练习1: 使用 Round Robin 调度算法（不需要编码）

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。

Round Robin 调度算法的主要实现在 default\_sched.c 之中，源码如下：

/\*

file\_path = kern/schedule/default\_sched.c

\*/

//RR\_init函数：这个函数被封装为 sched\_init 函数，用于调度算法的初始化，使用grep命令可以知道，该函数仅在 ucore 入口的 init.c 里面被调用进行初始化

static void RR\_init(struct run\_queue \*rq) { //初始化进程队列

list\_init(&(rq->run\_list));//初始化运行队列

rq->proc\_num = 0;//初始化进程数为 0

}

//RR\_enqueue函数：该函数的功能为将指定的进程的状态置成 RUNNABLE，并且放入调用算法中的可执行队列中，被封装成 sched\_class\_enqueue 函数，可以发现这个函数仅在 wakeup\_proc 和 schedule 函数中被调用，前者为将某个不是 RUNNABLE 的进程加入可执行队列，而后者是将正在执行的进程换出到可执行队列中去

static void RR\_enqueue(struct run\_queue \*rq, struct proc\_struct \*proc) {//将进程加入就绪队列

assert(list\_empty(&(proc->run\_link)));//进程控制块指针非空

list\_add\_before(&(rq->run\_list), &(proc->run\_link));//把进程的进程控制块指针放入到 rq 队列末尾

if (proc->time\_slice == 0 || proc->time\_slice > rq->max\_time\_slice) {//进程控制块的时间片为 0 或者进程的时间片大于分配给进程的最大时间片

proc->time\_slice = rq->max\_time\_slice;//修改时间片

}

proc->rq = rq;//加入进程池

rq->proc\_num ++;//就绪进程数加一

}

//RR\_dequeue 函数：该函数的功能为将某个在队列中的进程取出，其封装函数 sched\_class\_dequeue 仅在 schedule 中被调用，表示将调度算法选择的进程从等待的可执行的进程队列中取出进行执行

static void RR\_dequeue(struct run\_queue \*rq, struct proc\_struct \*proc) {//将进程从就绪队列中移除

assert(!list\_empty(&(proc->run\_link)) && proc->rq == rq);//进程控制块指针非空并且进程在就绪队列中

list\_del\_init(&(proc->run\_link));//将进程控制块指针从就绪队列中删除

rq->proc\_num --;//就绪进程数减一

}

//RR\_pick\_next 函数：该函数的封装函数同样仅在 schedule 中被调用，功能为选择要执行的下个进程

static struct proc\_struct \*RR\_pick\_next(struct run\_queue \*rq) {//选择下一调度进程

list\_entry\_t \*le = list\_next(&(rq->run\_list));//选取就绪进程队列 rq 中的队头队列元素

if (le != &(rq->run\_list)) {//取得就绪进程

return le2proc(le, run\_link);//返回进程控制块指针

}

return NULL;

}

//RR\_proc\_tick 函数：该函数表示每次时钟中断的时候应当调用的调度算法的功能，仅在进行时间中断的 ISR 中调用

static void RR\_proc\_tick(struct run\_queue \*rq, struct proc\_struct \*proc) {//时间片

if (proc->time\_slice > 0) {//到达时间片

proc->time\_slice --;//执行进程的时间片 time\_slice 减一

}

if (proc->time\_slice == 0) {//时间片为 0

proc->need\_resched = 1;//设置此进程成员变量 need\_resched 标识为 1，进程需要调度

}

}

//sched\_class 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 = "RR\_scheduler",

.init = RR\_init,

.enqueue = RR\_enqueue,

.dequeue = RR\_dequeue,

.pick\_next = RR\_pick\_next,

.proc\_tick = RR\_proc\_tick,

};

现在我们来逐个函数的分析，从而了解 Round Robin 调度算法的原理。

首先是 RR\_init 函数，函数完成了对进程队列的初始化。

//RR\_init函数：这个函数被封装为 sched\_init 函数，用于调度算法的初始化，使用grep命令可以知道，该函数仅在 ucore 入口的 init.c 里面被调用进行初始化

static void RR\_init(struct run\_queue \*rq) { //初始化进程队列

list\_init(&(rq->run\_list));//初始化运行队列

rq->proc\_num = 0;//初始化进程数为 0

}

其中的 run\_queue 结构体如下：

struct run\_queue {

list\_entry\_t run\_list;//其运行队列的哨兵结构，可以看作是队列头和尾

unsigned int proc\_num;//内部进程总数

int max\_time\_slice;//每个进程一轮占用的最多时间片

// For LAB6 ONLY

skew\_heap\_entry\_t \*lab6\_run\_pool;//优先队列形式的进程容器

};

而 run\_queue 结构体中的 skew\_heap\_entry 结构体如下：

struct skew\_heap\_entry {

struct skew\_heap\_entry \*parent, \*left, \*right;//树形结构的进程容器

};

typedef struct skew\_heap\_entry skew\_heap\_entry\_t;

