南京農業大學

计算机操作系统课外

实验报告

题　　目: 多进程并发环境模拟以及低级调度算法的仿真实现

姓　　名: 陈 扬

学　　院: 信息科学与技术学院

专　　业: 计算机科学与技术

班　　级: 网络工程 161 班

学　　号: 19316117

指导教师: 姜海燕　　职 称: 教 授

指导助教: 徐 灿

2018 年 11 月 05 日

姓名　陈 扬　　学号 19316117　　申请成绩等级　A +
QQ 号 824678119　　联系电话 17761703297　　E_mail 19316117@njau.edu.cn

一、实验目的

通过程序仿真掌握并发环境、进程 PCB 与控制操作原语、进程切换以及进程调度算法的原理、过程与实现步骤。

二、设计思路与实现内容

（一）设计思路

根据设计要求，提出了以下的设计工作流程以及代码编写时间安排。

1. 研究实验要求，理解每一个要求需要实现的内容及呈现效果。
2. 根据实验要求阅读书籍，了解相关知识。
3. 设计程序框架，理清各个程序模块本身的属性及所提供的功能，和模块之间的相互调用关系。
4. 为每一个模块编写代码，实现各个模块的独立的功能，并完成测试。
5. 设计控制模块，依照原先设计的程序结构，将各个模块连接，并实现 CPU 模式切换、文件读取、进程调度调度等功能。
6. 程序调制及 bug 修复，设计各种不同的情形进行进程调度的测试，与实际计算的值进行比较。
7. 进一步优化程序结构，力求在实现同样功能的前提下减少程序的多线程功能对计算机 CPU 资源及内存资源的占用。
8. 撰写设计报告，提交整套材料，等待答辩。

（二）设计概要

根据设计要求及实际的 CPU 模型，程序设计并实现了以下模块。通过控制模块对这些模块的操作及调用，可以很好地模拟出 CPU 的进程调度情况，并实时地显示出各个进程的执行情况。

这里只对这些模块进行简单的介绍，具体的介绍将会在后面的“四、程序结构及核心模块的实现”小节中做具体的介绍。

1. 时钟中断产生模块

该模块在代码中体现为一个 class 类，名称为“TimeClock”，主要负责时钟中断的产生与中断检测的反馈。该模块利用了多线程的设计思想，将 10ms 的计时与主程序分离开来，能够准确的为主程序提供 10ms 的计时。

2. 文件操作模块

该模块在代码中体现为一个 class 类，名称为“FileOperation”，主要负责对文件（pcb-inputs.txt 文件与 Results.txt 文件）进行读写操作。为了规避存在多个文件指针导致的文件写入错乱、文件拒绝访问等问题，故程序全局只有一个该模块的实例，要读写的文件的文件指针只存在有一个，以保证文件读取与写入的顺序。

3. 任务请求模块

该模块在代码中体现为一个 class 类，名称为“TaskRequest”，主要负责监测作业的请求及作业的调入操作。模块对需要调入的作业的调入时间进行实时监控，每当时间符合时，即开始调入，为其分配 PCB，进入就绪态运行。

4. PCB 结构模块

该模块在代码中体现为一个 class 类，名称为“PCB”，主要负责仿真实现一个

PCB 数据结构中所拥有的全部属性，以及为了访问、修改这些属性所需要的方法。

5. 进程调度模块

该模块在代码中体现为一个 class 类，名称为“ProcessSchedule”，主要负责各个进程的状态检测与进程调度功能。该模块中存在有三个队列，运行队列、就绪队列、等待队列，分别存放三种不同状态的 PCB，根据调度算法等待被调用。

6. 全局常量数据模块

该模块在代码中不再写为 class 类的形式，而是大部分以 constexpr 数据类型的形式给出，这些数据数值规定了一些程序运行所需要用到的常量。为了方便日后程序结构的重新设计与修改，故不将这些数值直接内嵌到代码中，而是独立的在该模块中给出，也可方便开发人员的二次设计。

7. CPU 模块

该模块在代码中体现为一个 class 类，名称为“CPU”，主要负责模拟仿真 CPU，其中含有各个 cpu 运行所需要的属性及访问、修改这些属性所需要的方法。CPU 模块的存在也是为了能够模拟仿真进程的恢复现场与保护现场的操作。

8. 控制模块

该模块在代码中体现为一个 class 类，名称为“Control”，主要负责各个模块之间的协调工作。为了保证程序功能的正常以及读写文件的顺序，故以上所述的所有模块在控制模块中都有唯一的一个实例，控制模块的各个方法只能够操作这些已经声明的实例而不能再另外声明，所有的传递及返回值都以引用的方式给出。

控制模块对其他模块的声明及调度关系如图 2-1 Control 类包含调度所给出。

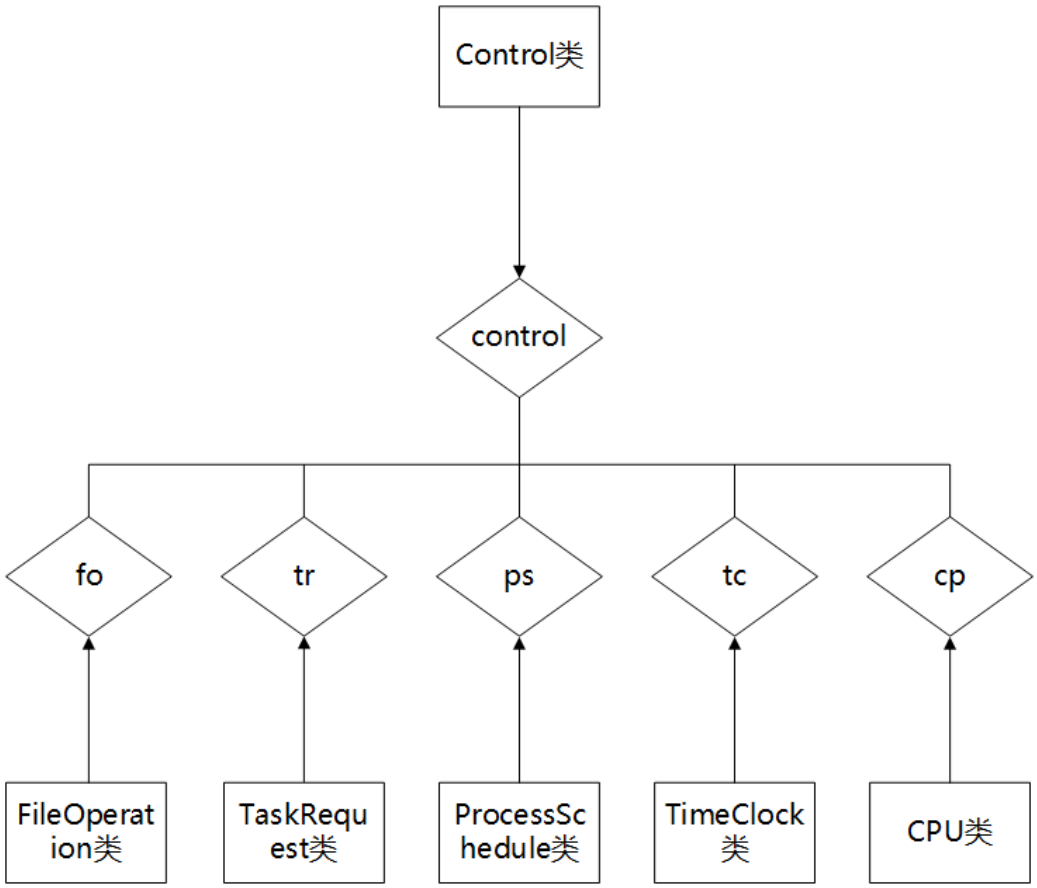

图 2-1 Control 类包含调度

（三）功能设计

为了实现设计要求的各种需求，则需要将以上所给出的模块进行详细的规划设计，在各个模块中尽可能多的提供操作该模块的方法。各个模块设计的越详细具体，最后系统所能够实现的功能也就越多。

此处将给出各个模块中的一些属性与方法的介绍，但只给出属性的定义及方法函数的形式，有些函数的实现过于简单，不做更详细的叙述。有些函数涉及到程序的核心功能，该部分请读者跳转至“四、程序结构及核心模块的实现”阅读。

因为模块的功能设计极大部分都体现在类的设计上面，所以此处直接给出该程序中各个类的设计，读者可以根据相对应的注释理解每个属性及方法的功能。

详情见**附件 1：程序类的设计**

（四）实现内容

1. CPU 的抽象设计

为了仿真真实的进程调度操作，则首先需要仿真一个可用的 CPU。

根据上文的介绍，本程序在设计时，将 CPU 单独抽象成为一个 CPU 模块，在该模块中，定义有 CPU 的不同属性与方法，这些方法为 CPU 的现场保护、现场恢复、中断执行等功能提供了支持。

2. 时钟中断设计

为了模拟仿真 10ms 间隔的时钟中断，故设计时钟中断模块，在该模块中，有一个专门的函数，被设计为多线程的一部分，专用来发生 10ms 的中断。

同时，该模块还提供中断信号的监测函数。即每当发生 10ms 一次的中断时，发生函数将模块中的 if_break 变量设置为 true，监测函数响应外部的请求，返回 if_break 的值，当返回后，再将 if_break 变量初始化，这样的功能分离的设计，可保证 10ms 间隔的准确。

3. PCB 模块设计

PCB 模块的数据结构已经在上文的“功能设计”中详细给出，即 class PCB 中的内容。因为考虑到需要保证 PCB 的稳定性，使得外部操作不能够轻易的改变 PCB 的内容，故放弃结构体的设计，转而采用 class 类的设计。

PCB 中基本的数据如下：

进程编号（ProID）、进程优先数（Priority）、进程创建时间（InTimes）、进程状态（ProState）、进程运行时间（RunTimes）、进程包含的指令数目（ InstrucNum）、编号指令（ Instruc_ID ）、每条指令的状态标志（Instruc_State，0 表示核心态、1 表示用户态、2 表示 PV 操作）、单指令运行时间（Instruc_Times）、进程当前状态（PSW）等信息。

请求运行的并发进程个数随机产生 5-10 以内随机整数，即说明在该程序中，总共最少执行 5 个进程，最多执行 10 个进程，这些进程之间一定是并发执行的。

进程创建时间（InTimes）：程序运行从 0 开始计时，生成进程的时间间隔控制在 1 分钟以内，需要保证出现进程并发情景。

进程运行时间（RunTime）：统计记录进程当前已运行了多少时间，此字段开始时为空，进程运行过程中不断保存和记录。

进程包含的指令数目（InstrucNum）：用 5-20 以内的随机整数产生；

编号指令（Instruc_ID）：进程所执行指令序号，根据 InstrucNum 数字生成。

本试验假设 CPU 在执行一条机器指令时必须完成，不可以中断。

每条指令的类型标志（Instruc_State，0 表示系统调用、1 表示用户态计算操作、2 表示 PV 操作）：用随机数产生 0、1、2。

当 Instruc_State=0 时，发生系统调用，CPU 进行模式切换，原进程进入阻塞

态；

当 Instruc_State=2 时，发生系统调用，进程进入阻塞态；

指令运行时间（Instruc_Times）：该条指令完成需要的时间，本实验假设 CPU 在执行一条机器指令时必须完成，不可以中断。

当 Instruc_State=0 或 1 时, Instruc_Times=产生[10-40]之间的随机 10ms 倍数的整数；

当 Instruc_State=2 时, Instruc_Times=50，用户进程发生 PV 阻塞请求，并假设完成 I/O 数据通信时间为 50ms，之后可唤醒。

PSW 中保存该进程当前执行的指令编号。

4. 系统 PCB 表设计

系统 PCB 表使用数组进行实现，根据题目的要求，整个运行期间的进程个数都是小于等于 10 的，所以此处只需要使用数组实现即可。

5. 进程控制原语

在 PCB 类中同时提供进程控制原语，即进程创建、进程撤销、进程阻塞、进程唤醒的原语。

6. 进程上下文切换

程序实现了进程的上下文切换，在基本要求的基础上，添加了阻塞态（等待态）的模式，对于 2 型（PV 操作），可以自动识别并立即将此进程加入阻塞队列等待，50ms 后唤醒进入就绪态。0 型（系统调用）与 1 型（用户计算）指令运行在就绪态与运行态之间。

7. 处理机模式切换

0 型（系统调用）与 2 型（PV 操作）指令都需要用到系统调用，当 CPU 运行这些指令时，将会自动识别并将 CPU 的模式切换为核心态运行。1 型（用户计算）指令，要求 CPU 工作在用户模式，此时 CPU 将会切换到用户模式运行。

8. 请求序列生成

程序利用自己的随机数算法，依据用户输入的总进程数量，严格按照一定的格式生成 PCB，并生成一定的序列导入。

9. 并发进程动态监测

程序事先读取所有的 PCB 序列，并将他们的调入时间进行记载。当程序在运行时，实时监测当前系统时间与某一个 PCB 的进入时间是否匹配，如果匹配，则立即调入，为期分配可用的 PCB，并进入就绪队列进行等待；如果不匹配，则继续等待，直至所有请求的 PCB 全部被调入才停止。

10. 调度算法

本程序中很好地实现了 Unix 的新进程调度算法。由于时间片轮转算法、时间片轮转+优先数调度算法、动态优先级算法的核心思想在 Unix 调度算法中都有体现，故此程序直接实现了 Unix 算法，而没有实现前面三种较简单的算法。

在 Unix 调度算法中，进程占用 CPU 的时间片的时机由每个进程的优先数来确定，规则如下：

① 进程的优先数计算公式：p-pri=(p-cpu/2)+用户基本优先数

② 用户基本优先数定位 60，核心态进程优先数在 60 以下，而用户态进程至少为 60，故前者优先权高于后者。

③ 每到 1 秒时，系统时钟中断处理程序会根据衰减函数计算 p-cpu：decay(p-cpu)=p-cpu/2

④ 一个新进程，未使用处理器，所以，(p-cpu/2)=0，新建进程最有可能被调

度。

⑤ 系统时钟中断处理程序每工作一次，将当前运行进程的 p-cpu 加 1，进程的优先数 p-pri 就会增大，随着进程占用处理器的时间不断地增加，它的优先权便会逐步降低。

⑥ 每到 1 秒时，系统就依次检查并计算所有进程的 p-cpu，当进程的 p-cpu 增加到一定程度时，在钟中断处理可能被其他的 p-cpu 较小的进程抢先，将进入到下一个等待调度队列的轮转中。

⑦ 每个时钟中断系统，都会对每个进程的 p-cpu 进行除 2 操作，对不占用处理器的进程来说 p-cpu 的值衰减很快，因此其优先数会不断发生变化。当值小于前被调度的进程及其他进程时，它就有机会重新占处理器并被调度执行。

11. 运行记录保存

程序会将进程事件请求调度的原始信息、进程执行状态、CPU 状态的变化情况、进程调度序列、每个进程的指令执行序列和情况、每个进程获得 CPU 的调度时间、周转时间、带权周转时间以及 CPU 利用率等动态信息加标记保存为进程运行记录文件。

12. 运行记录显示

程序会将进程请求调度信息、进程调度运行记录信息、CPU 寄存器信息等加以解释说明，显示在屏幕上。

三、公共数据结构设计与实现

（一）PCB 类

为了模拟仿真 PCB 模块的设计，故设计 PCB 的数据结构。同时，考虑到 PCB 中含有进程的控制原语以及一些为了保护数据而设计的方法，综合考虑下，将 PCB 设计为类，PCB 的属性定义为类的私有属性。其结构如下：

```
class PCB
{
private:
    int ProID;                      //进程编号
    int Priority;                   //进程优先数
    int InTimes;                    //进程创建时间
    int ProState;                   //进程状态（就绪、运行、等待）
    int RunTimes;                   //进程需要运行时间
    int InstrucNum;                 //进程中包含的指令数目
    int PSW;                        //程序状态字，进程当前执行的指令编号
    PCB_Instruc Instruc[MAX_INSTRUC_NUM];       //指令内容
    int p_pri;                      //Unix调度算法需要变量
    int p_cpu;                      //Unix 调度算法需要变量
public:
    ……
};
```

其中，对于指令内容的数据结构设计使用了结构体的数据结构，定义如下：

```
struct PCB_Instruc
{
    int Instruc_ID;           //指令编号
```

```
    int Instruc_State;          //指令类型
    int Instruc_Times;          //指令运行时间
};
```

