OS Lab6

练习0

在前几个实验的基础上，我们对其中一些代码做了修改，以完成Lab6的要求：

`proc.c`

在 `alloc_proc()` 函数中，我们对Lab6新增的变量进行了初始化：

代码块

```
// LAB6:YOUR CODE (update LAB5 steps)
/*
* below fields(add in LAB6) in proc_struct need to be initialized
*       struct run_queue *rq;                       // run queue contains Process
*       list_entry_t run_link;                      // the entry linked in run queue
*       int time_slice;                             // time slice for occupying the CPU
*       skew_heap_entry_t lab6_run_pool;            // entry in the run pool (lab6 stride)
*       uint32_t lab6_stride;                       // stride value (lab6 stride)
*       uint32_t lab6_priority;                     // priority value (lab6 stride)
*/
proc->rq = NULL;                    // 初始化运行队列指针为NULL
list_init(&(proc->run_link));       // 初始化运行队列链表节点
proc->time_slice = 0;               // 初始化时间片为0，后续由调度器设置
proc->lab6_run_pool.left = NULL;    // 初始化斜堆的左子树指针
proc->lab6_run_pool.right = NULL;   // 初始化斜堆的右子树指针
proc->lab6_run_pool.parent = NULL;  // 初始化斜堆的父节点指针
proc->lab6_stride = 0;              // 初始化stride值为0
proc->lab6_priority = 1;            // 初始化优先级为1（最小的优先级值，确保所有进程都有默认优先级）
```

`trap.c`

在 `interrupt_handler()` 函数中，做了以下修改：

```
case IRQ_S_TIMER:
    // "All bits besides SSIP and USIP in the sip register are
    // read-only." -- privileged spec1.9.1, 4.1.4, p59
    // In fact, Call sbi_set_timer will clear STIP, or you can clear it
    // directly.
    // clear_csr(sip, SIP_STIP);

    /* LAB3 :填写你在lab3中实现的代码 */
    /*(1)设置下次时钟中断- clock_set_next_event()
    *(2)计数器（ticks）加一
    *(3)当计数器加到100的时候，我们会输出一个`100ticks`表示我们触发了100次时钟中断，同时打印次数（num）加一
    * (4)判断打印次数，当打印次数为10时，调用<sbi.h>中的关机函数关机
    */

    // lab6: YOUR CODE  (update LAB3 steps)
    //  在时钟中断时调用调度器的 sched_class_proc_tick 函数

    /* (1) 设置下次时钟中断 */
    clock_set_next_event();

    /* (2) 计数器（ticks）加一 */
    ticks++;

    /* (3) 当计数器加到100的时候，输出一个`100ticks`表示触发了100次时钟中断，同时打印次数加一 */
    static int print_count = 0;

    if (ticks % TICK_NUM == 0) {
        print_ticks(); /* 打印 "100 ticks" */
        print_count++;

        if (print_count >= 10) {
            /* 调用 OpenSBI 的关机接口 */
            sbi_shutdown();
            /* 保险：若 shut_down 返回，防止继续运行 */
            while (1)
                ;
        }
    }
    /* LAB6: 调用调度器的tick处理函数 */
    /* 注意：current是当前正在运行的进程 */
    if (current != NULL) {
        sched_class_proc_tick(current);
    }
    break;
```

练习1：理解调度器框架的实现

调度器框架的实现

调度类结构体 sched_class 的分析

代码块

```
// sched.h中
struct sched_class
{
    // the name of sched_class
    const char *name;
    // Init the run queue
    void (*init)(struct run_queue *rq);
    // put the proc into runqueue, and this function must be called with rq_lock
    void (*enqueue)(struct run_queue *rq, struct proc_struct *proc);
    // get the proc out runqueue, and this function must be called with rq_lock
    void (*dequeue)(struct run_queue *rq, struct proc_struct *proc);
    // choose the next runnable task
    struct proc_struct *(*pick_next)(struct run_queue *rq);
    // dealer of the time-tick
    void (*proc_tick)(struct run_queue *rq, struct proc_struct *proc);
    /* for SMP support in the future
     *  load_balance
     *     void (*load_balance)(struct rq* rq);
     *  get some proc from this rq, used in load_balance,
     *  return value is the num of gotten proc
     *  int (*get_proc)(struct rq* rq, struct proc* procs_moved[]);
     */
};
```

各函数指针的作用与调度实时机

1. `name`

- 作用：调度策略的名称字符串（如 "RR_scheduler"、"stride_scheduler"）
- 意义：调试与日志输出时标识当前使用的调度算法。

2. `init(rq)`

- 作用：初始化运行队列 `rq` 的数据结构。
- 调用时机：
  - 系统启动时（`sched_init()`）

- 用于设置 `run_list`、`proc_num`、`lab6_run_pool` 等成员初始状态。

3. `enqueue(rq, proc)`

- 作用：将进程 `proc` 插入运行队列 `rq`。
- 调用时机：
  - 进程变为可运行状态时（`wakeup_proc()`）
  - 时间片用完但进程仍可运行时（在 `schedule()` 中重新入队）
  - 新进程创建后（`do_fork()` → `wakeup_proc()`）

4. `dequeue(rq, proc)`

- 作用：从运行队列 `rq` 中移除进程 `proc`。
- 调用时机：
  - 进程被调度执行前（`schedule()` 中选中该进程）
  - 进程退出、阻塞或变为不可运行时

5. `pick_next(rq)`

- 作用：从运行队列 `rq` 中选择下一个要运行的进程。
- 调用时机：
  - 调度器寻找可运行进程时（`schedule()`）
  - 返回 `NULL` 时表示队列为空，调度 idle 进程

6. `proc_tick(rq, proc)`

- 作用：处理时钟中断对当前进程的影响（如减少时间片）。
- 调用时机：
  - 每次时钟中断时（`trap.c` 中 `interrupt_handler(IRQ_S_TIMER)` 调用 `sched_class_proc_tick()`）
  - 可能触发 `need_resched` 标志

为何使用函数指针而非直接实现函数

实现调度算法与调度框架的解耦。通过将调度操作抽象为统一的函数指针接口，系统可以在运行时动态切换不同的调度策略（如RR、Stride等），而无需修改核心调度逻辑，增强了代码的可扩展性和可维护性，新增调度算法只需实现新的 `sched_class` 实例即可。

运行队列结构体 run_queue 的分析

代码块

```c
// sched.h 中
// lab5 的代码里找不到这个函数……
// lab6 的 run_queue 结构体实现
struct run_queue
{
    list_entry_t run_list;
    unsigned int proc_num;
    int max_time_slice;
    // For LAB6 ONLY
    skew_heap_entry_t *lab6_run_pool;  // 也许这就是 lab6 比 lab5 多出来的部分吧
};
```

lab 5 与 lab6 的差异

lab5 的代码中未找到与 `run_queue` 结构体相关的内容，但通过 lab6 的代码我们可以猜测，应该是在lab5代码的基础上新增了 `skew_heap_entry_t *lab6_run_pool;` 。

这是因为在 lab6 中引入了 Stride 调度算法，而 Stride 算法需要使用优先队列来高效选择最小 stride 值的进程。链表在寻找最小 stride 进程时需要遍历所有进程，效率为O(n)，而斜堆插入和删除最小元素的时间复杂度为 O(log n)，能更快地选出下一个要运行的进程。 `lab6_run_pool` 就是用来存储这个斜堆的根节点指针，专门为Stride调度算法服务。

为何 lab6 的 run_queue 需要支持链表和斜堆两种数据结构

因为 lab6 要同时支持两种不同的调度算法：传统的 RR 算法和新的 Stride 算法。RR算法只需要简单的FIFO队列，用链表实现就足够了；而Stride算法需要**优先队列**，用斜堆实现更高效。同时包含两种数据结构，可以让调度框架根据当前使用的调度算法来选择——用RR时就只用链表，用Stride时就只用斜堆。

