
   - 应用服务器（Tomcat, Node.js）
   - 数据库服务器

   ```
   优点：
   - 短连接快速响应
   - 长连接公平共享
   - 没有任何客户端饥荒
   ```

3. **交互式应用**
   - 桌面应用
   - 游戏引擎
   - IDE 编辑器

性能指标：

```
同样 3 个任务 (3,6,9)，时间片=5：

时间轴：
0-3:   P1 完成 (3 < 5)
3-8:   P2 运行 5ms (剩余 1ms)
8-13:  P3 运行 5ms (剩余 4ms)
13-14: P2 完成 (剩余 1ms)
14-19: P3 运行 5ms (剩余 -1ms，实际完成)

实际运行：
├─ P1: 完成时间=3,  等待时间=0,  周转时间=3
├─ P2: 完成时间=14, 等待时间=3,  周转时间=14
└─ P3: 完成时间=19, 等待时间=9,  周转时间=19

平均周转时间 = (3+14+19)/3 ≈ 12
平均等待时间 ≈ 4
响应时间 = 最大时间片 = 5  ← 重要！
```

时间片大小影响：

```
时间片太小（如 1ms）：
├─ ✓ 响应时间好（≤1ms）
├─ ✓ 交互性好
└─ ✗ 上下文切换频繁，开销大

时间片太大（如 1000ms）：
├─ ✓ 上下文切换少，效率高
├─ ✓ 缓存利用率好
└─ ✗ 响应时间差（≤1000ms），交互性差

经验值：
- 交互系统：10-100ms
- 批处理系统：100-500ms
- 科学计算：500-1000ms
```

优势总结：

```
RR 的关键特点：

1. 公平性完美
   所有进程获得 (N-1)/N × 总时间
   其中 N = 进程数

2. 响应时间可预测
   响应时间 ≤ (N-1) × 时间片

3. 无饥荒
   每个进程都能定期获得 CPU

4. 适应性强
   对进程长度没有假设
```

```
例：5 个进程，时间片 5ms
|        |        |        |        |        |
P1(5ms) P2(5ms) P3(5ms) P4(5ms) P5(5ms)
└─┬──────────────────────────────────┘
  一个完整轮转（25ms）
```

⚠️ 不适合的场景：

- ✗ 没有进程到达（浪费时间片）
- ✗ 对优先级敏感的系统
- ✗ 实时系统（需要优先级）

---

5. Stride - 步幅调度（优先级加权 RR）

特点：

- **算法原理**：基于比例公平的优先级调度
- **核心思想**：CPU 时间分配与优先级成正比
- **实现方式**：使用步幅值(stride)和通过(pass)
- **特性**：既有 RR 的公平性，又有 Priority 的优先级支持

算法原理详解：

```
基本概念：
- stride = BIG_STRIDE / priority
- 每次调度：选择 pass 最小的进程
- 执行后：该进程的 pass += stride

直观理解：
优先级为 5 的进程，stride = 0x7FFFFFFF / 5
优先级为 1 的进程，stride = 0x7FFFFFFF / 1

更新后：
- 优先级 5：pass 增加快（相对较小的步幅）
- 优先级 1：pass 增加慢（相对较大的步幅）

结果：优先级 5 的进程 pass 值始终较小，更频繁被调度！
```

数学模型：

```
定义：
- p_i: 进程 i 的优先级
- t_i: 进程 i 得到的总 CPU 时间
- T: 总的 CPU 时间

理论证明：
lim(T→∞) t_i / t_j = p_i / p_j

含义：CPU 时间的分配与优先级成比例！
```

适用场景：

✓ 最适合的场景：

1. **云计算/虚拟化环境**

```
VM 优先级分配：
- VM1 (数据库): 优先级 5 → 50% CPU
- VM2 (应用):   优先级 3 → 30% CPU
- VM3 (备份):   优先级 2 → 20% CPU

Stride 调度自动实现这个比例！
```

2. **容器编排系统（Kubernetes）**

```
Pod 资源限制：
requests: cpu=2
limits: cpu=4

Stride 调度确保获得约定的 CPU 份额
```

3. **QoS 保证系统**

```
多租户数据库：
- VIP 客户: 优先级 5 → 60% 吞吐量
- 普通客户: 优先级 3 → 30% 吞吐量
- 试用用户: 优先级 1 → 10% 吞吐量
```