PCB 的控制原语可写在其 public 部分，作为其对外的一个接口使用，方便 Control 调用以改变其自身的属性。

（二）PCB 任务队列

使用数组来模拟 PCB 任务队列，根据设计要求，队列中最多有 10 个进程。即设置数组的最大长度为 10，每一个元素为 Control 类的一个实例。因为每一个元素在开始模拟运行前已经申请空间，所以他们的位置都固定不变。每一次有新的任务进入时，需要扫描数组，找出可用的位置分配。数组的基本操作如图 3-1 PCB 任务队列所示。

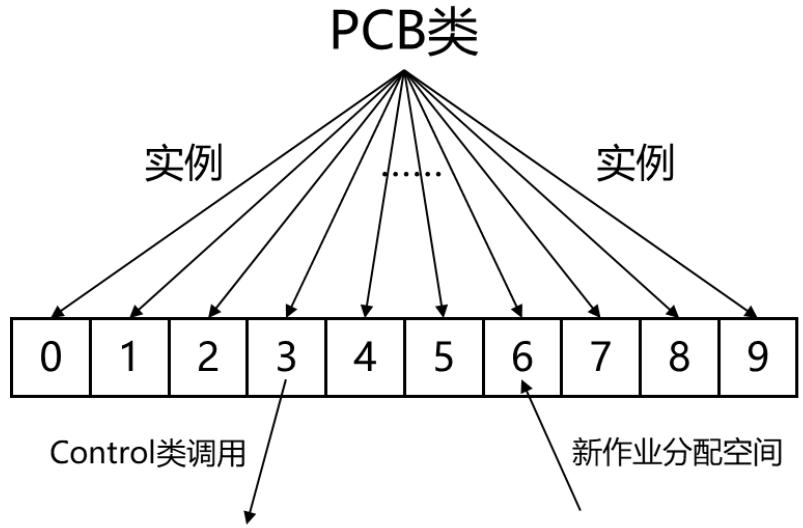

图 3-1 PCB 任务队列

（三）指令状态枚举体

定义指令状态的枚举体类型，对程序运行时可能会碰到的三种类型进行辨别。分别为：0 型—系统调用，1 型—用户操作，2 型—PV 操作。

定义的形式如下：

```
enum INSTRUC_STATE
{
    INSTRUC_STATE_SYSTEMCALL,           //指令类型—系统调用
    INSTRUC_STATE_USERMOD_OPERATION,    //指令类型—用户态计算操作
    INSTRUC_STATE_PV_OPERATION          //指令类型—PV操作
};
```

（四）PCB 状态枚举体

定义 PCB 状态的枚举体类型，对程序运行时的 PCB 实例进行标注，以帮助程序、CPU 更好的执行相应的操作。在 PCB 类中，也有相对应的方法对该变量进行修改与访问。

定义的形式如下：

```
enum PCB_STATE
{
    PCB_STATE_READY,            //PCB状态—就绪态
```

```
    PCB_STATE_RUNNING,        //PCB状态—运行态
    PCB_STATE_WAIT,           //PCB状态—等待态
    PCB_STATE_OVER            //PCB状态—结束
};
```

（五）CPU 状态枚举体

定义 CPU 状态的枚举体，对 CPU 的运行状态进行标注。CPU 是一个运行的、动态的结构，其状态会不断的改变。利用该枚举体及相关的检测函数、状态修改函数，可以很好地实时反映出 CPU 的当前状态，以供使用者查阅与查错。

CPU 的状态只有两个状态，当运行 0 型与 2 型指令时，CPU 切换到核心态，当运行 1 型指令时，CPU 切换到用户态。

定义的形式如下：

```
enum CPU_STATE
{
    USER_MODE,
    CORE_MODE
};
```

（六）程序全局常量参数

定义了一些程序启动时所需要使用的参数，这些参数都被定义成常量类型，用 constexpr 进行标记。这样设置，有利于程序结构的修改，方便日后开发人员的二次修改。

定义如下：

```
constexpr int BREAK_TIME_LENGTH = 10;          //中断产生时间
constexpr int TIMESLICE_TIME_LENGTH = 1000;    //时间片产生时间
constexpr int PCB_INIT_NUM = -1;               //PCB的初始化数值
constexpr int INSTRUC_INIT_NUM = -1;           //指令的初始化数值
constexpr int SYSTEM_MAX_PCB_NUM = 10;         //系统最大的PCB表的数量
constexpr int MAX_PRIORITY_NUM = 127;          //系统最大的优先数
constexpr int MAX_INTIMES_INTERVAL = 20;  //两个PCB产生之间的最大时间间隔
constexpr int MIN_INTIMES_INTERVAL = 10;  //两个PCB产生之间的最小时间间隔
constexpr int MAX_INSTRUC_NUM = 20;       //一个PCB中包含的最大的指令数目
constexpr int MIN_INSTRUC_NUM = 5;        //一个PCB中包含的最小的指令数目
constexpr int MAX_INSTRUC_TIMES = 4;           //一条指令最多需要运行的时间
constexpr int MIN_INSTRUC_TIMES = 1;           //一条指令最少需要运行的时间
constexpr int PV_INSTRUCTION_TIME = 50;        //PV 操作指令的时间
```

（七）调度队列

程序统一使用了队列的数据结构，来表示调度算法中所涉及到的三个队列：运行队列、就绪队列、等待（阻塞）队列。每一个队列在程序运行时都会被重复使用，以动态地展现调度算法的具体过程。

queue 模板类的定义在<queue>头文件中，是属于 STL 标准库的一个文件。

queue 的基本操作有：

入队，如：q.push(x)，将 x 接到队列的末端。

出队，如：q.pop()，弹出队列的第一个元素，注意，并不会返回被弹出元素的值。

访问队首元素，如：q.front()，即最早被压入队列的元素。

访问队尾元素，如：q.back()，即最后被压入队列的元素。
判断队列空，如：q.empty()，当队列空时，返回 true。
访问队列中的元素个数，如：q.size()，返回当前队列中的元素个数。
队列的基本结构及操作如图 3-2 队列结构及基本操作所示。

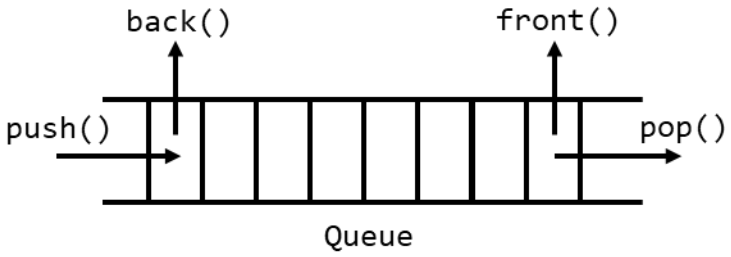

图 3-2 队列结构及基本操作

同时，因为队列的特性，无法从中取得特定位置的元素，所以需要自己另外编写代码，以实现对队列中的某一个元素进行删除操作。故编写代码如下，通过此代码，可实现特定序号的元素的删除。具体代码如下：

```
void OutQueue(int out_id)
{
    for (int i = 1; i <= queue.size(); i++)
    {
        PCB &temp = queue.front();
        queue.pop();
        if (temp.GetProID() != out_id + 1)
            running_queue.push(temp);
    }
}
```

同时，为了展示每个队列中所拥有的 PCB 的信息，故需要再编写函数将这些信息全部输出。该函数的实现原理特定元素删除的算法相似，也是通过不断的取元素弹出再入队列的操作实现，具体代码如下：

```
string ShowQueue()
{
    string str = "";
    for (int i = 1; i <= queue.size(); i++)
    {
        PCB &temp = queue.front();
        str += to_string(temp.GetProID()) + "  ";
        queue.pop();
        queue.push(temp);
    }
    str += "\n";
    return str;
}
```

有了这三个队列，便可以实现进程运行态、就绪态、阻塞态的切换，进程切换发生的时机如图 3-3 进程切换时机所示：

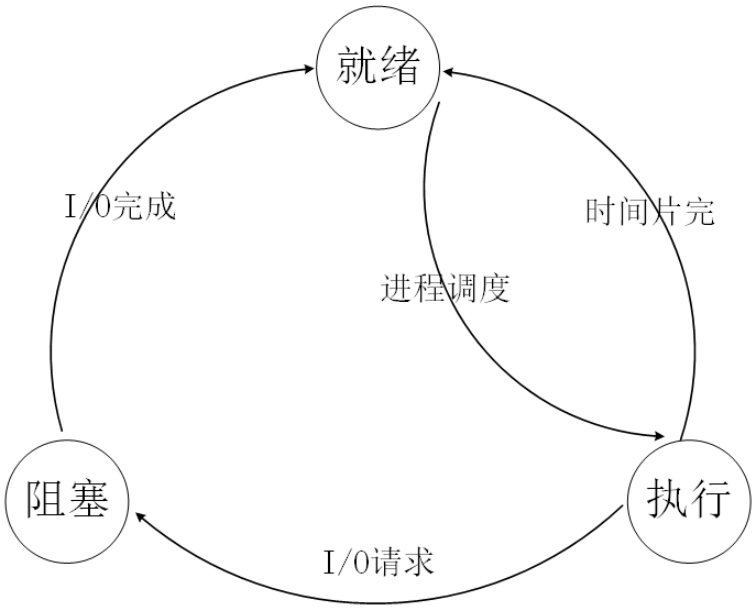

图 3-3 进程切换时机

（八）其他数据结构

上文已经详细描述了程序所使用的基本数据结构。另外，还有一些数据结构的定义与类的定义重复，此处不再赘述。程序的所有类的定义在“附件 1：程序类的设计”中已经全部给出，读者可以参考并查找并未列出的数据结构。

四、程序结构及核心模块的实现

（一）程序结构

1. 程序框架

报告的上文已经详细介绍了程序的模块设计，列出了程序中所有的模块。同时，在 Control 类中，将这些模块全部引入，除了全局常量参数之外，其他的模块类都在 Control 类中有一个唯一的实例。

根据以上的叙述，读者可以大概想象出该程序的程序框架。参考上文的图 2-1，再加入“全局常量参数”模块，即为该程序的程序框架。如图 4-1 程序框架所示。

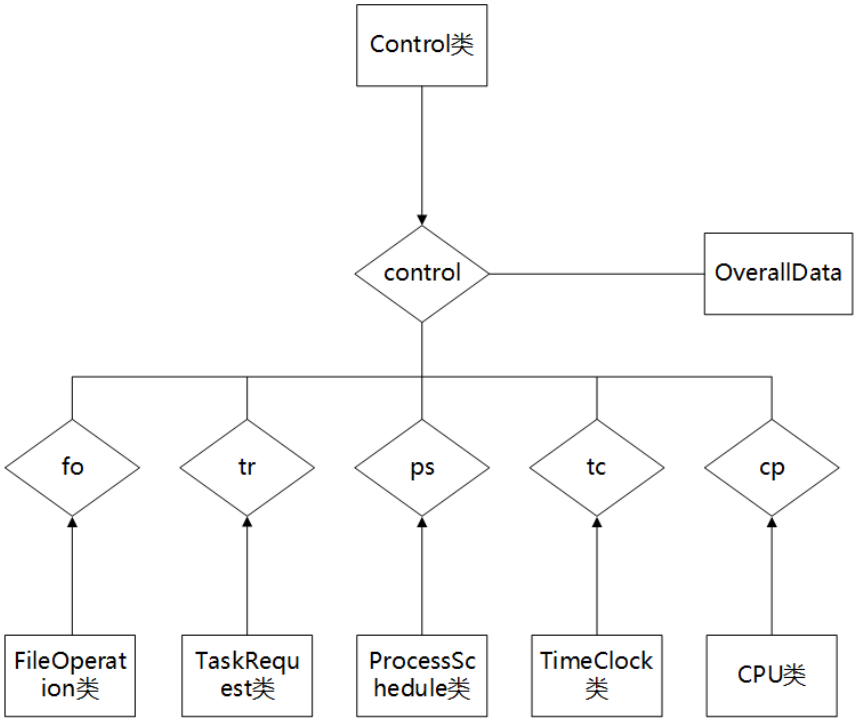

图 4-1 程序框架

2. 程序运行逻辑

当可执行文件 exe 被打开后，程序立即开始运行。

程序的功能选择是由菜单提供的，用户输入指令后，程序会检测输入的序号并启动相应的操作，菜单的功能设计将会在下文详细描述。

程序的核心功能为模拟多进程并发环境模拟以及低级调度算法的仿真实现，下面给出程序该功能的运行逻辑，如图 4-2 程序运行逻辑 所示。

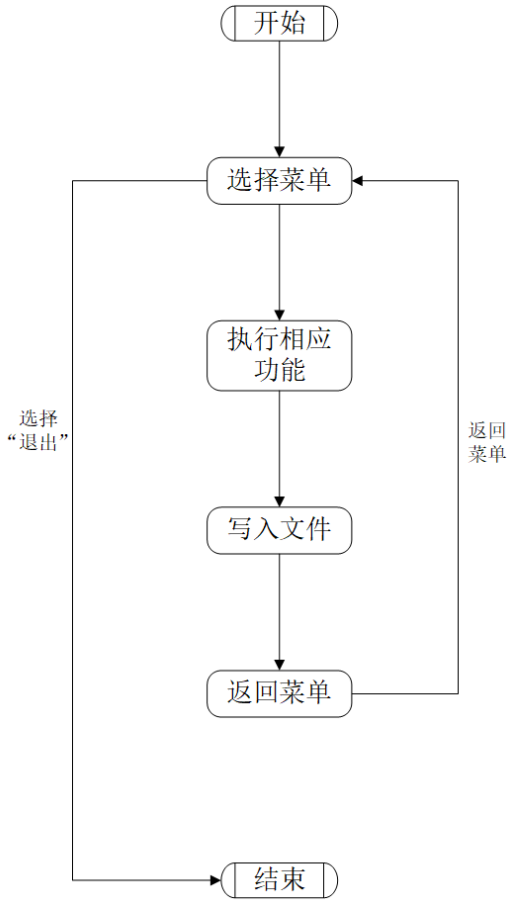

图 4-2 程序运行逻辑

程序主要功能：并发环境模拟与进程调度的流程图如图 4-3 并发环境模拟与进程调度流程图所示。

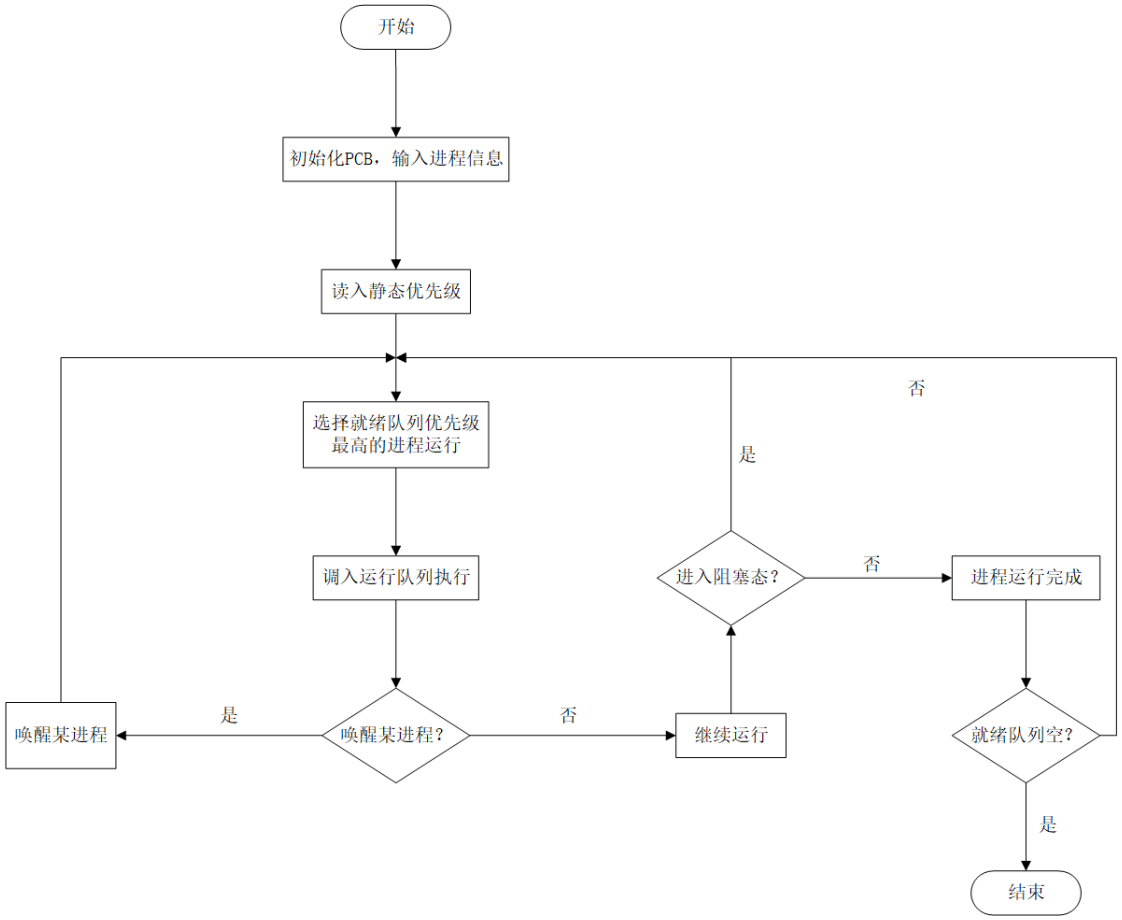

图 4-3　并发环境模拟与进程调度流程图

（二）时钟中断产生模块

1. 10ms 中断产生

使用多线程技术，结合 Windows.h 库中的 Sleep()系统函数调用，可以仿真模拟程序的 10ms 时间的间隔并发出信号。

程序的 10ms 信号不直接作为返回值传递，而是修改 class 类中的 bool 类型变量 if_break 为 true。程序代码如下：

```
void TimeClock::CreateBreak()
{
    while (1)
    {
        Sleep(BREAK_TIME_LENGTH);     //等待时间
        if_break = TRUE;              //产生中断
    }
}
```

