操作系统

实验报告

实验名称： 进程调度

院系： 计算机学院·网络空间安全学院

班级：

学号：

姓名：

指导老师：匡林爱

日期：2020年10月27日

1. **实验目的和要求：**

要求实现一个进程调度程序，通过该程序可以完成进程的创建、撤销、查看和调度。具体要求如下：

(1) 实现进程调度程序scheduleProcess，负责整个系统的运行。

这是一个不停循环运行的进(线)程，其任务是及时响应进程的创建、撤销及状态查看请求，要采用适当的进程调度策略调度进程的运行。

(2) 实现创建进程的命令。

用户通过本命令发送创建进程请求，将进程信息提交给系统。系统创建进程，为其分配一个唯一的进程标识PID，并将状态置为READY，然后放入就绪队列中。

(3) 实现撤销进程命令

输入撤销命令以后，系统就会删除待撤销的进程在缓冲区中的内容，如果输入有误，程序会给出出错提示。

(4) 实现查看进程状态的命令

该命令打印(显示)出当前运行进程和就绪队列中各进程的信息，状态信息应包括：

* 进程的PID
* 进程名字
* 进程状态（RUN、READY、WAIT）

(5) 实现时间片轮转调度算法

处理机总是优先调度运行就绪队列中的第一个进程，当时间片结束后就把该进程放在就绪队列的尾部。在系统的实现以及运行中，不必考虑Windows操作系统本身的进程调度。假设所有的作业均由scheduleProcess调度执行，而且进程在分配给它的时间片内总是不间断地运行。

1. **实验内容：**

本实验是在Windows 2000/XP+VC++6.0环境下实现的，利用Windows SDK提供的应用程序接口(API)完成程序的功能。实验中所使用的API是操作系统提供的用来进行应用程序设计的系统功能接口。要使用这些API，需要包含对这些函数进行说明的SDK头文件，最常见的是windows.h。一些特殊的API调用还需要包含其他的头文件。

在Windows 2000/XP中，每个进程至少包含一个线程，进程均由一个线程启动，该线程称为主线程，例如，我们用VC++语言编写了一个程序，编译运行后就是一个进程，该进程的主线程运行main函数的代码。进程(主线程)可以动态创建新的线程(子线程)。进程和线程均可以通过Win32 API函数CreateProcess和CreateThread动态创建(详见上机实验二和上机实验三)。

线程通常在用户态下运行，当它进行系统调用时，会切换到核心态运行，并继续作为同一线程并具有用户态下相同的属性和限制。当一个线程执行完毕时，它可以退出。当进程的最后一个线程退出时，该进程终止。

线程是一个调度的概念而不是一个占有资源的概念。任何一个线程可以访问它所属进程的所有对象(资源)，它所要做的只是获得对象句柄并做适当的Win32调用。

由于Windows 2000/XP中，调度的对象是线程而不是进程，所以每个线程具有一个状态(就绪、运行、阻塞等)，但是进程没有这些状态。线程构成了CPU调度的基础，因为操作系统会选择一个线程运行而不是一个进程。因此，Win32 API中有关于线程操作的相关函数，包括：线程的暂停(挂起)、线程的回复(唤醒)、线程的终止等等，利用这些操作，我们可以较方便地模拟进程调度的过程。设计本上机实验的最主要思路，就是用线程模拟进程。然后对进程(实际上是对线程)进行调度演示的。

### 相关线程的几个Win32 API函数介绍

在Windows 2000/XP系统中，要进行进程/线程的创建、撤销和调度等操作，都需要通过Win32 API调用来完成。涉及的API如下：

**1．CreateThread( )函数**

**2．SuspendThread( )函数**

**功能**：

挂起(暂停)线程。被挂起的线程将暂停执行，直到被回复(唤醒)为止。

**格式**：DWORD SuspendThread ( HANDLE hthread )

**参数说明：**hthread：线程的句柄。该句柄在调用CreateThread时获得。

**3．ResumeThread( )函数**

**功能**：

恢复(唤醒)线程。被挂起的线程恢复后才能为其分配CPU。该函数正确执行则返回一正整数，否则返回0xFFFFFFFF(即-1)。

**格式**：DWORD ResumeThread ( HANDLE hthread )

**参数说明：**hthread：线程的句柄。

**4．TerminateThread( )函数**

**功能**：终止线程运行。

**格式**：BOOL TerminateThread ( HANDLE hthread, DWORD dwExitCode )

**参数说明**

hthread：线程的句柄。

dwExitCode：线程终止时的退出代码(返回值)。

【**注**】TerminateThread终止线程时，该线程的初始堆栈未被回收，依附于该线程的DDLs也未被告知线程已经结束。TerminateThread是一个危险的函数，该函数仅仅在最极端的情况下使用。只有当你确切地知道目标线程正在工作以及在终止时能控制该线程所有可能正在运行的代码时才能够使用此函数。例如，TerminateThread可能导致如下问题：