调度器框架函数分析

代码块

```c
// 这些函数在 sched.c 中实现
void sched_init(void)
{
    list_init(&timer_list);

    sched_class = &default_sched_class;

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

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

```
14
15  void wakeup_proc(struct proc_struct *proc)
16  {
17      assert(proc->state != PROC_ZOMBIE);
18      bool intr_flag;
19      local_intr_save(intr_flag);
20      {
21          if (proc->state != PROC_RUNNABLE)
22          {
23              proc->state = PROC_RUNNABLE;
24              proc->wait_state = 0;
25              if (proc != current)
26              {
27                  sched_class_enqueue(proc);
28              }
29          }
30          else
31          {
32              warn("wakeup runnable process.\n");
33          }
34      }
35      local_intr_restore(intr_flag);
36  }
37
38  void schedule(void)
39  {
40      bool intr_flag;
41      struct proc_struct *next;
42      local_intr_save(intr_flag);
43      {
44          current->need_resched = 0;
45          if (current->state == PROC_RUNNABLE)
46          {
47              sched_class_enqueue(current);
48          }
49          if ((next = sched_class_pick_next()) != NULL)
50          {
51              sched_class_dequeue(next);
52          }
53          if (next == NULL)
54          {
55              next = idleproc;
56          }
57          next->runs++;
58          if (next != current)
59          {
60              proc_run(next);
```

```
61          }
62      }
63      local_intr_restore(intr_flag);
64  }
```

很可惜，在 lab5 的代码中仍然没有找到这三个函数。

解耦方式分析

这些函数通过与 `sched_class` 函数指针表交互来实现与具体调度算法的解耦。

sched_init() 如何解耦：

- 它只设置 `sched_class = &default_sched_class`，然后调用 `sched_class->init(rq)`
- 解耦关键：不知道也不关心 `default_sched_class` 是 RR 还是 Stride
- 无论调度算法怎么变，初始化流程都是 `sched_class->init(rq)` 这一个调用

wakeup_proc() 如何解耦：

- 它通过 `sched_class_enqueue(proc)` 将进程加入就绪队列
- `sched_class_enqueue(proc)` 内部调用 `sched_class->enqueue(rq, proc)`
- 解耦关键：唤醒进程时只说“把这个进程放回就绪队列”，具体怎么放（链表尾部还是斜堆插入）由调度算法决定

schedule() 如何解耦：

- 它只通过三个标准步骤与调度算法交互：
  a. `sched_class_enqueue(current)` - 当前进程重新入队
  b. `sched_class_pick_next()` - 询问“下一个该谁运行”
  c. `sched_class_dequeue(next)` - 把选中的进程从队列移除
- 解耦关键：调度框架只说“给我下一个进程”，不关心如何选择

调度器框架的使用流程

调度类的初始化流程

从内核启动到调度器初始化完成的完整流程

1. **内核入口启动：**硬件启动后，执行 kern/init/entry.S 中的汇编代码，设置初始栈指针，跳转到 C 语言入口 `kern_init()`。
2. **基础子系统初始化：**在 `kern_init()` 中按顺序初始化：

- 内存管理 `pmm_init()`：建立物理内存管理。
- 中断系统 `idt_init()`：设置中断向量表和异常处理。
- 控制台 `cons_init()`：初始化串口输出。

3. **进程管理初始化：**调用 `proc_init()`：
   - 创建 idleproc（空闲进程）：pid=0，内核第一个进程。
   - 创建 initproc（初始化进程）：pid=1，负责后续用户进程。
   - 设置 current = idleproc：当前运行的是空闲进程。
4. **调度系统初始化：**调用 `sched_init()`，这是调度器初始化的核心。

代码块

```
void sched_init(void) {
    list_init(&timer_list);                    // 1. 初始化定时器链表
    sched_class = &default_sched_class;        // 2. 选择调度算法（绑定RR）
    rq = &__rq;                                // 3. 获取运行队列结构体
    rq->max_time_slice = MAX_TIME_SLICE;       // 4. 设置时间片长度
    sched_class->init(rq);                     // 5. 调用具体算法的初始化
    cprintf("sched class: %s\n", sched_class->name); // 6. 打印调度器信息
}
```

5. **具体调度算法初始化：**执行 `default_sched_class.init(rq)` → `RR_init(rq)`：

代码块

```
static void RR_init(struct run_queue *rq) {
    list_init(&(rq->run_list));  // 初始化RR使用的链表
    rq->proc_num = 0;            // 进程计数器清零
    // lab6_run_pool保持NULL（RR用不到斜堆）
}
```

6. **初始化完成，进入调度循环：**内核继续执行其他初始化（文件系统等），最终进入 `cpu_idle()` 循环：

代码块

```
void cpu_idle(void) {
    while (1) {
        if (current->need_resched) {  // 检查是否需要调度
            schedule();               // 触发第一次调度
        }
    }
}
```

总结一下时间线：

代码块

```
硬件启动
  ↓
entry.S（汇编入口）
  ↓
kern_init()（C语言入口）
  ├── pmm_init()   内存管理
  ├── idt_init()   中断系统
  ├── cons_init()  控制台
  ├── proc_init()  创建idleproc和initproc ← 有了"进程"的概念
  └── sched_init() 调度器初始化完成 ← 可以开始调度了！
       ↓
cpu_idle() 循环等待调度
```

default_sched_class 如何与调度器框架关联

先来看一下这个结构体：

代码块

```
// default_sched.c 中
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,
};
```

关联的关键步骤：

1. **指针赋值**：`sched_class = &default_sched_class`（这句话在`sched_init()`里）
   - 将全局调度器指针指向 RR 调度算法的函数表
2. **函数表挂接**：将RR调度算法函数集合的地址赋给全局调度器指针，使调度框架中所有通过该指针发起的函数调用都被定向到RR算法的函数上，完成调度接口与RR算法的绑定。
   - `sched_class->init(rq)` → `RR_init(rq)`
   - `sched_class->enqueue(rq, proc)` → `RR_enqueue(rq, proc)`

总结一下关联过程：

代码块

```
调度框架 (sched.c)
    |
    | 通过函数指针调用
    ↓
default_sched_class (RR算法)
    |
    | 具体实现
    ↓
RR_init() / RR_enqueue() / RR_dequeue() ...
    |
    | 操作具体数据结构
    ↓
运行队列的链表 (run_list)
```

进程调度流程

完整进程调度流图

代码块

