**实验三、调度器**

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。

1、

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;

proc->run\_link.prev = proc->run\_link.next = NULL;

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;

}

2、

ticks ++;

assert(current != NULL);

run\_timer\_list();

break;

练习1: 使用 Round Robin 调度算法（不需要编码）

1、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。

函数指针的用法：

RR\_init完成了对进程队列的初始化。

RR\_enqueue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。

RR\_dequeue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。

RR\_pick\_next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。

RR\_proc\_tick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。

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，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。

2、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

假设进程一共有4个调度优先级，分别为0、1、2、3，其中0位最高优先级，3位最低优先级。为了支持4个不同的优先级，在运行队列中开4个队列，分别命名为rq -> run\_list[0..3]。除此之外，在proc\_struct中加入priority成员表示该进程现在所处的优先级，初始化为0。

1、MLFQ\_init：进程队列初始化，和RR算法实现一样，不同之处在于需要初始化4个队列，分别对应0、1、2、3。

2、MLFQ\_enqueue(struct run\_queue \*rq, struct proc\_struct \*proc)：判断proc进程的时间片proc -> time\_slice是否为0，如果为0，则proc -> priority += 1，否则不变。根据proc加入到对应优先级的列表中去。时间片的长度也和优先级有关，低优先级的时间片长度设置为高优先级的两倍。

3、MLFQ\_dequeue(struct run\_queue \*rq, struct proc\_struct \*proc)：将proc进程从相应的优先级运行队列中删除。

4、MLFQ\_pick\_next(struct run\_queue \*rq)：为了避免优先级较低的进程出现饥饿现象，对每个优先级设置一定的选中概率，高优先级是低优先级选中概率的两倍，然后选出一个优先级，找到这个优先级中的第一个进程返回。

5、MLFQ\_proc\_tick(struct run\_queue \*rq, struct proc\_struct \*proc)：和RR算法相似。

练习2: 实现 Stride Scheduling 调度算法（需要编码）

1、简要说明设计实现过程

（1）比较器定义

#define BIG\_STRIDE 0x7FFFFFFF

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;

}

（2）运行队列初始化

static void

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

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

rq->lab6\_run\_pool = NULL;

rq->proc\_num = 0;

}

（3）入队操作

static void

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

#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 ++;

}

（4）出队操作

static void

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

#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 --;

}

（5）选择进程调度

static struct proc\_struct \*

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

#if USE\_SKEW\_HEAP

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

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

}

（6）时间片部分

static void

stride\_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;

}

}

（7）定义一个c语言类的实现，提供调度算法的切换接口

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,

};

结果：