* 如果目标线程拥有一个临界区，该临界区将不被释放。
* 如果目标线程被终止时正在执行某些kernel32调用，则将导致该线程进程的kernel32状态的不一致性。
* 如果目标线程正在操纵共享的DDL的全局状态，则DDL的状态将被破坏从而影响使用DDL的其他用户。

**5．Sleep函数**

**6．CloseHandle函数**

**功能**：关闭内核对象(线程)的句柄，将对象引用计数减1，或者释放堆栈资源。

**格式**：BOOL CloseHandle (HANDLE hobj )

**参数说明：**hobj：对象的句柄。

多个线程操作相同的数据时，一般是需要按顺序访问的(互斥访问)，否则会引起数据错乱，使其无法控制数据。为了解决这个问题，需引入互斥变量，让这些线程都按顺序地访问变量(程序中是指互斥访问链队列这类临界资源)。这样就需要使用如下三个函数。

**7．InitializeCriticalSection( ) 函数**

**8．EnterCriticalSection( ) 函数**

**9．LeaveCriticalSection ( ) 函数**

### 重要的数据结构

**1．进程控制块**

typedef struct PCB // 进程控制块

{

int id; // 进程标识PID

char name[20]; // 进程名

enum STATUS status; // 进程状态：RUN, READY, WAIT

int flag; //为了不重复显示，额外增加此成员

HANDLE hThis; // 进程句柄(实际上是线程句柄)

DWORD threadID; // 线程ID

int count; // 进程计划运行时间长度，以时间片为单位

struct PCB \*next; // 指向就绪队列或缓冲区(空闲PCB)链的指针

} PCB,\*pPCB;

该数据结构定义了用于进程控制的若干属性，包括PID(进程标识号)、进程名字、进程状态(有三种：RUN、READY、WAIT)、操作进程的句柄和进程剩余时间长度等。若采用优先级调度，还需要有优先数等。

**2．用于管理进程就绪队列和空闲PCB队列的结构**

为了操作方便，在程序中还定义了一个数据结构用于操作进程队列。如下：

typedef struct // 就绪队列和空闲PCB队列管理用的结构

{

pPCB head; // 队首指针

pPCB tail; // 队尾指针(基本的时间片轮转算法使用)

int pcbNum; // 队列中的进程数

} readyList, freeList, \*pList;

1. **实验步骤：**

**1．主函数**

在主函数main( )中，首先打开用于记录信息的文件，并调用init( )函数初始化各数据结构并启动进程调度线程，然后在一个无限循环中接收用户输入的命令：创建进程(create)、撤销进程(remove)或查看信息(current)。并调用相应的函数进行响应。为了更清楚观察进程调度过程，在进入无限循环前，预先自动创建了6个进程。

**2．初始化环境函数**

在程序中，设置了一个可调度进程数的上限PCB\_LIMIT。在初始化环境函数init( )中为每个可用的PCB结构分配了空间，称为缓存区freeList。每当新创建一个进程，则从缓存区中取一个PCB结构放入就绪队列readyList中。当一个进程结束时，则需把该PCB内容清空并放回缓存区。

**3．创建线程用于模拟进程**

为了能使程序中的进程调度程序scheduleProcess能够调度进程，在程序中使用线程来模拟用户进程，调度程序也是由线程模拟。这里以用户进程的模拟线程为例说明线程函数的编写格式和线程的创建方法。

模拟用户进程的线程的工作代码(线程函数)如下：

DWORD WINAPI processThread(LPVOID lpParameter)

{

pPCB currentPcb=(pPCB)lpParameter;

while (true)

{

if (currentPcb->flag==1)//若调度后第一次运行，则显示信息

{

currentPcb->flag=0;

EnterCriticalSection(&cs\_SaveInfo);

//“进程正在运行”信息保存到文件

log<<"Process "<<currentPcb->id<<':'<<currentPcb->name

<<" is running............"<<endl;

LeaveCriticalSection(&cs\_SaveInfo);//离开临界区

}

Sleep(900); // 等待900ms

}

return 1;

}

这个线程只有一个死循环，除了每次被调度(实验中用唤醒模拟调度，真正的调度实际上是由操作系统完成)后输出一个“正在运行”的信息外，什么都不做，直到线程被终止。DWORD WINAPI表明这是一个线程入口函数，LPVOID lpParameter是线程函数的参数表。线程函数必须有一个DWORD（即unsigned long）类型的返回值，作为该线程的退出标识(因此线程的返回值是一个无符号整数)。

创建该线程(用于模拟进程)的代码如下：

void createProcess(char \*name,int count)