```
┌──────────────────────────────┐
│   时钟中断发生               │ 1. 硬件时钟触发中断
└──────────────┬───────────────┘
               ↓
┌──────────────────────────────┐
│ interrupt_handler()  │ 2. 进入中断处理函数
│ (trap.c)             │    - 判断中断类型为IRQ_S_TIMER
└──────────────┬───────────────┘
               ↓
┌──────────────────────────────┐
│ sched_class_proc_tick() │ 3. 调用调度器的tick处理
│ (sched.c)            │    - 传入当前进程current
└──────────────┬───────────────┘
               ↓
┌──────────────────────────────┐
│ RR_proc_tick()       │ 4. 具体算法的tick处理
│ (default_sched.c)    │    - 减少当前进程时间片
│                      │    - 检查时间片是否用完
└──────────────┬───────────────┘
               ├──────────────────────┐
               ↓                      │
时间片>0? │                  │ 时间片=0?
   否     │                  是
               ↓                      │
```


继续执行
当前进程
设置need_resched
= 1
5. 关键标志位设置
中断返回前检查
need_resched
6. 在trap()返回前
或cpu_idle()循环中
need_resched=1?
是
schedule()
(sched.c)
7. 调用调度函数
schedule()内部执行流程
1. 关中断，保存状态
2. if (current->state ==
PROC_RUNNABLE)
sched_class_enqueue()
→ RR_enqueue()
8. 当前进程重新入队
3. next = sched_class_
pick_next()
→ RR_pick_next()
9. 选择下一个进程
4. if (next != NULL)
sched_class_
dequeue(next)
→ RR_dequeue()
10. 从队列移除
5. if (next == NULL)
next = idleproc
6. next->runs++
7. if (next != current)
proc_run(next)
11. 切换上下文
8. 开中断，恢复状态

`need_resched` 标志位在调度过程中的作用

`need_resched` 标志位是进程调度的核心触发信号，它作为一个“**延迟调度请求标记**”，在进程时间片用尽时由时钟中断处理函数置为1，随后系统会检查该标志位，若为1则调用 `schedule()` 执行实际调度，从而实现了“请求调度”与“执行调度”的解耦，保证了调度的及时性，避免了在中断上下文中直接进行复杂的进程切换。

调度算法的切换机制

添加新的调度算法需要修改哪些代码

1. 实现算法本身的代码文件
2. 算法接口声明（在 `default_sched.h` 中添加声明）
3. 算法结构体的定义
4. 实现切换调度算法

为何当前的设计便于切换调度算法

当前设计通过 “策略模式” 实现了**调度框架与算法的完全分离**，具体体现在：

1. **统一的接口设计**：所有调度算法都必须实现相同的 `sched_class` 函数表（包含 `init`、`enqueue`、`dequeue`、`pick_next`、`proc_tick` 五个标准函数），调度框架只通过这个统一接口调用算法功能，不关心具体实现细节。
2. **运行时动态绑定**：只需在 `sched_init()` 中修改一行代码——将 `sched_class` 指针指向不同的算法结构体，就能立即切换整个系统的调度策略，无需修改任何框架逻辑。

练习2：实现 Round Robin 调度算法

lab5 lab6同名函数比较分析

代码块

```
1   // lab5 中的 schedule()函数
2   void schedule(void)
3   {
4       bool intr_flag;
5       list_entry_t *le, *last;
6       struct proc_struct *next = NULL;
7       local_intr_save(intr_flag);
8       {
9           current->need_resched = 0;
10          last = (current == idleproc) ? &proc_list : &(current->list_link);
11          le = last;
12          do
13          {
```

```
            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);
}

// lab6 中的 schedule()函数
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);
        }
        if ((next = sched_class_pick_next()) != NULL)
        {
            sched_class_dequeue(next);
        }
        if (next == NULL)
        {
            next = idleproc;
        }
        next->runs++;
        if (next != current)
        {
            proc_run(next);
        }
```

```
61        }
62        local_intr_restore(intr_flag);
63    }
```

核心区别对比

方面	Lab5 schedule()	Lab6 schedule()
进程选择方式	遍历所有进程的全局链表	调用调度算法的 pick_next()
就绪队列管理	无独立就绪队列，扫描全部进程	有专门的运行队列，算法管理入队/出队
调度逻辑	硬编码的简单轮询	通过函数指针调用算法实现
代码复杂度	包含循环和状态判断	简洁，职责单一
数据结构耦合	直接操作 proc_list	通过调度类抽象操作

改动原因

1. **以支持多种调度算法**（核心原因）
   - Lab5：`schedule()` 函数硬编码了"遍历进程链表，选第一个就绪进程"的逻辑
   - Lab6：要支持RR、Stride等多种调度算法，每种算法选择进程的逻辑不同
2. **提高调度效率**
   - Lab5：每次调度都要遍历所有进程，复杂度O(n)
   - Lab6：调度算法维护就绪队列，直接取队首元素，复杂度O(1)或O(log n)
3. **职责分离，符合单一职责原则**
   - Lab5：`schedule()` 函数承担了太多职责——查找就绪进程、管理进程状态、执行进程切换
   - Lab6：`schedule()` 只做调度框架的工作——调用算法接口、执行进程切换
4. **以支持时间片调度**
   - Lab5：没有时间片概念，无法实现RR轮转
   - Lab6：需要时间片管理，当前进程重新入队

不改动的后果

1. **无法实现 Stride 调度算法**。
2. **调度逻辑混乱，难以维护**：不同算法的代码耦合在一起，一个算法的修改可能会影响其他算法。
3. **性能问题严重**：遍历所有进程，只能使用O(n)。

4. **无法通过函数指针动态切换算法。**

```
代码块
1   // Lab5框架下:
2   sched_class = &stride_sched_class;  // 赋值了也没用!
3   // 因为schedule()根本不调用sched_class->pick_next()
4   // 还是按自己的方式遍历proc_list
```

代码编写及分析

RR_init()

```
代码块
1   static void
2   RR_init(struct run_queue *rq)
3   {
4       // LAB6: YOUR CODE
5       list_init(&(rq->run_list));         // 初始化运行队列链表
6       rq->proc_num = 0;                   // 进程数量初始为0
7   }
8   /* list_init() 创建一个空的双向循环链表
9         将链表头的 prev 和 next 都指向自身
10        形成 head → head 的空链表结构
11      proc_num = 0 记录当前就绪进程数为0 */
```

具体思路和方法

- **目的**：初始化运行队列，为RR调度做好准备
- **方法**：使用双向循环链表的初始化方法

为何选择特定的链表操作方法——list_init()

- ucore 的双向循环链表设计：`list_head` 作为哨兵节点
- 空队列时：`head.next = head.prev = head`
- 这种设计简化了边界条件处理

边界情况处理

- 空队列：链表初始状态就是空的，无需特殊处理
- 重复初始化：即使多次调用也不会出错

RR_enqueue()

代码块

```
static void
RR_enqueue(struct run_queue *rq, struct proc_struct *proc)
{
    // LAB6: YOUR CODE
    // 将进程添加到运行队列尾部
    // 注意：list_add_before(&head, node) 将节点添加到链表头部（head之前）
    // 要实现FIFO队列，应该使用list_add_after或list_add_tail
    // 但根据ucore的list实现，使用list_add_before(&head, node)实际上是添加到尾部

    // 检查进程是否已经在队列中
    if (list_empty(&(proc->run_link))) { // 进程不在任何队列中时，其 run_link 的 prev 和 next 指向自身
        list_add_before(&(rq->run_list), &(proc->run_link));

        // 设置进程的时间片
        if (proc->time_slice == 0 || proc->time_slice > rq->max_time_slice) {
            proc->time_slice = rq->max_time_slice;
        }

        // 设置进程的运行队列指针
        proc->rq = rq;

        // 增加运行队列中的进程计数
        rq->proc_num++;
    }
}
```

具体思路和方法

- 目标：将进程插入运行队列尾部（FIFO）
- 方法：检查进程是否已在队列中，若不在则添加到尾部并设置时间片

为何选择特定的链表操作方法——list_add_before()

- `list_add_before(head, node)` 插入到 `head` 前
- 由于循环链表，`head` 前就是当前链表的最后一个位置
- 实现了先进先出：先入队的在头部，后入队的在尾部

边界情况处理

- 重复入队：通过 `list_empty()` 检查避免
- 时间片异常：强制标准化为 `max_time_slice`

- 空队列：`list_add_before(head, node)` 在空队列中能正确建立 `head ↔ node ↔ head`

RR_dequeue()

代码块

```
static void
RR_dequeue(struct run_queue *rq, struct proc_struct *proc)
{
    // LAB6: YOUR CODE
    // 从运行队列中移除进程
    if (!list_empty(&(proc->run_link))) {
        list_del_init(&(proc->run_link));

