**Lab6 调度**

练习0：填写已有实验

本实验依赖实验1/2/3/4/5。请把你做的实验2/3/4/5的代码填入本实验中代码中有“LAB1”/“LAB2”/“LAB3”/“LAB4”“LAB5”的注释 相应部分。并确保编译通过。注意：为了能够正确执行lab6的测试应用程序，可能需对已完成的实验1/2/3/4/5的代码进行进 一步改进。

alloc\_proc函数

完整代码如下：

static struct proc\_struct \*

alloc\_proc(void) {

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

if (proc != NULL) {

proc->state = PROC\_UNINIT; //设置进程初始状态

proc->pid = -1; //进程id=-1

proc->runs = 0; //初始化时间片为0

proc->kstack = 0; //初始化内存栈的地址为0

proc->need\_resched = 0; //是否需要调度设为不需要

proc->parent = NULL; //将父节点置空

proc->mm = NULL; //置空虚拟内存

memset(&(proc->context), 0, sizeof(struct context));//初始化上下文

proc->tf = NULL; //将中断帧指针置空

proc->cr3 = boot\_cr3; //将页目录设内核页目录表的基址

proc->flags = 0; //初始化标志位

memset(proc->name, 0, PROC\_NAME\_LEN);//置空进程名

proc->wait\_state = 0; //初始化进程等待状态

proc->cptr=proc->yptr=proc->optr = NULL;//初始化进程相关指针

proc->rq = NULL;//置空运行队列

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

proc->lab6\_stride = 0; //初始化当前运行步数

proc->lab6\_priority = 0; //初始化优先级

}

return proc;

}

相比于lab5，lab6对proc\_struct结构体再次做了扩展，这里主要是多出了以下部分

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

trap\_dispatch函数

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1: 使用 Round Robin 调度算法（不需要编码）

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。执行make grade，大部分测试用例应该通过。但执行priority.c应该过不去。 请在实验报告中完成： 请理解并分析sched\_calss中各个函数指针的用法，并接合Round Robin 调度算法描ucore的调度执行过程 请在实验报告中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

0x1 执行过程

让所有runnable态的进程分时轮流使用CPU时间。RR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调度算法的就绪队列在组织结构上也是一个双向链表，只是增加了一个成员变量，表明在此就绪进程队列中的最大执行时间片。而且在进程控制块proc\_struct中增加了一个成员变量time\_slice，用来记录进程当前的可运行时间片段。这是由于RR调度算法需要考虑执行进程的运行时间不能太长。在每个timer到时的时候，操作系统会递减当前执行进程的time\_slice，当time\_slice为0时，就意味着这个进程运行了一段时间（这个时间片段称为进程的时间片），需要把CPU让给其他进程执行，于是操作系统就需要让此进程重新回到rq的队列尾，且重置此进程的时间片为就绪队列的成员变量最大时间片max\_time\_slice值，然后再从rq的队列头取出一个新的进程执行。

0x2 算法实现

RR\_init完成了对进程队列的初始化

static void

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

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

rq->proc\_num = 0;

}

RR\_enqueue的函数实现如下表所示。即把某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rq成员变量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。

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 ++;

}

RR\_dequeue的函数实现如下表所示。即把就绪进程队列rq的进程控制块指针的队列元素删除，并把表示就绪进程个数的proc\_num减一。

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 --;

}

RR\_pick\_next的函数实现如下表所示。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。

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;

}

RR\_proc\_tick的函数实现如下表所示。即每次timer到时后，trap函数将会间接调用此函数来把当前执行进程的时间片time\_slice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。

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;

}

}

练习2: 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，即用default\_sched\_stride\_c覆盖default\_sched.c。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。 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 location1 strid-shed paper location2 也可GOOGLE “Stride Scheduling” 来查找相关资料 执行：make grade。如果所显示的应用程序检测都输出ok，则基本正确。如果只是priority.c过不去，可执行 make run- priority 命令来单独调试它。大致执行结果可看附录。（ 使用的是 qemu-1.0.1 ）。 请在实验报告中简要说明你的设计实现过程。

首先，根据实验指导书的要求，先用default\_sched\_stride\_c覆盖default\_sched.c，即覆盖掉Round Robin调度算法的实现。

覆盖掉之后需要在该框架上实现Stride Scheduling调度算法。

关于Stride Scheduling调度算法，经过查阅资料和实验指导书，我们可以简单的把思想归结如下：

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最小的进程

首先是初始化函数stride\_init。

开始初始化运行队列，并初始化当前的运行队，然后设置当前运行队列内进程数目为0。

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;

}

然后是入队函数stride\_enqueue，根据之前对该调度算法的分析，这里函数主要是初始化刚进入运行队列的进程 proc 的stride属性，然后比较队头元素与当前进程的步数大小，选择步数最小的运行，即将其插入放入运行队列中去，这里并未放置在队列头部。最后初始化时间片，然后将运行队列进程数目加一。

static void

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

/\* LAB6: YOUR CODE \*/

#if USE\_SKEW\_HEAP

rq->lab6\_run\_pool = //在使用优先队列的实现中表示当前优先队列的头元素

skew\_heap\_insert(rq->lab6\_run\_pool, &(proc->lab6\_run\_pool), proc\_stride\_comp\_f);//比较队头元素与当前进程的步数大小，选择步数最小的运行

#else

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

list\_add\_before(&(rq->run\_list), &(proc->run\_link));//将 proc插入放入运行队列中去

#endif

if (proc->time\_slice == 0 || proc->time\_slice > rq->max\_time\_slice) {//初始化时间片

proc->time\_slice = rq->max\_time\_slice;

}

proc->rq = rq;

rq->proc\_num ++;

}

然后是出队函数stride\_dequeue，即完成将一个进程从队列中移除的功能，这里使用了优先队列。最后运行队列数目减一。

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);// 在斜堆中删除相应元素

#else

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

list\_del\_init(&(proc->run\_link));// 从运行队列中删除相应元素

#endif

rq->proc\_num --;

}

接下来就是进程的调度函数stride\_pick\_next，观察代码，它的核心是先扫描整个运行队列，返回其中stride值最小的对应进程，然后更新对应进程的stride值，将步长设置为优先级的倒数，如果为0则设置为最大的步长。

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

//更新对应进程的stride值

if (p->lab6\_priority == 0)//优先级设置

p->lab6\_stride += BIG\_STRIDE;//步长为0则设置为最大步长保持相减的有效性

else p->lab6\_stride += BIG\_STRIDE / p->lab6\_priority;//步长设置为优先级的倒数

return p;

}

函数stride\_proc\_tick的主要工作是检测当前进程的时间片是否已经用完。如果时间片已经用完,就会按照正确的流程进行进程的切换工作。这里和之前实现的Round Robin调度算法一样，所采用的思想也是一致的

优先队列比较函数proc\_stride\_comp\_f的实现，主要利用思路是通过相减之后的值，进行判断大小

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;//步数相减，通过正负比较大小关系

if (c > 0) return 1;

else if (c == 0) return 0;

else return -1;

}