{

……

newPcb->hThis = CreateThread (NULL, 0, processThread,newPcb,

CREATE\_SUSPENDED, &(newPcb->threadID ) );

……

}

这段代码用CreateThread ( )函数创建了一个线程，它对应的线程工作代码为processThread函数，即上面定义的模拟用户进程的线程函数，线程的参数为一个指向PCB结构的指针newPcb，线程的初始状态为暂停(CREATE\_SUSPENDED创建的线程不利己运行，而是先挂起)，该线程的线程号也记录到newPcb中。关于CreateThread ( )函数各参数的含义可参见前面的API介绍。

**4．进程调度函数**

进程调度函数scheduleProcess ( )是本实验中的重点。调度要做的主要工作是暂停当前运行的进程，如果运行时间未用完，则将其挂起停止运行(使用SuspendThread函数)，并将其PCB插入就绪队列，同时从就绪队列中选择一个进程(就绪队列队首进程)让其运行(使用ResumeThread函数让其从挂起状态恢复)，同时更新进程状态。

在每次调度完新进程后，还要利用Sleep()函数使调度程序睡眠一段时间。睡眠时间就是时间片的大小，从而表示消耗了一个时间片(本实验的时间片是1000ms，即1s)。

参考程序中采用的是轮转调度算法。当一个进程运行时间片到期时，要调用SuspendThread()函数暂停进程运行，并判断此进程是否运行完毕。如果进程运行时间结束，就调用TerminateThread()函数终止该进程，并且回收PCB空间；如果该进程没有结束，就把它放到就绪队列尾部，最后取出就绪队列队首的进程，调用resumeThread()函数恢复其运行。

**5．命令处理函数**

命令处理函数分别处理create、remove、current三种命令。

1. **实验源程序**

**schedule.h:**

#include <string.h>

#include <windows.h>

#include <iostream>

#include <fstream>

using namespace std;

#ifndef SCHEDULE\_H

#define SCHEDULE\_H

typedef struct PCB // 进程控制块

{

int id; // 进程标识PID

char name[20]; // 进程名

enum STATUS status; // 进程状态：RUN, READY, WAIT

int flag; //为了不重复显示，额外增加此成员

HANDLE hThis; // 进程句柄

DWORD threadID; // 线程ID

int count; // 进程计划运行时间长度，以时间片为单位

struct PCB\* next; // 指向就绪队列或缓冲区(空闲PCB)链的指针

} PCB, \* pPCB;

// 就绪队列或空白PCB队列的管理用的结构

typedef struct

{

pPCB head; // 队首指针

pPCB tail; // 队尾指针

int pcbNum; // 队列中的进程数

} readyList, freeList, \* pList;

enum STATUS { RUN, READY, WAIT }; // 进程的三种状态列表

// 进程控制块数为30(最多允许创建30个进程)

const int PCB\_LIMIT = 30;

HANDLE init();

void createProcess(char\* name, int ax);

void scheduleProcess();

void removeProcess(char\* name);

void fprintReadyList();

void printReadyList();

void printCurrent();

void stopAllThreads();

#endif

**schedule.cpp文件**

#include "schedule.h"

readyList readylist; //定义就绪队列的管理结构

pList pReadyList = &readylist; //定义该结构的指针

freeList freelist; //定义空闲PCB队列的管理结构

pList pFreeList = &freelist; //定义该结构的指针

PCB pcb[PCB\_LIMIT]; //定义30个PCB

pPCB runPCB; //指向当前运行的进程PCB的指针

int pid = 0; //进程id(创建进程用)

//临界区对象

CRITICAL\_SECTION cs\_ReadyList; //用于互斥访问就绪队列

CRITICAL\_SECTION cs\_SaveInfo; //用于互斥访问保存信息的文件

//输出文件

extern ofstream log1;

extern volatile bool exiting; //当exiting=true时程序结束

//初始化进程控制块

void initialPCB(pPCB p)

{

p->id = 0; //进程id

strcpy\_s(p->name, "NoName");//进程名字

p->status = WAIT; //进程状态

p->next = NULL; //PCB的next指针

p->hThis = NULL; //进程句柄

p->threadID = 0; //线程id

p->flag = 1; //开始时允许线程显示信息

p->count = 0; //进程计划运行时间

}

//从空白PCB队列取一空闲进程控制块

pPCB getPcbFromFreeList()

{

pPCB freePCB = NULL;

if (pFreeList->head != NULL && pFreeList->pcbNum > 0)

{

freePCB = pFreeList->head;

pFreeList->head = pFreeList->head->next;

pFreeList->pcbNum--;

}

return freePCB;

}

//释放PCB使之插入空闲PCB队列

void returnPcbToFreeList(pPCB p)