然后是 RR\_enqueue 函数，首先，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\_enqueue函数：该函数的功能为将指定的进程的状态置成 RUNNABLE，并且放入调用算法中的可执行队列中，被封装成 sched\_class\_enqueue 函数，可以发现这个函数仅在 wakeup\_proc 和 schedule 函数中被调用，前者为将某个不是 RUNNABLE 的进程加入可执行队列，而后者是将正在执行的进程换出到可执行队列中去

static void RR\_enqueue(struct run\_queue \*rq, struct proc\_struct \*proc) {//将进程加入就绪队列

assert(list\_empty(&(proc->run\_link)));//进程控制块指针非空

list\_add\_before(&(rq->run\_list), &(proc->run\_link));//把进程的进程控制块指针放入到 rq 队列末尾

if (proc->time\_slice == 0 || proc->time\_slice > rq->max\_time\_slice) {//进程控制块的时间片为 0 或者进程的时间片大于分配给进程的最大时间片

proc->time\_slice = rq->max\_time\_slice;//修改时间片

}

proc->rq = rq;//加入进程池

rq->proc\_num ++;//就绪进程数加一

}

然后是 RR\_dequeue 函数，它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\_dequeue 函数：该函数的功能为将某个在队列中的进程取出，其封装函数 sched\_class\_dequeue 仅在 schedule 中被调用，表示将调度算法选择的进程从等待的可执行的进程队列中取出进行执行

static void RR\_dequeue(struct run\_queue \*rq, struct proc\_struct \*proc) {//将进程从就绪队列中移除

assert(!list\_empty(&(proc->run\_link)) && proc->rq == rq);//进程控制块指针非空并且进程在就绪队列中

list\_del\_init(&(proc->run\_link));//将进程控制块指针从就绪队列中删除

rq->proc\_num --;//就绪进程数减一

}

接下来是 RR\_pick\_next 函数，即选取函数。它选取就绪进程队列 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\_pick\_next 函数：该函数的封装函数同样仅在 schedule 中被调用，功能为选择要执行的下个进程

static struct proc\_struct \*RR\_pick\_next(struct run\_queue \*rq) {//选择下一调度进程

list\_entry\_t \*le = list\_next(&(rq->run\_list));//选取就绪进程队列 rq 中的队头队列元素

if (le != &(rq->run\_list)) {//取得就绪进程

return le2proc(le, run\_link);//返回进程控制块指针

}

return NULL;

}

最后是 RR\_proc\_tick，它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。

//RR\_proc\_tick 函数：该函数表示每次时钟中断的时候应当调用的调度算法的功能，仅在进行时间中断的 ISR 中调用

static void RR\_proc\_tick(struct run\_queue \*rq, struct proc\_struct \*proc) {//时间片

if (proc->time\_slice > 0) {//到达时间片

proc->time\_slice --;//执行进程的时间片 time\_slice 减一

}

if (proc->time\_slice == 0) {//时间片为 0

proc->need\_resched = 1;//设置此进程成员变量 need\_resched 标识为 1，进程需要调度

}

}

sched\_class 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 = "RR\_scheduler",

.init = RR\_init,

.enqueue = RR\_enqueue,

.dequeue = RR\_dequeue,

.pick\_next = RR\_pick\_next,

.proc\_tick = RR\_proc\_tick,

};

请理解并分析 sched\_calss 中各个函数指针的用法，并结合 Round Robin 调度算法描述 ucore 的调度执行过程;

首先我们可以查看一下 sched\_class 类中的内容：

struct sched\_class {

const char \*name;// 调度器的名字

void (\*init) (struct run\_queue \*rq);// 初始化运行队列

void (\*enqueue) (struct run\_queue \*rq, struct proc\_struct \*p);// 将进程 p 插入队列 rq

void (\*dequeue) (struct run\_queue \*rq, struct proc\_struct \*p);// 将进程 p 从队列 rq 中删除

struct proc\_struct\* (\*pick\_next) (struct run\_queue \*rq);// 返回运行队列中下一个可执行的进程

void (\*proc\_tick)(struct run\_queue\* rq, struct proc\_struct\* p);// timetick 处理函数

};

描述 ucore 调度执行过程：

在ucore中调用调度器的主体函数（不包括 init，proc\_tick）的代码仅存在在 wakeup\_proc 和 schedule，前者的作用在于将某一个指定进程放入可执行进程队列中，后者在于将当前执行的进程放入可执行队列中，然后将队列中选择的下一个执行的进程取出执行；

当需要将某一个进程加入就绪进程队列中，则需要将这个进程的能够使用的时间片进行初始化，然后将其插入到使用链表组织的队列的对尾；当需要将某一个进程从就绪队列中取出的时候，只需要将其直接删除即可；

当需要取出执行的下一个进程的时候，只需要将就绪队列的队头取出即可；

每当出现一个时钟中断，则会将当前执行的进程的剩余可执行时间减 1，一旦减到了 0，则将其标记为可以被调度的，这样在 ISR 中的后续部分就会调用 schedule 函数将这个进程切换出去；

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;

设计如下：

在 proc\_struct 中添加总共 N 个多级反馈队列的入口，每个队列都有着各自的优先级，编号越大的队列优先级约低，并且优先级越低的队列上时间片的长度越大，为其上一个优先级队列的两倍；并且在 PCB 中记录当前进程所处的队列的优先级；

处理调度算法初始化的时候需要同时对 N 个队列进行初始化；

在处理将进程加入到就绪进程集合的时候，观察这个进程的时间片有没有使用完，如果使用完了，就将所在队列的优先级调低，加入到优先级低 1 级的队列中去，如果没有使用完时间片，则加入到当前优先级的队列中去；

在同一个优先级的队列内使用时间片轮转算法；

在选择下一个执行的进程的时候，有限考虑高优先级的队列中是否存在任务，如果不存在才转而寻找较低优先级的队列；（有可能导致饥饿）

从就绪进程集合中删除某一个进程就只需要在对应队列中删除即可；

处理时间中断的函数不需要改变；