**操作系统课程设计报告** 日期：19.5.30

|  |  |  |  |
| --- | --- | --- | --- |
| 实习题目：作业调度系统 | | | |
| 完成人 | 姓名：龙佳琪 | 班级：1730040107 | 学号2017141461065 |
| 实习内容简要描述 | 实现用户空间内的作业调度系统，且实现以下的操作  (1)提交自己的作业。  (2)将自己提交的作业移出。  (3)查看作业状态。  作业的调度以时间片轮转和优先级结合的算法进行 | | |
| 主要代码结构  （附注释） | 数据结构：  //命令的类型，分别是入队、删除和查看作业状态  enum cmdtype{  ENQ,  DEQ,  STAT  };  //命令结构体  struct jobcmd{  enum cmdtype \*type;  int argnum;  int owner;  int defpri;  char data[BUFLEN];  };  //作业状态  enum jobstate{  READY,  RUNNING,  DONE  };  struct jobinfo{  int jid;  int pid;  char cmdarg[BUFLEN][BUFLEN];  int defpri;  int curpri;  int ownerid;  int wait\_time;  time\_t creat\_time;  int run\_time;  enum jobstate state;  };  typedef struct jobinfo\* Elementype; // 定义数据类型  // 定义节点结构  typedef struct Node {  Elementype Element; // 数据域  struct Node \* Next;  }NODE, \*PNODE;  // 定义队列结构体  typedef struct QNode {  PNODE Front, Rear; // 队列头，尾指针  } Queue, \*PQueue;  主要函数：   1. 作业的创建   int creatJobInfo(struct jobcmd \*jobCmd,struct jobinfo \*jobInfo);  //根据作业命令创建作业进程  int splitCommand(char argv[BUFLEN],char commands[BUFLEN][BUFLEN]);  //将输入的字符串按空格分割，返回分割出的命令数  void RunScheduel(PQueue WaitQueue);  //作业运行  void InitialCmd(struct jobcmd \*jobCmd);  //初始化作业命令   1. 作业的运行   int get\_pri\_info(PQueue ReadyQueue); //找出优先级最高的进程号  void fresh(PQueue ReadyQueue,int id); //刷新 :num-进程总数 id-运行的那个进程号；  int isFinish(PQueue ReadyQueue); //判断当前队列是否处理完  void handler(int sig);//句柄函数，设置的定时器得到相关信号后就调用它   1. 队列处理函数   void InitQueue(PQueue); // 创建队列函数  int IsEmptyQueue(PQueue); // 判断队列是否为空函数  void InsertQueue(PQueue, struct jobinfo\* val); // 入队函数  void DeleteQueue(PQueue,struct jobinfo \* val); // 出队函数  void DestroyQueue(PQueue); // 摧毁队列函数  void TraverseQueue(PQueue); // 遍历队列函数  void ClearQueue(PQueue); // 清空队列函数  int LengthQueue(PQueue); // 求队列长度函数  void DeleteItem(PQueue,int pid); //清除队列中进程号为pid的进程   1. 全局变量   Queue ReadyQueue; //准备队列  int jidx=0; //作业号  int pidx=0; //进程号 | | |
| 结果分析（或错误原因分析） | 插入命令并查看    删掉1号    运行中    显然0号优先级高，由于我的队列是从头开始找的，后续的要大于它才可以运行，所以0、1号虽然优先级相同但是0先运行    仍然在运行0号，但是其他进程的等待时间发生变化，累计一定后优先级也会变化 | | |
| 注释：如果上述表格空间不够，可以另附表格进行说明 | | | |

设计文档：

输入使用fgets获取缓冲区的一行，再分割，注意每一次要清理缓冲且要处理掉转行符’\n’ 不同下标的对应不同的参数

enq命令中[-p]可以设置优先级

atoi函数可以将字符串转换为整形，由于分割后所取得的优先级是以字符串的形式存在的，故用此函数转换

deq实现队列中项目的删除，根据的是进程号pid：遍历，如果下一个就是要删除的，那么把当前点的next设置为删除点的next

stat输入作业的相关消息

定时器设定为10，通过查阅资料，用setitimer实现，同时用signal函数处理信号

struct itimerval new\_value,old\_value;//set the time slip=10

new\_value.it\_value.tv\_sec=4;

new\_value.it\_value.tv\_usec=0;

new\_value.it\_interval.tv\_sec=4;

new\_value.it\_interval.tv\_usec=0;

setitimer(ITIMER\_REAL,&new\_value,&old\_value);