4. **多应用协作系统**

```
前端应用: 优先级 5
后台服务: 优先级 3
日志收集: 优先级 1

高优先级应用优先响应，不会完全饿死低优先级
```

性能示例：

```
3 个进程，优先级 (5, 3, 1)，BIG_STRIDE = 1024

初始状态：
P1: pass=0, stride=1024/5=204
P2: pass=0, stride=1024/3=341
P3: pass=0, stride=1024/1=1024

调度顺序（总共 9 个时间片）：

轮 1: pass最小 = P1(0),P2(0),P3(0) → 选P1, pass=204
轮 2: pass最小 = P2(0),P3(0),P1(204) → 选P2, pass=341
轮 3: pass最小 = P3(0),P1(204),P2(341) → 选P3, pass=1024
轮 4: pass最小 = P1(204),P2(341),P3(1024) → 选P1, pass=408
轮 5: pass最小 = P2(341),P1(408),P3(1024) → 选P2, pass=682
轮 6: pass最小 = P1(408),P2(682),P3(1024) → 选P1, pass=612
```

```
轮 7: pass最小 = P2(682),P1(612),P3(1024) → 选P1, pass=816
轮 8: pass最小 = P2(682),P1(816),P3(1024) → 选P2, pass=1023
轮 9: pass最小 = P1(816),P2(1023),P3(1024) → 选P1, pass=1020

最终执行次数:
P1: 5 次 (55.6%)  → 优先级 5/9 ≈ 55.6% ✓
P2: 3 次 (33.3%)  → 优先级 3/9 ≈ 33.3% ✓
P3: 1 次 (11.1%)  → 优先级 1/9 ≈ 11.1% ✓

精确按比例分配!
```

实现关键:

```c
// Stride Scheduler 实现
#define BIG_STRIDE 0x7FFFFFFF  // 2^31 - 1

struct proc_struct {
    uint32_t lab6_stride;      // 当前的 pass 值
    uint32_t lab6_priority;    // 优先级 1-5
};

void stride_enqueue(struct run_queue *rq, struct proc_struct *proc) {
    // 使用最小堆插入，保持 pass 值最小的在堆顶
    skew_heap_insert(&rq->lab6_run_pool, &proc->lab6_run_pool,
                     proc_stride_comp_f);
}

struct proc_struct *stride_pick_next(struct run_queue *rq) {
    // 取堆顶（pass 最小）
    return le2proc(rq->lab6_run_pool, lab6_run_pool);
}

void stride_proc_tick(struct run_queue *rq, struct proc_struct *proc) {
    // 时间片耗尽后更新 stride
    if (proc->time_slice == 0) {
        proc->time_slice = rq->max_time_slice;
        // 更新 stride: pass += BIG_STRIDE / priority
        proc->lab6_stride += BIG_STRIDE / proc->lab6_priority;
    }
}
```

⚠️ 注意事项：

```
1. 新进程处理
   - 新进程 pass = 0（可能获得过多 CPU）
   - 解决：pass = 当前最小pass值

2. 溢出处理
   - stride 累计可能溢出
   - 解决：使用足够大的 BIG_STRIDE（2^31-1）
   - 或使用模运算处理环形缓冲区

3. 动态优先级
   - Stride 假设优先级固定
   - 若要支持动态优先级，需要调整 stride 计算
```

✓ 优势总结：

```
Stride 的完美之处：

1. 比例公平
   每个进程获得 CPU 时间 ∝ 优先级

2. 无饥荒
   即使优先级低，仍会定期获得 CPU

3. 优先级支持
   支持任意优先级值

4. 可预测性
   响应时间与优先级相关，可分析

5. 平衡
   既有优先级的灵活性，又有公平性

数据中心应用：
Amazon, Google 都使用类似算法进行虚拟机和容器调度！
```

三、选择指南矩阵

按系统类型选择

系统类型	首选	次选	说明
实时系统	Priority	Stride	需要确定响应时间上界
分时系统	RR	Stride	需要交互响应
批处理系统	SJF/FIFO	Priority	吞吐量优先
云计算	Stride	RR	QoS 保证
嵌入式系统	FIFO/Priority	RR	资源受限