代码中，BREAK_TIME_LENGTH 为全局的常量参数，程序中默认设置为 10，即代表 10ms 的间隔时间。

2. 中断检测函数

中断产生模块为了能够与程序主模块并行执行，所以需要将 if_break 变量单独的保护起来，只提供一个唯一的函数接口进行调用，该函数的返回值即为是否产生中断的标记。当该函数检测到程序发生中断后，立即将 if_break 变量设置为 false，以便与中断产生函数不发生冲突，及时复位。程序代码如下：

```
void TimeClock::ResetBreak()
{
    if_break = FALSE;
}

bool TimeClock::GetIfBreak()
{
    return if_break;
}
```

（三）文件操作模块

1. PCB 信息读取

程序开始运行时，需要提前从文件中读取全部的 PCB 信息，但又因为在 txt 文件中，PCB 信息有自己的组织方式，无法直接读取，故需要依据其特定格式，逐个读取。并根据 InstruNum 数值，继续判断指令的条数，从而读取到完整的 PCB 信息。

在 txt 文件中，PCB 文件的表头信息如下：

ProID　Priority　InTimes RunTimes　InstrucNum　PSW　Instruc_ID　Instruc_State　Instruc_Times

依据该格式，即可写出相应的读取函数：

```
PCB FileOperation::ReadOnePCBFromFile()
{
    int data[6], instruc[MAX_INSTRUC_NUM][3];
    //如果是第一次读取，则将表头给读取
    if (if_firsttime_read == true)
    {
        fscanf(read_txt,
        "ProID\tPriority\tInTimes\tRunTimes\tInstrucNum\tPSW\tInstruc
        _ID\tInstruc_State\tInstruc_Times\n");
        if_firsttime_read = false;
    }
    fscanf(read_txt,
    "%d\t\t%d\t\t\t%d\t\t%d\t\t\t%d\t\t\t%d\t%d\t\t\t%d\t\t\t\t%d\n",
    &data[0], &data[1], &data[2], &data[3], &data[4],
    &data[5],&instruc[0][0],&instruc[0][1],&instruc[0][2]);
    read_from_file.SetProID(data[0]);
    read_from_file.SetPriority(data[1]);
    read_from_file.SetInTimes(data[2]);
    read_from_file.SetRunTimes(data[3]);
    read_from_file.SetInstrucNum(data[4]);
    read_from_file.SetPSW(data[5]);
    for (int i = 1; i < read_from_file.GetInstrucNum(); i++)
    {
        fscanf(read_txt,
        "\t\t\t\t\t\t\t\t\t\t\t\t%d\t\t\t%d\t\t\t\t%d\n",
```

```
            &instruc[i][0], &instruc[i][1], &instruc[i][2]);
        }
        for (int i = 0; i < read_from_file.GetInstrucNum(); i++)
        {
            read_from_file.SetInstrucID(i, instruc[i][0]);
            read_from_file.SetInstrucState(i, instruc[i][1]);
            read_from_file.SetInstrucTimes(i, instruc[i][2]);
        }
        return read_from_file;
    }
```

2. PCB 信息写入

当程序自动生成新的 PCB 时，需要将其写入到文件中，PCB 的文件头的格式与上文所述读取的格式相同。依据改格式，可写出代码：

```
    void FileOperation::WritePCBToFile()
    {
        if (if_write_pcb_head == false)
            WriteHeadToFile();
        fprintf(write_txt,
        "%d\t\t%d\t\t\t%d\t\t%d\t\t\t%d\t\t\t%d\t%d\t\t\t%d\t\t\t\t%d\n",
        read_from_file.GetProID(), read_from_file.GetPriority(),
        read_from_file.GetInTimes(),read_from_file.GetRunTimes(),
        read_from_file.GetInstrucNum(),
        read_from_file.GetPSW(),read_from_file.GetInstrucID(0),
        read_from_file.GetInstrucState(0),
        read_from_file.GetInstrucTimes(0));
        for (int i = 1; i < read_from_file.GetInstrucNum(); i++)
        {
            fprintf(write_txt, "\t\t\t\t\t\t\t\t\t\t\t\t\t\t
            \t%d\t\t\t%d\t\t\t\t%d\n",
                read_from_file.GetInstrucID(i),
                read_from_file.GetInstrucState(i),
                read_from_file.GetInstrucTimes(i));
        }
    }
```

3. 运行结果写入

程序需要将程序模拟并发环境及调度的实时结果写入到文件中，需要专门的函数进行写入。同时，又因为在 Control 类中只有 FileOperation 模块的唯一实例，所以其他的函数需要写入文件时，必须调用该函数，能够保证文件指针与文件写入位置的顺序。

```
    void FileOperation::WriteResults(string data)
    {
        ofstream out("19316117_RunResults.txt", ios::app);
        out << data;
        out.close();
```

```
}
```

4. 运行结果显示

程序需要读取位于当前目录下的 Results.txt 文件，并显示该内容。读取的操作可以使用 fgetc()函数完成，通过循环判断是否到达文件尾（feof()函数）即可。

```
void FileOperation::ShowRunResults()
{
    run_results = fopen("19316117_RunResults.txt", "a+");
    char s[1000], ch;
    int i = 0;
    while ((ch = fgetc(run_results)) != EOF)
        s[i++] = ch;
    s[i] = '\0';
    printf("%s\n", s);
    fclose(run_results);
}
```

（四）任务请求模块

1. PCB 完成检测

对进程进行模拟调度，其结束条件便是所有的进程全部结束。要判断所有读入的进程是否已经结束，则需要编写函数，每一次对存放 PCB 的数组进行遍历操作，判断已经分配空间的进程是否已经运行结束。

```
bool TaskRequest::IfAllOK()
{
    for (int i = 0; i < SYSTEM_MAX_PCB_NUM; i++)
    {
        if ((system_PCB[i].GetIfEnd() == false) &&
        (system_PCB[i].GetIfUsed()==true))
            return false;
    }
    return true;
}
```

2. 作业进入时间监测

当程序读取完全部的 PCB 信息后，便开始根据当前的 cpu 时间实时监测，检查等待调入的 PCB 中是否有与当前时间符合的，若有，则立即为其分配空间，调入到就绪队列，若没有，则继续等待，直到调入完成所有的 PCB 为止。

```
bool TaskRequest::TaskJoinCheck(int time,int& num)
{
    for (int i = 0; i < SYSTEM_MAX_PCB_NUM; i++)
    {
        if (time == system_PCB[i].GetInTimes())
        {
            num = i;
            return true;
        }
    }
```

```
    return false;
}
```

3. 新 PCB 生成

程序同时提供了随机生成 PCB 信息的功能，用户输入指定的 PCB 数量，则可以随机生成 PCB 信息，并写入到文件中。

该功能随机生成的 PCB 信息之间的间隔时间、最少指令数量、最多指令数量、最少指令运行时间、最多指令运行时间等信息都在全局常量参数中有定义，读者可以阅读附件一。

```
PCB TaskRequest::CreateNewPCB()
{
    PCB temp;
    temp.SetProID(PCB_num++);                    //设置PCB编号
    srand((unsigned int)__rdtsc());              //设置随机数种子
    temp.SetPriority(rand() % MAX_PRIORITY_NUM);//生成0-5的随即优先数
    if (PCB_num - 1 == 1)                //使用第一个PCB，没有上一个
        temp.SetInTimes(0);
    else       //使用的不是第一个PCB，则有上一个PCB可供参考
    {
        temp.SetInTimes(last_one.GetInTimes() + ((rand() %
(MAX_INTIMES_INTERVAL - MIN_INTIMES_INTERVAL + 1)) +
MIN_INTIMES_INTERVAL)*10);
    }
    temp.SetInstrucNum((rand() % (MAX_INSTRUC_NUM - MIN_INSTRUC_NUM +
1)) + MIN_INSTRUC_NUM);       //设置指令个数
    temp.SetPSW(1);
    int all_time = 0;
    for (int i = 0; i < temp.GetInstrucNum(); i++)
    {
        temp.SetInstrucID(i, i + 1);
        temp.SetInstrucState(i, rand() % 3);
        temp.SetInstrucTimes(i, ((rand() % (MAX_INSTRUC_TIMES -
        MIN_INSTRUC_TIMES + 1)) + MIN_INSTRUC_TIMES) * 10);
        if (temp.GetInstrucState(i) == 2)
            temp.SetInstrucTimes(i, PV_INSTRUCTION_TIME);
        all_time += temp.GetInstrucTimes(i);
    }
    temp.SetRunTimes(all_time);
    last_one = temp;
    return temp;
}
```

4. PCB 可用检测

当 PCB 被调入或完成时，都会有特定的变量来记录该 PCB 是否可用及是否已经完成，为 bool 类型变量 if_used 及 if_end。

通过检测这两个变量，即可判断该 PCB 是否被使用或已经完成。

```
bool TaskRequest::CheckIfJoin(int no)
{
    if (system_PCB[no].GetIfUsed()==true)
        return true;
    else return false;
}
```

（五）PCB 模块

1. p-cpu 自增

根据 Unix 算法，每 10ms 中断产生时，将所有位于队列中的 PCB 的 p-cpu 值加 1，根据该算法思想，可写出如下代码。

```
void PCB::P_CPU_Add()
{
    if (GetIfUsed() == true && GetIfEnd() != true)
        p_cpu++;
}
```

2. p-cpu 衰减

根据 Unix 算法，每次当时间片到来时，将现在正在运行队列中的 PCB 的 p-cpu 值进行衰减，每次衰减的量为原值的一半，根据该算法思想，可写出如下代码：

```
void PCB::PCPU_DECAY()
{
    if (GetIfUsed() == true && GetIfEnd() != true)
        p_cpu = p_cpu / 2;
}
```

3. p-pri 计算

根据 Unix 算法，每次当时间片到来时，依据 PCB 现在的 p-cpu 值，重新计算得到 p-pri 的值，并据此进行重新调度，计算的公式为：

p-pri=(p-cpu/2)+用户基本优先数

根据该公式，即可写出代码。

```
void PCB::Cal_P_PRI()
{
    p_pri = (p_cpu / 2) + GetPriority();
}
```

4. 进程控制原语

进程控制原语包括进程创建、进程撤销、进程阻塞、进程唤醒。因为程序的 PCB 模块中已经有了相关的变量来标识该 PCB 的信息，所以这些控制原语只需要改变这些变量即可。同时，进程控制原语集成在 PCB 模块中，作为 PCB 模块的一个方法被外部调用。

进程创建原语：

```
void PCB::ProcessCreate()
{
    if_used = true;
    if_end = false;
}
```

进程撤销原语：

```
void PCB::ProcessCancel(int time)
{
    if_used = true;
    if_end = true;
    this->end_time = time;
    this->SetProState(PCB_STATE_OVER);
}
```

进程阻塞原语：

```
void PCB::ProcessBlocked()
{
    this->ProState = PCB_STATE_WAIT;
}
```

进程唤醒原语：

```
void PCB::ProcessWakeUp()
{
    this->ProState = PCB_STATE_READY;
}
```

（六）进程调度模块

1. 加入队列

程序运行时，需要对队列进行加入操作，同时，还需要考虑当前队列中是否已经有该序号的元素。所以，在加入前，必须事先检测，检测通过之后，才能够加入。

```
void ProcessSchedule::JoinQueue(PCB &data, FileOperation &fo, int
cpu_time)
{
    bool if_exist = false;
    for (int i = 1; i <= queue.size(); i++)
    {
        PCB &temp = queue.front();
        if (temp.GetProID() == data.GetProID())
        {
            if_exist = true;
            break;
        }
        queue.pop();
        queue.push(temp);
    }
    if (if_exist == false)
    {
        queue.push(data);
        data.SetProState(PCB_STATE_RUNNING);
        string str = "";
        str = str + "cpu时间：" + to_string(cpu_time) + "，进程" +
```

```
            to_string(data.GetProID()) + "调入运行队列！\n";
            cout << str;
            fo.WriteResults(str);
        }
    }
```

2. 从队列移除元素

当 PCB 的状态转换时，需要从当前队列中移出。但是，因为队列的特性，只能够先进先出，无法直接移出指定位置的元素。所以，需要编写代码实现从队列中删除指定序号的元素。该算法利用了队列先进先出的特性，每一次弹出一个元素进行比较，若不是，则将其插入到队列的最后，若是，则不将其插入到最后。

```
    void ProcessSchedule::OutQueue(int out_id)
    {
        for (int i = 1; i <= queue.size(); i++)
        {
            PCB &temp = queue.front();
            queue.pop();
            if (temp.GetProID() != out_id + 1)
                queue.push(temp);
        }
    }
```

3. 展示队列元素

程序运行时，需要实时查看当前三个队列的情况。同样的，因为队列先进先出的特定，无法选择指定位置的元素查看，所以需要像上文一样，逐个弹出读取信息，读取完毕后再重新加入到队列的最后，以保证队列的完整性不被破坏。

```
    string ProcessSchedule::GetAllQueueInfo()
    {
        string str = "";
        for (int i = 1; i <= queue.size(); i++)
        {
            PCB &temp = queue.front();
            str += to_string(temp.GetProID()) + "  ";
            queue.pop();
            queue.push(temp);
        }
        str += "\n";
        return str;
    }
```

（七）CPU 模块

1. 恢复现场

当进程进入到 CPU 后，即进入了运行态。此时，将 PCB 的信息控制权交给 CPU 处理，需要实现恢复现场，将 PCB 内部的一些必要信息提供给 CPU，更新 CPU 当前的信息。

```
    void CPU::RecoverSpot(PCB& data)
    {
```

```
    SetStorePCB(data);
    this->PC = store.GetProID();
    this->PSW = store.GetPSW();
    PCB_Instruc temp;
    temp.Instruc_ID = store.GetInstrucID(PSW);
    temp.Instruc_State = store.GetInstrucState(PSW);
    temp.Instruc_Times = store.GetInstrucTimes(PSW);
    this->IR = temp;
    this->current_time = store.GetCurrentRunTime();
}
```

2. 保护现场

当 CPU 运行完毕或者发生进程切换时，CPU 需要将当前已经处理的信息保存到原来的 PCB 中，才能够接受下一个调入的 PCB。

```
void CPU::ProtectSpot(PCB& data)
{
    data.SetPSW(this->PSW);
    data.SetCurrentTime(this->current_time);
}
```

3. 刷新 PSW 状态

当 CPU 发生保护现场操作前，需要更新当前的 IR 指针，及目前所运行到的指令编号。该算法根据每一条指令所占用的时间以及当前已经运行的时间计算得到当前已经运行到的指令编号。

```
void CPU::RefreshPSW()
{
    int i =  0, sum = 0;
    while (sum < current_time)
    {
        sum += store.GetInstrucTimes(i);
        i++;
    }
    PSW = (i == 0 ? 1 : i);
}
```