运行中时，首先遍历整个队列找到优先级最大的并记下其进程号，然后进行刷新：

对于进行中的作业：等待时间清零，优先级置为默认优先级，运行时间-10（因为时间片设定为10）；等待作业：等待时间加上这一段的时间变化，同时若等待时间>100，优先级+1；依次类推；当所有作业都运行完时发出提示信息

获取时间：定义time\_t 型变量，通过time(& time\_t time)获取当前实时时间

附代码:

Queue.h

// 队列的单链表实现

// 头节点：哨兵作用，不存放数据，用来初始化队列时使队头队尾指向的地方

// 首节点：头节点后第一个节点，存放数据

#ifndef \_QUEUE\_H\_

#define \_QUEUE\_H\_

#include<stdio.h>

#include<malloc.h>

#include<stdlib.h>

#define BUFLEN 255

enum jobstate{

READY,

RUNNING,

DONE

};

/\*struct waitqueue{

struct waitqueue \*next;

struct jobinfo\* job;

};\*/

struct jobinfo{

int jid;

int pid;

char cmdarg[BUFLEN][BUFLEN];

int defpri;

int curpri;

int ownerid;

int wait\_time;

time\_t creat\_time;

int run\_time;

enum jobstate state;

};

typedef struct jobinfo\* Elementype; // 定义数据类型

// 定义节点结构

typedef struct Node {

Elementype Element; // 数据域

struct Node \* Next;

}NODE, \*PNODE;

// 定义队列结构体

typedef struct QNode {

PNODE Front, Rear; // 队列头，尾指针

} Queue, \*PQueue;

// 声明函数体

void InitQueue(PQueue); // 创建队列函数

int IsEmptyQueue(PQueue); // 判断队列是否为空函数

void InsertQueue(PQueue, struct jobinfo\* val); // 入队函数

void DeleteQueue(PQueue,struct jobinfo \* val); // 出队函数

void DestroyQueue(PQueue); // 摧毁队列函数

void TraverseQueue(PQueue); // 遍历队列函数

void ClearQueue(PQueue); // 清空队列函数

int LengthQueue(PQueue); // 求队列长度函数

void DeleteItem(PQueue,int pid); //清除队列中进程号为pid的进程

// 定义队列初始化函数

void InitQueue(PQueue queue) {

queue->Front = queue->Rear = (PNODE)malloc(sizeof(NODE)); // 动态创建头节点，使队头，队尾指向该节点

//头节点相当于哨兵节点的作用，不存储数据（区别于首节点）

if (queue->Front == NULL) { // 判断内存是否分配成功

printf("创建队列，无法分配所需内存...");

exit(-1);

}

queue->Front->Next = NULL; // 初始队列头节点指针指向为空

printf("创建队列成功...\n");

}

// 定义判断队列是否为空函数

int IsEmptyQueue(PQueue queue) {

if (queue->Front == queue->Rear) {

printf("队列为空...\n");

return 1;

}

else {

//printf("队列不为空...\n");

return 0;

}

}

// 定义入队函数

// 从队列尾部插入数据val

void InsertQueue(PQueue queue,struct jobinfo\* val) {

PNODE P = (PNODE)malloc(sizeof(NODE)); // 创建一个新节点用于存放插入的元素

if (P == NULL) {

printf("内存分配失败，无法插入数据%d...", val->pid);

exit(-1);

}

P->Element = val; // 把要插入的数据放到节点数据域

P->Next = NULL; // 新节点指针指向为空

queue->Rear->Next = P; // 使上一个队列尾部的节点指针指向新建的节点

queue->Rear = P; // 更新队尾指针，使其指向队列最后的节点

printf("插入数据 %d 成功...\n", val->pid);

}

// 定义出队函数

// 从队列的首节点开始出队

// 若出队成功，用val返回其值

void DeleteQueue(PQueue queue,struct jobinfo\* val) {

if (IsEmptyQueue(queue)) {

printf("队列已经空，无法出队...\n");

exit(-1);