        // 减少运行队列中的进程计数
        rq->proc_num--;

        // 清除进程的运行队列指针
        proc->rq = NULL;
    }
}
```

具体思路和方法

- 目标：从运行队列中移除指定进程
- 方法：检查进程是否在队列中，若在则移除并更新计数

为何选择特定的链表操作方法——list_del_init()而非list_del()

- `list_del` 只断开链接，节点指针可能指向无效内存
- `list_del_init` 将节点重置为独立状态
- 便于后续 `list_empty()` 检查能正确工作

边界情况处理

- 删除不存在的进程：先检查避免误操作
- 队列计数一致性：确保 `proc_num` 与实际链表长度匹配
- 指针清理：`proc->rq = NULL` 防止后续误用

RR_pick_struct()

代码块

```
static struct proc_struct *
```

```
RR_pick_next(struct run_queue *rq)
{
    // LAB6: YOUR CODE
    list_entry_t *le = list_next(&(rq->run_list));

    // 如果运行队列为空，返回NULL
    if (le == &(rq->run_list)) {
        return NULL;
    }

    // 返回运行队列中的第一个进程（FIFO）
    struct proc_struct *proc = le2proc(le, run_link);

    // 确保进程状态是可运行的
    if (proc->state == PROC_RUNNABLE) {
        return proc;
    }

    return NULL;
}
```

具体思路和方法

- 目标：选择下一个要运行的进程（RR的队首进程）
- 方法：返回运行队列链表中的第一个进程

为何选择链表头部作为下一个进程

- RR核心思想：公平轮转，先入先服务
- 队列结构：
  - 队列: head ↔ P1 ↔ P2 ↔ P3 ↔ head
  - 调度顺序: P1 → P2 → P3 → P1 → ...
  - 每次pick_next都取head.next

边界情况处理

- 空队列：返回 `NULL`，调度器会选 `idleproc`
- 进程状态异常：检查并返回 `NULL`，让调度器处理
- 宏转换安全：确保 `le2proc` 参数正确（`run_link` 字段名）

RR_proc_tick()

代码块

```c
static void
RR_proc_tick(struct run_queue *rq, struct proc_struct *proc)
{
    // LAB6: YOUR CODE
    if (proc->time_slice > 0) {
        proc->time_slice--;
    }

    // 当时间片用完时，设置需要重新调度的标志
    if (proc->time_slice == 0) {
        proc->need_resched = 1;
    }
}
```

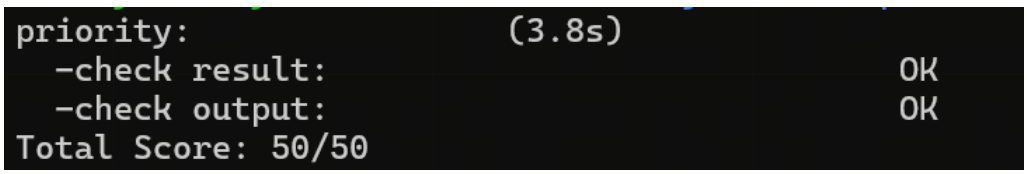
具体思路和方法

- 目标：处理时钟中断，更新进程时间片
- 方法：减少时间片，检查是否用完，用完后设置重调度标志

边界情况处理

- 时间片为0：直接设置 `need_resched`，避免负值
- idle进程：不调用此函数（在 `sched_class_proc_tick` 中过滤）
- 时间片重置：在 `RR_enqueue` 中重置，不在这里处理

输出结果展示



QEMU调度现象分析

代码块

```
check_alloc_page() succeeded!
check_pgdir() succeeded!
check_boot_pgdir() succeeded!
use SLOB allocator
kmalloc_init() succeeded!
check_vma_struct() succeeded!
check_vmm() succeeded.
sched class: RR_scheduler       // 确认使用的是RR调度器
++ setup timer interrupts
kernel_execve: pid = 2, name = "priority".
```

```
set priority to 6      // 进程优先级的设置被忽略，因为所有进程几乎同时结束
main: fork ok,now need to wait pids.
set priority to 1      // 如果使用Stride调度，高优先级进程应该明显更早结束
set priority to 2
set priority to 3
set priority to 4
set priority to 5
100 ticks              // 时间片轮转的证据
100 ticks
child pid 3, acc 508000, time 2010    // 进程按顺序轮流执行
child pid 4, acc 564000, time 2010
child pid 5, acc 580000, time 2010    // 每个进程的累计值(acc)相差不大
child pid 6, acc 532000, time 2020    // 表明每个进程获得的CPU时间大致相等
child pid 7, acc 544000, time 2020    // 体现了RR调度器的公平性
main: pid 0, acc 508000, time 2020
main: pid 4, acc 564000, time 2030
main: pid 5, acc 580000, time 2030
main: pid 6, acc 532000, time 2030
main: pid 7, acc 544000, time 2030
main: wait pids over
sched result: 1 1 1 1 1     // 所有结果都是1，表示每个进程的运行时间比例基本相等
all user-mode processes have quit.
init check memory pass.
```

RR调度算法分析

优点

1. **公平性优秀**：每个进程都能获得均等的CPU时间片，确保了公平性，不会出现低优先级进程被长期饿死的情况。
2. **响应时间确定**：由于采用固定时间片轮转，任何进程的最长等待时间都是可计算的，用户操作总能在一个可预期的时间内得到响应。
3. **实现简单高效**：算法逻辑清晰，只需要维护一个FIFO队列和简单的时间片计数，调度开销小，在进程数量适中时性能表现良好。
4. **避免进程垄断**：通过强制时间片中断，有效防止了单个计算密集型进程长期占用CPU，保证了系统的整体响应能力。

缺点

1. **时间片选择困难**：时间片设置需要权衡，过短会导致频繁上下文切换降低效率，过长又会影响系统响应性，且没有一个普适的最佳值。
2. **对I/O型进程不友好**：I/O密集型进程经常在时间片未用完时就主动阻塞，浪费了剩余时间片，但重新就绪后仍需等待完整轮转周期。

3. **忽略进程优先级**：完全平等对待所有进程，无法体现不同任务的重要性差异。
4. **上下文切换开销大**：当进程数量较多时，频繁的进程切换会消耗大量CPU时间，特别是在时间片设置过短的情况下，切换开销可能超过实际工作时间。

如何调整时间片大小来优化系统性能

1. **根据系统类型确定基准**：交互式系统（如桌面OS）需要较小时间片（10-100ms）保证响应速度；服务器系统则可设置较大时间片（100-500ms）减少切换开销，这需首先明确系统主要负载类型。
2. **基于进程混合比例动态调整**：系统可监控I/O密集型与CPU密集型进程比例，当I/O进程多时缩短时间片提升响应性，CPU进程多时则延长时间片减少切换损失，实现自适应优化。
3. **考虑硬件性能设置合理范围**：时间片至少应大于上下文切换耗时的10倍以避免效率浪费，并随CPU频率提升而适当增加。

为何需要在 RR_proc_tick 中设置 need_resched 标志

在 RR_proc_tick 中设置 need_resched 标志是因为时间片用完时需要**异步通知调度器进行进程切换**，由于该函数在时钟中断上下文中被调用，不能直接执行复杂的调度操作，通过设置标志位可以让调度器在安全的内核上下文中统一处理调度请求，实现了**中断处理与调度执行的解耦**。

拓展思考

实现优先级RR调度的代码修改思路

要实现优先级RR调度，需要在RR_enqueue函数中为**不同优先级进程分配不同的时间片长度**，高优先级进程获得更大时间片，同时在RR_pick_next函数中**优先选择高优先级进程**，这可以通过维护多个优先级队列或为进程结构添加优先级权重并修改选择逻辑来实现。

当前实现是否支持多核调度

当前实现不支持多核调度，证据如下：

1. **全局单一运行队列**：

