

总体介绍和调度过程

总体介绍

- 目标
 - 理解操作系统的调度管理机制
 - 熟悉 ucore 调度器框架
 - 理解 Round-Robin 调度算法
 - 理解并实现 Stride 调度算法
- 练习
 - 分析 ucore 调度器框架
 - 分析 Round-Robin 调度算法
 - 分析并实现 Stride 调度算法
- 流程概述

hello 应用程序

```
1  #include <stdio.h>
2  #include <ulib.h>
3
4  int main(void) {
5      cprintf("Hello world!!.\n");
6      cprintf("I am process %d.\n", getpid());
7      cprintf("hello pass.\n");
8      return 0;
9  }
```

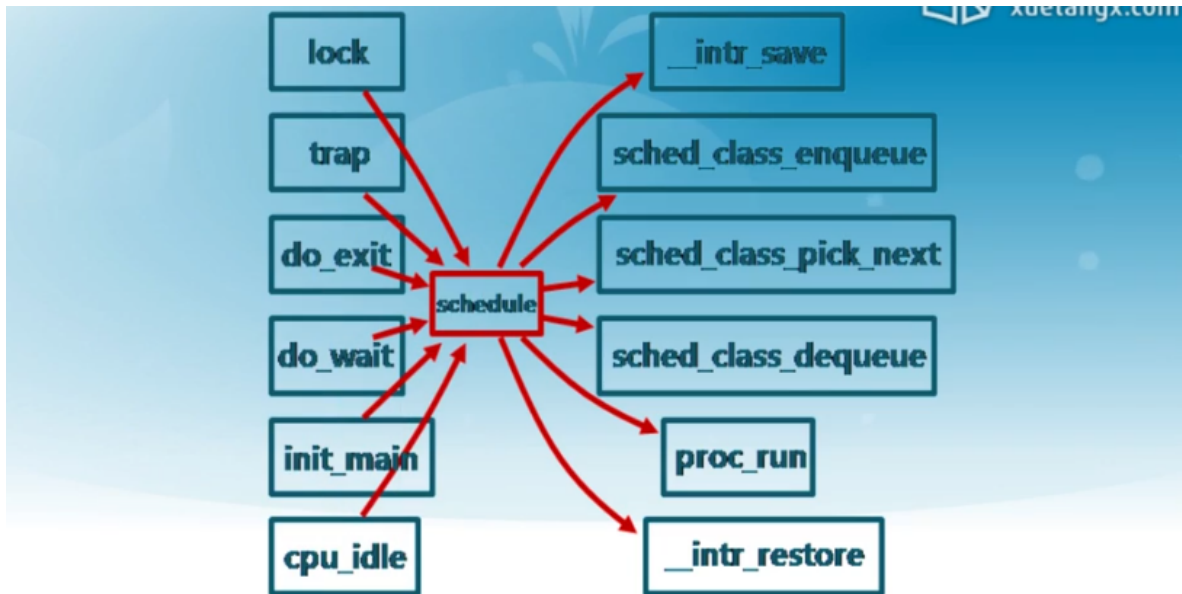
- 回到 lab5
 - 一般进程
 - 从头进行
 - 直到结束
 - 结束后进行进程切换
 - idle 进程
 - 不断地遍历进程池
 - 直到找到第一个 runnable 状态的进程
 - 调用并通过进程切换来执行新进程
- lab6 重新设计调度框架



调度过程

1. 触发: trigger scheduling
2. 入队: 'enqueue'
3. 选取: pick up
4. 出队: 'dequeue'
5. 切换: process switch

调度算法支撑框架



1. 触发: trigger scheduling ----> proc_tick
2. 入队: 'enqueue' ----> enqueue
3. 选取: pick up ----> proc_next
4. 出队: 'dequeue' ----> dequeue
5. 切换: process switch ----> switch_to

- 问题: 调度算法如何知道进程的事件使用情况
- 答案: 让调度算法能够感知到时钟中断的发生
- 进入/离开就绪队列 —— 机制
 - 抽象数据结构, 可以不是队列
 - 可根据调度算法的需求采用各种具体数据结构
- 调度算法的核心 —— 策略
 - 与调度算法无关, 与硬件相关

```
1 struct sched_class {
2     const char *name;
3     (*init)(struct run_queue *rq);
4     (*enqueue)(struct run_queue *rq, ...);
5     (*dequeue)(struct run_queue *rq, ...);
6     (*pick_next)(struct run_queue *rq);
7     (*proc_tick)(struct run_queue *r, ...);
8 };
```

```
1 void
2 schedule(void) {
3     bool intr_flag;
```

```

4      struct proc_struct *next;
5      local_intr_save(intr_flag);
6      {
7          current->need_resched = 0;
8          if (current->state == PROC_RUNNABLE) {
9              sched_class_enqueue(current);
10         }
11         if ((next = sched_class_pick_next()) != NULL) {
12             sched_class_dequeue(next);
13         }
14         if (next == NULL) {
15             next = idleproc;
16         }
17         next->runs ++;
18         if (next != current) {
19             proc_run(next);
20         }
21     }
22     local_intr_restore(intr_flag);
23 }

