**北京邮电大学软件学院**

**2017-2018学年第1学期实验报告**

**课程名称： 操作系统**

**实验名称： 实验四 进程调度**

**实验完成人：**

**姓名：**\_ 李博\_ \_**学号：**\_\_2016212003\_\_\_**成绩：**\_\_\_\_\_\_\_\_

**姓名：**\_王子妍\_\_**学号：**\_\_2016211980\_\_**成绩：**\_\_\_\_\_\_\_\_

**姓名：**\_\_\_朱岩 \_\_**学号：**\_\_2016522039\_**成绩：**\_\_\_\_\_\_\_\_

**指导教师：**\_\_\_\_\_\_\_\_\_\_\_\_\_陈晋鹏\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

**日 期： 2018 年 5 月 16 日**

1. **实验目的**

1. 理解 Linux 管理进程所用到的数据结构。

2. 理解 Linux 的进程调度算法的处理逻辑及其实现所用到的数据结构。

1. **实验内容**

1. 通过查阅参考书或者上网找资料，熟悉/usr/src/linux（注意：这里最后一级目录名可能是个含有具体内核版本号和“linux”字符串的名字）下各子目录的内容，即所含 Linux源代码的情况。

2. 分析 Linux 进程调度有关函数的源代码，主要是 schedule()函数，并且要对它们引用的头文件等一并分析。

3. 实现 Linux 的进程调度算法及理解其实现所用的主要数据结构。

1. **实验环境**

在VMware虚拟机下安装Ubuntu系统作为Linux实验环境。采用以Vi +GCC+GDB为开发环境的C语言实验环境。其中Vi 作为编辑器，GCC作为编译器，GDB作为调试器。

1. **实验过程描述**
2. **通过查阅参考书或者上网找资料，熟悉/usr/src/linux（注意：这里最后一级目录名可能是个含有具体内核版本号和“linux”字符串的名字）下各子目录的内容，即所含 Linux源代码的情况。**

实验思路：

先在网上搜索了关于linux源码的一些资料，学习并理解。

获得结果：

Linux内核源代码位于/usr/src/linux目录下，每一个目录或子目录可以看作一个模块，其目录之间的连线表示“子目录或子模块”的关系。下面是对每一个目录的简单描述。

Arch

目录包括了所有和体系结构相关的核心代码。它下面的每一个子目录都代表一种Linux支持的体系结构，例如X86就是Intel CPU及与之相兼容体系结构的子目录。Arch目录下的子目录包括：/arch/kernel、/arch/mm、/arch/include、/arch/boot、/arch/configs。其中/arch/kernel目录包含了与CPU相关的中断和SMP等信息，/arch/boot包含了系统启动的相关信息和make之后生成的启动镜像文件bzImage。

Copying

目录下是GPL版权申明。对具有GPL版权的源代码改动而形成的程序，或使用GPL工具产生的程序，具有使用GPL发表的义务，如公开源代码。

Credits

目录下是光荣榜。对Linux做出过很大贡献的一些人的信息。

Documentation

目录下是一些文档，linux-3.0.4版本该目录下有217个文件，文件00-INDEX对该目录下的所有文件进行了简要说明，其他文件分别对linux操作系统的各个部分进行了说明。IRQ.txt描述了什么是中断及中断在系统中的枚举定义和头文件的位置；Changes描述了当前版本的系统对各个软件版本的需求，linux 3.0.4要求make的版本为3.8.0，grub的版本为0.93。

Drivers

目录中是系统中所有的设备驱动程序。它又进一步划分成几类设备驱动，每一种有对应的子目录，如声卡的驱动对应于drivers/sound; block 下为块设备驱动程序，比如ide（ide.c）。如果你希望查看所有可能包含文件系统的设备是如何初始化的，你可以看drivers/block/genhd.c中的device\_setup()。它不仅初始化硬盘，也初始化，因为安装nfs文件系统的时候需要网络其他: 如, Lib放置核心的库代码; Net,核心与网络相关的代码; Ipc,这个目录包含核心的进程间通讯的代码; Fs,所有的文件系统代码和各种类型的文件操作代码，它的每一个子目录支持一个文件系统，例如fat和ext2。