}

PNODE P= queue->Front->Next; // 临时指针

val = P->Element; // 保存其值

queue->Front->Next = P->Next; // 更新头节点

if (queue->Rear==P)

queue->Rear = queue->Front;

free(P); // 释放头队列

P = NULL; // 防止产生野指针

printf("出栈成功，出栈值为 %d\n", val->pid);

}

// 定义队列遍历函数

void TraverseQueue(PQueue queue) {

if (IsEmptyQueue(queue)) {

exit(-1);

}

PNODE p = queue->Front->Next; //从队列首节点开始遍历（非头节点，注意区分）

printf("遍历队列结果为：\n");

while (p != NULL) {

//printf("%d ", P->Element);

printf("pid: %d\n",p->Element->pid);

printf("user name: %d\n",p->Element->ownerid);

printf("run time: %d\n",p->Element->run\_time);

printf("wait time: %d\n",p->Element->wait\_time);

printf("creat time: %ld\n",p->Element->creat\_time);

printf("job state: %d\n",p->Element->state);

printf("job priority: %d\n",p->Element->curpri);

printf("\n");

p = p->Next;

}

}

// 定义队列的摧毁函数

// 删除整个队列，包括头节点

void DestroyQueue(PQueue queue) {

//从头节点开始删除

while (queue->Front != NULL) {

queue->Rear = queue->Front->Next;

free(queue->Front);

queue->Front = queue->Rear;

}

printf("摧毁队列成功...\n");

}

int LengthQueue(PQueue queue){

int len=0;

if (IsEmptyQueue(queue)) {

return 0;

}

PNODE p = queue->Front->Next; //从队列首节点开始遍历（非头节点，注意区分）

printf("遍历队列结果为：\n");

while (p != NULL) {

//printf("%d ", P->Element);

len++;

p = p->Next;

}

return len;

}

// 定义清空队列函数

void ClearQueue(PQueue queue) {

PNODE P = queue->Front->Next; // 临时指针

PNODE Q = NULL; // 临时指针

queue->Rear = queue->Front; // 使队尾指针指向头节点

queue->Front->Next = NULL;

// 从首节点开始清空

while (P != NULL) {

Q = P;

P = P->Next;

free(Q);

}

printf("清空队列成功...\n");

}

void DeleteItem(PQueue queue,int pid)

{

PNODE P = queue->Front->Next;

while(P!=NULL){

if(P->Next->Element->pid==pid)

{

printf("清除%d号:\n",P->Next->Element->pid);

//if(P->Next->Next!=NULL)

P->Next=P->Next->Next;

break;

}

P=P->Next;

}

}

#endif

Schedule.c

#include<stdio.h>

#include<stdlib.h>

#include<time.h>

#include<unistd.h>

#include<math.h>

#include<sys/time.h>

#include<signal.h>

#include<sys/types.h>

#include<sys/stat.h>

#include<string.h>

#include "queue.h"

#define BUFLEN 255

#define BUFERSIZE 1024

int jidx=0;

int pidx=0;

enum cmdtype{

ENQ,

DEQ,

STAT

};

struct jobcmd{

enum cmdtype \*type;

int argnum;

int owner;

int defpri;

char data[BUFLEN];

};

int creatJobInfo(struct jobcmd \*jobCmd,struct jobinfo \*jobInfo);

int splitCommand(char argv[BUFLEN],char commands[BUFLEN][BUFLEN]);

void RunScheduel(PQueue WaitQueue);

void InitialCmd(struct jobcmd \*jobCmd);

//运行处理

int get\_pri\_info(PQueue ReadyQueue); //找出优先级最高的进程号

void fresh(PQueue ReadyQueue,int id); //刷新 :num-进程总数 id-运行的那个进程号；

int isFinish(PQueue ReadyQueue); //判断当前队列是否处理完

Queue ReadyQueue;

void handler(int sig);

int main()

{

struct jobinfo jobs[50];

InitQueue(&ReadyQueue);

int i=0;

while(1){

char argv[BUFERSIZE];

char buf\_in[BUFERSIZE];

int fd[2];//fd[0] read; fd[1] write

int pipe\_ret;

ssize\_t write\_ret;

ssize\_t read\_ret;

char buf\_out[BUFERSIZE]={"0"};

strcpy(buf\_in,"");

pipe\_ret=pipe(fd);

if(pipe\_ret==-1)

