#include <stdio.h>  
#include <stdlib.h>  
#define N 5  
#include <string.h>  
enum State { Ready, Run, Block, Finish };  
struct PCB //数据结构  
{  
 int id;  
 int priority;  
 int cputime;  
 int alltime;  
 int startblock;//在cpu中能运行的时间  
 int blocktime;//阻塞了多久后，进入就绪队列  
 int waittime;//\*在阻塞队列中等待的时间  
 State state;  
 PCB\* next;  
 PCB\* pre;  
}\*ready\_pro, \* block\_pro, \* ready\_tail, \* block\_tail;//就绪,阻塞队列,记录其头指针,  
int ready\_num, block\_num;//就绪，运行，阻塞队列中进程的数量。  
PCB record[N];  
//原始数据  
int id[] = { 0,1,2,3,4 };  
int priority[] = { 9,38,30,29,0 };  
int cputime[] = { 0,0,0,0,0 };  
int alltime[] = { 3,3,6,3,4 };  
int startblock[] = { 2,-1,-1,-1,-1 };//在cpu中能运行的时间  
int blocktime[] = { 3,0,0,0,0 }; //阻塞了多久后，进入就绪队列  
State state[] = { Ready,Ready,Ready,Ready,Ready };  
//将新的进程插入到就绪队列中  
//用插入排序的方法，保证了就绪队列的值是按照大小排序的，  
void ready\_push(PCB\* pro)  
{  
 pro->state = Ready;  
 if (!ready\_num)//如果是就绪队列为空，则放在头指针的位置,尾指针也赋值为pro  
 {  
 ready\_pro = pro;  
 ready\_tail = pro;  
 }  
 else  
 {  
 PCB\* this\_pro = ready\_pro;  
 while (this\_pro != NULL)//插入的进程和就绪队列中的进程进行比较，插入到合适的位置  
 {  
 if (this\_pro->priority > pro->priority)//如果当前进程的优先级高则和下一个进程比较  
 this\_pro = this\_pro->next;  
 else//如果当前进程的优先级比pro低，则插在当前进程前面  
  
 {  
 if (this\_pro == ready\_pro)//如果是头指针则要改ready\_pro的值  
 {  
 pro->next = this\_pro;  
 this\_pro->pre = pro;  
 ready\_pro = pro;  
 }  
 else//如果不是头指针则，插入队列中  
 {  
 pro->next = this\_pro;  
 pro->pre = this\_pro->pre;  
 this\_pro->pre->next = pro;  
 this\_pro->pre = pro;  
 }  
 break;//插入后退出循环  
 }  
 }  
 if (this\_pro == NULL)//如果pro的优先级最小则插到队伍的最后  
 {  
 ready\_tail->next = pro;  
 pro->pre = ready\_tail;  
 ready\_tail = pro;//把ready\_tail赋值为pro；  
 }  
 }  
 ready\_num++;  
}  
PCB\* ready\_pop()//就绪进程出队,将优先级最高的进程pop出去  
{  
 PCB\* max\_pro;  
 //优先级最高的进程就是第一个进程  
 max\_pro = ready\_pro;  
 ready\_pro = ready\_pro->next;  
 if (ready\_pro == NULL)//如果就绪队列出队后为空，则尾指针也需要赋值为NULL  
 ready\_tail = NULL;  
 else//否则就绪队列头指针的前驱为空  
 {  
 ready\_pro->pre = NULL;  
 max\_pro->next = NULL;  
 }  
 ready\_num--;  
 return max\_pro;  
}  
void ready\_updata()//每个时刻就绪队列中进程优先级的更新  
{  
 PCB\* pro = ready\_pro;  
 while (pro != NULL)  
 {  
 pro->priority++;  
 pro = pro->next;  
 }  
}  
void ready\_work()  
{  
 printf("READY\_QUEUE:");  
 PCB\* pro = ready\_pro;  
 if (pro == NULL)  
 printf("NULL");  
 while (pro != NULL)  
 {  
 printf("->%d", pro->id);  
 record[pro->id].id = pro->id;  
 record[pro->id].priority = pro->priority;  
 record[pro->id].cputime = pro->cputime;  
 record[pro->id].alltime = pro->alltime;  
 record[pro->id].startblock = pro->startblock;  
 record[pro->id].blocktime = pro->blocktime;  
 record[pro->id].state = pro->state;  
 pro = pro->next;  
 }  
 printf("\n");  
}  
void block\_push(PCB\* pro)//进入阻塞队列  
{  
 block\_num++;  
 pro->state = Block;  
 if (block\_pro == NULL)//如果是阻塞队列空，则加入放在第一位  
 {  
 pro->next = NULL;  
 pro->pre = NULL;  
 block\_pro = pro;  
 block\_tail = pro;  
 }  
 else//如果非空，插入排序，则按照所需要的时间长短放在阻塞队列中，时间越短越靠前，  
  
 {  
 PCB\* this\_pro = block\_pro;  
 while (this\_pro != NULL)  
 {  
 if (this\_pro->blocktime < pro->blocktime)//如果当前进程的阻塞时间小于pro则pro与下一个比较  