系统类型	首选	次选	说明
多用户系统	RR	Stride	公平性重要

按性能指标选择

目标	最佳选择	次选	理由
最小平均周转时间	SJF	FIFO	数学最优
最小平均等待时间	SJF	RR	SJF 理论最优
最小响应时间	RR	Stride	时间片保证
最小方差（公平性）	RR	Stride	完全轮转
优先级保证	Stride	Priority	Stride 无饥荒
最小上下文切换	FIFO	Priority	无抢占

四、实际案例分析

案例 1：Web 服务器（Apache）

场景：处理多个 HTTP 请求

- 请求 A：静态文件（快速，5ms）
- 请求 B：数据库查询（中等，50ms）
- 请求 C：报表生成（长，500ms）

各算法表现：

```
FIFO 处理：
时间0-5ms:   请求A 完成 ✓
时间5-55ms:  请求B 运行，用户等待 50ms
时间55-555ms: 请求C 运行，用户等待 500ms
总响应时间：A=5, B=50, C=500, 平均=185ms

SJF 处理：
时间0-5ms:   请求A 完成 ✓ (立即响应)
时间5-55ms:  请求B 完成 ✓ (等待5ms)
时间55-555ms: 请求C 完成 (等待50ms)
总响应时间：A=5, B=55, C=555, 平均=205ms
└ 注：SJF 需要预知请求长度，难以应用于 web

RR 处理（时间片 10ms）：
时间0-5:     请求A 完成 ✓ (响应5ms)
时间5-15:    请求B 运行 (10ms)
时间15-25:   请求C 运行 (10ms)
时间25-35:   请求B 运行 (10ms)
时间35-45:   请求C 运行 (10ms)
时间45-50:   请求B 完成 ✓ (响应45ms)
时间50-60:   请求C 运行 (10ms)
```

```
...
总响应时间：A=5，B≈45，C≈500，平均≈150ms ← 更均衡

Stride 处理（优先级 A=3，B=2，C=1）：
与 RR 类似，但 A 更频繁得到 CPU
总响应时间：A=3，B≈40，C≈500，平均≈148ms
```

结论：Web 服务器应选 **RR 或 Stride**

案例 2：实时控制系统

场景：工业机器人控制

```
紧急停止(P=5):    1ms 任务（最高优先）
电机控制(P=4):   10ms 任务
传感器采样(P=3): 20ms 任务
日志记录(P=2):   50ms 任务
诊断(P=1):      200ms 任务
```

各算法表现：

```
FIFO：✗ 不好
  诊断任务长（200ms），可能延迟紧急停止

Priority：✓ 良好但有风险
  紧急停止立即执行
  但诊断任务可能永不执行（饥荒）

RR：✗ 不合适
  时间片导致实时任务延迟不可控
  例：紧急停止可能延迟 200ms（长任务不断重新排队）

Stride：✓✓ 最佳
  - 紧急停止获得最小 pass，优先执行
  - 低优先级任务（诊断）仍能定期运行，不会饥荒
  - 各任务响应时间可预测且有界

  实现：
  if (紧急信号) {
       调整诊断任务优先级 = 0（不运行）
       保证核心任务运行
  }
```

结论：实时系统应选 **Stride 或 Priority（带老化）**

案例 3：批处理数据中心

场景：夜间数据处理，5 个任务：

```
任务A: 数据导入    (已知 10 秒)
任务B: 数据清洗    (已知 2 秒)
任务C: 数据分析    (已知 30 秒)
任务D: 生成报表    (已知 15 秒)
任务E: 数据导出    (已知 5 秒)
```

各算法表现:

```
FIFO（按到达顺序）：
A(10) → B(2) → C(30) → D(15) → E(5) = 62秒
├─ A: 完成10,  等待0,  周转10
├─ B: 完成12,  等待10, 周转12
├─ C: 完成42,  等待12, 周转42
├─ D: 完成57,  等待42, 周转57
└─ E: 完成62,  等待57, 周转62
平均周转时间 = 36.6秒

SJF（按长度排序）：
B(2) → E(5) → A(10) → D(15) → C(30) = 62秒（总时间相同）
├─ B: 完成2,   等待0,  周转2
├─ E: 完成7,   等待2,  周转7
├─ A: 完成17,  等待7,  周转17
├─ D: 完成32,  等待17, 周转32
└─ C: 完成62,  等待32, 周转62
平均周转时间 = 24秒 ← 最优!

Priority（优先级）：
C(30, P5) → D(15, P4) → A(10, P3) → E(5, P2) → B(2, P1)
= 62秒（总时间相同）
平均周转时间 = 28秒
```

