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

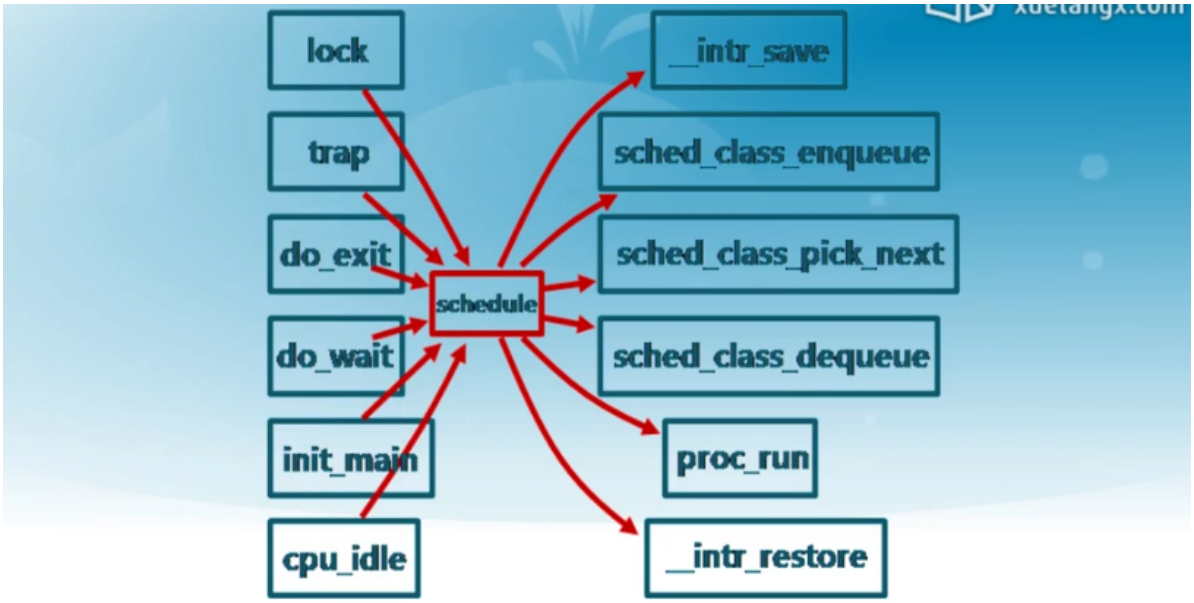
```c
#include <stdio.h>
#include <ulib.h>

int main(void) {
    cprintf("Hello world!!.\n");
    cprintf("I am process %d.\n", getpid());
    cprintf("hello pass.\n");
    return 0;
}
```

  - 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：tigger 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：tigger 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

```
struct sched_class {
    const char *name;
    (*init)(struct run_queue *rq);
    (*enqueue)(struct run_queue *rq, ...);
    (*dequeue)(struct run_queue *rq, ...);
    (*pick_next)(struct run_queue *rq);
    (*proc_tick)(struct run_queue *r, ...);
};
```

```
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);
        }
    }
    local_intr_restore(intr_flag);
}
```

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)

```
struct sched_class {
    const char *name;
    (*init)(struct run_queue *rq);
    (*enqueue)(struct run_queue *rq, ...);
    (*dequeue)(struct run_queue *rq, ...);
    (*pick_next)(struct run_queue *rq);
    (*proc_tick)(struct run_queue *r, ...);
};
```

```
static void
RR_init(struct run_queue *rq) {
    list_init(&(rq->run_list));
    rq->proc_num = 0;
}
```

```
struct run_queue {
    list_entry_t run_list;
    unsigned int proc_num;
    int max_time_slice;
};
```

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

```
static void
RR_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;
    }
}
```

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

```
static void
RR_enqueue(struct run_queue *rq, struct proc_struct *proc) {
    assert(list_empty(&(proc->run_link)));
    list_add_before(&(rq->run_list), &(proc->run_link));
    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 ++;
}
```

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

```
static struct proc_struct *
RR_pick_next(struct run_queue *rq) {
    list_entry_t *le = list_next(&(rq->run_list));
    if (le != &(rq->run_list)) {
        return le2proc(le, run_link);
    }
    return NULL;
}
```

- 问题：NULL?
- 答案：NULL 将被 idle 进程取代

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

```
static void
RR_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 --;
}
```

Round Robin 调度算法 - 绑定 / 公布

```
===== default_sched.c =====
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,
};
===== sched.c =====
void sched_init(void) {
    ......
    sched_class = &default_sched_class;
    ......
}
```

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

```
struct skew_heap_entry {
     struct skew_heap_entry *parent, *left, *right;
};
(*compare_f)(void *a, void *b);
skew_heap_init(skew_heap_entry_t *a);
skew_heap_insert(skew_heap_entry_t *a, ...);
skew_heap_remove(skew_heap_entry_t *a, ...);
```

```
struct proc_struct {
    ......
    skew_heap_entry_t lab6_run_pool;
    uint32_t lab6_stride;
    uint32_t lab6_priority;
};
struct run_queue {
    ......
    skew_heap_entry_t *lab6_run_pool;
}
```

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）