4. 检测是否允许执行下一条指令

根据设计要求，每一条指令的运行不能够被打断，包括时间片切换。为了能够避免这种冲突，将程序设计为：若下一条指令不够时间运行时，则直接放弃执行，等待时间片的轮转切换。

```
bool CPU::CheckIfBorder()
{
    if (current_time == 0 && PSW == 1)
        return true;
    for (int i = 1, sum = 0; i <= store.GetInstrucNum(); i++)
    {
        sum += store.GetInstrucTimes(i - 1);
        if (current_time == sum)
```

```
            return true;
        }
        return false;
    }

    void CPU::CheckIfCouldGo(int cpu_time)
    {
        int next_timeslice = (cpu_time / 1000 + 1) * 1000;
        //下个时间片到来的时间
        int remain_time = next_timeslice - cpu_time;
        //现在时间到下个时间片到来时间的剩余时间
        if (CheckIfBorder() == true)
        {
            if (current_time == 0 && PSW == 1)      //第一条指令的特殊情况
            {
                if (store.GetInstrucTimes(0) > remain_time)
                    if_could_go = false;
                else
                    if_could_go = true;
            }
            if (current_time != 0)
            {
                if (store.GetInstrucTimes(PSW) > remain_time)
                    if_could_go = false;
                else
                    if_could_go = true;
            }
        }
    }
```

5. 更新 CPU 状态

CPU 有两种工作模式，即用户模式与核心模式。这两种运行模式可以根据指令当前的类型来进行切换，即当指令为 0 型或者 2 型时，CPU 模式切换为核心模式，当指令为 1 型时，CPU 模式切换为用户模式。

```
    int CPU::UpdateCPUMode()
    {
        switch (store.GetInstrucState(PSW))
        {
        case INSTRUC_STATE_SYSTEMCALL:
            this->cpu_state = CORE_MODE;         //核心态
            break;
        case INSTRUC_STATE_USERMOD_OPERATION:
            this->cpu_state = USER_MODE;         //用户态
            break;
        case INSTRUC_STATE_PV_OPERATION:
```

```
            this->cpu_state = CORE_MODE;          //核心态
            break;
        default:
            break;
        }
        return this->cpu_state;
    }
```

（八）控制模块

1. 线程创建与并行

控制模块开始模拟并发环境与调度算法时，需要得到多个线程的支持，包括：任务请求线程、时间片调度线程与进程运行线程。每一个线程都有其特殊的用途，需要协调配合才能够共同完成任务。此处只给出线程的创建及调用代码，具体的调度思路与策略将在下文详细描述。

```
    void Control::UnixDispatch()
    {
        /*数值重新初始化*/
        ValueReset();
        cp.ValueReset();
        fo.ValueReset();
        ps.ValueReset();
        tr.ValueReset();
        /*数值重新初始化*/
        int join_task_num = 0, max_intime = -1;
        max_intime = tr.CalMaxInTime();
        thread task_request_thread(&Control::TaskRequestThread, this);
        //任务请求调入
        thread time_slice_thread(&Control::TimeSliceThread, this);
        //时间片调度
        thread PCB_run_thread(&Control::PCBRunThread, this);
        //进程运行
        tc.BeginCreate();              //开始计时
        task_request_thread.detach();
        time_slice_thread.detach();
        PCB_run_thread.join();         //主线程等待执行，所以使用join
        LastTimeSliceDone();           //最后一个时间片的处理
    }
```

2. 菜单调用

程序的功能选择是由菜单进行调度的，即用户选择对应的编号，菜单根据编号执行相应的命令。

```
    void Control::ChoiceOperation(int choice)
    {
        switch (choice)
        {
        case 1:
```

```
            system("cls");
            UnixDispatch();
            LastShowAllInfo();
            ShowSavedFileInfo();
            AnyKeyToBackToMenu();
            break;
        case 2:
            system("cls");
            fo.ResetPCBHeadShowInfo();
            fo.ReadAndShowAll();
            AnyKeyToBackToMenu();
            break;
        case 3:
            system("cls");
            fo.ResetPCBHeadShowInfo();
            fo.ResetPCBHeadWriteInfo();
            ReCreateAllPCB();
            AnyKeyToBackToMenu();
            break;
        case 4:
            system("cls");
            fo.ShowRunResults();
            AnyKeyToBackToMenu();
            break;
        case 5:
            exit(0);
            break;
        default:
            break;
        }
    }
```

3. 时间片中断开关

0 型指令与 2 型指令为系统调用指令，此时，程序需要关闭其中断开关，使得时间片调度线程暂时不工作。当指令执行完毕切换为 1 型指令时，时间片又再次开启，允许时间片调度线程进行调度。

```
    void Control::CloseTimeSliceBreak()
    {
        if_close_timeslice_break = true;
    }
    void Control::OpenTimeSliceBreak()
    {
        if_close_timeslice_break = false;
    }
```

4. 寻找 p-pri 最小的进程

每一次当时间片进行调度时，程序需要找出当前最高优先权的进程进行调度，而最高优先权的依据便是 p-pri 的值。

程序遍历当前所有已经分配空间的 PCB，逐个比较他们的 p-pri 的值，然后进行选择。

```
int Control::GetMaxPriorityProcessID()
{
    int min_no = -1;
    int min = 99999;
    for (int i = 0; i < SYSTEM_MAX_PCB_NUM; i++)
    {
        PCB &temp = tr.GetOnePCB(i);
        if(((temp.GetProState()==PCB_STATE_READY)||(temp.GetProState(
        ) ==
        PCB_STATE_RUNNING))&&(temp.GetIfUsed()==true)&&(temp.GetIfEnd
        ()!=true))
            if (temp.Get_P_PRI() <= min)
            {
                min = temp.Get_P_PRI();
                min_no = i;
            }
    }
    return min_no;
}
```

5. cpu 时间控制

cpu 每接收到一次 10ms 的时间中断，则将当前记录的时间加 10ms，以保证系统时间的准确。这里只给出具体的实现代码，具体的实现思想将在下文详细描述。

```
void Control::RefreshCpuTime()
{
    cpu_current_time = receive_break * BREAK_TIME_LENGTH;
}
```

6. 阻塞态控制

当程序运行 2 型 PV 指令时，发生系统调用，并将该进程调入到阻塞态。直至 50ms 时间后才能够将其从阻塞态调入到就绪态。

这里只给出阻塞态控制的代码，具体的实现思想将在下文详细描述。

```
void Control::PVOperation(PCB &pcb)
{
    string str = "";
    str = str + "\ncpu时间: " + to_string(cpu_current_time) + ", 进程"
    + to_string(pcb.GetProID()) + "进入阻塞态！\n";
    cout << str;
    fo.WriteResults(str);
    pcb.SetCurrentTime(pcb.GetCurrentRunTime() - BREAK_TIME_LENGTH);
    //原来的时间-10
```

```
        pcb.SetPSW(pcb.GetPSW() - 1);
        if (pcb.GetPSW() == 0)
            pcb.SetPSW(1);
        ps.OutRunningQueue(pcb.GetProID() - 1); //将该进程从运行队列移出
        ps.JoinWaitQueue(pcb, fo, cpu_current_time);//将该进程移入等待队列
        //找出最高优先级的PCB进程
        int max = GetMaxPriorityProcessID();     //获取当前的最大优先权的ID
        if (ps.GetReadyQueueSize() > 0)
        {
            PCB& temp2 = tr.GetOnePCB(max);      //取出就绪态中适合的元素
            ps.JoinRunningQueue(temp2, fo, cpu_current_time);
                                                 //将该进程加入到运行队列
            cp.RecoverSpot(temp2);               //恢复现场
        }
        ps.OutReadyQueue(max);                   //将该进程从就绪队列中删除
        pcb.InitWaitTime();                      //开始等待时间计时
    }
```

7. 阻塞态时间监测与状态恢复

当有进程进入到阻塞态时，需要对其进行计时，当阻塞时间满 50ms 时，才能将其从阻塞态调入到就绪态。

阻塞态的时间监测与状态恢复需要比较复杂的操作才能够完成，此处只给出代码，具体的实现思想将在下文详细描述。

```
    void Control::PVOperationTimeAdd()
    {
        for (int i = 1; i <= SYSTEM_MAX_PCB_NUM; i++)
        {
            PCB &temp = tr.GetOnePCB(i - 1);
            if (temp.GetProState()==PCB_STATE_WAIT)
            {
                temp.WaitTimeSelfAdd();       //等待时间自增
            }
        }
    }

    void Control::CheckIfPVOperationOK()
    {
        for (int i = 1; i <= SYSTEM_MAX_PCB_NUM; i++)
        {
            PCB &temp = tr.GetOnePCB(i - 1);
            if ((temp.GetWaitTime() == PV_INSTRUCTION_TIME) &&
            (temp.GetProState() == PCB_STATE_WAIT))
            //等待时间已经到
            {
                string str = "";
```

```
                PVProcessToReady(temp);
                str = str + "cpu时间: " + to_string(cpu_current_time +
                BREAK_TIME_LENGTH) + ", 进程" + to_string(temp.GetProID())
                + "的I/O操作完成! 从阻塞态进入就绪态! \n";
                cout << str;
                fo.WriteResults(str);
            }
        }
    }

    void Control::PVProcessToReady(PCB &pcb)
    {
        ps.OutWaitQueue(pcb.GetProID() - 1);        //移出等待队列
        ps.JoinReadyQueue(pcb, fo, cpu_current_time + BREAK_TIME_LENGTH);
                                        //加入就绪队列
        pcb.InitWaitTime();             //等待时间复位
        pcb.ProcessWakeUp();            //进程唤醒
        pcb.SetCurrentTime(pcb.GetCurrentRunTime() +
        PV_INSTRUCTION_TIME);
        if(pcb.GetPSW()!=1)
            pcb.SetPSW(pcb.GetPSW() + 1);
    }
```

8. 任务请求线程

任务请求线程是程序进行并发环境模拟与进程调度的基本支持，为其提供任务的实时检测功能。该线程与主程序并行执行，既不受其他模块的影响，又能够实时提供当前的状态信息给主程序。

这里只给出代码，具体的实现思想将在下文详细描述。

```
    void Control::TaskRequestThread()
    {
        int join_task_num = 0, max_intime = -1;
        max_intime = tr.CalMaxInTime();
        while (1)
        {
            //符合时间的任务请求处理
            if ((tr.TaskJoinCheck(cpu_current_time,join_task_num) ==
            true) && (cpu_current_time < max_intime + BREAK_TIME_LENGTH *
            2))
            {
                /*任务调度到就绪队列*/
                PCB &rj = tr.GetOnePCB(join_task_num);
                if (rj.GetIfUsed() == false)
                {
                    rj.ProcessCreate();             //原语—进程创建
                    ps.JoinReadyQueue(rj, fo, cpu_current_time);
```

```
                string str = "cpu时间: " + to_string(cpu_current_time)
                + ", 作业" + to_string(rj.GetProID()) + "调入内存，等待
                执行！\n";
                cout << str;
                fo.WriteResults(str);
            }
        }
        if (cpu_current_time >= max_intime + BREAK_TIME_LENGTH * 2)
            break;
    }
}
```

9. 时间片调度线程

时间片调度线程是程序进行并发环境模拟与进程调度的基本支持，为其提供时间片调度的功能。该线程与主程序并行执行，既不受其他模块的影响，又能够实时提供当前的状态信息给主程序。

这里只给出代码，具体的实现思想将在下文详细描述。

```
void Control::TimeSliceThread()
{
    while (!tr.IfAllOK())
    {
        if (break_time_num == (TIMESLICE_TIME_LENGTH /
BREAK_TIME_LENGTH))
        {
            string str = "";
            str += "\n第" + to_string(time_slice_num) + "个时间片到来！
            \n";
            cout << str;
            //每次时间片信号到来时的处理
            if (!if_close_timeslice_break)//有时间片信号且没有被关中断
            {
                fo.WriteResults(str);
                str.clear();
                //将每个进程的p-cpu进行衰减并计算每个进程的p-pri
                for (int i = 0; i < SYSTEM_MAX_PCB_NUM; i++)
                {
                    PCB &ts = tr.GetOnePCB(i);
                    ts.PCPU_DECAY();  //p-cpu值衰减（所有进程同时衰减）
                    ts.Cal_P_PRI();   //计算所有进程的p-pri
                }
                int current_priority_id = GetMaxPriorityProcessID();
                if (current_priority_id != last_priority_ID)
                {
                    if (ps.GetRunningQueueSize() > 0)
                    //上一次进程从运行态调入就绪态
```

```
                    {
                        PCB& temp = tr.GetOnePCB(last_priority_ID);
                        //取出运行队列中的元素
                        ps.JoinReadyQueue(temp, fo, cpu_current_time);
                        //将该元素加入就绪队列
                        ps.OutRunningQueue(last_priority_ID);
                    }
                    PCB& re = tr.GetOnePCB(current_priority_id);
                    //获取优先权最大的进程
                    ps.JoinRunningQueue(re, fo, cpu_current_time);
                    //将该进程加入到运行队列
                    ps.OutReadyQueue(current_priority_id);
                    //将该进程从就绪队列中删除
                    last_priority_ID = current_priority_id;
                }
            }
            AllPCBInfoShow_TimeSlice();
            str += ps.ShowRunningQueue();
            str += ps.ShowReadyQueue();
            str += ps.ShowWaitQueue();
            time_slice_num++;
            break_time_num = 0;       //时间片信号恢复
            fo.WriteResults(str);
        }
    }
}
```

10. 进程运行线程

进程运行线程是程序进行并发环境模拟与进程调度的基本支持，为其提供控制进程运行的功能。该线程与主程序并行执行，既不受其他模块的影响，又能够实时提供当前的状态信息给主程序。

这里只给出代码，具体的实现思想将在下文详细描述。

```
void Control::PCBRunThread()
{
    while (!tr.IfAllOK())
    {
        //每次的中断处理
        if (tc.GetIfBreak())      //产生一次10ms中断
        {
            ShowEveryTimeDetail();    //每个时间点显示一次详细信息
            if (ps.GetRunningQueueFirstID() != PCB_INIT_NUM)
            //运行队列中有元素
            {
                PCB &temp = tr.GetOnePCB(ps.GetRunningQueueFirstID());
                //temp为当前运行队列中的元素
```

```
if (temp.GetStartTime() == PCB_INIT_NUM)
//若进程从未被创建，则开始计时
{
    temp.SetStartTime(cpu_current_time);
}
cp.RecoverSpot(temp);          //CPU恢复现场
cp.CurrentRunTimeSelfAdd(cpu_current_time, fo);
//进程实际使用时间自增，更新CPU状态，更新时间片bool参数
cp.RefreshOverRunTime();       //恢复超出的实际使用时间
temp.P_CPU_Add();              //当前运行的进程的p-cpu自增
cp.ProtectSpot(temp);          //CPU保护现场
CheckIfCloseTimeSlice(temp);
//检测是否需要关闭时间片中断

if (temp.GetInstrucState(temp.GetPSW() - 1) ==
INSTRUC_STATE_PV_OPERATION)       //PV操作的检测
    PVOperation(temp);
PVOperationTimeAdd(); //处于等待态的进程的等待时间自增
CheckIfPVOperationOK();//检查是否有处于等待态的进程完成

//temp引用的重新声明
PCB &temp_new =
tr.GetOnePCB(ps.GetRunningQueueFirstID());//temp为当前
运行队列中的元素
if (temp_new.GetStartTime() == PCB_INIT_NUM)
//若进程从未被创建，则开始计时
{
    temp_new.SetStartTime(cpu_current_time);
}

if (cp.CheckIfOK() == true)       //进程完成的检测
{
    string str = "";
    str += "√cpu时间：" + to_string(cpu_current_time +
    BREAK_TIME_LENGTH) + "，进程" +
    to_string(cp.GetPC()) + "运行结束！\n";
    cout << str;
    fo.WriteResults(str);
    //该进程出运行队列，并且将if_end标志设置为true
    ps.OutRunningQueue(temp_new.GetProID() - 1);
    temp_new.ProcessCancel(cpu_current_time +
    BREAK_TIME_LENGTH);       //进程撤销原语
    //马上调入下一个就绪态的PCB
    int max = GetMaxPriorityProcessID();
```

```
                    //获取当前的最大优先权的ID
                    if (ps.GetReadyQueueSize() > 0)
                    {
                        PCB& temp2 = tr.GetOnePCB(max);
                        //取出就绪态中适合的元素
                        ps.JoinRunningQueue(temp2, fo, cpu_current_time
                        + BREAK_TIME_LENGTH); //将该进程加入到运行队列
                        cp.RecoverSpot(temp2);//恢复现场
                    }
                    ps.OutReadyQueue(max);    //将该进程从就绪队列中删除
                    last_priority_ID = max;
                }
            }
            else    //运行队列中没有元素
            {
                cp.WaitTimeSelfAdd();
                PVOperationTimeAdd();//处于等待态的进程的等待时间自增
                CheckIfPVOperationOK();//检查是否有处于等待态的进程完成
                //极端情况解决——运行态为空，就绪态为1，进不去的情况
                if (ps.GetReadyQueueSize() !=0 &&
                ps.GetRunningQueueSize() == 0)
                {
                    int max = GetMaxPriorityProcessID();
                    PCB& temp = tr.GetOnePCB(max);
                    ps.OutReadyQueue(temp.GetProID() - 1);
                    ps.JoinRunningQueue(temp, fo, cpu_current_time +
                    BREAK_TIME_LENGTH);
                }
            }
            break_time_num++;
            receive_break++;
            tc.ResetBreak();      //中断信号恢复
            RefreshCpuTime();
        }
    }
}
```