Fs

目录存放Linux支持的文件系统代码和各种类型的文件操作代码。每一个子目录支持一个文件系统，包括我们常用的文件系统类型：Ext2、Ext3、Ext4、 Proc、Fat和Nfs。其中Ext3文件系统对应的就是/fs/ext3目录。

Include

目录包括编译核心所需要的大部分头文件，例如与平台无关的头文件在include/linux子目录下，与 intel cpu相关的头文件在include/asm-i386子目录下,而include/scsi目录则是有关scsi设备的头文件目录。

Init

目录包含核心的初始化代码（不是系统的引导代码），有main.c和Version.c两个文件。这是研究核心如何工作的好起点。

Ipc

目录包含了核心进程间的通信代码。

Kernel

内核管理的核心代码，此目录下的文件实现了大多数linux系统的内核函数，其中最重要的文件当属sched.c；同时与处理器结构相关代码都放在archlib/目录下。

Maintainers

目录存放了维护人员列表，对当前版本的内核各部分都有谁负责。

Makefile

目录第一个Makefile文件。用来组织内核的各模块，记录了个模块间的相互这间的联系和依托关系，编译时使用；仔细阅读各子目录下的Makefile文件对弄清各个文件这间的联系和依托关系很有帮助。

Mm

目录包含了所有独立于 cpu 体系结构的内存管理代码，如页式存储管理内存的分配和释放等。与具体硬件体系结构相关的内存管理代码位于arch/\*/mm目录下，例如arch/i386/mm/Fault.c 。

Net

目录里是核心的网络部分代码，其每个子目录对应于网络的一个方面。

ReadMe

文件提供内核的各种编译方法；生成文件的查看方法，如 nm vmlinux | sort | less

Reporting-bugs

目录里是有关报告Bug 的一些内容

Rules.make

目录里是各种Makefilemake所使用的一些共同规则

Scripts

目录包含用于配置核心的脚本文件等。

一般在每个目录下都有一个.depend文件和一个Makefile文件。这两个文件都是编译时使用的辅助文件。仔细阅读这两个文件对弄清各个文件之间的联系和依托关系很有帮助。另外有的目录下还有Readme文件，它是对该目录下文件的一些说明，同样有利于对内核源码的理解。

隐藏文件：

.Config

Make oldconfig 和make defconfig会把生成的默认的configuration放到文件.config中。执行Make之后会根据Makefile上下层级依赖关系编译整个系统，.config作为编译各个模块的依据。

下面是linux0.11的内核简单目录结构：

**2. 分析 Linux 进程调度有关函数的源代码，主要是 schedule()函数，并且要对它们引用的头文件等一并分析。**

首先，先将linux中的源代码以及代码分析写下来，如下：

schedule();

struct task\_struct \*tsk = current; //current是当前进程

sched\_submit\_work(tsk); //避免死锁

\_\_schedule();//这就是调度的主函数了

static void \_\_sched \_\_schedule(void)

{

struct task\_struct \*prev, \*next;

unsigned long \*switch\_count;

struct rq \*rq;

int cpu;

need\_resched:

preempt\_disable(); //关闭内核抢占，关于内核抢占详见注释1

cpu = smp\_processor\_id();

rq = cpu\_rq(cpu); //跟当前进程相关的runqueue的信息被保存在rq中

rcu\_note\_context\_switch(cpu);

prev = rq->curr; //当前进程放入prev

schedule\_debug(prev);

if (sched\_feat(HRTICK))

hrtick\_clear(rq);

raw\_spin\_lock\_irq(&rq->lock);

switch\_count = &prev->nivcsw;

//如果内核态没有被抢占，并且内核抢占有效

if (prev->state && !(preempt\_count() & PREEMPT\_ACTIVE)) {

//如果当前进程有非阻塞等待信号，并且它的状态是TASK\_INTERRUPTIBLE

if (unlikely(signal\_pending\_state(prev->state, prev))) {

prev->state = TASK\_RUNNING; //将当前进程的状态设为：TASK\_RUNNING

} else {

deactivate\_task(rq, prev, DEQUEUE\_SLEEP);//将当前进程从runqueue(运行队列)中删除

prev->on\_rq = 0; //标识当前进程不在runqueue中

if (prev->flags & PF\_WQ\_WORKER) {

struct task\_struct \*to\_wakeup;

to\_wakeup = wq\_worker\_sleeping(prev, cpu);

if (to\_wakeup)

try\_to\_wake\_up\_local(to\_wakeup);