{

if (pFreeList->head == NULL) //若当前空闲PCB队列为空

{

pFreeList->head = p;

pFreeList->tail = p;

p->next = NULL;

pFreeList->pcbNum++;

}

else //若空白PCB队列不空，则将释放的PCB插入队首

{

p->next = pFreeList->head;

pFreeList->head = p;

pFreeList->pcbNum++;

}

}

// 模拟用户进程的线程之执行代码

DWORD WINAPI processThread(LPVOID lpParameter)

{

pPCB currentPcb = (pPCB)lpParameter;

while (true)

{

if (currentPcb->flag == 1)//若调度后第一次运行，则显示信息

{

currentPcb->flag = 0;

EnterCriticalSection(&cs\_SaveInfo);

//“进程正在运行”信息保存到文件

log1 << "Process " << currentPcb->id << ':' << currentPcb->name

<< " is running............" << endl;

LeaveCriticalSection(&cs\_SaveInfo);//离开临界区

}

Sleep(800); // 等待800ms

}

return 1;

}

// 调度线程的执行代码

DWORD WINAPI scheduleThread(LPVOID lpParameter)

{

// pList preadyList=(pList)lpParameter;//实际上此参数无作用

//循环调用进程调度函数，直到exiting=true为止

while (!exiting)//若exiting=false,则循环执行调度程序

{

scheduleProcess();

}

stopAllThreads();//若exiting=true，则结束所有进程(线程)

return 1;

}

// 初始化操作

HANDLE init()//函数正确执行后，返回调度线程的句柄

{

pPCB p = pcb; //指向第一个PCB

//就绪队列初始化为空(初始化其管理结构)

pReadyList->head = NULL;

pReadyList->tail = NULL;

pReadyList->pcbNum = 0;

//空闲队列初始化为空(初始化其管理结构)

pFreeList->head = &pcb[0];

pFreeList->tail = &pcb[PCB\_LIMIT - 1];

pFreeList->pcbNum = PCB\_LIMIT;

//构成空闲PCB队列

for (int i = 0; i < PCB\_LIMIT - 1; i++)//PCB\_LIMIT已在头文件中定义为30

{

initialPCB(p);

p->next = &pcb[i + 1];

p++;

}

initialPCB(p);

pcb[PCB\_LIMIT - 1].next = NULL;

InitializeCriticalSection(&cs\_ReadyList); //初始化临界区对象

InitializeCriticalSection(&cs\_SaveInfo);

exiting = false; //使调度程序不断循环

// 创建调度程序的监控线程

HANDLE hSchedule; //程序调度线程的句柄

hSchedule = CreateThread(

NULL, //返回的句柄不能被子线程继承

0, //新线程堆栈大小与主线程相同

scheduleThread, //新线程执行此参数指定的函数的代码

pReadyList, //传递给函数scheduleThread的参数

//(本程序中此参数实际上没有用处)

0, //新线程的初始状态为运行状态

NULL //对新创建线程的id不感兴趣

);

//预先创建6个进程

char pName[6] = "p00";

for (int i = 0; i < 6; i++)

{

pName[2] = '0' + i;

createProcess(pName, 10);//addApplyProcess(pName,10);

}

return hSchedule;

}

// 创建进程(此函数并非API函数CreateProcess)

void createProcess(char\* name, int count)

{

EnterCriticalSection(&cs\_ReadyList); //准备进入临界区

if (pFreeList->pcbNum > 0) // 若有用于创建进程的空白PCB

{

pPCB newPcb = getPcbFromFreeList();// 从空白PCB队列获取一个空白PCB

newPcb->status = READY; // 新进程状态为“READY”

strcpy\_s(newPcb->name, name); // 填写新进程的名字

newPcb->count = count; // 填写新进程的运行时间

newPcb->id = pid++; // 进程id

newPcb->next = NULL; // 填写新PCB的链接指针

//若就绪队列空，则新PCB为ReadyList的第一个结点

if (pReadyList->pcbNum == 0)

{

pReadyList->head = newPcb;

pReadyList->tail = newPcb;

pReadyList->pcbNum++;

}

else//否则，就绪队列不空，新PCB插入就绪队列尾部

{

pReadyList->tail->next = newPcb;

pReadyList->tail = newPcb;

pReadyList->pcbNum++;

}

cout << "New Process Created, Process ID:"

<< newPcb->id << ", Process Name:"

<< newPcb->name << ", Process Length:" << newPcb->count << endl;

cout << "Current ReadyList is:" << endl;

printReadyList();

//向信息文件输出相关信息以便查看程序执行过程

EnterCriticalSection(&cs\_SaveInfo);

log1 << "New Process Created, Process ID:"

<< newPcb->id << ", Process Name:"

