**六、进程调度**

**6.1 实验目的**

1、进程调度是处理机管理的核心内容

2、通过本实验加深对进程控制块、就绪队列等概念的理解

3、体会和了解优先权调度算法的具体实现方法

**6.2 预备知识**

在操作系统中，由于进程总数多于处理机，它们必然竞争处理机。进程调度的功能就是

按一定策略、动态地把处理机分配给处于就绪队列中的某一进程并使之执行。根据不同的系

统设计目标，可有多种选择进程的策略。例如系统开销较少的静态优先数法，适合于分时系

统的轮轮法以及 UNIX 采用的动态优先数反馈法等。

有2种基本的进程调度方式，即剥夺方式(preemptive mode)和非剥夺方式(non-preemptive

mode)。前者指就绪队列中一旦有优先级高于现行进程优先级的进程出现时，系统便立即把

处理机分配给高优先级的进程。当然，被剥夺了处理机的进程的有关状态和上下文都必须妥

善保存以便今后恢复。后者是，一旦处理机分给了某进程，除非该进程的时间片已满或它主

动放弃处理机，系统不得以任何理由剥夺该现行进程的处理机。

引起进程调度的原因与操作系统的类型有关，大体可归结为以下几种：

(1) 进程运行完毕

(2) 进程提出 I/O 请求

(3) 进程执行某种原语操作(如 P 操作)导致进程阻塞

(4) 时间片已满

(5) 可剥夺方式中，就绪队列中某进程的优先级变得高于现行进程；

要上述情况出现就引发进程调度。由于进程调度的使用频率高，其性能优劣直接影响操

作系统的性能。

**6.3 实验内容**

用 C 语言编写和调试一个基于优先权的进程调度程序。

**6.4 实验指导**

基于优先权的进程调度，把所有的进程按照优先权顺序插入就绪队列，调度的时候，从

就绪队列中选择优先权最高的进程，调度运行。当一个进程运行的时候，动态修改其优先级

系数和 CPU 时间，如果 CPU 时间增加到大于或等于所需的全部时间，则表示该进程运行完成，退出；否则，继续插入就绪队列，等待再一次被调度运行。

**步骤：**

1) 初始化工作：随机产生 5 个进程，设置好进程标志符 pid、优先级 pri、当前已获得的CPU时间统计cputime、运行完成总共需要的时间alltime、状态state（全部定义为1，即运行状态）；

2) 创建一个有序链表，作为就绪队列，并把上述 5 个进程按优先级从高到低的次序插

入就绪队列；

3) 如果就绪队列不为空，从就绪队列中取下第一个进程（优先级最高的进程）模拟运

行；就绪队列为空，退出运行；

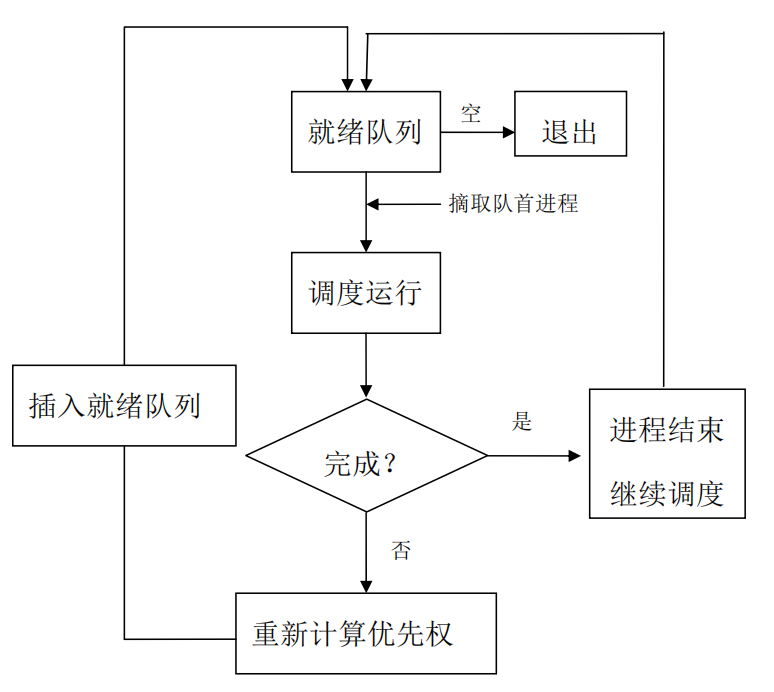
4) 修改该进程的优先级数值 pri，增加 CPU 运行时间统计，即增加 cputime；