}

}

switch\_count = &prev->nvcsw;

}

pre\_schedule(rq, prev);

if (unlikely(!rq->nr\_running))//如果runqueue中没有正在运行的进程

idle\_balance(cpu, rq); //就会从其它CPU拉入进程

put\_prev\_task(rq, prev); //通知调度器，当前进程要被另一个进程取代，做好准备

next = pick\_next\_task(rq); //从runqueue中选择最适合的进程

clear\_tsk\_need\_resched(prev); //清除当前进程的重调度标识

rq->skip\_clock\_update = 0;

//当前进程与所选进程是否是同一进程，不属于同一进程才需要切换

if (likely(prev != next)) {

rq->nr\_switches++;

rq->curr = next; //所选进程代替当前进程

++\*switch\_count;

context\_switch(rq, prev, next); //负责底层上下文切换

cpu = smp\_processor\_id();

rq = cpu\_rq(cpu);

} else

raw\_spin\_unlock\_irq(&rq->lock); //如果不需要切换进程，则只需要解锁

post\_schedule(rq);

sched\_preempt\_enable\_no\_resched();

if (need\_resched())

goto need\_resched;

}

注释1：

内核抢占基础知识

1、内核抢占概念

当进程位于内核空间，有一个更高优先级的任务出现时，如果该内核支持抢占的话，则可以将当前任务挂起，执行更高优先级的任务！

2、用户抢占的概念

内核即将返回用户空间的时候，如果need resched标志被设置，会导致schedule()被调用，此时就会发生用户抢占。内核无论是在从中断处理程序还是在系统调用后返回，都会检查need resched标志。如果它被设置了，那么，内核会选择一个其他(更合适的)进程投入运行。

3、内核抢占好处

首先，这是实时系统所要求的。试想一下，如果硬件中断开启了一个实时进程，如果内核不支持抢占的话，被开启的实时进程就要等到当前进程执行完毕才能被调度，这就带来了延时，对实时性不好！如果内核支持抢占的话，就可以将当前进程挂起，来执行实时进程，这样对实时性有利！

4、什么情况下不能抢占内核

（1）内核正进行中断处理

（2）内核正在进行中断上下文的Bottom Half(中断的底半部)处理

（3）内核的代码段正持有spinlock自旋锁、writelock/readlock读写锁等锁，处干这些锁的保护状态中。

（4）内核正在执行调度程序Scheduler，这种情况正对应我们的schedule函数分析

（5）内核正在对每个CPU“私有”的数据结构操作

为保证Linux内核在以上情况下不会被抢占，抢占式内核使用了一个变量preempt\_count，称为内核抢占锁。这一变量被设置在进程的PCB结构task\_struct中。每当内核要进入以上几种状态时，变量preempt\_ count就加1，指示内核不允许抢占。每当内核从以上几种状态退出时，变量preempt\_ count就减1，同时进行可抢占的判断与调度。

**3. 实现 Linux 的进程调度算法及理解其实现所用的主要数据结构。**

**实验准备：**

**1.Linux进程状态的描述**

Linux将进程状态描述为如下五种：

TASK\_RUNNING：可运行状态。处于该状态的进程可以被调度执行而成为当前进程。

TASK\_INTERRUPTIBLE：可中断的睡眠状态。处于该状态的进程在所需资源有效时被唤醒，也可以通过信号或定时中断唤醒。

TASK\_UNINTERRUPTIBLE：不可中断的睡眠状态。处于该状态的进程仅当所需资源有效时被唤醒。

TASK\_ZOMBIE：僵死状态。表示进程结束且已释放资源，但其 task\_struct 仍未释放。

TASK\_STOPPED：暂停状态。处于该状态的进程通过其他进程的信号才能被唤醒。

**2.进程的虚拟地址空间**

调度方式Linux中的每个进程都分配有一个相对独立的虚拟地址空间。该虚存空间分为两部分：用户空间包含了进程本身的代码和数据；内核空间包含了操作系统的代码和数据。Linux采用“有条件的可剥夺”调度方式。对于普通进程，当其时间片结束时，调度程序挑选出下一个处于TASK\_RUNNING状态的进程作为当前进程（自愿调度）。对于实时进程，若其优先级足够高，则会从当前的运行进程中抢占CPU 成为新的当前进程（强制调度）。发生强制调度时，若进程在用户空间中运行，就会直接被剥夺CPU；若进程在内核空间中运行，即使迫切需要其放弃CPU，也仍要等到从它系统空间返回的前夕才被剥夺CPU。