```
代码块
1   static struct run_queue __rq;  // 只有一个全局运行队列
2   rq = &__rq;                    // 所有CPU共享同一个队列
```

2. **全局当前进程指针**：

```
代码块
1   struct proc_struct *current = NULL;  // 所有CPU共享的current
```

3. **无CPU标志且同步机制不足：**
    - 进程没有绑定到特定CPU，调度器不知道有几个CPU可用，所有调度决策基于单一队列
    - 只有简单的关中断保护，无自旋锁等多核同步机制，多个CPU同时操作同一队列会导致竞态条件

需要改进的核心方面：

1. 为每个CPU建立独立运行队列并按亲和性分配进程，减少多核锁竞争。
2. 监控各CPU负载差异并定期迁移进程，避免负载不均。
3. 为每个运行队列添加锁并规范加锁顺序，防止多核同步问题。
4. 为进程添加CPU亲和性字段支持绑定策略，优化缓存局部性。

Challenge1：实现 Stride Scheduling 调度算法

多级反馈队列（MLFQ）概要设计

- 队列层级：设 L 层队列，Q0 最高优先级、最短时间片，往下时间片倍增（如 1,2,4,8…）；新建或刚唤醒进程入 Q0。
- 选择策略：调度时从高到低扫描，取首个非空队列的队头进程（各层内部 FIFO/RR）。
- 时间片与降级：进程在当前层用完时间片且仍可运行则降到下一层队尾；未用完时间片就阻塞/主动让出则保持当前层。
- 优先级提）：周期性“全局提升”——将所有可运行进程移回高层（如全部回 Q0 或回上层），并重置时间片；周期可按时钟滴答或调度次数设定。
- I/O 友好：I/O 密集型因频繁阻塞，常在高层获得更多及时响应；CPU 密集型会逐步下沉到低层，得到长时间片、低响应优先级。
- 实现要点：每层一个 run_queue，调度器维护当前最高非空层索引；全局提升时需避免 O(n^2) 移动，可批量重链表；需要配合抢占（时钟中断驱动），以及在 fork/wakeup 时决定入队层级。

Stride 算法时间片与优先级成正比的说明

- 定义：每进程有步长 stride += BIG_STRIDE / priority。优先级越大，增量越小；调度器每次选择最小 stride 的进程。
- 不变量：令进程 i 的累积 stride 为 S_i，运行次数为 k_i。有 S_i = k_i * (BIG_STRIDE / p_i)。
- 选择性质：调度序列保持 S_i 之间差值有界（最多一个增量）；因为每次都取最小值并对其加上自己的步长。
- 比例推导：假设总调度次数 N，若调度策略稳定，则存在常数 C 使对所有 i 有 S_i ≈ C（或相差在一个步长以内）。代入得 k_i ≈ C * p_i / BIG_STRIDE。两进程 i,j 比值 k_i / k_j ≈ p_i / p_j。

- 结论：当 N 充分大时，各进程获得的调度次数（时间片数）与其优先级成比例；误差上界为每进程一个步长的差分，因此随 N 增大误差比例趋近 0。

实现思路与过程

- 定义大步长：在 kern/schedule/default_sched_stride.c 设定 BIG_STRIDE=0x7fffffff，保证 BIG_STRIDE/priority 为正且足够大，使 stride 累加的精度和公平性稳定。
- 运行队列初始化：stride_init 初始化 run_list、lab6_run_pool=NULL、proc_num=0，为后续斜堆/链表操作准备。
- 入队/出队：stride_enqueue/stride_dequeue 以斜堆（lab6_run_pool）维护按最小 stride 的优先队列；入队时校正 time_slice（上限 rq->max_time_slice），确保 lab6_priority>=1，更新 rq 指针与 proc_num；出队对应地从斜堆移除并减少计数。
- 选取下一个进程：stride_pick_next 从斜堆取最小 stride 的进程（为空即返回 NULL），选中后按 lab6_stride += BIG_STRIDE / lab6_priority 更新步长，形成“走最小、累加步长”的比例公平调度。
- 时钟滴答：stride_proc_tick 递减 time_slice，耗尽则置 need_resched=1 触发抢占。
- 调度器切换开关：在 kern/schedule/sched.c 加入 USE_STRIDE_SCHED 宏，默认保持 RR 以通过现有测试；编译时加 -DUSE_STRIDE_SCHED=1 可切换到本实现的 Stride 调度器。

Challenge2：多种基本调度算法的实现与测试

算法实现

一、FIFO算法

1. 算法概述

FIFO是一种最基本的**非抢占式**调度算法。其核心思想是按照进程进入就绪队列的先后顺序进行调度。一旦一个进程获得 CPU，它将一直运行，直到该进程主动放弃 CPU（如等待 I/O、结束运行 `exit`）或调用 `yield`，调度器不会因为时间片耗尽而强制中断其运行。

2. 数据结构与初始化

在 ucore 中，调度器利用 `struct run_queue` 结构体来管理就绪进程。

- **实现方式**：利用双向链表 (`run_list`) 维护就绪队列。
- **初始化 (`FIFO_init`)**：
  - 调用 `list_init` 初始化链表头，使其指向自己，表示队列为空。
  - 将进程计数器 `proc_num` 置为 0。

3. 核心调度逻辑实现

(1) 入队策略：First-In (`FIFO_enqueue`)

FIFO 算法要求新到达的进程必须排在队伍的末尾。

代码块

```
static void FIFO_enqueue(struct run_queue *rq, struct proc_struct *proc) {
    // 确保进程当前不在任何队列中
    assert(list_empty(&(proc->run_link)));

    // list_add_before 在头结点之前插入，相当于插到链表的尾部// 这样保证了先来的进程在链表前面，后来的在后面
    list_add_before(&(rq->run_list), &(proc->run_link));

    // 标记该进程归属于当前的运行队列（防止出队时指针错误导致 panic）
    proc->rq = rq;

    // 队列进程数加 1
    rq->proc_num ++;
}
```

ucore 的链表是循环双向链表。 `list_add_before` 将新节点插入到 `run_list` 头结点的前面，物理上等同于插入到了逻辑队列的队尾。

(2) 调度选择：First-Out (`FIFO_pick_next`)

调度器总是选择队列中最靠前的进程，即等待时间最长的进程。

代码块

```
static struct proc_struct *FIFO_pick_next(struct run_queue *rq) {
    // 获取链表头部的第一个节点
    list_entry_t *le = list_next(&(rq->run_list));

    // 如果链表不为空（le 不指向头结点本身）
    if (le != &(rq->run_list)) {
        // 通过宏将链表节点转换为进程控制块（PCB）指针返回
        return le2proc(le, run_link);
    }
    return NULL;
}
```

直接取链表的第一个节点（Head Next），这就是最早进入队列的进程。算法复杂度为O(1)。

(3) 出队操作 (`FIFO_dequeue`)

当进程被调度执行、阻塞或退出时，需要将其从就绪队列中移除。

代码块

```
static void FIFO_dequeue(struct run_queue *rq, struct proc_struct *proc) {
    // 断言检查：进程确实在队列中，且属于当前队列
    assert(!list_empty(&(proc->run_link)) && proc->rq == rq);

    // 从链表中删除该节点
    list_del_init(&(proc->run_link));

    rq->proc_num --;
}
```

(4) 非抢占式的实现 (`FIFO_proc_tick`)

这是 FIFO 区别于 RR最本质的地方。

代码块