五、测试用例及运行结果分析

（说明：按照功能测试来设计测试用例，测试用例要有测试样本的输入数据及输出结果以及界面呈现，并对结果进行文字分析）

（一）测试前准备

为了能够测试程序的正确性，故需要使用不同类型，不同搭配，不同数量，尽可能多的数据来进行测试。

程序很好地实现了三种指令的调度功能，分别为：0 型系统调用指令，1 型用

户计算操作指令，2 型 PV 操作指令。在本节中，将会依次测试他们的正确性。

测试共分为三种情况。第一种情况下，全部使用 1 型指令，即用户计算操作指令，该指令不需要进行系统调用，CPU 始终工作在用户态，CPU 的空闲只有发生在没有进程被调度或者指令无法继续工作的情况。第二种情况下，全部使用 1 型指令与 0 型指令的混合，0 型指令会使用系统调用，此时 CPU 的时间片中断信号将会被屏蔽，直至系统调用完成为止，CPU 会在用户态与核心态之间进行切换。第三种情况下，使用 0 型、1 型与 2 型指令的混合，是最复杂的情况，在该种情况下，除了上文所说的情况，还会加入第三种情况，即阻塞情况的发生，若指令为 2 型指令，则系统将会自动识别并将其直接从运行队列送入阻塞队列，同时根据 Unix 调度算法从就绪队列中选择合适的进程调入运行态执行，调入阻塞队列的进程需要运行满 50ms 后才能够被释放重新进入到就绪队列。

为了测试这三种情况，准备了三组不同的数据进行测试。

1. 全 1 型指令 数据准备

使用程序的生成功能，自动生成 10 个全 1 型的 PCB 信息，等待被调用。PCB 信息过多，此处不进行展示，读者可以在报告的附件 2：全 1 型指令数据中查看。

2. 全 0 型指令与 1 型指令 数据准备

使用程序的生成功能，自动生成 10 个全 0 型与 1 型指令的 PCB 信息，等待被调用。PCB 信息过多，此处不进行展示，读者可以在报告的附件 3：全 0 型指令与 1 型指令数据中查看。

3. 混合指令 数据准备

使用程序的生成功能，自动生成 10 个混合型指令的 PCB 信息，等待被调用。PCB 信息过多，此处不进行展示，读者可以在报告的附件 4：混合指令数据中查看。

（二）测试一：全 1 型指令

将“pcb-inputs.txt”文件移入合适的位置，打开程序开始执行。

程序执行的结果如图 5-1 全 1 型指令结果所示：

```
- cpu时间：2830
- 当前运行进程：9，当前指令：5，类型：1-用户态计算操作，已经运行时间：120
- 运行队列：9
- 就绪队列：
- 等待队列：
- CPU状态：USER_MODE，时间片信号：未屏蔽


- cpu时间：2840
- 当前运行进程：9，当前指令：5，类型：1-用户态计算操作，已经运行时间：130
- 运行队列：9
- 就绪队列：
- 等待队列：
- CPU状态：USER_MODE，时间片信号：未屏蔽


- cpu时间：2850
- 当前运行进程：9，当前指令：6，类型：1-用户态计算操作，已经运行时间：140
- 运行队列：9
- 就绪队列：
- 等待队列：
- CPU状态：USER_MODE，时间片信号：未屏蔽


- cpu时间：2860
- 当前运行进程：9，当前指令：7，类型：1-用户态计算操作，已经运行时间：150
- 运行队列：9
- 就绪队列：
- 等待队列：
- CPU状态：USER_MODE，时间片信号：未屏蔽


- cpu时间：2870
- 当前运行进程：9，当前指令：8，类型：1-用户态计算操作，已经运行时间：160
- 运行队列：9
- 就绪队列：
- 等待队列：
- CPU状态：USER_MODE，时间片信号：未屏蔽

√cpu时间：2880，进程9运行结束！
```

图 5-1 全 1 型指令结果

程序最后的结果分析如图 5-2 全 1 型指令结果分析所示：

```
进程1:
提交时间：0；需要运行时间：320；包含指令数：11；开始运行时间：0；结束运行时间：320；
周转时间：320；带权周转时间：1.000000；
进程2:
提交时间：130；需要运行时间：170；包含指令数：6；开始运行时间：320；结束运行时间：490；
周转时间：360；带权周转时间：2.117647；
进程3:
提交时间：300；需要运行时间：330；包含指令数：14；开始运行时间：490；结束运行时间：820；
周转时间：520；带权周转时间：1.575758；
进程5:
提交时间：640；需要运行时间：370；包含指令数：17；开始运行时间：2170；结束运行时间：2540；
周转时间：1900；带权周转时间：5.135135；
进程6:
提交时间：740；需要运行时间：330；包含指令数：12；开始运行时间：820；结束运行时间：1170；
周转时间：430；带权周转时间：1.303030；
进程7:
提交时间：860；需要运行时间：160；包含指令数：7；开始运行时间：2010；结束运行时间：2170；
周转时间：1310；带权周转时间：8.187500；
进程8:
提交时间：1060；需要运行时间：470；包含指令数：20；开始运行时间：1170；结束运行时间：1640；
周转时间：580；带权周转时间：1.234043；
进程9:
提交时间：1240；需要运行时间：170；包含指令数：8；开始运行时间：2710；结束运行时间：2880；
周转时间：1640；带权周转时间：9.647058；
进程10:
提交时间：1420；需要运行时间：520；包含指令数：19；开始运行时间：1640；结束运行时间：2710；
周转时间：1290；带权周转时间：2.480769；

CPU总共运行时间：2880；有效工作时间：2840；等待时间：40；
CPU利用率：98.611110%；



进程已经全部运行结束!
进程PCB信息存储在：D:\C++\操作系统实验 进程调度\x64\Release\19316117-pcbs-input.txt中!
进程运行结果存储在：D:\C++\操作系统实验 进程调度\x64\Release\19316117_RunResults.txt中!
```

图 5-2 全 1 型指令结果分析

仔细观察实验结果，与手动推算的实验结果相同。

由于结果输出数据庞大，只截取部分图片进行展示，详细的输出结果请阅读附件 5：全 1 型指令输出结果。

（三）测试二：全 0 型指令与 1 型指令

将“pcb-inputs.txt”文件移入合适的位置，打开程序开始执行。

程序执行的结果如图 5-3 全 0 型指令与 1 型指令结果所示：

```
- cpu时间：140
- 当前运行进程：1，当前指令：4，类型：1-用户态计算操作，已经运行时间：140
- 运行队列：1
- 就绪队列：
- 等待队列：
- CPU状态：USER_MODE，时间片信号：未屏蔽


- cpu时间：150
- 当前运行进程：1，当前指令：4，类型：1-用户态计算操作，已经运行时间：150
- 运行队列：1
- 就绪队列：
- 等待队列：
- CPU状态：USER_MODE，时间片信号：未屏蔽


- cpu时间：160
- 当前运行进程：1，当前指令：5，类型：1-用户态计算操作，已经运行时间：160
- 运行队列：1
- 就绪队列：
- 等待队列：
- CPU状态：USER_MODE，时间片信号：未屏蔽

◣cpu时间：170，进程2调入就绪队列！
△cpu时间：170，作业2调入内存，等待执行！

- cpu时间：170
- 当前运行进程：1，当前指令：5，类型：1-用户态计算操作，已经运行时间：170
- 运行队列：1
- 就绪队列：2
- 等待队列：
- CPU状态：USER_MODE，时间片信号：未屏蔽


- cpu时间：180
- 当前运行进程：1，当前指令：5，类型：1-用户态计算操作，已经运行时间：180
- 运行队列：1
- 就绪队列：2
- 等待队列：
- CPU状态：USER_MODE，时间片信号：未屏蔽
```

图 5-3 全 0 型指令与 1 型指令结果

程序最后的结果分析如图 5-4 全 0 型指令与 1 型指令结果分析所示：

```
进程1:
提交时间：0；需要运行时间：220；包含指令数：7；开始运行时间：0；结束运行时间：220；
周转时间：220；带权周转时间：1.000000；
进程2:
提交时间：170；需要运行时间：490；包含指令数：16；开始运行时间：220；结束运行时间：710；
周转时间：540；带权周转时间：1.102041；
进程3:
提交时间：300；需要运行时间：160；包含指令数：6；开始运行时间：1980；结束运行时间：2150；
周转时间：1850；带权周转时间：11.562500；
进程4:
提交时间：430；需要运行时间：310；包含指令数：13；开始运行时间：710；结束运行时间：1040；
周转时间：610；带权周转时间：1.967742；
进程5:
提交时间：560；需要运行时间：150；包含指令数：5；开始运行时间：1830；结束运行时间：1980；
周转时间：1420；带权周转时间：9.466666；
进程6:
提交时间：740；需要运行时间：510；包含指令数：19；开始运行时间：2410；结束运行时间：2920；
周转时间：2180；带权周转时间：4.274510；
进程7:
提交时间：920；需要运行时间：270；包含指令数：10；开始运行时间：1040；结束运行时间：1310；
周转时间：390；带权周转时间：1.444444；
进程8:
提交时间：1080；需要运行时间：200；包含指令数：8；开始运行时间：1310；结束运行时间：1510；
周转时间：430；带权周转时间：2.150000；
进程9:
提交时间：1280；需要运行时间：260；包含指令数：9；开始运行时间：2150；结束运行时间：2410；
周转时间：1130；带权周转时间：4.346154；
进程10:
提交时间：1480；需要运行时间：320；包含指令数：11；开始运行时间：1510；结束运行时间：1830；
周转时间：350；带权周转时间：1.093750；

CPU总共运行时间：2920；有效工作时间：2890；等待时间：30；
CPU利用率：98.972601%；



进程已经全部运行结束!
进程PCB信息存储在：D:\C++\操作系统实验 进程调度\x64\Release\19316117-pcbs-input.txt中!
进程运行结果存储在：D:\C++\操作系统实验 进程调度\x64\Release\19316117_RunResults.txt中!
```

图 5-4 全 0 型指令与 1 型指令结果分析

仔细观察实验结果，与手动推算的实验结果相同。

由于结果输出数据庞大，只截取部分图片进行展示，详细的输出结果请阅读附件 6：全 0 型指令与 1 型指令输出结果。

（四）测试三：混合指令

将“pcb-inputs.txt”文件移入合适的位置，打开程序开始执行。

程序执行的结果如图 5-5 混合指令结果所示：

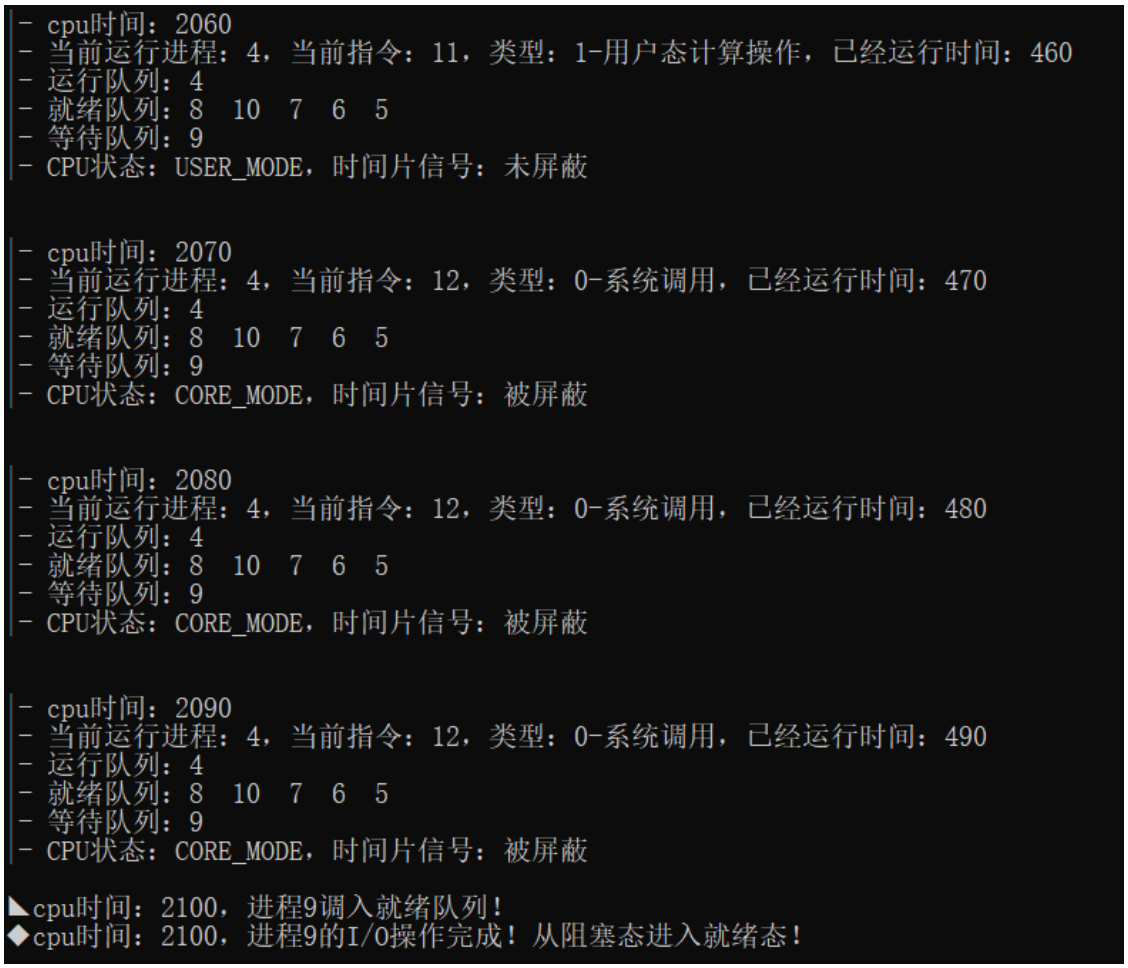

图 5-5 混合指令结果

程序最后的结果分析如图 5-6 混合指令结果分析所示：

```
进程1:
提交时间：0；需要运行时间：290；包含指令数：9；开始运行时间：0；结束运行时间：290；
周转时间：290；带权周转时间：1.000000；
进程2:
提交时间：200；需要运行时间：820；包含指令数：20；开始运行时间：290；结束运行时间：1670；
周转时间：1470；带权周转时间：1.792683；
进程3:
提交时间：380；需要运行时间：220；包含指令数：6；开始运行时间：380；结束运行时间：630；
周转时间：250；带权周转时间：1.136364；
进程4:
提交时间：580；需要运行时间：560；包含指令数：14；开始运行时间：630；结束运行时间：2170；
周转时间：1590；带权周转时间：2.839286；
进程5:
提交时间：760；需要运行时间：400；包含指令数：13；开始运行时间：830；结束运行时间：2830；
周转时间：2070；带权周转时间：5.175000；
进程6:
提交时间：890；需要运行时间：370；包含指令数：11；开始运行时间：900；结束运行时间：2480；
周转时间：1590；带权周转时间：4.297297；
进程7:
提交时间：1060；需要运行时间：310；包含指令数：9；开始运行时间：2370；结束运行时间：2720；
周转时间：1660；带权周转时间：5.354839；
进程8:
提交时间：1220；需要运行时间：460；包含指令数：16；开始运行时间：1470；结束运行时间：2440；
周转时间：1220；带权周转时间：2.652174；
进程9:
提交时间：1320；需要运行时间：610；包含指令数：19；开始运行时间：1390；结束运行时间：2370；
周转时间：1050；带权周转时间：1.721311；
进程10:
提交时间：1490；需要运行时间：200；包含指令数：6；开始运行时间：1690；结束运行时间：2240；
周转时间：750；带权周转时间：3.750000；

CPU总共运行时间：2830；有效工作时间：2520；等待时间：310；
CPU利用率：89.045936%；



进程已经全部运行结束!
进程PCB信息存储在：D:\C++\操作系统实验 进程调度\x64\Release\19316117-pcbs-input.txt中!
进程运行结果存储在：D:\C++\操作系统实验 进程调度\x64\Release\19316117_RunResults.txt中!
```