**3.调度策略**

1）SCHED\_OTHER

SCHED\_OTHER 是面向普通进程的时间片轮转策略。采用该策略时，系统为处于TASK\_RUNNING状态的每个进程分配一个时间片。当时间片用完时，进程调度程序再选择下一个优先级相对较高的进程，并授予CPU 使用权。

2）SCHED\_FIFO

SCHED\_FIFO 策略适用于对响应时间要求比较高，运行所需时间比较短的实时进程。采用该策略时，各实时进程按其进入可运行队列的顺序依次获得 CPU。除了因等待某个事件主动放弃CPU，或者出现优先级更高的进程而剥夺其CPU 之外，该进程将一直占用CPU运行。

3）SCHED\_RR

SCHED\_RR 策略适用于对响应时间要求比较高，运行所需时间比较长的实时进程。采用该策略时，各实时进程按时间片轮流使用 CPU。当一个运行进程的时间片用完后，进程调度程序停止其运行并将其置于可运行队列的末尾。

**4.相关函数**

1）Schedule()函数

Schedule()函数首先对所有任务（进程）进行检测，唤醒任何一个得到信号的任务。具 体方法是针对任务数组中的每个任务，检查其报警定时值alarm。如果任务的alarm时间已经 过期（alarm<jiffies），则在它的信号位图中设置SIGALRM信号，然后清alarm值。jiffies是 系统从开机开始算起的滴答数（10ms/滴答），在sched..h 中定义。如果进程的信号位图中 除去被阻塞的信号外还有其他信号，并且任务处于可中断睡眠状态 （TASK\_INTERRUPTIBLE），则置任务为就绪状态（TASK\_RUNNING）。

2）sleep\_on()函数

sleep\_on()函数的主要功能是当一个进程（或任务）所请求的资源正忙或不在内存中时 暂时切换出去，放在等待队列中等待一段时间，当切换回来后再继续运行。放入等待队列的 方式是利用了函数中的tmp指针作为各个正在等待任务的联系。

3）wake\_up()函数 唤醒操作函数wake\_up()把正在等待可用资源的指定任务置为就绪状态。该函数是一个通用唤醒函数。在有些情况下，例如读取磁盘上的数据块，由于等待队列中的任何一个任务 都可能被先唤醒，因此还需要把被唤醒任务结构的指针置空。这样，在其后进入睡眠的进程 被唤醒而又重新执行sleep\_on()时，就无需唤醒该进程了。