<< newPcb->name << ", Process Length:" << newPcb->count << endl;

log1 << "Current ReadyList is:" << endl;

fprintReadyList();

LeaveCriticalSection(&cs\_SaveInfo);

//创建用户线程，初始状态为暂停

newPcb->hThis = CreateThread(NULL, 0, processThread,

newPcb, CREATE\_SUSPENDED, &(newPcb->threadID));

}

else //空闲PCB用完

{

cout << "New process intend to append. But PCB has been used out!" << endl;

EnterCriticalSection(&cs\_SaveInfo);

log1 << "New process intend to append. But PCB has been used out!" << endl;

LeaveCriticalSection(&cs\_SaveInfo);

}

LeaveCriticalSection(&cs\_ReadyList); // 退出临界区

}

// 进程调度

void scheduleProcess()

{

EnterCriticalSection(&cs\_ReadyList);

if (pReadyList->pcbNum > 0) // 就绪队列中有进程则调度

{

runPCB = pReadyList->head; // 调度程序选择就绪队列中第一个进程

pReadyList->head = pReadyList->head->next; //修改就绪队列的头指针

if (pReadyList->head == NULL) // 若就绪队列已空，则需修改其尾指针

pReadyList->tail = NULL;

pReadyList->pcbNum--; // 就绪队列节点数减1

runPCB->count--; // 新进程时间片数减1

runPCB->flag = 1; //进程每次被调度，只显示1次

EnterCriticalSection(&cs\_SaveInfo);

log1 << "Process " << runPCB->id << ':' << runPCB->name << " is to be scheduleed." << endl;

LeaveCriticalSection(&cs\_SaveInfo);

ResumeThread(runPCB->hThis);// 恢复线程(进程)，在本程序中实际上是启动线程运行

runPCB->status = RUN; // 进程状态设置为“RUN”

// 时间片为1s

Sleep(1000); // 等待1秒钟，用此模拟(时间片为1s的)定时中断

EnterCriticalSection(&cs\_SaveInfo);

log1 << "\nOne time slot used out!\n" << endl;

LeaveCriticalSection(&cs\_SaveInfo);

runPCB->status = READY;

SuspendThread(runPCB->hThis); // 当前运行进程被挂起

// 判断进程是否运行完毕

if (runPCB != NULL && runPCB->count <= 0)

{

cout << "\n\*\*\*\*\*\* Process " << runPCB->id << ':' << runPCB->name

<< " has finished. \*\*\*\*\*\*" << endl;

cout << "Current ReadyList is:" << endl;

printReadyList();

cout << "COMMAND>"; cout << flush;

EnterCriticalSection(&cs\_SaveInfo);

log1 << "\*\*\*\*\*\* Process " << runPCB->id << ':' << runPCB->name

<< " has finished. \*\*\*\*\*\*\n" << endl;

log1 << "Current ReadyList is:" << endl;

fprintReadyList();

log1 << flush;

LeaveCriticalSection(&cs\_SaveInfo);

// 终止进程(线程)

if (!TerminateThread(runPCB->hThis, 1))

{ //若终止线程失败，则给出出错信息，结束整个程序

EnterCriticalSection(&cs\_SaveInfo);

log1 << "Terminate thread failed! System will abort!" << endl;

LeaveCriticalSection(&cs\_SaveInfo);

exiting = true; //结束程序

}

else //终止继承操作正确执行

{

CloseHandle(runPCB->hThis);

//终止进程的PCB释放到空白PCB链队列中

returnPcbToFreeList(runPCB);

runPCB = NULL;

}

}

else if (runPCB != NULL) //进程未运行完毕，则将其插入就绪队列

{

if (pReadyList->pcbNum <= 0)//就绪队列为空时的处理

{

pReadyList->head = runPCB;

pReadyList->tail = runPCB;

}

else//就绪队列为不空时将原运行进程的PCB接到就绪队列尾部

{

pReadyList->tail->next = runPCB;

pReadyList->tail = runPCB;

}

runPCB->next = NULL;

runPCB = NULL;

pReadyList->pcbNum++; //就绪队列进程数增1

}

}

else if (pReadyList != NULL) // 清空就绪队列

{

pReadyList->head = NULL;

pReadyList->tail = NULL;

pReadyList->pcbNum = 0;

}

LeaveCriticalSection(&cs\_ReadyList);

}

// 撤销进程

void removeProcess(char\* name)

{

pPCB removeTarget = NULL;

pPCB preTemp = NULL;

EnterCriticalSection(&cs\_ReadyList); //互斥访问就绪队列

// 若撤销的是当前运行进程

if (runPCB != NULL && strcmp(name, runPCB->name) == 0)

{

removeTarget = runPCB;

if (!(TerminateThread(removeTarget->hThis, 1)))