{

perror("pipe error!");

exit(-1);

}

//setbuf(stdin,NULL);

//fflush(stdin);

fgets(buf\_in,BUFERSIZE,stdin);

//printf("清空前%s\n",buf\_in);

//scanf("%\*[^\n]%\*c");

setbuf(stdin,NULL);

//printf("清空后%s\n",buf\_in);

int len=strlen(buf\_in);

if(buf\_in[len-1]=='\n')

buf\_in[len-1]=0;

//write\_ret=write(fd[1],buf\_in,strlen(buf\_in));

//printf("write\_ret:%ld\n",write\_ret);

if(write\_ret==-1)

{

perror("write error!");

exit(-1);

}

else{//deal the command

//read\_ret=read(fd[0],argv,BUFERSIZE);

printf("输入的是%s\n",buf\_in);

if(read\_ret==-1)

{

perror("read error!");

exit(-1);

}

char command[BUFLEN][BUFLEN];

int cnt=splitCommand(buf\_in,command);//一命令中分割后的个数

struct jobcmd jobCmd;

InitialCmd(&jobCmd);

if(!strcmp(command[0],"enq"))//enq jobCmd.type==0

{

jobCmd.type=READY;

jobCmd.argnum=cnt-1;

jobCmd.owner=i;

if(cnt==5&&(!strcmp(command[1],"-p"))){

jobCmd.defpri=atoi(command[2]);

}

else if(cnt==4)

jobCmd.defpri=0;

strcpy(jobCmd.data,buf\_in);

creatJobInfo(&jobCmd,&jobs[i]);

InsertQueue(&ReadyQueue,&jobs[i]);

}

else if(!strcmp(command[0],"deq"))//(strcmp(jobCmd.type,"DEQ"))

{

jobCmd.type=READY;

jobCmd.argnum=cnt-1;

jobCmd.owner=i;

jobCmd.defpri=0;

strcpy(jobCmd.data,buf\_in);

creatJobInfo(&jobCmd,&jobs[i]);

int pidx=atoi(command[1]);

printf("待删除号：%d:\n",pidx);

DeleteItem(&ReadyQueue,pidx);

}

else if(!strcmp(command[0],"stat"))

{

jobCmd.type=READY;

jobCmd.argnum=cnt-1;

jobCmd.owner=i;

jobCmd.defpri=0;

strcpy(jobCmd.data,buf\_in);

creatJobInfo(&jobCmd,&jobs[i]);

TraverseQueue(&ReadyQueue);

}

else

{

perror("No such Command!");

}

//printf("------------jobcmd-------------\n");

//printf(" type: %d\n argnum: %d\n defpri: %d\n data:%s \nowner:%d\n",jobCmd.type,jobCmd.argnum,jobCmd.defpri,jobCmd.data,jobCmd.owner);

// printf("------------jobinfo--------------\n");

// printf(" jid: %d\n pid: %d\n wait\_time: %d\n creat\_time:%ld\n",jobs[i].jid,jobs[i].pid,jobs[i].wait\_time,jobs[i].creat\_time);

//printf("------------jobinfo-------------\n");

//printf("jid:%d\npid:%d\ndefpri:%d\ncurpri:%d\nownerid:%d\n",jobs[i].jid,jobs[i].pid,jobs[i].defpri,jobs[i].curpri,jobs[i].ownerid);

//设定时间片 没输入指令时就run

//printf("运行完了没？%d\n",isFinish(&ReadyQueue));

signal(SIGALRM,handler);

struct itimerval new\_value,old\_value;//set the time slip=10

new\_value.it\_value.tv\_sec=4;

new\_value.it\_value.tv\_usec=0;

new\_value.it\_interval.tv\_sec=4;

new\_value.it\_interval.tv\_usec=0;

setitimer(ITIMER\_REAL,&new\_value,&old\_value);

}

i++;

}

}

int creatJobInfo(struct jobcmd \*jobCmd,struct jobinfo \*jobInfo)

{

jobInfo->jid=jidx;

jidx+=1;

jobInfo->pid=pidx;

pidx+=1;

int command[BUFLEN][BUFLEN];

char arg[BUFLEN]={"0"};

strncpy(arg,jobCmd->data,jobCmd->argnum);

splitCommand(arg,jobInfo->cmdarg);

jobInfo->defpri=jobCmd->defpri;

jobInfo->curpri=jobCmd->defpri;

jobInfo->ownerid=jobCmd->owner;

jobInfo->wait\_time=0;

time(&jobInfo->creat\_time);

jobInfo->run\_time=rand()%50;//随机生成运行时间

jobInfo->state=READY;