**5.程序**

代码比较长，具体见附录1。

程序中的pcb结构为：

typedef struct node

{

char name[20]; /\*进程的名字\*/

int prio; /\*进程的优先级\*/

int round; /\*分配CPU的时间片\*/

int cputime; /\*CPU执行时间\*/

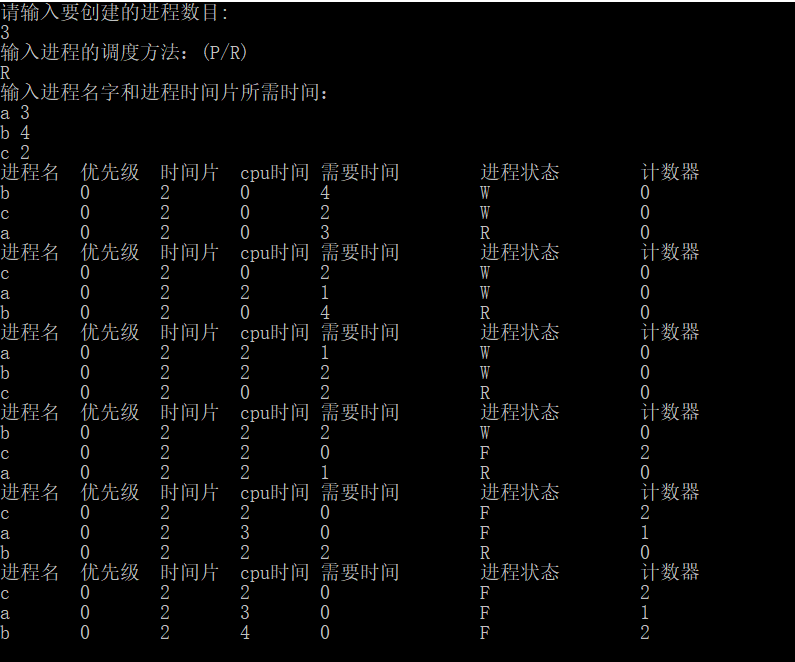
int needtime; /\*进程执行所需要的时间\*/

char state; /\*进程的状态，W--就绪态，R--执行态，F--完成态\*/

int count; /\*记录执行的次数\*/

struct node \*next; /\*链表指针\*/

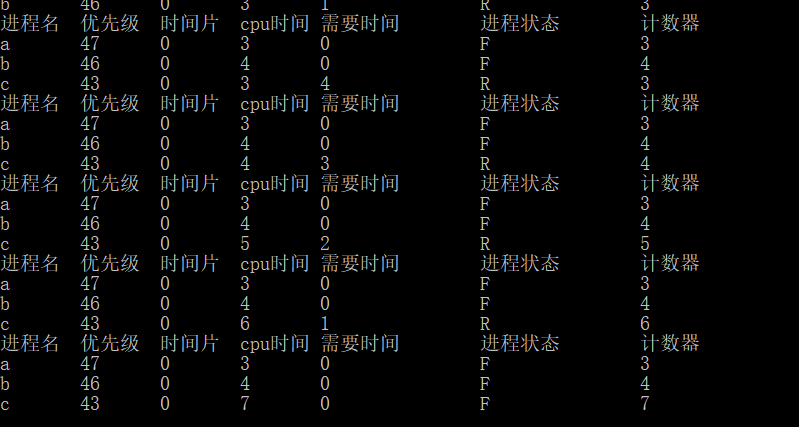
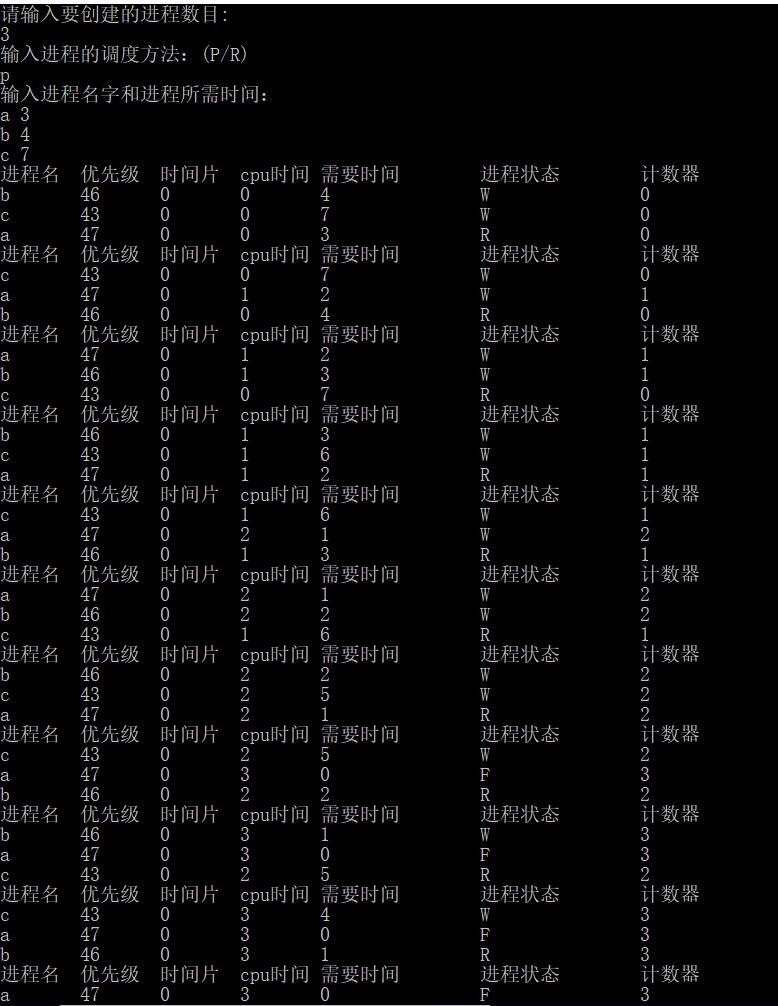
}PCB;