 this\_pro = this\_pro->next;  
 else  
 {  
 if (this\_pro == block\_pro)//如果是头指针则要改block\_pro的值  
 {  
 pro->next = this\_pro;  
 pro->pre = NULL;  
 this\_pro->pre = pro;  
 block\_pro = pro;  
 }  
 else//如果不是头指针则，插入队列中  
 {  
 pro->next = this\_pro;  
 pro->pre = this\_pro->pre;  
 this\_pro->pre->next = pro;  
 this\_pro->pre = pro;  
 }  
 break;//插入后退出循环  
 }  
 if (this\_pro == NULL)//如果在最后则修改尾指针  
 {  
 block\_tail->next = pro;  
 pro->pre = block\_tail;  
 block\_tail = pro;  
 }  
 }  
 }  
}  
PCB\* block\_pop()//取阻塞队列的队首  
{  
 PCB\* pro;  
 pro = block\_pro;  
 block\_num--;  
 if (!block\_num)//只有一个值  
 {  
 block\_pro = NULL;  
 block\_tail = NULL;  
 }  
 else//不止一个值  
 {  
 block\_pro = block\_pro->next;  
 block\_pro->pre = NULL;  
 }  
 pro->next = NULL;  
 return pro;  
}  
void block\_updata()  
{  
 PCB\* pro = block\_pro;  
 while (pro != NULL)  
 {  
 pro->waittime++;  
 //如果等待时间大于阻塞时间，则将进程放到就绪队列中  
 if (pro->waittime > pro->blocktime)//如果等待时间大于阻塞时间，则将进程放到就绪队列中  
 {  
 PCB\* process = pro;  
 process->priority--;//先运行阻塞队列，需要减去1，在就绪队列中加回  
 if (process == block\_pro)//如果是在队首  
 {  
 process = block\_pop();//取出队首元素  
 pro = block\_pro;//下一个需要判断的pro就等于队首元素  
 }  
 else if(process == block\_tail)//如果在队尾  
 {  
 block\_tail = block\_tail->pre;  
 process->pre = NULL;  
 process->waittime = 0;  
 pro = NULL;  
 }  
 else//如果在队伍中间  
 {  
 pro->pre->next = pro->next;  
 pro->next->pre = pro->pre;  
 pro = pro->next;  
 process->pre = NULL;  
 process->next = NULL;  
 process->waittime = 0;  
 }  
 ready\_push(process);  
 }  
 else  
 pro = pro->next;  
 }  
}  
void block\_work()//打印阻塞队列的信息  
{  
 printf("BLOCK\_QUEUE:");  
 PCB\* pro = block\_pro;  
 if (pro == NULL)  
 printf("NULL");  
 while (pro != NULL)  
 {  
 printf("->%d", pro->id);  
 record[pro->id].id = pro->id;  
 record[pro->id].priority = pro->priority;  
 record[pro->id].cputime = pro->cputime;  
 record[pro->id].alltime = pro->alltime;  
 record[pro->id].startblock = pro->startblock;  
 record[pro->id].blocktime = pro->blocktime;  
 record[pro->id].state = pro->state;  
 pro = pro->next;  
 }  
 printf("\n");  
}  
void cpu\_work(PCB\* cpu\_pro)  
{  
 if (cpu\_pro == NULL)  
 {  
 printf("RUNNINGPROG:NULL\n");  
 return;  
 }  
 if (!cpu\_pro->alltime)  
 return;  
 cpu\_pro->priority -= 3;  
 cpu\_pro->alltime--;  
 cpu\_pro->cputime++;  
 printf("RUNNINGPROG:%d\n", cpu\_pro->id);  
 record[cpu\_pro->id].id = cpu\_pro->id;  
 record[cpu\_pro->id].priority = cpu\_pro->priority;  
 record[cpu\_pro->id].cputime = cpu\_pro->cputime;  
 record[cpu\_pro->id].alltime = cpu\_pro->alltime;  
 record[cpu\_pro->id].startblock = cpu\_pro->startblock;  
 record[cpu\_pro->id].blocktime = cpu\_pro->blocktime;  
 record[cpu\_pro->id].state = cpu\_pro->state;  
}  
void init()//初始化  
{  
 //初始化就绪队列，运行队列和阻塞队列  
 block\_pro = NULL;  
 block\_tail = NULL;  
 block\_num = 0;  
 ready\_num = 0;  
 ready\_pro = NULL;  
 ready\_tail = NULL;  
 int i;  
 PCB\* pro;  
 for (i = 0; i < N; i++)  
 {  
 pro = (PCB\*)malloc(sizeof(PCB));  
 pro->id = id[i];  
 pro->priority = priority[i];  
