姓名；张磊

学号：171491311

班级：硬件二班

**实验题目**：调度器

**实验目的**：理解操作系统的调度管理机制

熟悉 ucore 的系统调度器框架，以及缺省的Round-Robin 调度算法

基于调度器框架实现一个(Stride Scheduling)调度算法来替换缺省的调度算法

**实验内容：**

**练习0：**填写已有实验

需要修改的文件：proc.c

default\_pmm.c

pmm.c

swap\_fifo.c

vmm.c

trap.c

修改alloc\_proc:

static struct proc\_struct \*

alloc\_proc(void) {

struct proc\_struct \*proc = kmalloc(sizeof(struct proc\_struct));

if (proc != NULL) {

proc->state = PROC\_UNINIT;

proc->pid = -1;

proc->runs = 0;

proc->kstack = 0;

proc->need\_resched = 0;

proc->parent = NULL;

proc->mm = NULL;

memset(&(proc->context), 0, sizeof(struct context));

proc->tf = NULL;

proc->cr3 = boot\_cr3;

proc->flags = 0;

memset(proc->name, 0, PROC\_NAME\_LEN);

proc->wait\_state = 0;

proc->cptr = proc->optr = proc->yptr = NULL;

proc->rq = NULL;

list\_init(&(proc->run\_link));

proc->time\_slice = 0;

proc->lab6\_run\_pool.left = proc->lab6\_run\_pool.right = proc->lab6\_run\_pool.parent = NULL;

proc->lab6\_stride = 0;

proc->lab6\_priority = 0;

}

return proc;

}

对proc\_stuct结构体扩展为：

proc->rq = NULL;//初始化运行队列为空

list\_init(&(proc->run\_link));//初始化运行队列的指针

proc->time\_slice = 0;//初始化时间片

proc->lab6\_run\_pool.left = proc->lab6\_run\_pool.right proc->lab6\_run\_pool.parent = NULL; //初始化各类指针为空，包括父进程等待

proc->lab6\_stride = 0;//步数初始化

proc->lab6\_priority = 0;//初始化优先级

trar\_dispatch函数修改为：

static void

trap\_dispatch(struct trapframe \*tf) {

......

......

ticks ++;

assert(current != NULL);

run\_timer\_list(); //更新定时器，并根据参数调用调度算法

break;

......

......

}

**练习一：使用 Round Robin 调度算法（不需要编码）**

Round Robin调度算法的调度思想是让所有 runnable 态的进程分时轮流使用 CPU 时间。Round Robin 调度器维护当前 runnable进程的有序运行队列。当前进程的时间片用完之后,调度器将当前进程放置到运行队列的尾部，再从其头部取出进程进行调度。

RR\_init函数，函数比较简单，不再罗列，完成了对进程队列的初始化

RR\_enqueue函数，它把进程的进程控制块指针放入到rq队列末尾，且如果进程控制块的时间片为0，则需要把它重置为max\_time\_slice。这表示如果进程在当前的执行时间片已经用完，需要等到下一次有机会运行时，才能再执行一段时间。然后在依次调整rq和rq的进程数目加一

RR\_dequeue函数：简单的把就绪进程队列rq的进程控制块指针的队列元素删除，然后使就绪进程个数的proc\_num减一

RR\_pick\_next函数：选取函数，即选取就绪进程队列rq中的队头队列元素，并把队列元素转换成进程控制块指针，即置为当前占用CPU的程序。

RR\_proc\_tick函数：即每一次时间片到时的时候，当前执行进程的时间片time\_slice便减一。如果time\_slice降到零，则设置此进程成员变量need\_resched标识为1，这样在下一次中断来后执行trap函数时，会由于当前进程程成员变量need\_resched标识为1而执行schedule函数，从而把当前执行进程放回就绪队列末尾，而从就绪队列头取出在就绪队列上等待时间最久的那个就绪进程执行

**练习2: 实现 Stride Scheduling 调度算法（需要编码）**

思想：

1、为每个runnable的进程设置一个当前状态stride，表示该进程当前的调度权。另外定义其对应的pass值，表示对应进程在调度后，stride 需要进行的累加值。

2、每次需要调度时，从当前 runnable 态的进程中选择 stride最小的进程调度。对于获得调度的进程P，将对应的stride加上其对应的步长pass（只与进程的优先权有关系）。

3、在一段固定的时间之后，回到步骤2，重新调度当前stride最小的

代码：#include <defs.h>

#include <list.h>

#include <proc.h>

#include <assert.h>

#include <default\_sched.h>

#define USE\_SKEW\_HEAP 1

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;

if (c > 0) return 1;

else if (c == 0) return 0;

else return -1;

}

static void

stride\_init(struct run\_queue \*rq) {

/\* LAB6: YOUR CODE \*/

list\_init(&(rq->run\_list));

rq->lab6\_run\_pool = NULL;

rq->proc\_num = 0;

}

static void

stride\_enqueue(struct run\_queue \*rq, struct proc\_struct \*proc) {

/\* LAB6: YOUR CODE \*/

#if USE\_SKEW\_HEAP

rq->lab6\_run\_pool = skew\_heap\_insert(rq->lab6\_run\_pool, &(proc->lab6\_run\_pool), proc\_stride\_comp\_f);

#else

assert(list\_empty(&(proc->run\_link)));

list\_add\_before(&(rq->run\_list), &(proc->run\_link));

#endif

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 ++;

}

static void

stride\_dequeue(struct run\_queue \*rq, struct proc\_struct \*proc) {

/\* LAB6: YOUR CODE \*/

#if USE\_SKEW\_HEAP

rq->lab6\_run\_pool =

skew\_heap\_remove(rq->lab6\_run\_pool, &(proc->lab6\_run\_pool), proc\_stride\_comp\_f);

#else

assert(!list\_empty(&(proc->run\_link)) && proc->rq == rq);

list\_del\_init(&(proc->run\_link));

#endif

rq->proc\_num --;

}

static struct proc\_struct \*

stride\_pick\_next(struct run\_queue \*rq) {

/\* LAB6: YOUR CODE \*/

#if USE\_SKEW\_HEAP

if (rq->lab6\_run\_pool == NULL) return NULL;

struct proc\_struct \*p = le2proc(rq->lab6\_run\_pool, lab6\_run\_pool);

#else

list\_entry\_t \*le = list\_next(&(rq->run\_list));

if (le == &rq->run\_list)

return NULL;

struct proc\_struct \*p = le2proc(le, run\_link);

le = list\_next(le);

while (le != &rq->run\_list)

{

struct proc\_struct \*q = le2proc(le, run\_link);

if ((int32\_t)(p->lab6\_stride - q->lab6\_stride) > 0)

p = q;

le = list\_next(le);

}

#endif

if (p->lab6\_priority == 0)

p->lab6\_stride += BIG\_STRIDE;

else p->lab6\_stride += BIG\_STRIDE / p->lab6\_priority;

return p;

}

static void

stride\_proc\_tick(struct run\_queue \*rq, struct proc\_struct \*proc) {

/\* LAB6: YOUR CODE \*/

if (proc->time\_slice > 0) {

proc->time\_slice --;

}

if (proc->time\_slice == 0) {

proc->need\_resched = 1;

}

}

struct sched\_class default\_sched\_class = {

.name = "stride\_scheduler",

.init = stride\_init,

.enqueue = stride\_enqueue,

.dequeue = stride\_dequeue,

.pick\_next = stride\_pick\_next,

.proc\_tick = stride\_proc\_tick,

};