代码执行效果如下：

首先是对于RR时间片轮转的执行效果（可以从图片中展示的效果看到时间片轮转的效果）

下面是优先级算法的执行效果：



这个算法展示的步骤为：

1.输入进程数，选择p算法

2.创建3个进程，输入进程运行需要时间

3.优先级的定义方式为50减去运行所需时间

4．动态算法：进程每被执行一遍后，若未完成则优先级减3后加入就绪队列。若就绪队列中两个进程优先级一样，则后进入的在后。

1. **实验结果**

该实验已全部完成，代码部分见附录，实验效果方法以及截图见实验过程描述。

1. **附件**

**6.1 附件1：源代码**

#include <stdio.h>

#include <stdlib.h>

#include <string.h>

typedef struct node

{

char name[20]; /\*进程的名字\*/

int prio; /\*进程的优先级\*/

int round; /\*分配CPU的时间片\*/

int cputime; /\*CPU执行时间\*/

int needtime; /\*进程执行所需要的时间\*/

char state; /\*进程的状态，W--就绪态，R--执行态，F--完成态\*/

int count; /\*记录执行的次数\*/

struct node \*next; /\*链表指针\*/

}PCB;

PCB \*ready=NULL,\*run=NULL,\*finish=NULL; /\*定义三个队列，就绪队列，执行队列和完成队列\*/

int num;

void GetFirst(); /\*从就绪队列取得第一个节点\*/

void Output(); /\*输出队列信息\*/

void InsertPrio(PCB \*in); /\*创建优先级队列，规定优先数越小，优先级越高\*/

void InsertTime(PCB \*in); /\*时间片队列\*/

void InsertFinish(PCB \*in); /\*时间片队列\*/

void PrioCreate(); /\*优先级输入函数\*/

void TimeCreate(); /\*时间片输入函数\*/

void Priority(); /\*按照优先级调度\*/

void RoundRun(); /\*时间片轮转调度\*/

int main(void)

{

char chose;

printf("请输入要创建的进程数目:\n");

scanf("%d",&num);

getchar();

printf("输入进程的调度方法：(P/R)\n");

scanf("%c",&chose);

switch(chose)

{

case 'P':

case 'p':

PrioCreate();

Priority();

break;

case 'R':

case 'r':

TimeCreate();

RoundRun();

break;

default:break;

}

Output();

return 0;

}

void GetFirst() /\*取得第一个就绪队列节点\*/

{

run = ready;

if(ready!=NULL)

{

run ->state = 'R';

ready = ready ->next;

run ->next = NULL;

}

}

void Output() /\*输出队列信息\*/

{

PCB \*p;

/\*p = ready;\*/

printf("进程名\t优先级\t时间片\tcpu时间\t需要时间\t进程状态\t计数器\n");

p = ready;

while(p!=NULL)

{

printf("%s\t%d\t%d\t%d\t%d\t\t%c\t\t%d\n",p->name,p->prio,p->round,p->cputime,p->needtime,p->state,p->count);

p = p->next;

}

p = finish;

while(p!=NULL)

{

printf("%s\t%d\t%d\t%d\t%d\t\t%c\t\t%d\n",p->name,p->prio,p->round,p->cputime,p->needtime,p->state,p->count);

p = p->next;

}

p = run;

while(p!=NULL)

{

printf("%s\t%d\t%d\t%d\t%d\t\t%c\t\t%d\n",p->name,p->prio,p->round,p->cputime,p->needtime,p->state,p->count);

p = p->next;

}

}

void InsertPrio(PCB \*in) /\*创建优先级队列，规定优先数越小，优先级越低\*/

{

PCB \*fst,\*nxt;

fst = nxt = ready;

if(ready == NULL) /\*如果队列为空，则为第一个元素\*/

{

in->next = ready;

ready = in;

}

else /\*查到合适的位置进行插入\*/

{

if(in ->prio > fst ->prio) /\*比第一个还要大(大于等于)，则插入到队头\*/

{

in->next = ready;

ready = in;