{

cout << "Terminate thread failed! System will abort!" << endl;

EnterCriticalSection(&cs\_SaveInfo);

log1 << "Terminate thread failed! System will abort!" << endl;

LeaveCriticalSection(&cs\_SaveInfo);

exit(0); //结束程序

}

else // 撤销操作成功时

{

CloseHandle(removeTarget->hThis); //关闭进程句柄

returnPcbToFreeList(removeTarget); //该进程的PCB插入空闲PCB队列

runPCB = NULL;

//显示进程已撤销的信息

cout << "\nProcess " << removeTarget->id

<< ':' << removeTarget->name << " has been removed." << endl;

cout << "Current ReadyList is:\n";

printReadyList();

//同时将进程已撤销的信息保存到文件

EnterCriticalSection(&cs\_SaveInfo);

log1 << "\nProcess " << removeTarget->id

<< ':' << removeTarget->name << " has been removed." << endl;

log1 << "Current ReadyList is:\n";

fprintReadyList();

log1 << flush;

LeaveCriticalSection(&cs\_SaveInfo);

LeaveCriticalSection(&cs\_ReadyList);

return;

}

}

// 否则，在就绪队列中寻找要撤销的进程

if (pReadyList->head != NULL)

{

removeTarget = pReadyList->head;

while (removeTarget != NULL)

{

if (strcmp(name, removeTarget->name) == 0)//找到要撤销的进程

{

if (removeTarget == pReadyList->head)//是就绪队列中的第一个进程

{

pReadyList->head = pReadyList->head->next;

if (pReadyList->head == NULL)

pReadyList->tail = NULL;

}

else // 找到的不是就绪队列中第一个进程

{

preTemp->next = removeTarget->next;

if (removeTarget == pReadyList->tail)

pReadyList->tail = preTemp;

}

if (!TerminateThread(removeTarget->hThis, 0))//执行撤销进程的操作

{ //撤销操作失败时的输出信息

cout << "Terminate thread failed! System will abort!" << endl;

EnterCriticalSection(&cs\_SaveInfo);

log1 << "Terminate thread failed! System will abort!" << endl;

LeaveCriticalSection(&cs\_SaveInfo);

LeaveCriticalSection(&cs\_ReadyList);

exit(0); //结束程序

}

//撤销操作成功后的处理

CloseHandle(removeTarget->hThis);

returnPcbToFreeList(removeTarget);

pReadyList->pcbNum--;

cout << "Process " << removeTarget->id

<< ':' << removeTarget->name << " has been removed." << endl;

cout << "currentreadyList is:" << endl;

printReadyList();

EnterCriticalSection(&cs\_SaveInfo);

log1 << "Process " << removeTarget->id

<< ':' << removeTarget->name << " has been removed." << endl;

log1 << "currentreadyList is:" << endl;

fprintReadyList();

log1 << flush;

LeaveCriticalSection(&cs\_SaveInfo);

LeaveCriticalSection(&cs\_ReadyList);

return;

}

else //未找到，继续找

{

preTemp = removeTarget;

removeTarget = removeTarget->next;

}

}

}

LeaveCriticalSection(&cs\_ReadyList);

cout << "Sorry, there's no process named " << name << endl;

return;

}

// 向文件中打印就绪队列信息

void fprintReadyList()

{

pPCB tmp = NULL;

tmp = pReadyList->head;

if (tmp != NULL)

for (int i = 0; i < pReadyList->pcbNum; i++)

{

log1 << "--" << tmp->id << ':' << tmp->name << "--";

tmp = tmp->next;

}

else

log1 << "NULL";

log1 << endl << endl;

}

// 向标准输出打印就绪队列信息

void printReadyList()

{

pPCB tmp = NULL;

tmp = pReadyList->head;

if (tmp != NULL)

for (int i = 0; i < pReadyList->pcbNum; i++)

{

cout << "--" << tmp->id << ':' << tmp->name << "--";

tmp = tmp->next;

}

else

cout << "NULL";

cout << endl;

}

// 打印当前运行进程信息

void printCurrent()

{

if (runPCB != NULL)

cout << "Process " << runPCB->name << " is running..." << endl;

else

cout << "No process is running." << endl;

cout << "Current readyList is:" << endl;

printReadyList();

}

// 结束所有子线程

void stopAllThreads()

{

if (runPCB != NULL)

{

TerminateThread(runPCB->hThis, 0);

CloseHandle(runPCB->hThis);

}

// 结束所有就绪队列中的线程

pPCB q, p = pReadyList->head;

while (p != NULL)

{

if (!TerminateThread(p->hThis, 0))

{

cout << "Terminate thread failed! System will abort!" << endl;

exit(0); //结束程序

}

CloseHandle(p->hThis);

q = p->next;

returnPcbToFreeList(p);

p = q;