```
static void FIFO_proc_tick(struct run_queue *rq, struct proc_struct *proc) {
    // 【核心逻辑】这里什么都不做
    // RR 算法会在这里减少时间片，并在时间片为 0 时设置 proc->need_resched = 1
    // FIFO 算法忽略时钟中断对调度的影响，从而实现非抢占
}
```

在 ucore 中，每次时钟中断都会调用 sched_class_proc_tick。

- **RR 算法**：会递减时间片，减到 0 就设置 `need_resched` 标志，触发 `schedule()` 。
- **FIFO 算法**：**函数体为空**。这意味着无论时钟中断触发多少次，当前进程永远不会因为时间原因被标记为“需要重新调度”。它会一直持有 CPU，直到它自己代码中调用 `do_yield` 或 `do_exit` 。

4. 总结

`fifo_sched_class` 严格遵循了 FIFO 算法定义：

- **公平性**：通过链表尾部插入 (`enqueue`) 和头部取出 (`pick_next`) 保证顺序。
- **非抢占性**：通过空的 `proc_tick` 函数，剥夺了时钟中断强制切换进程的能力。
- **复杂度**：入队和出队操作均为 O(1)，是所有调度算法中开销最小的。

二、SJF算法

1. 算法概述

SJF (Shortest Job First) 也就是短作业优先调度算法。其核心思想是：当 CPU 空闲时，总是从就绪队列中选择估计运行时间最短的进程进行调度。

- 复用了 `proc_struct` 结构体中的 `lab6_priority` 成员变量来代表进程的估计运行时间
- 数值越小，代表估计运行时间越短，应当被优先执行
- 非抢占式SJF。一旦进程开始运行，即使有更短的作业到达，当前进程也会继续运行直到结束或主动放弃 CPU。

2. 数据结构与初始化

与 FIFO 类似，SJF 也是利用 `struct run_queue` 中的双向链表 (`run_list`) 来管理就绪进程，但维护方式不同。

- **初始化 (`SJF_init`)**:
  - 调用 `list_init` 初始化链表头。
  - 将进程计数器 `proc_num` 置为 0。

3. 核心调度逻辑实现

(1) 入队策略：有序插入 (`SJF_enqueue`)

这是 SJF 算法实现中最关键的部分。与 FIFO 直接插入队尾不同，SJF 在进程入队时必须维护队列的有序性（按运行时间从小到大排序）。

代码块

```
static void SJF_enqueue(struct run_queue *rq, struct proc_struct *proc) {
    assert(list_empty(&(proc->run_link)));

    // 获取队列的第一个元素list_entry_t *le = list_next(&(rq->run_list));

    // 【核心逻辑】遍历队列，寻找合适的插入位置while (le != &(rq->run_list)) {
        struct proc_struct *next = le2proc(le, run_link);

        // 比较当前进程与队列中进程的估计运行时间 (lab6_priority)
        // 如果 proc 的时间更短，说明应该排在这个 next 进程的前面
        if (proc->lab6_priority < next->lab6_priority) {
            break; // 找到了位置，跳出循环
        }
        le = list_next(le);
    }

    // 将进程插入到 le 指向的节点之前
    // 情况1：中途 break，插入到比它长的作业前面
    // 情况2：遍历完整个链表，le 指向头结点，插入到队尾（说明它是当前最长的）
    list_add_before(le, &(proc->run_link));

    proc->rq = rq;
    rq->proc_num ++;
```

```
24  }
```

- 函数的时间复杂度为 O(N)，其中 N 是就绪队列的长度。
- 通过在入队时进行线性扫描和比较，保证了 `run_list` 始终是有序的：队头元素永远是 `lab6_priority` 最小的进程，队尾是最大的。

(2) 调度选择：取队头 (`SJF_pick_next`)

由于 `enqueue` 阶段已经完成了排序工作，调度器的选择变得非常简单。

代码块

```
1  static struct proc_struct *SJF_pick_next(struct run_queue *rq) {
2      list_entry_t *le = list_next(&(rq->run_list));
3
4      // 只要队列不为空，直接取第一个
5      if (le != &(rq->run_list)) {
6          return le2proc(le, run_link);
7      }
8      return NULL;
9  }
```

- 直接获取链表的第一个节点，它必然是当前队列中估计运行时间最短的进程。
- 时间复杂度为 O(1)。

(3) 出队操作 (`SJF_dequeue`)

标准的链表删除操作，与 FIFO 一致。

代码块

```
1  static void SJF_dequeue(struct run_queue *rq, struct proc_struct *proc) {
2      assert(!list_empty(&(proc->run_link)) && proc->rq == rq);
3      list_del_init(&(proc->run_link));
4      rq->proc_num --;
5  }
```

(4) 关键特性：非抢占式的实现 (`SJF_proc_tick`)

在我的实现中，SJF 选择采用非抢占策略。

代码块

```
1  static void SJF_proc_tick(struct run_queue *rq, struct proc_struct *proc) {
2      // 函数体为空// 不对时间片进行计数，不强制设置 need_resched
```

```
3   }
```

4. 总结

`sjf_sched_class` 具有以下特点：

- **有序性**：通过 `enqueue` 时的插入排序，保证就绪队列始终按“估计运行时间”递增排列。
- **贪心策略**：`pick_next` 总是选择当前最优（最短）的作业，试图最小化平均周转时间。
- **非抢占性**：忽略时钟滴答，减少了上下文切换的开销，但也可能导致长作业长时间占用 CPU（如果不主动放弃）。
- **复杂度**：入队操作为 O(N)，出队和选择为 O(1)。相比 FIFO 的 O(1) 入队，SJF 牺牲了少量的入队性能以换取调度上的优化。

三、FP算法

1. 算法概述

FP (Fixed Priority) 即**固定优先级**调度算法。其核心思想是系统总是选择优先级最高的就绪进程来执行。

- 利用 `proc_struct` 结构体中的 `lab6_priority` 成员变量来代表进程的优先级。
- 数值越大，代表优先级越高（与 SJF 中数值越小越优先相反）。
- 该实现是抢占式调度。如果有一个优先级比当前正在运行的进程更高的进程进入就绪队列，调度器会在时钟中断时剥夺当前进程的 CPU 使用权。

2. 数据结构与初始化

FP 同样利用 `struct run_queue` 中的双向链表 (`run_list`) 管理就绪进程。

- **初始化 (`FP_init`)**:
  - 调用 `list_init` 初始化链表头。
  - 将进程计数器 `proc_num` 置为 0。

3. 核心调度逻辑实现

(1) 入队策略：直接插入 (`FP_enqueue`)

与 SJF 在入队时进行排序不同，FP 算法入队时不排序，而是直接放到队尾。

代码块

```
1   static void FP_enqueue(struct run_queue *rq, struct proc_struct *proc) {
2       assert(list_empty(&(proc->run_link)));
3   
4       // 【实现逻辑】直接将新进程插入到链表头部之前（即逻辑上的队尾）
```

```
    // 不进行任何优先级比较或排序操作
    list_add_before(&(rq->run_list), &(proc->run_link));

    // 维护归属队列指针和计数器
    proc->rq = rq;
    rq->proc_num++;
}
```

- 入队操作的时间复杂度为 O(1)。
- 就绪队列 `run_list` 是无序的，因此查找最高优先级的开销被转移到了调度选择阶段（`pick_next`）。

(2) 调度选择：遍历查找（`FP_pick_next`）

由于队列是无序的，调度器必须遍历整个队列来找到优先级最高的那个进程。

代码块

```
static struct proc_struct *FP_pick_next(struct run_queue *rq) {
    list_entry_t *le = list_next(&(rq->run_list));
    if (le == &(rq->run_list)) return NULL; // 队列为空
    struct proc_struct *max_p = NULL;
    uint32_t max_priority = 0;

    // 【核心逻辑】遍历整个链表，寻找优先级最高的进程
    while (le != &(rq->run_list)) {
        struct proc_struct *p = le2proc(le, run_link);