```

调度点

- 调度点：触发做调度相关的工作

编号	位 置	原 因
1	proc.c:do_exit	用户线程执行结束，主动放弃CPU
2	proc.c:do_wait	用户线程等待子进程结束，主动放弃CPU
3	proc.c:init_main	1.Initproc内核线程等待所有用户进程结束 2.所有用户进程结束后，回收系统资源
4	proc.c::cpu_idle	idleproc内核线程等待处于就绪态的进程或线程，如果有选取一个并切换进程
5	sync.h::lock	进程如果无法得到锁，则主动放弃CPU
6	Trap.c::trap	修改当前进程时间片，若时间片用完，则设置need_resched为1，让当前进程放弃CPU

与调度算法相关

时间片轮转调度算法

Round Robin 调度算法 - 初始化 (default_sched.c)

```

1 struct sched_class {
2     const char *name;
3     (*init)(struct run_queue *rq);
4     (*enqueue)(struct run_queue *rq, ...);
5     (*dequeue)(struct run_queue *rq, ...);
6     (*pick_next)(struct run_queue *rq);
7     (*proc_tick)(struct run_queue *r, ...);
8 };

```

```

1 static void
2 RR_init(struct run_queue *rq) {
3     list_init(&(rq->run_list));
4     rq->proc_num = 0;
5 }

```

```

1 struct run_queue {
2     list_entry_t run_list;
3     unsigned int proc_num;
4     int max_time_slice;
5 };

```

Round Robin 调度算法 - proc_tick (default_sched.c)

```

1 static void
2 RR_proc_tick(struct run_queue *rq, struct proc_struct *proc) {
3     if (proc->time_slice > 0) {
4         proc->time_slice--;
5     }
6     if (proc->time_slice == 0) {
7         proc->need_resched = 1;
8     }
9 }

```

Round Robin 调度算法 - enqueue (default_sched.c)

```

1 static void
2 RR_enqueue(struct run_queue *rq, struct proc_struct *proc) {
3     assert(list_empty(&(proc->run_link)));
4     list_add_before(&(rq->run_list), &(proc->run_link));
5     if (proc->time_slice == 0 || proc->time_slice > rq->max_time_slice) {
6         proc->time_slice = rq->max_time_slice;
7     }
8     proc->rq = rq;
9     rq->proc_num++;
10 }

```

Round Robin 调度算法 - pick_next (default_sched.c)

```

1 static struct proc_struct *
2 RR_pick_next(struct run_queue *rq) {
3     list_entry_t *le = list_next(&(rq->run_list));
4     if (le != &(rq->run_list)) {
5         return le2proc(le, run_link);
6     }
7     return NULL;
8 }

```

- 问题: NULL?
- 答案: NULL 将被 idle 进程取代

Round Robin 调度算法 - dequeue (default_sched.c)

```

1 static void
2 RR_dequeue(struct run_queue *rq, struct proc_struct *proc) {
3     assert(!list_empty(&(proc->run_link)) && proc->rq == rq);
4     list_del_init(&(proc->run_link));
5     rq->proc_num --;
6 }

```

Round Robin 调度算法 - 绑定 / 公布

```

1 ===== default_sched.c =====
2 struct sched_class default_sched_class = {
3     .name = "RR_scheduler",
4     .init = RR_init,
5     .enqueue = RR_enqueue,
6     .dequeue = RR_dequeue,
7     .pick_next = RR_pick_next,
8     .proc_tick = RR_proc_tick,
9 };
10 ===== sched.c =====
11 void sched_init(void) {
12     .....
13     sched_class = &default_sched_class;
14     .....
15 }

```

Stride调度算法

特征

- 基于优先级 -- Priority-based
- 调度选择是确定的 -- Deterministic

实现 (YOUR WORK)

- 选择合适的数据结构 (list, priority queue, etc,) init()
 - 初始化数据结构: in init()
 - 更新数据结构: 涉及 enqueue() 和 dequeue()
- 实现 Stride 调度算法选取下一个进程: in pick_next()
- 处理时钟 ticks: in proc_tick()

- 如果认为当前进程用完了时间片，则 `proc->need_resched` 为 1
- 实现入队/出队：`enqueue()`, `dequeue()`
替换 `default_sched_class`：in `sched_init()`
- 执行 'make run-priority' 来测试你实现的 Stride 调度算法

Skep heap (斜堆) 数据结构

priority queue

```

1 struct skew_heap_entry {
2     struct skew_heap_entry *parent, *left, *right;
3 };
4 (*compare_f)(void *a, void *b);
5 skew_heap_init(skew_heap_entry_t *a);
6 skew_heap_insert(skew_heap_entry_t *a, ...);
7 skew_heap_remove(skew_heap_entry_t *a, ...);

```

```

1 struct proc_struct {
2     .....
3     skew_heap_entry_t lab6_run_pool;
4     uint32_t lab6_stride;
5     uint32_t lab6_priority;
6 };
7 struct run_queue {
8     .....
9     skew_heap_entry_t *lab6_run_pool;
10 }

```

步进值 `pass` 与优先级 `priority` 的关系

`pass = BIG_VALUE / lab6_priority`

如何避免 `stride` 溢出？

`STRIDE_MAX - STRIDE_MIN <= PASS_MAX`

`stride`, `pass` 是无符号整数

用有符号整数表示 (`Proc.A.stride - Proc.B.stride`)