图 5-6 混合指令结果分析

仔细观察实验结果，与手动推算的实验结果相同。

由于结果输出数据庞大，只截取部分图片进行展示，详细的输出结果请阅读附件 7：混合指令输出结果。

六、技术问题及解决方案

（一）多线程并行

1. 问题描述

为了仿真模拟程序内部多个独立执行单元的并发或并行执行，则需要使用到多线程技术，使得这些函数能够同时独立的工作，以仿真出程序的并发环境。

2. 解决方法

C++11 引入了多线程类的新特性，C++11 在标准库中为多线程提供组件，使用线程需要包含头文件 thread，其命名空间为 std，阅读文献[1]-[5]，以及 cplusplus 官网的说明，可以大概知晓其工作原理及使用方法。

程序使用了多个函数的线程，分别为：时钟中断产生线程、任务请求线程、时间片调度线程、进程运行线程。

线程的基本定义方式为（以时钟中断产生线程为例）：

```
thread create_break(&TimeClock::CreateBreak, this);
create_break.detach();
```

当程序开始运行后，4 个线程便开始独立工作，互不干扰但共享变量，或向外提供查询接口，线程运行之间的时间关系图如图 6-1 线程运行时间关系图所示。

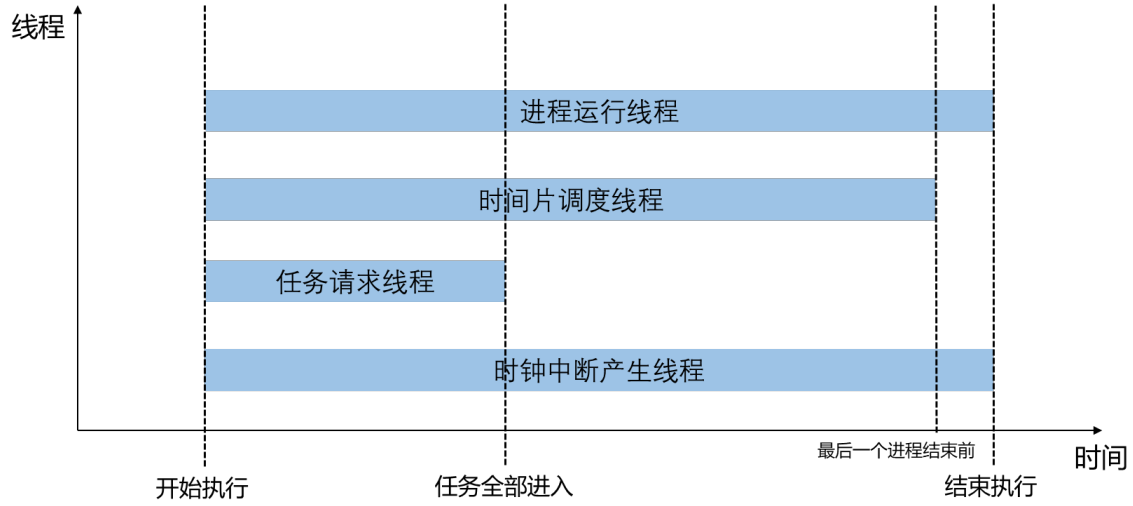

图 6-1 线程运行时间关系图

（二）cpu 内部时间计数

1. 问题描述

为了能够精确的描述系统内部的时间，故需要使用一个全局唯一的 cpu_time 变量来给出时间，并且随着每一次时钟中断的产生，其时间也应相应的做出变化。

cpu_time 变量应只让一个函数拥有修改权限，对其他所有的函数只有读取权限。但是，这个权限如何分配是一个问题。

若让时钟中断产生线程接管该权限，则会有时候导致时间不准的问题。具体表现为：当一个线程的一次处理操作未进行完成时，时钟中断已经到来，这时，该线程无法准确接收中断信号，导致后面的操作被整体的延后，与前面的时间产生脱节，如图 6-2 cpu 时间延后问题所示。

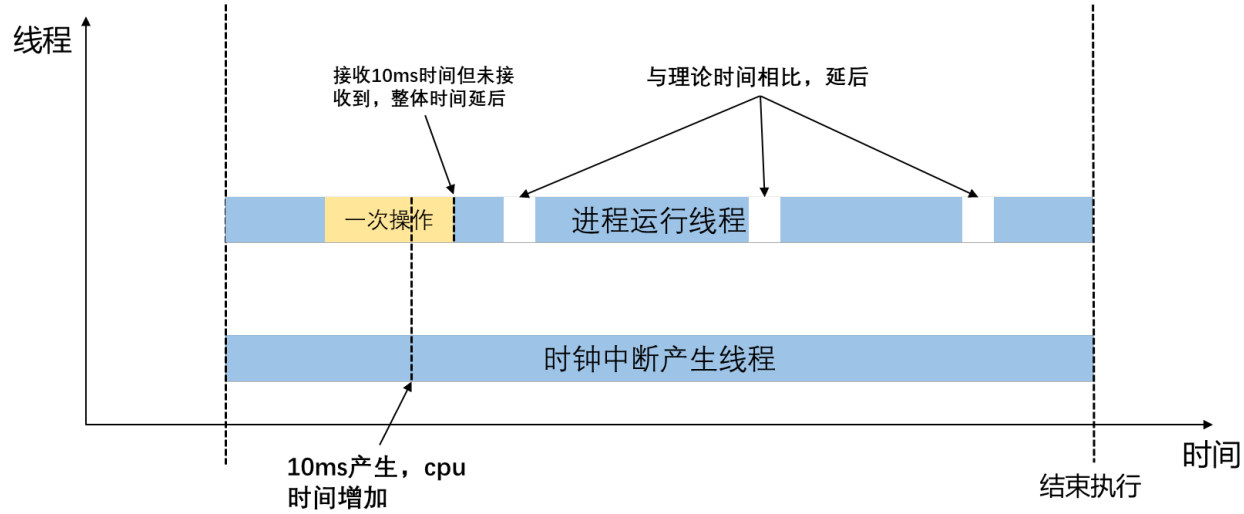

图 6-2 cpu 时间延后问题

2. 解决方法

为了解决时间不同步的问题，可以将时间的修改权限转移至“进程运行进程”。因为进程运行进程与时钟中断产生进程是同步开始与结束的，所以权限的转移并不会影响。对于真实时间与 cpu 时间不同步的问题，可以有选择的忽略一些 10ms 的时钟中断，每一次的 cpu_time 的计时以线程实际接收的为准。

每一次当线程接收到中断信号后，cpu_time 才能够自增。利用这种方式，虽

然于现实的时间略有误差，但能够保证 cpu 时间的绝对准确。同时，给除了时钟中断产生线程和进程运行线程之外的其他线程加上信号检测，可以实现多个线程之间的信号传递，进一步保证时间的准确性。该实现方法的原理图如图 6-3 时间延后解决原理所示。

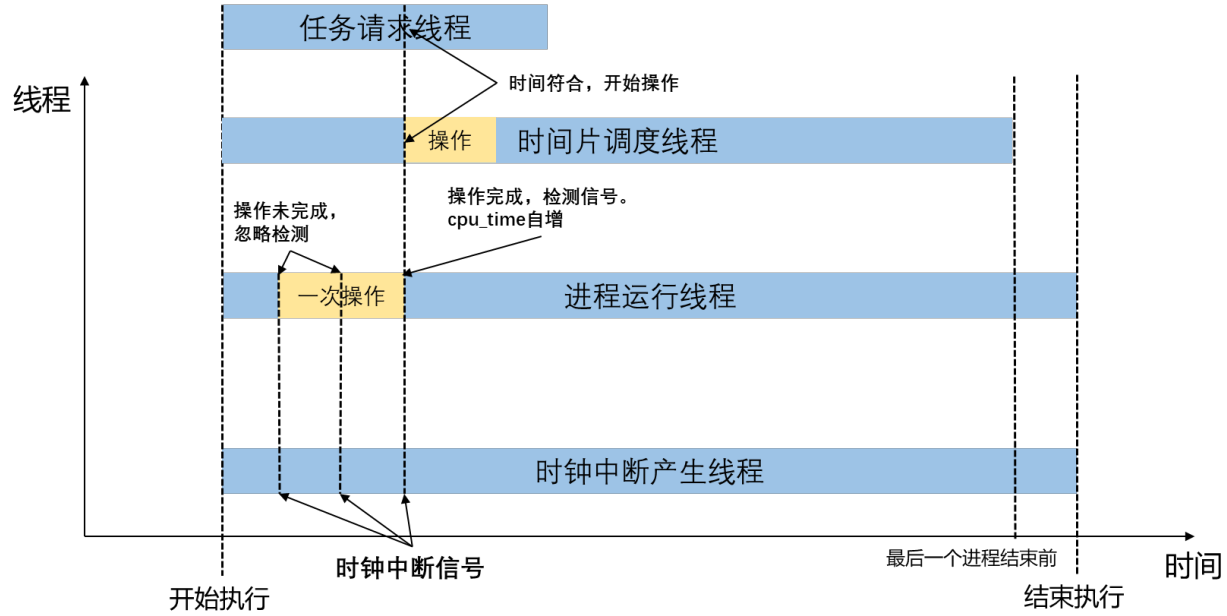

图 6-3 时间延后解决原理

（二）精确时间计时

1. 问题描述

为了能够准确的发生 10ms 的中断时间，故需要设计中断的产生函数。

开始，拟采取每一次检测当前系统时间，调用系统 API 函数 GetLocalTime() 进行检测，然后每一次与上一次获得的时间相减，判断是否为 10ms。

经过分析，该种方法所需要消耗的 CPU 资源过于庞大，不适合该程序的运行与设计。

2. 解决方法

利用 Windows.h 库中的 Sleep()函数，可以将睡眠时间精确到 ms 级别。同时，又因为该函数为线程，独立的执行，因为不会对其他程序产生影响。

（三）队列指定元素删除

1. 问题描述

当 PCB 的状态转换时，需要从当前队列中移出。但是，因为队列的特性，只能够先进先出，无法直接移出指定位置的元素。所以，需要编写代码实现从队列中删除指定序号的元素。

2. 解决方法

设计了算法进行该操作，将指定元素进行删除。该算法利用了队列先进先出的特性，每一次弹出一个元素进行比较，若不是，则将其插入到队列的最后，若是，则不将其插入到最后。

代码为：

```
void ProcessSchedule::OutQueue(int out_id)
{
    for (int i = 1; i <= queue.size(); i++)
```

```
    {
        PCB &temp = queue.front();
        queue.pop();
        if (temp.GetProID() != out_id + 1)
            queue.push(temp);
    }
}
```

实现过程的原理如图 6-4 移除队列元素所示。

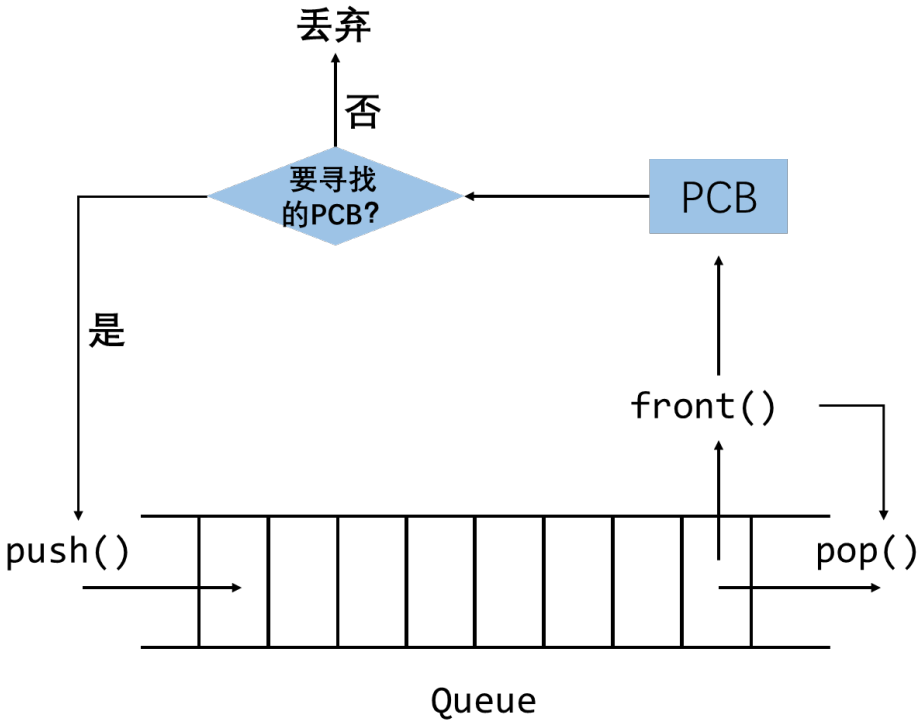

图 6-4 移除队列元素

（四）开/关中断控制

1. 问题描述

根据设计要求，程序中需要体现开/关中断的思想。

2. 解决方法

在 Control 模块中添加 bool 类型变量 if_close_timeslice，当每次的指令为 0 型或 2 型指令时，将该变量设置为 true。

在时间片调度时，用一个 if()语句包括起来，判断的条件即为 if_close_timeslice 的值是否为 true，如果为 true 则终止执行，如果为 false，则可以执行。

（五）任务请求调度算法

1. 问题描述

根据程序设计要求，需要设计一个任务请求的调度算法，以解决实时的任务请求问题。

2. 解决方法

设计如下的算法逻辑：

①计算最后的线程进入时间

②开始循环执行

①如果有进程时间符合且 cpu 当前时间没有超出
①获取要进入的进程信息
①进程创建原语，分配空间
②加入就绪队列
②如果当前 cpu 时间超时，则退出
③关闭线程

（六）时间片调度算法

1. 问题描述

根据程序设计要求，需要设计一个时间片调度算法，以解决实时的时间片调度问题。

2. 解决方法

设计如下的算法逻辑：

①当所有进程并未全部完成
①当时间片到来
①没有被关中断
①将每个进程的 p-cpu 进行衰减并计算每个进程的 p-pri
②取得优先级最大的 PCB
③上一次进程从运行态调入就绪态
④取出运行队列中的元素
⑤将该元素加入就绪队列
⑥原来的 PCB 进入就绪队列
⑦新进程加入到运行队列
⑧将原进程从就绪队列中删除
②时间片次数自增
②关闭线程

（七）进程运行算法

1. 问题描述

根据程序设计要求，需要设计一个进程运行的调度算法，以解决实时的进程运行调度问题。

2. 解决方法

设计如下的算法逻辑：

①当进程没有全部执行完成
①产生一次 10ms 中断
①运行队列中有元素
①取出当前运行队列中的元素
①若进程从未被创建，则调用进程创建原语
②CPU 恢复现场
③进程实际使用时间自增，更新 CPU 状态，更新时间片 bool 参数
④恢复超出的实际使用时间
⑤当前运行的进程的 p-cpu 自增
⑥CPU 保护现场
⑦检测是否需要关闭时间片中断
⑧检测是否为 PV 操作

①调入阻塞队列
⑨处于等待态的进程的等待时间自增
⑩检查是否有处于等待态的进程完成
②进程完成的检测
①该进程出运行队列，并且将 if_end 标志设置为 true
②进程撤销原语
③马上调入下一个就绪态的 PCB
④将该进程加入到运行队列
⑤恢复现场
⑥将该进程从就绪队列中删除
②时间中断数量自增
③刷新当前 cpu 时间
②关闭线程

七、实验心得

经过了 2 个星期的时间，我付出了艰苦卓绝的努力，终于完成了此次的课外实验，颇为不易。

系统完成了设计任务书的全部要求，且能够很好地运行，且进行了适当的扩展，完成了运行态、就绪态、阻塞态的三态转换。

进程是操作系统中最重要和最基本的概念之一，引入进程是由系统资源的有限性和系统内的并发性所决定的。进程具有生命周期，由创建而产生，由调度而执行，由撤销而消亡，操作系统的基本功能是进程的创建、管理和撤销。为了实现对进程的管理，操作系统维护每个进程的资料结构——进程控制块，这是进程的唯一标志，创建进程必须为其创建进程控制块，撤销进程时系统便回收进程控制块。