        // 擂台法：如果当前进程优先级 > 已知的最大优先级，则更新候选人
        if (max_p == NULL || p->lab6_priority > max_priority) {
            max_p = p;
            max_priority = p->lab6_priority;
        }
        le = list_next(le);
    }
    return max_p;
}
```

- 该函数的时间复杂度为 O(N)，其中 N 是就绪队列长度。
- 如果有多个进程拥有相同的最高优先级，该算法会选择最先被遍历到的那一个（即队列靠前的，也就是先到达的）。

(3) 出队操作（`FP_dequeue`）

标准的链表删除操作。

代码块

```
static void FP_dequeue(struct run_queue *rq, struct proc_struct *proc) {
    assert(!list_empty(&(proc->run_link)) && proc->rq == rq);
    list_del_init(&(proc->run_link));
    rq->proc_num--;
}
```

(4) 关键特性：抢占式的实现 (`FP_proc_tick`)

这是 FP 算法与本实验中实现的 FIFO 和 SJF 最本质的区别。FP 实现了基于优先级的抢占。

代码块

```
static void FP_proc_tick(struct run_queue *rq, struct proc_struct *proc) {
    // 特殊情况：如果当前是 idle 进程，必须立刻调度
    if (proc == idleproc) {
        proc->need_resched = 1;
        return;
    }

    // 【抢占逻辑】遍历就绪队列
    list_entry_t *le = list_next(&(rq->run_list));
    while (le != &(rq->run_list)) {
        struct proc_struct *p = le2proc(le, run_link);

        // 如果发现队列中有一个进程的优先级 > 当前正在运行进程(proc)的优先级
        if (p->lab6_priority > proc->lab6_priority) {
            // 设置调度标记，触发 schedule()，剥夺当前进程 CPU
            proc->need_resched = 1;
            return;
        }
        le = list_next(le);
    }
}
```

- 每次时钟中断都会触发此函数。
- 函数会检查就绪队列中是否存在比当前进程优先级更高的进程。
- 一旦发现，立即设置 `need_resched = 1` 。这保证了高优先级任务一旦就绪，能以极低的延迟（最长为一个 tick 周期）获得 CPU，实现了实时响应。

4. 总结

`fp_sched_class` 具有以下特点：

- **抢占性**：这是其实时性的保证。只要有更高优先级的任务就绪，低优先级任务会被强制暂停。

- **无序队列**：入队 O(1)，但选择进程 O(N)。这与 SJF（入队 O(N)，选择 O(1)）形成了鲜明对比。
- **优先级定义**：`lab6_priority` 值越大，优先级越高。
- **开销分析**：由于 `proc_tick` 中也包含了遍历队列的操作（用于检查抢占条件），因此在高负载（队列进程很多）情况下，时钟中断的处理开销会比 FIFO/RR 显著增加。

测试用例设计

一、响应时间测试（`test_metric_rt`）

1. 测试目的

验证调度算法的抢占特性 (Preemptive)。

- **指标定义**：响应时间 (Response Time) = 进程首次获得 CPU 的时刻 - 进程到达（提交）时刻。
- **核心问题**：当一个高优先级的紧急任务到达时，调度器能否立即中断当前运行的低优先级长作业。

2. 代码说明

模拟了一个“长作业霸占 CPU，紧急任务突然到达”的场景。

1. 制造阻塞背景：首先创建一个低优先级（lab6_setpriority(10)）的长作业进程，该进程执行大量计算 (WORK_LOAD) 以占满 CPU。

```
代码块
1   // test_metric_rt.cif ((pid = fork()) == 0) {
2       lab6_setpriority(10); // Low Priority
3       work(WORK_LOAD);      // 模拟长作业占用 CPUexit(0);
4   }
```

2. 模拟随机到达：父进程利用 while 循环延时约 200ms，模拟系统运行一段时间后，紧急任务才被提交。

```
代码块
1   unsigned int delay_start = gettime_msec();
2   while (gettime_msec() - delay_start < 200);
```

3. 构造高优先级紧急任务：为了确保紧急任务一经创建就具有高优先级（触发 FP 抢占），父进程在 fork 前先提升自己的优先级。利用 ucore 的 do_fork 优先级继承机制，子进程将继承这个高优先级。

```
代码块
1   // 父进程提升优先级，准备创建紧急任务
```

```
    lab6_setpriority(100);
    unsigned int arrival_time = gettime_msec(); // 记录到达时间(T_arrival)
    if ((pid = fork()) == 0) {
        // 子进程(Urgent Job)开始执行// ...
    }
```

4. 测量响应时间：子进程获得 CPU 后执行的第一行代码用于记录 start_time。如果是抢占式调度，start_time 应几乎等于 arrival_time；如果是非抢占，start_time 会被推迟到长作业结束。

代码块

```
    // 子进程代码
    lab6_setpriority(100);
    unsigned int start_time = gettime_msec(); // 记录开始运行时间(T_start)
    cprintf("... Response Time = %d ms\n", start_time - arrival_time);
```

二、平均周转时间测试 (`test_metric_att.c`)

1. 测试目的

验证调度算法的吞吐率和抗“护航效应”能力。

- **指标定义**：周转时间 (Turnaround Time) = 进程结束时刻 - 进程到达时刻。
- **核心问题**：调度器能否智能地优先处理短作业，从而降低所有任务的平均等待时间。

2. 代码说明

该测试构建了一个“长短作业混合”的就绪队列场景。

1. 长作业主动让权：首先创建一个长作业 P1。为了让 SJF 算法有机会对队列进行重排，长作业在启动后立即调用 yield()。

代码块

```
    // test_metric_att.cif ((pid = fork()) == 0) {
        lab6_setpriority(100); // SJF中代表长作业(优先级数值大)
        yield();               // 【关键】主动让出CPU，允许短作业进入就绪队列
        work(LONG_WORK);       // 执行长任务exit(0);
    }
```

如果不调用 `yield()` ，在非抢占式调度下，长作业一旦开始就会跑到底。 `yield()` 将 P1 放回就绪队列尾部，此时父进程继续创建短作业，使得队列中同时存在 `{P1, P2, P3, P4}` ，SJF 调度器即可从中挑选最优进程。

2. 短作业密集到达：父进程连续创建 3 个短作业，优先级设为 10（代表短作业）。

```
代码块
1   for (int i = 0; i < 3; i++) {
2       arrival_ts = gettime_msec(); // 在入队前记录到达时间
3       if ((pid = fork()) == 0) {
4           lab6_setpriority(10);    // SJF中代表短作业
5           work(SHORT_WORK);        // 执行短任务
6           end_ts = gettime_msec(); // 记录完成时间
7           // 计算周转时间
8           cprintf("Short_Job_%d Turnaround: %d ms\n", i+1, end_ts - arrival_ts);
9           exit(0);
10      }
11  }
```

通过对比短作业的周转时间，可以清晰区分算法策略。

三、调度开销测试 (`test_metric_overhead.c`)

1. 测试目的

评估调度算法的计算复杂度 (Computational Complexity)。

- **指标定义**：调度开销 ≈ (N个极短任务的总耗时) / N。
- **核心问题**：复杂的调度逻辑（如 SJF/FP 的链表排序 O(N)）是否会显著拖慢系统运行速度。

2. 代码说明

该测试采用“放大法”，通过海量微小进程的创建与销毁，将微秒级的调度开销放大到毫秒级以便观测。

1. 高并发负载设定：将进程数 PROCESS_NUM 设定为 1000，工作量 TINY_WORK 设定为极小值（100次循环）。

```
代码块
1   // test_metric_overhead.c
2   #define PROCESS_NUM 1000
3   #define TINY_WORK 100
```

2. 累积时间测量：父进程记录从第一子进程创建前到最后一个子进程回收后的总时间。