5) 判断，如果cputime>=alltime，该进程运行完毕退出；否则，该进程继续插入就绪

队列，等待下一次调度运行；

6) 继续执行步骤 3)

算法流程：



1、数据结构说明

就绪队列是一个按照优先权排列的队列，所以我们采用有序链表结构（SortedList），在插入链表时，自动比较优先权，把尚未完成退出的进程插入到合适的位置。链表的首元素，

一定是优先权最高的进程。

有序链表（SortedList）函数说明：

Node\* InitSortedList( ) //初始化有序链表，返回一个链表的头结点

void insertSortedList(LinkList L,PROC proc) //按优先级把进程插入有序链表

PROC\* getHeadElement(LinkList L) //摘取链表的头结点，即优先权最高的进程

两个自定义结构：

typedef struct PROC{ //进程的 PCB 结构

int pid; //进程标志

int pri; //优先级

int cputime; //以用的 CPU 时间

int alltime; //运行完成所需的所有时间

int state; //状态

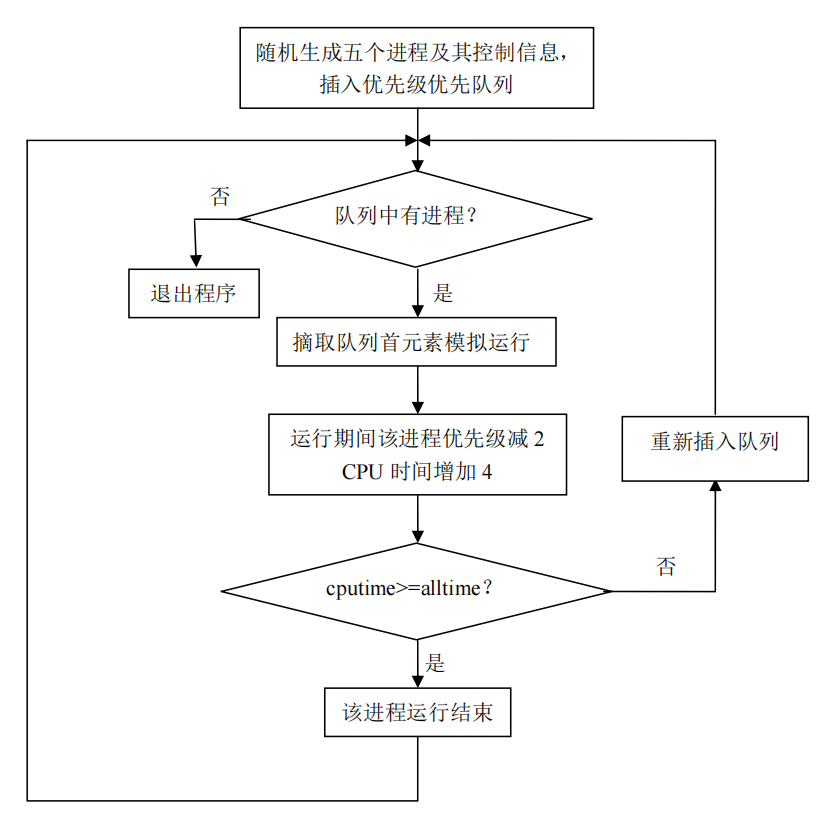
}PROC;

typedef struct Node{ //有序链表结点定义

struct PROC data;

struct Node \*next;

}Node;



**6.5 参考源代码**

#include<stdlib.h>

#include<stdio.h>

#include<math.h>

#include<malloc.h>

/\*结构定义和有序链表实现\*/

typedef struct PROC

{

int pid; //进程标志符

int pri; //进程优先级

int cputime; //cpu 时间统计

int alltime; //运行所需时间

int state; //进程状态

}PROC; //进程控制块结构

typedef struct Node

{

PROC info; //进程控制块信息

struct Node\* next; //下一个结点指针

}Node; //链表结点结构

typedef Node\* LinkList;