进程是并发程序的执行，是动态的概念，但是为了在处理器上执行仍然需要静态描述。一个进程在运行过程中或执行系统调用，或产生中断/异常事件，处理器都会进行状态转换，操作系统接收控制权，内核完成必需的操作之后，或恢复被中断进程运行，或切换至新进程。当系统调度新进程占有处理器时，新老进程随之会发生上下文切换。

本程序很好地模拟操作系统的低级调度，以 10ms 为基本的时间单位，展现进程在操作系统内部的切换规律，从而帮助更好的理解进程切换的含义。

但是，程序的设计在很多方面仍有待完善，具体表现在以下几个方面：

1. 没有设计 GUI，所有的程序操作都在控制台下进行。
2. 自动生成的 PCB 信息的分布仍不够均匀，需要更随机的随机函数。
3. 程序输出的时间点信息过于复杂，阅读起来较为辛苦。
4. 没有能够将过程以图形的方式展现，对结果的理解需要读者阅读结果输出文件。
5. 程序的结构略复杂，不易其他人的二次开发。

希望日后这些问题能被别人很好的解决。

对于操作系统课程的学习、对于编程能力的提升远远不止如此，未来仍需要不断努力，继续学习。

八、用户手册

（一）开发环境

系统：Windows 10 x64 1809

CPU：Intel Core i7-6700HQ

内存：8GB

硬盘：256GB SSD

IDE：Microsoft Visual Studio Community 2017 15.8.9

Microsoft Visual C++ 2017

Windows SDK：10.0.17134.0

（二）运行条件

系统：Windows10 x64

环境：装有 Viusal Studio 2017 软件及 10.0.17134.0 SDK

注：其他配置环境未经过测试，如果不能够运行请检查自己的环境配置，或将源代码重新编译运行。

（三）操作步骤

1. 启动程序

打开“可执行文件.exe”文件或者用户自己编译的可执行程序，即可开始运行。

2. 菜单选择

程序开启后，会打开菜单面板，要求用户输入相应的序号执行相应的操作。如图 8-1 菜单面板。各个功能简洁明了，用户根据菜单描述可直接推测其含义，在此不做解释。

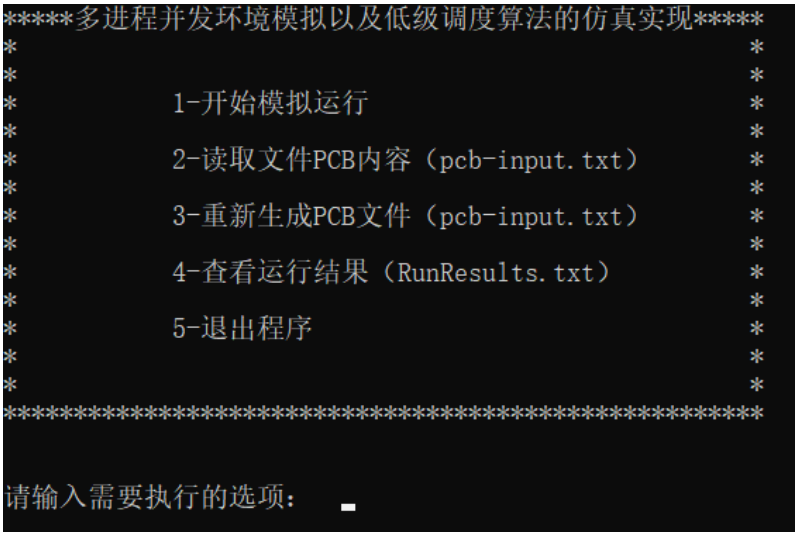

图 8-1 菜单面板

3. 模拟运行

程序读取目录下的“pcb-inputs.txt”文件，导入 PCB 信息，开始并发环境模拟与进程调度。

模拟运行分为两部分，一部分为实时信息展示，如图 8-2 实时信息展示所示；另一部分为最后结果的展示，如图 8-3 最后结果展示所示。

当程序执行完毕实时信息展示后，会要求用户按任意键以开始输出最后的结果，如图 8-4 操作跳转所示。此时只需要输入任意数字，并回车确定即可。

```
- cpu时间：1700
- 当前运行进程：10，当前指令：3，类型：2-PV操作，已经运行时间：70
- 运行队列：10
- 就绪队列：5  3  7  8  6  9
- 等待队列：
- CPU状态：CORE_MODE，时间片信号：被屏蔽


◇cpu时间：1700，进程10进入阻塞态！
◥cpu时间：1700，进程10调入等待队列！
◢cpu时间：1700，进程6调入运行队列！

- cpu时间：1710
- 当前运行进程：6，当前指令：11，类型：0-系统调用，已经运行时间：360
- 运行队列：6
- 就绪队列：9  5  3  7  8
- 等待队列：10
- CPU状态：CORE_MODE，时间片信号：被屏蔽


- cpu时间：1720
- 当前运行进程：6，当前指令：12，类型：2-PV操作，已经运行时间：370
- 运行队列：6
- 就绪队列：9  5  3  7  8
- 等待队列：10
- CPU状态：CORE_MODE，时间片信号：被屏蔽
```

图 8-2 实时信息展示

```
进程1:
提交时间：0；需要运行时间：250；包含指令数：11；开始运行时间：0；结束运行时间：250；
周转时间：250；带权周转时间：1.000000；
进程2:
提交时间：140；需要运行时间：620；包含指令数：20；开始运行时间：250；结束运行时间：1410；
周转时间：1270；带权周转时间：2.048387；
进程3:
提交时间：280；需要运行时间：450；包含指令数：15；开始运行时间：310；结束运行时间：2500；
周转时间：2220；带权周转时间：4.933333；
进程4:
提交时间：480；需要运行时间：240；包含指令数：8；开始运行时间：490；结束运行时间：1520；
周转时间：1040；带权周转时间：4.333333；
进程5:
提交时间：650；需要运行时间：520；包含指令数：19；开始运行时间：660；结束运行时间：2160；
周转时间：1510；带权周转时间：2.903846；
进程6:
提交时间：770；需要运行时间：460；包含指令数：14；开始运行时间：1110；结束运行时间：1930；
周转时间：1160；带权周转时间：2.521739；
进程7:
提交时间：960；需要运行时间：190；包含指令数：6；开始运行时间：2280；结束运行时间：2530；
周转时间：1570；带权周转时间：8.263158；
进程8:
提交时间：1100；需要运行时间：190；包含指令数：6；开始运行时间：1250；结束运行时间：1750；
周转时间：650；带权周转时间：3.421053；
进程9:
提交时间：1270；需要运行时间：390；包含指令数：12；开始运行时间：1610；结束运行时间：2220；
周转时间：950；带权周转时间：2.435897；
进程10:
提交时间：1420；需要运行时间：270；包含指令数：10；开始运行时间：1620；结束运行时间：2100；
周转时间：680；带权周转时间：2.518518；

CPU总共运行时间：2530；有效工作时间：2340；等待时间：190；
CPU利用率：92.490119%；



进程已经全部运行结束！
进程PCB信息存储在：D:\C++\操作系统实验 进程调度\x64\Release\19316117-pcbs-input.txt中！
进程运行结果存储在：D:\C++\操作系统实验 进程调度\x64\Release\19316117_RunResults.txt中！



请输入任意值以返回主菜单！
```

图 8-3 最后结果展示

图 8-4 操作跳转

4. 读取文件 PCB 内容

程序读取当前目录下的“pcb-inputs.txt”文件，向用户展示其文件内容。

5. 重新生成 PCB 文件

程序重新生成 PCB 信息文件，并将他们写入到“pcb-inputs.txt 文件中”。注意，重新生成的 PCB 信息将会强行覆盖原有的文件内容，在操作前请做好资料备份工作，以免数据的丢失。

当用户选择此项目后，程序会要求用户给出要生成的 PCB 文件的个数，如图 8-5 PCB 生成数量输入所示。用户给出数目后，程序随即开始生成 PCB 文件，并在屏幕上显示出来，之后写入到文件中，如图 8-6 PCB 信息生成所示。

图 8-5 PCB 生成数量输入

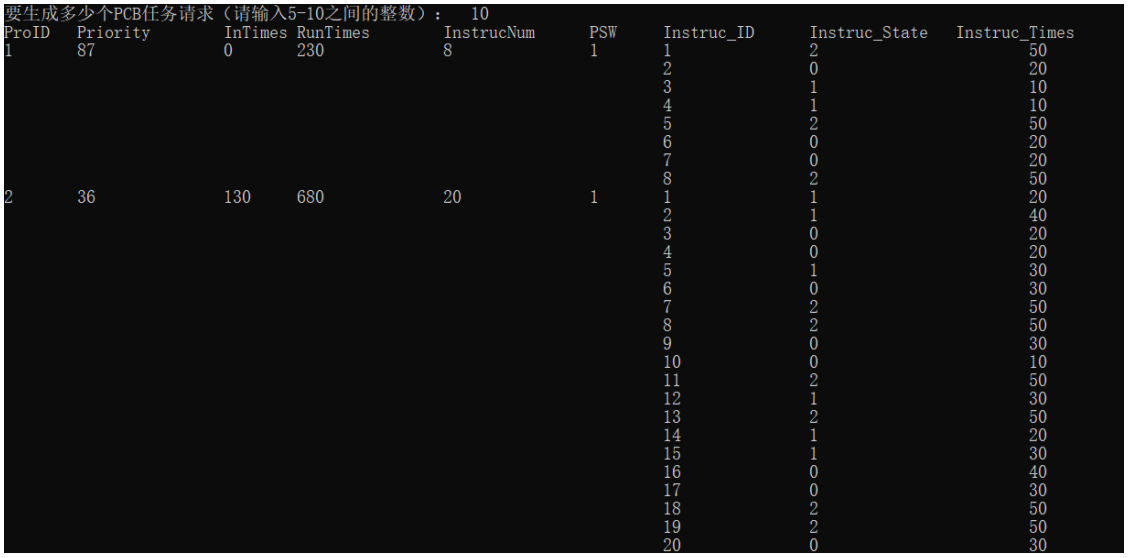

图 8-6 PCB 信息生成

6. 查看运行结果

程序读取当前目录下的“Results.txt”文件，向用户展示其文件内容。

7. 退出程序

清理程序内存并安全退出程序。

（四）使用说明

程序的操作如上文所述，读者在使用前只需要熟读这些操作步骤即可。

可执行文件.exe 使用 Visual Studio 2017 的 Release 模式编译，并使用了“多线程 (/MT)”的运行库，将需要连接的库内置在程序中。若非极特殊环境，一般情况下都能够正常运行。

同时，由于笔者使用配置与各位读者使用配置的不同，难免会存在有相同的输入而产生不同的结果的情况，请读者多次重复并检查自己的环境配置。

程序代码中引入了 C++11 的新特性，可能不受一些编译器的支持，请读者在编译前进行调试。同时，当读者在编译代码时，请使用 Release 模式，因为无法保证 Debug 模式的性能与结果。

若出现问题，请认真反思总结自己的不足与缺漏，及时改正错误，重新测试。

九、备注

（一）提交电子资料目录

目录：

```
└──19316117_必修实验一_申请优秀 A+
   │   19316117-pcbs-input.txt
   │   19316117_RunResults.txt
   │   可执行程序.exe
   │
   ├──code
   │   │   操作系统实验 进程调度.sln
   │   │
   │   └──操作系统实验 进程调度
   │           Control.cpp
   │           Control.h
   │           CPU.cpp
   │           CPU.h
   │           FileOperation.cpp
   │           FileOperation.h
   │           main.cpp
   │           OverallData.h
   │           PCB.cpp
   │           PCB.h
   │           ProcessSchedule.cpp
   │           ProcessSchedule.h
   │           TaskRequest.cpp
   │           TaskRequest.h
   │           TimeClock.cpp
   │           TimeClock.h
   │           操作系统实验 进程调度.vcxproj
   │           操作系统实验 进程调度.vcxproj.filters
   │           操作系统实验 进程调度.vcxproj.user
   │
   └──doc
           课外实验报告_申请优秀 A+.docx
           附件 5：全 1 型指令输出结果.txt
           附件 6：全 0 型指令与 1 型指令输出结果.txt
           附件 7：混合指令输出结果.txt
```

（二）源程序文件清单

1. 时钟中断模块

TimeClock.h

TimeClock.cpp

2. 文件操作模块

FileOperation.h
FileOperation.cpp

3. 任务请求模块

TaskRequest.h
TaskRequest.cpp

4. PCB 结构模块

PCB.h
PCB.cpp

5. 进程调度模块

ProcessSchedule.h
ProcessSchedule.cpp

6. 全局常量数据模块

OverallData.h

7. CPU 模块

CPU.h
CPU.cpp

8. 控制模块

Control.h
Control.cpp

9. 主函数

main.cpp

（三）文件说明

1. 19316117-pcbs-input.txt

PCB 进程信息文件

2. 19316117_RunResults.txt

并发环境模拟及任务调度结果输出文件

3. 可执行程序.exe

通过 VS2017 软件编译的可执行程序

4. 操作系统实验 进程调度.sln

VS2017 的工程文件

5. 操作系统实验 进程调度.vcxproj

操作系统实验 进程调度.vcxproj.filters
操作系统实验 进程调度.vcxproj.user
导入 VS2017 工程时所需要使用的配置信息

6. 课外实验报告_申请优秀 A+.docx

本次实验的实验报告

7. 附件

附件 5：全 1 型指令输出结果.txt
附件 6：全 0 型指令与 1 型指令输出结果.txt
附件 7：混合指令输出结果.txt

实验报告的附件，为三种测试情况的测试结果文件。

十、参考文献

[1] 刘正阳. C/C++多线程程序并发访存问题的软件分析研究[D].北京邮电大学,2017.
[2] 刘晓娇.Unix 系统中进程调度算法的分析与评价[J].科技信息,2010(17):39-40.
[3] 王得利,马月坤.基于回调函数与 Visual C++的多线程应用程序的实现[J].唐山学院学报,2006(02):106-108.
[4] 赵思奇,钱巨,陈林,徐宝文.面向并发的 C++程序重构模式研究[J].计算机与数字工程,2009,37(10):68-73.
[5] 杨延中,王为,田籁声.具有并发类库的 C++[J].软件学报,1998(06):2-5.
[6] 曲广平.管理 Linux 进程与终端[J].网络安全和信息化,2018(03):95-98.
[7] 仇阳.Linux 内核进程调度算法发展[J].电子世界,2017(07):85+87.
[8] 丁富淮. 嵌入式 Linux 系统的二级调度策略优化技术及应用[D].苏州大学,2015.
[9] 吴振亚. Linux 实时调度研究及改进[D].西安电子科技大学,2014.
[10] 杨兴强,刘翔鹏,刘毅.Linux 进程状态演化过程的图形学表示[J].系统仿真学报,2013,25(10):2444-2448.
[11] 苏淑霞.Linux 进程调度算法的研究及改进[J].数字技术与应用,2014(09):121-122.
[12] 俞露.基于 Linux 随机进程调度算法的实现[J].福建电脑,2013,29(02):108-109.
[13] C++文件读写详解
https://www.cnblogs.com/hdk1993/p/5853233.html
[14] C++ 文件和流
http://www.runoob.com/cplusplus/cpp-files-streams.html
[15] C++11 并发编程基础
https://www.cnblogs.com/lpxblog/p/5190438.html
[16] C++ 11 多线程
https://www.cnblogs.com/wangguchangqing/p/6134635.html
[17] C++实现多线程及其三种方法实现多线程同步
https://blog.csdn.net/u011028345/article/details/73440873/
[18] <thread> - C++ Reference
http://www.cplusplus.com/reference/thread/
[19] C++ queue
https://blog.csdn.net/Lison_Zhu/article/details/77095198
[20] [C++ STL] Queue
https://www.cnblogs.com/ChinaHook/p/6985553.html
[21] queue - C++ Reference
http://www.cplusplus.com/reference/queue/queue/

十一、附件

（一）附件 1：程序类的设计

1. 时钟中断产生模块　TimeClock.h

```
class TimeClock
{
private:
    bool if_break;            //是否产生中断
public:
```

```
    TimeClock();              //默认构造函数
    void BeginCreate();       //开始产生中断信号与时间片信号
    void CreateBreak();       //循环产生中断
    void ResetBreak();        //重置中断信号
    bool GetIfBreak();        //获取当前中断状态
};
```