```
代码块
1   unsigned int start = gettime_msec();
2
3   for (int i = 0; i < PROCESS_NUM; i++) {
4       if ((pid = fork()) == 0) {
5           work(TINY_WORK); // 子进程几乎不做任何事，主要开销在调度和切换
```

```
6           exit(0);
7       }
8   }
9
10  // 等待所有 1000 个子进程结束
11  for (int i = 0; i < PROCESS_NUM; i++) {
12      wait();
13  }
14
15  unsigned int total_time = gettime_msec() - start;
```

3. 结果计算：总时间 total_time 包含了 1000 次 do_fork、exit 以及数千次 schedule（入队、出队、选择下一个进程）的开销。由于 do_fork 等内核操作对所有算法是固定的，total_time 的差异直接反映了调度算法 enqueue 和 pick_next 函数的效率差异。

测试结果

一、测试用例test_metric_rt.c

1. FP算法

代码块

```
1   sched class: FP_scheduler
2   ++ setup timer interrupts
3   kernel_execve: pid = 2, name = "test_metric_rt".
4   set priority to 100
5   Urgent_Job: Arrived (Forked) at 200 ms
6   set priority to 100
7   Urgent_Job: Started Running at 200 ms
8   Urgent_Job: Response Time = 0 ms
9   set priority to 10
10  all user-mode processes have quit.
11  init check memory pass.
12  kernel panic at kern/process/proc.c:574:
13      initproc exit.
```

2. FIFO算法

代码块

```
1   sched class: FIFO_scheduler
2   ++ setup timer interrupts
3   kernel_execve: pid = 2, name = "test_metric_rt".
4   set priority to 100
```

```
Urgent_Job: Arrived (Forked) at 200 ms
set priority to 10
set priority to 100
Urgent_Job: Started Running at 600 ms
Urgent_Job: Response Time = 400 ms
all user-mode processes have quit.
init check memory pass.
kernel panic at kern/process/proc.c:574:
    initproc exit.
```

3. SJF算法

代码块

```
sched class: SJF_scheduler
++ setup timer interrupts
kernel_execve: pid = 2, name = "test_metric_rt".
set priority to 100
Urgent_Job: Arrived (Forked) at 200 ms
set priority to 10
set priority to 100
Urgent_Job: Started Running at 600 ms
Urgent_Job: Response Time = 400 ms
all user-mode processes have quit.
init check memory pass.
kernel panic at kern/process/proc.c:574:
    initproc exit.
```

结果对比：

调度算法	调度策略类型	紧急任务到达时刻	紧急任务开始运行时刻	响应时间	现象分析
FP (Fixed Priority)	**抢占式**	200 ms	200 ms	**0 ms**	**实时响应**。高优先级进程一经创建，立即抢占 CPU，无需等待。
FIFO (先来先服务)	**非抢占**	200 ms	600 ms	**400 ms**	**严重阻塞**。即使新任务很紧急，也必须等待当前运行的长作业彻底结束。
SJF (短作业优先)	**非抢占**	200 ms	600 ms	**400 ms**	**同 FIFO**。虽然新任务很短，但因不支持抢占，只能排队等待 CPU 释放。

二、测试用例test_metric_att.c

1. FP算法

代码块

```
sched class: FP_scheduler
++ setup timer interrupts
kernel_execve: pid = 2, name = "test_metric_att".
TEST: ATT/AWT Analysis Started.
set priority to 100
set priority to 10
Short_Job_1 Turnaround: 430 ms
set priority to 10
Short_Job_2 Turnaround: 470 ms
set priority to 10
Short_Job_3 Turnaround: 510 ms
TEST: ATT/AWT Analysis Finished.
all user-mode processes have quit.
init check memory pass.
kernel panic at kern/process/proc.c:574:
    initproc exit.
```

2. FIFO算法

代码块

```
sched class: FIFO_scheduler
++ setup timer interrupts
```

```
kernel_execve: pid = 2, name = "test_metric_att".
TEST: ATT/AWT Analysis Started.
set priority to 100
set priority to 10
Short_Job_1 Turnaround: 440 ms
set priority to 10
Short_Job_2 Turnaround: 470 ms
set priority to 10
Short_Job_3 Turnaround: 510 ms
TEST: ATT/AWT Analysis Finished.
all user-mode processes have quit.
init check memory pass.
kernel panic at kern/process/proc.c:574:
    initproc exit.
```

3. SJF算法

代码块

```
sched class: SJF_scheduler
++ setup timer interrupts
kernel_execve: pid = 2, name = "test_metric_att".
TEST: ATT/AWT Analysis Started.
set priority to 100
set priority to 10
Short_Job_1 Turnaround: 50 ms
set priority to 10
Short_Job_2 Turnaround: 90 ms
set priority to 10
Short_Job_3 Turnaround: 130 ms
TEST: ATT/AWT Analysis Finished.
all user-mode processes have quit.
init check memory pass.
kernel panic at kern/process/proc.c:574:
    initproc exit.
```

调度算法	短作业 1 周转	短作业 2 周转	短作业 3 周转	平均周转时间	现象分析
SJF (短作业优先)	50 ms	90 ms	130 ms	**90 ms (最优)**	**极佳**。SJF 成功识别出队列中的短任务，让它们插队在长任务之前执行，消除了护航效应。
FIFO (先来先服务)	440 ms	470 ms	510 ms	**473 ms (差)**	**严重护航效应**。短作业必须等待前面的长作业完全跑完才能开始，等待时间极长。
FP (固定优先级)	430 ms	470 ms	510 ms	**470 ms (差)**	**取决于设置**。因测试中长作业优先级 (100) > 短作业 (10)，FP 选择先运行长作业，导致表现与 FIFO 一致。

三、测试用例test_metric_overhead.c

1. FP算法

代码块

```
sched class: FP_scheduler
++ setup timer interrupts
kernel_execve: pid = 2, name = "test_metric_overhead".
TEST: Overhead Analysis Started (N=1000)...
Total time for 1000 tiny processes: 200 ms
Average overhead per process: 0.2 ms
all user-mode processes have quit.
init check memory pass.
kernel panic at kern/process/proc.c:574:
    initproc exit.
```

2. FIFO算法

代码块

```
sched class: FIFO_scheduler
++ setup timer interrupts
kernel_execve: pid = 2, name = "test_metric_overhead".
TEST: Overhead Analysis Started (N=1000)...
Total time for 1000 tiny processes: 190 ms
```

```
Average overhead per process: 0.1 ms
all user-mode processes have quit.
init check memory pass.
kernel panic at kern/process/proc.c:574:
    initproc exit.
```

3. SJF算法

代码块

```
sched class: SJF_scheduler
++ setup timer interrupts
kernel_execve: pid = 2, name = "test_metric_overhead".
TEST: Overhead Analysis Started (N=1000)...
Total time for 1000 tiny processes: 200 ms
Average overhead per process: 0.2 ms
all user-mode processes have quit.
init check memory pass.
kernel panic at kern/process/proc.c:574:
    initproc exit.
```

调度算法	总耗时 (N=1000)	平均单进程开销	算法复杂度	现象分析
FIFO (先来先服务)	**190 ms**	**0.19 ms**	**O(1)**	**开销最低**。直接入队尾，无需遍历和排序，适合对系统吞吐量要求极高的场景。
SJF (短作业优先)	**200 ms**	**0.20 ms**	**O(N)**	**开销稍高**。每次入队都需要遍历运行队列以保持有序，进程越多开销越大。
FP (固定优先级)	**200 ms**	**0.20 ms**	**O(N)**	**开销稍高**。同SJF，需维护优先级队列，带来了额外的排序计算成本。