//初始化有序链表，返回一个链表的头结点

Node\* InitSortedList( )

{

Node\* head = ( Node\* )malloc( sizeof( Node ) );

head -> next = NULL;

return head;

}

//按优先级高低次序，把进程 Proc 插入到有序链表 L 中

void InsertSortedList( LinkList L, PROC Proc )

{

Node\* pre = L;

Node\* p = pre -> next;

Node\* NewProc = ( Node\* )malloc( sizeof( Node ) );

NewProc -> info = Proc;

while( p != NULL && Proc.pri <= ( p -> info ).pri )

{ //寻找合适的插入点

pre = p;

p = p -> next;

}

if( p == NULL )

{

pre -> next = NewProc;

NewProc -> next = NULL;

}

else

{

NewProc -> next = p;

pre -> next = NewProc;

}

}

//摘取有序链表的第一个结点（这个结点从链表中移出）

PROC\* GetHeadElement( LinkList L )

{

PROC\* p;

if( L -> next == NULL )

return NULL;

else

{

p = &(( L -> next ) -> info);

L -> next = ( L -> next ) -> next;

return p;

}

}

//打印有序链表内容

void PrintSortedList( LinkList L )

{

Node\* p = L -> next;

if( p == NULL )

printf( "\nThis SortedList Has No Node !" );

else

{

printf( "\nShow The Node(s) Of The SortedList: " );

while( p != NULL )

{

printf( "\nPid: %d", ( p -> info ).pid );

printf( " Pri: %d", ( p -> info ).pri );

printf( " CPUTime: %d", ( p -> info ).cputime );

printf( " AllTime: %d", ( p -> info ).alltime );

printf( " State: %d\n", ( p -> info ).state );

p = p -> next;

}

}

}

//打印进程控制块信息

void PrintPROC( PROC\* Proc )

{

if(Proc==NULL)

{

printf("\nProcess Point is NULL!");

}else{

printf( "\nPid: %d", Proc->pid );

printf( " Pri: %d", Proc->pri );

printf( " CPUTime: %d", Proc->cputime );

printf( " AllTime: %d", Proc->alltime );

printf( " State: %d\n", Proc->state );

}

}/\*\*\*\*\*\*以上部分可以略看，知道每个函数的功能即可\*\*\*\*\*\*\*\*\*/

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*进程调度程序，仔细理解\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

int main( ){

int i,count;

PROC p[ 5 ];

PROC\* pp;

LinkList L = InitSortedList( ); /\* 初始化一个有序链表 \*/

/\*产生五个随机不同优先级的进程\*/

for( i = 0; i < 5; i++ ) {

p[ i ].pid = i+1;

p[ i ].pri = rand()%20; //优先级最高 20

p[ i ].cputime = rand()%30; //cpu时间统计限定在 30 以内

p[ i ].alltime = p[ i ].cputime + rand()%20; //所需全部时间

p[ i ].state = 1;

}

/\* 插入进程, 产生就绪队列 \*/

for( i = 0; i < 5; i++ )

InsertSortedList( L, p[ i ] );

PrintSortedList( L ); /\* 打印就绪队列 \*/

count=1; //调度次数计数器

while(L->next!=NULL) { //就绪队列不空，调度进程运行

pp =GetHeadElement( L ); //取得优先级最高的进程

printf("/\*\*\*\*\*\*\*Schedule #%d\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/",count);

printf("\nProcess %d info before running:",pp->pid);

PrintPROC( pp );

pp->pri=(pp->pri) - 2; //每运行一次优先级降低 2

if (pp->pri<0){

pp->pri=0;

}

pp->cputime=pp->cputime+4; //每运行一次 CPU 时间增加 4

if(pp->cputime>=pp->alltime){ //运行完成，退出

printf("\nProcess %d is over!\n",pp->pid);

}else{ //还没有运行完成，继续插入就绪队列，等待调度

printf("\nProcess %d info after running:",pp->pid);

PrintPROC( pp );

InsertSortedList( L, \*pp );

PrintSortedList( L ); /\* 打印就绪队列 \*/

}

count++;

}

return 0;

}