结论：批处理应选 **SJF**（如果能预知长度）或 **FIFO**（简单可靠）

五、高级话题

1. 适应性调度

理想调度器应该:

- 学习任务特征（CPU密集 vs I/O密集）
- 动态调整策略
- 自适应时间片

实现思路:

```
class AdaptiveScheduler:
    def measure_job_characteristics(job):
        cpu_time, io_waits = measure_during_execution()
        io_ratio = io_waits / (cpu_time + io_waits)
        return io_ratio

    def choose_algorithm(jobs):
```

```
        if all_cpu_bound(jobs):
            return SJF  # 最小化周转时间
        elif all_io_bound(jobs):
            return RR   # 保持响应性
        else:
            return Stride  # 平衡两者

    def adapt_time_slice(job):
        if job.is_io_bound():
            return smaller_time_slice  # 快速响应
        else:
            return larger_time_slice   # 减少切换
```

2. 多级队列调度（MLFQ）

结合多个算法:

```
┌──────────────────────────────────────┐
| 级别1（优先级最高）: RR队列     |  新进程进入
| 级别2（中优先级）:  RR队列     |
| 级别3（低优先级）:  SJF队列    |  需要长时间的进程
└──────────────────────────────────────┘

特点:
- 新进程在高级别快速响应
- 如果超过时间限制，降级到低级别
- 周期性老化，防止饥荒
- 综合了 RR、Priority、SJF 的优点
```

3. 公平性度量

衡量调度器的公平性:

```
Jain's Fairness Index:

F = (Σ x_i)² / (n × Σ x_i²)

其中:
- x_i = 进程 i 获得的 CPU 时间
- n = 进程数

范围: 1/n ≤ F ≤ 1
- F = 1: 完全公平（所有进程相等）
- F = 1/n: 完全不公平（一个进程得所有）

例（3个进程）:
RR:     [100, 100, 100] → F = 1.0 ✓ 完全公平
Priority: [150, 50, 0]  → F = 0.4   ✗ 不公平
Stride: [150, 100, 50]  → F = 0.81  ✓ 相对公平
```

六、总结建议

快速决策树

```
你的系统是什么？
├─ 实时系统
|  └─ Stride（推荐） 或 Priority（简单）
|
├─ 分时/交互式系统
|  └─ RR（标准选择）
|
├─ 批处理系统
|  ├─ 如果知道任务长度 → SJF
|  └─ 否则 → FIFO
|
├─ 云/虚拟化系统
|  └─ Stride（资源隔离）
|
└─ 嵌入式系统
   ├─ 固定任务流 → FIFO
   └─ 多优先级 → Priority
```

实现难度 vs 收益

```
实现难度→

FIFO
  |
  |    ✓ 简单实现
Priority  |    ✓ 支持优先级
  |    X 可能饥荒
  |
SJF    |    ✓ 最优周转时间
  |    X 需要预知长度
  |
RR     |    ✓ 公平无饥荒
  |    ✓ 交互性好
  |
Stride |    ✓ 比例公平
       |    ✓ 无饥荒
       |    ✓ 优先级支持
       |    X 实现复杂

                ↓ 收益增大，响应性和公平性提升
```

性能对比总结

```
                  FIFO  SJF  Priority  RR   Stride
平均周转时间       ✗✗   ✓✓    ✗✓      ✗✓   ✗✓
响应时间          ✗✗   ✗✗    ✗✓      ✓✓   ✓✓
公平性            ✗✗   ✗✗    ✗✗      ✓✓   ✓✓
优先级支持         ✗    ✗     ✓✓      ✗    ✓✓
无饥荒            ✓    ✓     ✗✗      ✓    ✓
实现简度           ✓✓   ✓     ✓       ✓    ✗

✓✓ = 优秀  ✓ = 良好  ✗ = 一般  ✗✗ = 差
```