}

int splitCommand(char argv[BUFLEN],char commands[BUFLEN][BUFLEN])

{

int num = 0;

int i, j;

int len = strlen(argv);

for (i=0, j=0; i<len; ++i) {

if (argv[i] != ' ') {

commands[num][j++] = argv[i];

} else {

if (j != 0) {

commands[num][j] = '\0';

++num;

j = 0;

}

}

}

if (j != 0) {

commands[num][j] = '\0';

++num;

}

return num;

}

void InitialCmd(struct jobcmd \*jobCmd)

{

jobCmd->type=READY;

jobCmd->argnum=0;

jobCmd->owner=0;

jobCmd->defpri=0;

strcpy(jobCmd->data,"");

}

int get\_pri\_info(PQueue ReadyQueue)

{

if (IsEmptyQueue(ReadyQueue)) {

exit(-1);

}

PNODE p = ReadyQueue->Front->Next; //从队列首节点开始遍历（非头节点，注意区分）

int id=p->Element->pid; //设定初始号，即第一个

int maxPri=0; //初始最大优先级

printf("开始找最大优先级：\n");

while (p != NULL) {

//printf("%d ", P->Element);

printf("新结点优先级:%d\n",p->Element->curpri);

printf("新结点状态:%d\n",p->Element->state);

if( (p->Element->curpri)>maxPri )

{

//printf("新结点状态:%d\n",p->Element->state);

if((p->Element->state)!=2)

{

printf("刷新最大值时新结点状态:%d\n",p->Element->state);

id=p->Element->pid;

maxPri=p->Element->curpri;

}

}

p = p->Next;

}

printf("最高为%d级，是%d号\n",maxPri,id);

return id;

}

void fresh(PQueue ReadyQueue,int id)

{ //当所有都执行完时退出

if(isFinish(ReadyQueue)==1) {

printf("finish scheduel!\n");

//signal(SIGSTOP,handler);

exit(1);

}

PNODE p = ReadyQueue->Front->Next; //从队列首节点开始遍历（非头节点，注意区分）

time\_t current;

time(&current);//获取当前运行的系统时间

while (p != NULL) {

//struct jobinfo \*job=p->Element;

if((p->Element->pid)!=id && ((p->Element->state)!=2)) //非执行的进程刷新，每100ms优先级+1,同时增加等待时间

{

int delta=current-p->Element->creat\_time;

p->Element->wait\_time=delta;

if(( p->Element->curpri + (p->Element->wait\_time)/100 )<4){

p->Element->curpri+=(p->Element->wait\_time)/100;

}

else{

p->Element->curpri=3;

}

}

else if((p->Element->pid)==id) //执行的那个

{

printf("执行结点状态:%d\n",p->Element->state);

int temp\_time=p->Element->run\_time;

p->Element->wait\_time=0;

p->Element->curpri=p->Element->defpri;

p->Element->run\_time-=10;

temp\_time-=10;

p->Element->state=RUNNING;

if(temp\_time<0)

{

printf("It goes to be DONE\n");

p->Element->state=DONE;

}

}

p = p->Next;

}

}

int isFinish(PQueue ReadyQueue)

{

int flag=1;

if (IsEmptyQueue(ReadyQueue)) {

exit(-1);

}

PNODE p = ReadyQueue->Front->Next; //从队列首节点开始遍历（非头节点，注意区分）

while (p != NULL) {

if((p->Element->state)!=2)

{

flag=0;

break;

}

p = p->Next;

}

return flag;

}

void handler(int sig)

{

printf("信号：%d!\n",sig);

if(isFinish(&ReadyQueue)==0)

{

RunScheduel(&ReadyQueue);

}

else{

printf("Finish! No more infos\n");

exit(-1);

}

}

void RunScheduel(PQueue WaitQueue)

{

printf("Run\n");

int id;

id=get\_pri\_info(WaitQueue); //找出优先级最高的进程号

fresh(WaitQueue,id); //刷新

}

运行结果截屏：