}

else

{

while(fst->next != NULL) /\*移动指针查找第一个别它小的元素的位置进行插入\*/

{

nxt = fst;

fst = fst->next;

}

if(fst ->next == NULL) /\*已经搜索到队尾，则其优先级数最小，将其插入到队尾即可\*/

{

in ->next = fst ->next;

fst ->next = in;

}

else /\*插入到队列中\*/

{

nxt = in;

in ->next = fst;

}

}

}

}

void InsertTime(PCB \*in) /\*将进程插入到就绪队列尾部\*/

{

PCB \*fst;

fst = ready;

if(ready == NULL)

{

in->next = ready;

ready = in;

}

else

{

while(fst->next != NULL)

{

fst = fst->next;

}

in ->next = fst ->next;

fst ->next = in;

}

}

void InsertFinish(PCB \*in) /\*将进程插入到完成队列尾部\*/

{

PCB \*fst;

fst = finish;

if(finish == NULL)

{

in->next = finish;

finish = in;

}

else

{

while(fst->next != NULL)

{

fst = fst->next;

}

in ->next = fst ->next;

fst ->next = in;

}

}

void PrioCreate() /\*优先级调度输入函数\*/

{

PCB \*tmp;

int i;

printf("输入进程名字和进程所需时间：\n");

for(i = 0;i < num; i++)

{

if((tmp = (PCB \*)malloc(sizeof(PCB)))==NULL)

{

perror("malloc");

exit(1);

}

scanf("%s",tmp->name);

getchar(); /\*吸收回车符号\*/

scanf("%d",&(tmp->needtime));

tmp ->cputime = 0;

tmp ->state ='W';

tmp ->prio = 50 - tmp->needtime; /\*设置其优先级，需要的时间越多，优先级越低\*/

tmp ->round = 0;

tmp ->count = 0;

InsertPrio(tmp); /\*按照优先级从高到低，插入到就绪队列\*/

}

}

void TimeCreate() /\*时间片输入函数\*/

{

PCB \*tmp;

int i;

printf("输入进程名字和进程时间片所需时间：\n");

for(i = 0;i < num; i++)

{

if((tmp = (PCB \*)malloc(sizeof(PCB)))==NULL)

{

perror("malloc");

exit(1);

}

scanf("%s",tmp->name);

getchar();

scanf("%d",&(tmp->needtime));

tmp ->cputime = 0;

tmp ->state ='W';

tmp ->prio = 0;

tmp ->round = 2; /\*假设每个进程所分配的时间片是2\*/

tmp ->count = 0;

InsertTime(tmp);

}

}

void Priority() /\*按照优先级调度，每次执行一个时间片\*/

{

int flag = 1;

GetFirst();

while(run != NULL) /\*当就绪队列不为空时，则调度进程如执行队列执行\*/

{

Output(); /\*输出每次调度过程中各个节点的状态\*/

while(flag)

{

run->prio -= 0; /\*优先级减去三,若设为0则优先级不变\*/

run->cputime++; /\*CPU时间片加一\*/

run->needtime--;/\*进程执行完成的剩余时间减一\*/

if(run->needtime == 0)/\*如果进程执行完毕，将进程状态置为F，将其插入到完成队列\*/

{

run ->state = 'F';

run->count++; /\*进程执行的次数加一\*/

InsertFinish(run);

flag = 0;

}

else /\*将进程状态置为W，入就绪队列\*/

{

run->state = 'W';

run->count++; /\*进程执行的次数加一\*/

InsertTime(run);

flag = 0;

}

}

flag = 1;

GetFirst(); /\*继续取就绪队列队头进程进入执行队列\*/

}

}

void RoundRun() /\*时间片轮转调度算法\*/

{

int flag = 1;

GetFirst();

while(run != NULL)

{

Output();

while(flag)

{

run->count++;

run->cputime++;

run->needtime--;

if(run->needtime == 0) /\*进程执行完毕\*/

{

run ->state = 'F';

InsertFinish(run);

flag = 0;

}

else if(run->count == run->round)/\*时间片用完\*/

{

run->state = 'W';

run->count = 0; /\*计数器清零，为下次做准备\*/

InsertTime(run);

flag = 0;

}

}

flag = 1;

GetFirst();

}

}