}

}

**main.cpp**

#include "schedule.h"

ofstream log1; //保存进程调度信息的文件

volatile bool exiting; //是否退出程序

void helpInfo()

{

cout << "\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\n";

cout << "COMMAND LIST:\n";

cout << "create process\_name process\_length (create p0 8)\n";

cout << "\t append a process to the process list\n";

cout << "remove process\_name (remove p0)\n";

cout << "\t remove a process from the process list\n";

cout << "current\t show current runprocess readyList\n";

cout << "exit\t exit this simulation\n";

cout << "help\t get command imformation\n";

cout << "\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\n\n";

}

int main()

{

char name[20] = { '\0' };

HANDLE hSchedule; //调度线程的句柄

log1.open("Process\_log.txt");

helpInfo();

hSchedule = init(); //hSchedule是调度程序的句柄

if (hSchedule == NULL)

{

cout << "\nCreate schedule-process failed. System will abort!" << endl;

exiting = true;

}

char command[30] = { 0 };

while (!exiting)

{

cout << "COMMAND>";

cin >> command;

if (strcmp(command, "exit") == 0)

break;

else if (strcmp(command, "create") == 0)

{

char name[20] = { '\0' };

int time = 0;

cin >> name >> time;

createProcess(name, time);

}

else if (strcmp(command, "remove") == 0)

{

cin >> name;

removeProcess(name);

}

else if (strcmp(command, "current") == 0)

printCurrent();

else if (strcmp(command, "help") == 0)

helpInfo();

else

cout << "Enter help to get command information!\n";

}

exiting = true;

if (hSchedule != NULL)

{ //无限等待，直到Schedule进程(线程)终止为止

WaitForSingleObject(hSchedule, INFINITE);

CloseHandle(hSchedule);

}

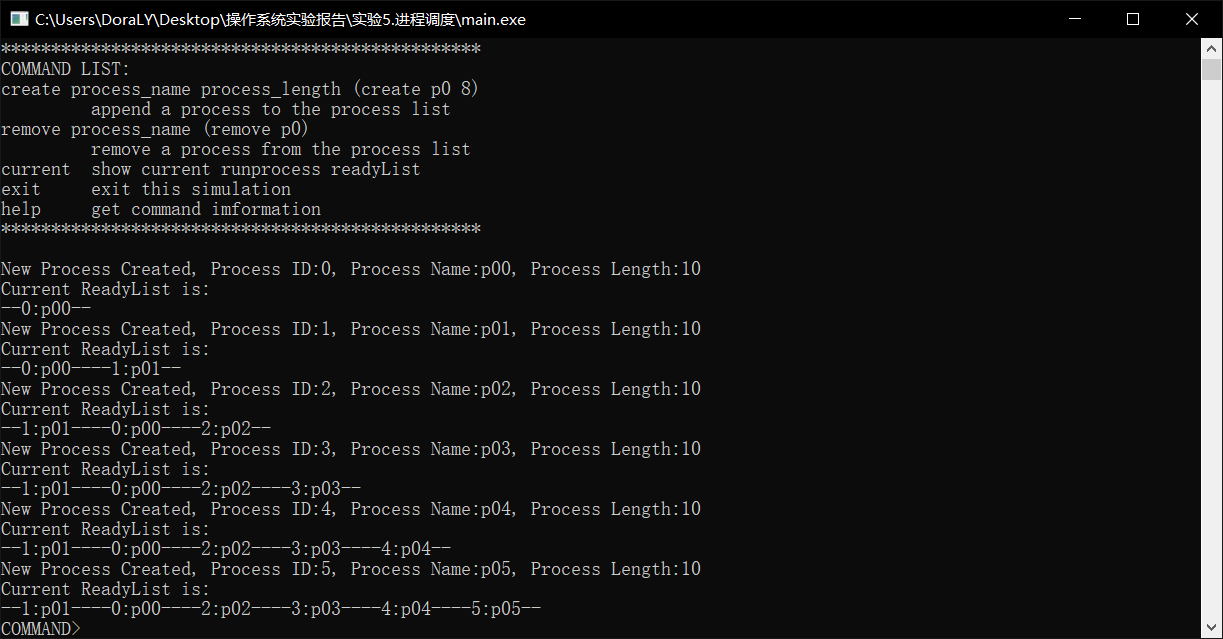
log1.close();