 pro->cputime = cputime[i];  
 pro->alltime = alltime[i];  
 pro->startblock = startblock[i];  
 pro->blocktime = blocktime[i];  
 pro->waittime = 0;  
 pro->pre = NULL;  
 pro->next = NULL;  
 ready\_push(pro);  
 }  
}  
void print\_table()  
{  
 int i;  
 printf("===========================================================\n");  
 printf("ID \t%d\t%d\t%d\t%d\t%d\n",  
 record[0].id, record[1].id, record[2].id, record[3].id, record[4].id);  
 printf("PRIORITY(优先) \t%d\t%d\t%d\t%d\t%d\n",  
 record[0].priority, record[1].priority, record[2].priority, record[3].priority, record[4].priority);  
 printf("CPUTIME(使用时间） \t%d\t%d\t%d\t%d\t%d\n",  
 record[0].cputime, record[1].cputime, record[2].cputime, record[3].cputime, record[4].cputime);  
 printf("ALLTIME \t%d\t%d \t%d \t%d\t%d\n",  
 record[0].alltime, record[1].alltime, record[2].alltime, record[3].alltime, record[4].alltime);  
 printf("STARTBLOCK(在cpu中能运行的时间)\t%d\t%d\t%d \t%d\t%d\n",  
 record[0].startblock, record[1].startblock, record[2].startblock, record[3].startblock, record[4].startblock);  
 printf("BLOCKTIME(阻塞了多久后，进入就绪队列)\t%d \t%d \t%d\t%d \t%d \n",  
 record[0].blocktime, record[1].blocktime, record[2].blocktime, record[3].blocktime, record[4].blocktime);  
 printf("STATE \t");  
 for (i = 0; i < N; i++)  
 {  
 if (record[i].state == Run)  
 printf("RUN \t");  
 else if(record[i].state == Ready)  
 printf("READY \t");  
 else if(record[i].state == Finish)  
 printf("FINISH\t");  
 else if(record[i].state == Block)  
 printf("BLOCK \t");  
 }  
 printf("\n");  
}  
int main()  
{  
 init();//初始化；  
 PCB\* cpu\_pro = NULL;  
 int cpu\_num = 0;  
 int times = 0;  
 int i = 0;  
 printf("第%d个时间片后:\n", times);  
 cpu\_work(cpu\_pro);  
 ready\_work();  
 block\_work();  
 print\_table();  
 while (1)  
 {  
 //如果三个队列中都没有进程则推出调用  
 if (!ready\_num && !block\_num && !cpu\_num)  
 break;  
 if (!cpu\_num)//如果cpu空闲则调用一个就绪队列中的进程  
 {  
 if (ready\_num > 0)//如果就绪队列非空，则调出优先级最高的进程  
 {  
 cpu\_pro = ready\_pop();  
 }  
 else//如果就绪队列为空，则调出阻塞队列的第一个值  
 {  
 cpu\_pro = block\_pop();  
 }  
 cpu\_num = 1;  
 cpu\_pro->state = Run;  
 cpu\_pro->cputime = 0;//开始运行的时间为0；  
 }  
 times++;  
 printf("\n第%d个时间片后:\n", times);  
 cpu\_work(cpu\_pro);  
 //如果alltime == 0，进程结束，释放内存  
 if (!cpu\_pro->alltime)  
 {  
 cpu\_pro->state = Finish;  
 record[cpu\_pro->id].id = cpu\_pro->id;  
 record[cpu\_pro->id].priority = cpu\_pro->priority;  
 record[cpu\_pro->id].cputime = cpu\_pro->cputime;  
 record[cpu\_pro->id].alltime = cpu\_pro->alltime;  
 record[cpu\_pro->id].startblock = cpu\_pro->startblock;  
 record[cpu\_pro->id].blocktime = cpu\_pro->blocktime;  
 record[cpu\_pro->id].state = cpu\_pro->state;  
 free(cpu\_pro);  
 cpu\_num = 0;  
 }  
 //更新阻塞和就绪队列中的信息  
 block\_updata();  
 ready\_updata();  
 //显示就绪、阻塞队列的信息  
 ready\_work();  
 block\_work();  
 print\_table();  
 //如果在cpu上运行时间达到了startblock并且alltime还不为0，则放到阻塞队列中，  
 if (cpu\_pro->cputime == cpu\_pro->startblock && cpu\_pro->alltime > 0)  
 {  
 //在放入阻塞队列中；  
 cpu\_pro->cputime = 0;  
 cpu\_pro->waittime = 0;  
 block\_push(cpu\_pro);  
 cpu\_num = 0;  
 }  
 }  
 printf("\n模拟进程调度算法结束!\n");  
 return 0;  
}