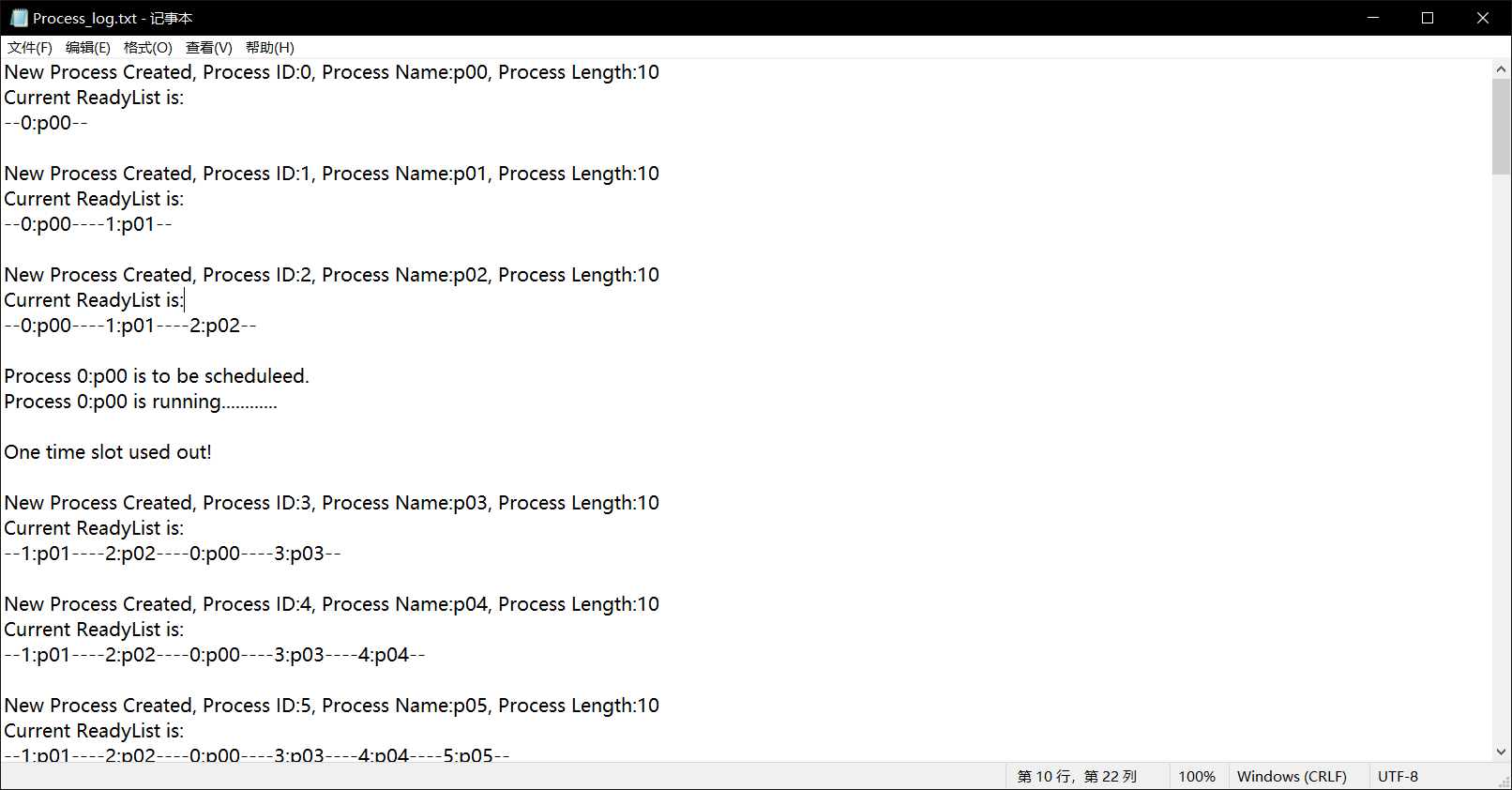
cout << "\n\*\*\*\*\*\*\* End \*\*\*\*\*\*\*\n" << endl;

return 0;

}

1. **实验结果**





1. **实验总结：**

通过这次实验，我学会了如何编码运行一个进程调度程序，同时也学会了完成进程的基础操作如创建、撤销、查看和调度。这个实验涉及了前面所学的进程线程知识，但在前面知识的基础上多了更多复杂的代码和操作，刚开始我怎么运行代码都不成功，后面也是通过自己的检查和思索发现了问题并将它解决掉，这种经历对我代码能力的提升有很大的帮助。在这次实验过程中我学习了怎么实现时间片轮转调度算法，还复习了之前的进程线程的相关操作，同时我也了解并掌握了一些新的WIN32 API函数如TerminateThread()函数，SuspendThread()函数等。我觉得这次实验中，最重要的部分就是进程控制模块和管理队列的结构，这两个部分是代码的核心部分，而且这次实验指导书中画了流程图，我觉得这种方式能够帮助我们更好的理解程序运行的原理和流程，总而言之，这次实验对我的帮助很大，谢谢。