2. 文件操作模块　FileOperation.h

```
class FileOperation
{
private:
    PCB read_from_file;   //读入的数据的暂存
    FILE *read_txt;       //要读入的文件指针  19316117-pcbs-input.txt
    FILE *write_txt;      //要写入的文件指针  19316117-pcbs-input.txt
    FILE *run_results;    //运行结果 19316117-RunResults.txt
    bool if_firsttime_read;   //是否第一次读取文件
    bool if_show_pcb_head;    //是否已经输出PCB文件头
    bool if_write_pcb_head;   //是否已经写入PCB文件头
public:
    FileOperation();
    void ValueReset();            //重新设置初始化参数
    PCB ReadOnePCBFromFile();     //读取一条PCB的信息
    string ReadAndShowAll();      //读取并显示全部的PCB信息
    void SetSystemPCB(PCB&);      //设置传入的系统的PCB
    void SetAllSystemPCB(PCB&, PCB&, PCB&, PCB&, PCB&, PCB&, PCB&,
PCB&, PCB&, PCB&);
    string ShowPCBHead();         //显示PCB的表目信息
    void ResetPCBHeadShowInfo();
    //重置PCB的表目的信号  if_show_pcb_head
    void ResetPCBHeadWriteInfo();
    //重置PCB的表目的信号  if_write_pcb_head
    string ShowOwnPCBDetail();        //显示PCB的信息
    string ShowPCBDetail(PCB);        //显示传入的PCB的信息
    void SetPCB(PCB data);            //设置当前的PCB
    PCB GetPCB();                     //获取当前的PCB
    void WriteHeadToFile();           //将PCB的头写入文件
    void WritePCBToFile();            //将PCB写入文件
    void ShowRunResults();            //显示RunResults文件
    void ClearPCBFile();              //清空PCB文件
    void ClearResultsFile();          //清空Results文件
    void ReOpenWriteFile();           //重新打开写入文件
    void ReOpenReadFile();            //重新打开读取文件
    void WriteResults(string);        //向结果文件中写入数据
};
```

3. 任务请求模块　TaskRequest.h

```
class TaskRequest
{
private:
    PCB system_PCB[SYSTEM_MAX_PCB_NUM];          //系统内可用的PCB表
    bool if_PCB_used[SYSTEM_MAX_PCB_NUM];        //10个PCB表是否被占用
    int next_PCB_no;              //下一个将要使用的PCB编号
    PCB last_one;                 //上一个产生的PCB
    int PCB_num;                  //生成的PCB的编号
public:
    TaskRequest();                //默认构造函数
    void ValueReset();            //重新设置初始化参数
    bool IfExistAvailablePCB();   //是否存在可用的PCB
    void RefreshPCB(PCB);         //更新PCB last_one
    PCB CreateNewPCB();           //生成新的PCB文件
    void InitPCBNum();            //重置PCB的编号
    bool IfAllOK();               //判断所有的PCB是否都已经完成
    PCB& GetOnePCB(int);          //获取某一个具体的PCB
    void SetOnePCB(int, PCB);     //设置某一个具体的PCB
    int CalMaxInTime();           //计算PCB中最大的InTime
    bool TaskJoinCheck(int,int&);//作业进入的实时检查
    bool CheckIfJoin(int);        //检查某一个PCB是否被使用
    int CheckPCBState(int);       //检查某一个PCB的当前状态
};
```

4. PCB 结构模块　PCB.h

```
class PCB
{
private:
    int ProID;                        //进程编号
    int Priority;                     //进程优先数
    int InTimes;                      //进程创建时间
    int ProState;                     //进程状态（就绪、运行、等待）
    int RunTimes;                     //进程需要运行时间
    int InstrucNum;                   //进程中包含的指令数目
    int PSW;                          //程序状态字，进程当前执行的指令编
号
    PCB_Instruc Instruc[MAX_INSTRUC_NUM];       //指令内容

    bool if_used;             //该PCB在系统PCB表中是否被用
    bool if_end;              //PCB是否已经执行结束
    int start_time;           //进程的实际开始时间
    int end_time;             //进程的实际结束时间
    int current_use_time;     //进程已经运行的时间（已经占用CPU的时间）

    int p_pri;                        //Unix调度算法需要变量
```

```
    int p_cpu;                          //Unix调度算法需要变量

    int wait_state_time;                //已经处于阻塞队列的时长
public:
    PCB();                      //默认构造函数
    void InitInstruc();         //初始化指令内容数组
    void InitPCB();             //初始化PCB内容
    int GetProID();             //获取进程编号
    int GetPriority();          //获取进程优先数
    int GetInTimes();           //获取进城创建时间
    int GetProState();          //获取进程状态
    int GetRunTimes();          //获取进程运行时间
    int GetInstrucNum();        //获取进程中包含的指令数目
    int GetPSW();               //获取程序状态字，进程当前执行的指令编号
    int GetInstrucID(int);          //获取指定的指令的编号
    int GetInstrucState(int);       //获取指定的指令的状态标志
    int GetInstrucTimes(int);       //获取指定的指令的运行时间
    int GetCurrentRunTime();        //获取已经运行的时间
    int Get_P_PRI();                //获取p_pri
    void SetProID(int);             //设置进程编号
    void SetPriority(int);          //设置进程优先数
    void SetInTimes(int);           //设置进城创建时间
    void SetProState(int);          //设置进程状态
    void SetRunTimes(int);          //设置进程运行时间
    void SetInstrucNum(int);        //设置进程中包含的指令数目
    void SetPSW(int);           //设置程序状态字，进程当前执行的指令编号
    void SetInstrucID(int, int);        //设置指定的指令的编号
    void SetInstrucState(int, int);  //设置指定的指令的状态标志
    void SetInstrucTimes(int, int);  //设置指定的指令的运行时间
    void SetCurrentTime(int);           //设置已经运行的时间

    void ProcessCreate();           //进程控制原语—进程创建
    void ProcessCancel(int);        //进程控制原语—进程撤销
    void ProcessBlocked();          //进程控制原语—进程阻塞
    void ProcessWakeUp();           //进程控制原语—进程唤醒

    bool GetIfUsed();               //返回PCB是否被用
    bool GetIfEnd();                //返回PCB是否已经结束
    int GetStartTime();             //返回PCB真正开始时间
    int GetEndTime();               //返回PCB结束的时间
    void SetStartTime(int);         //设置PCB真正开始的时间

    void P_CPU_Add();               //p_cpu+1
    void PCPU_DECAY();              //p_cpu参数的衰减
```

```
    void Cal_P_PRI();              //计算该PCB的p_pri参数

    void InitWaitTime();           //初始化等待状态时间
    void WaitTimeSelfAdd();        //等待时间自增
    int GetWaitTime();             //返回已经处于等待态的时间
};
```

5. 进程调度模块　ProcessSchedule.h

```
class ProcessSchedule
{
private:
    queue<PCB>running_queue;       //运行队列
    queue<PCB>ready_queue;         //就绪队列
    queue<PCB>wait_queue;          //等待队列
public:
    ProcessSchedule();             //默认构造函数
    void ValueReset();             //重新设置初始化参数
    void JoinRunningQueue(PCB&, FileOperation &, int); //加入运行队列
    void JoinReadyQueue(PCB&, FileOperation &, int);   //加入就绪队列
    void JoinWaitQueue(PCB&, FileOperation &, int);    //加入等待队列
    void OutReadyQueue(int);       //将特定序号的PCB从就绪队列中移除
    void OutRunningQueue(int);     //将特定序号的PCB从就绪队列中移除
    void OutWaitQueue(int);        //将特定序号的PCB从就绪队列中移除
    int GetReadyQueueSize();       //获取就绪态队列长度
    int GetRunningQueueSize();     //获取就绪态队列长度
    int GetWaitQueueSize();        //获取就绪态队列长度
    PCB& GetReadyQueueOnlyID();    //获取就绪态仅有的唯一的元素
    void RunningQueuePop();        //运行态序列弹出
    int GetRunningQueueFirstID();//取得当前运行队列中的进程的ProID

    string ShowRunningQueue();     //展示运行队列的所有进程编号
    string ShowReadyQueue();       //展示就绪队列的所有进程编号
    string ShowWaitQueue();        //展示等待队列的所有进程编号
    string GetAllRunningQueue();  //获得运行队列的所有进程编号
    string GetAllReadyQueue();     //获得就绪队列的所有进程编号
    string GetAllWaitQueue();      //获得等待队列的所有进程编号

    PCB& GetRunningPCB();      //获取当前运行队列中的PCB
    bool IfExistRunningPCB(); //查看当前运行队列中是否有正在运行的进程
};
```

6. 全局常量数据模块　OverallData.h

```
constexpr int BREAK_TIME_LENGTH = 10;           //中断产生时间
constexpr int TIMESLICE_TIME_LENGTH = 1000;     //时间片产生时间

//PCB中的指令结构体
```

```
struct PCB_Instruc
{
    int Instruc_ID;       //指令编号
    int Instruc_State;    //指令类型
    int Instruc_Times;    //指令运行时间
};

//指令状态枚举体
enum INSTRUC_STATE
{
    INSTRUC_STATE_SYSTEMCALL,        //指令类型——系统调用
    INSTRUC_STATE_USERMOD_OPERATION, //指令类型——用户态计算操作
    INSTRUC_STATE_PV_OPERATION       //指令类型——PV操作
};

//PCB状态枚举体
enum PCB_STATE
{
    PCB_STATE_READY,      //PCB状态——就绪态
    PCB_STATE_RUNNING,    //PCB状态——运行态
    PCB_STATE_WAIT,       //PCB状态——等待态
    PCB_STATE_OVER        //PCB状态——结束
};

constexpr int PCB_INIT_NUM = -1;     //PCB的初始化数值

constexpr int INSTRUC_INIT_NUM = -1;//指令的初始化数值

constexpr int SYSTEM_MAX_PCB_NUM = 10;  //系统最大的PCB表的数量
constexpr int MAX_PRIORITY_NUM = 127;   //系统最大的优先数
constexpr int MAX_INTIMES_INTERVAL = 20;//两个PCB产生之间的最大时间间隔
constexpr int MIN_INTIMES_INTERVAL = 10;//两个PCB产生之间的最小时间间隔
constexpr int MAX_INSTRUC_NUM = 20;     //一个PCB中包含的最大的指令数目
constexpr int MIN_INSTRUC_NUM = 5;      //一个PCB中包含的最小的指令数目
constexpr int MAX_INSTRUC_TIMES = 4;    //一条指令最多需要运行的时间
constexpr int MIN_INSTRUC_TIMES = 1;    //一条指令最少需要运行的时间

enum CPU_STATE
{
    USER_MODE,     //用户态
    CORE_MODE      //核心态
};

constexpr int PV_INSTRUCTION_TIME = 50;     //PV 操作指令的时间
```

7. CPU 模块　CPU.h

```
class CPU
{
private:
    int PC;                         //程序计数器，当前正在运行的进程序号
    PCB_Instruc IR;                 //指令寄存器，当前正在运行的指令状态
    int PSW;                        //状态寄存器，当前正在运行的指令编号
    int current_time;               //指令已经运行的时间
    int cpu_state;                  //CPU的状态
    int wait_time;                  //CPU的等待时间
    PCB store;                      //传入到CPU运行的PCB

    bool if_could_go;               //CPU是否可以自增的标识
public:
    CPU();                          //默认构造函数
    void ValueReset();              //重新设置初始化参数
    int GetPC();                    //获取PC程序计数器
    PCB_Instruc GetIR();            //获取IR指令寄存器
    int GetPSW();                   //获取PSW状态寄存器
    int GetCpuState();              //获取CPU的状态
    PCB& GetStorePCB();             //获取当前存储的PCB
    int GetWaitTime();              //获取等待时间
    void SetPC(int);                //设置PC程序计数器
    void SetIR(PCB_Instruc);        //设置IR指令寄存器
    void SetPSW(int);               //设置PSW状态寄存器
    void SetCpuState(int);          //设置CPU的状态
    void SetStorePCB(PCB&);         //设置当前存储的PCB
    void SetWaitTime(int);          //设置等待时间
    void WaitTimeSelfAdd();         //等待时间自增
    void RefreshPSW();              //刷新PSW状态

    void RecoverSpot(PCB&);         //恢复现场
    void ProtectSpot(PCB&);         //保护现场

    void SetUserState();            //CPU切换成用户态
    void SetCoreState();            //CPU切换成核心态

    void CurrentRunTimeSelfAdd(int, FileOperation&);//已经运行时间自增
    void RefreshOverRunTime();       //恢复已经超出的实际运行时间
    bool CheckIfOK();                //查看进程是否已经运行完成
    bool CheckIfBorder();            //进程是否运行到了时间边界
    void CheckIfCouldGo(int);        //检查是否可以运行下一条指令
    int UpdateCPUMode();             //更新CPU的当前模式，用户态、核心态
};
```

8. 控制模块　Control.h

```
class Control
{
private:
    FileOperation fo;
    TaskRequest tr;
    ProcessSchedule ps;
    TimeClock tc;
    CPU cp;
    int cpu_current_time;
    bool if_close_timeslice_break;   //是否关闭时间片中断
    int last_priority_ID;            //上一次位于运行态的进程的序号
    int break_time_num;              //中断——时间片计数
    int time_slice_num;              //时间片计数
    int receive_break;               //实际收到的中断计数
public:
    Control();                       //默认构造函数
    void ValueReset();               //重新设置初始化参数
    void MenuShow();                 //显示菜单选项
    void ChoiceOperation(int);       //对应的选项操作
    void AnyKeyToBackToMenu();       //询问是否返回主菜单
    void ReCreateAllPCB();           //重新生成全部的PCB文件并写入到文件
    void UnixDispatch();             //Unix新进程调度算法实现
    void CloseTimeSliceBreak();      //关时间片中断
    void OpenTimeSliceBreak();       //开时间片中断
    int GetMaxPriorityProcessID();
    //通过p-pri比较，求得p-pri最小，即优先权最高的进程ID
    string GetOnePCBInfo_TimeSlice(int);
    //一次时间片下，获取某一个PCB进程的详细信息
    void TaskRequestThread();            //任务请求模块 线程
    void TimeSliceThread();              //时间片调度模块 线程
    void PCBRunThread();                 //进程运行模块 线程
    void AllPCBInfoShow_TimeSlice();     //一次时间片下，各进程状态输出
    void ShowCurrentTimeSlice();         //显示当前时间片时间
    void RefreshCpuTime();               //刷新当前cpu时间
    void LastTimeSliceDone();            //最后一个时间片的善后处理
    void LastShowAllInfo();      //当进程运行完成时，输出全部的详细信息
    void ShowSavedFileInfo();            //显示保存文件的信息
    void CheckIfCloseTimeSlice(PCB&);    //检测是否需要关闭
    void PVOperation(PCB&);
    //2型指令的操作——进入阻塞队列、重新导入、50ms的操作
    void PVOperationTimeAdd();
    //处于等待态的PCB进程当前等待时间自增
    void CheckIfPVOperationOK();
```

```
    //检查所有的PCB中，等待时间是否已经结束
    void PVProcessToReady(PCB&);          //将等待态的PCB进程放入就绪态
    void ShowEveryTimeDetail();           //显示每个时间点的详细信息
};
```

（二）附件 2：全 1 型指令数据

ProID	Priority	InTimes	RunTimes	InstrucNum	PSW	Instruc_ID	Instruc_State	Instruc_Times
1	59	0	320	11	1	1	1	30
						2	1	30
						3	1	30
						4	1	40
						5	1	20
						6	1	30
						7	1	30
						8	1	30
						9	1	40
						10	1	30
						11	1	10
2	55	130	170	6	1	1	1	40
						2	1	20
						3	1	10
						4	1	30
						5	1	30
						6	1	40
3	61	300	330	14	1	1	1	10
						2	1	30
						3	1	10
						4	1	30
						5	1	20
						6	1	10
						7	1	20
						8	1	40
						9	1	40
						10	1	30
						11	1	20
						12	1	30
						13	1	10
						14	1	30
4	108	300	440	14	1	1	1	30
						2	1	30
						3	1	40
						4	1	30
						5	1	30

						6	1	30
						7	1	20
						8	1	20
						9	1	40
						10	1	40
						11	1	30
						12	1	30
						13	1	40
						14	1	30
5	68	640	370	17	1	1	1	30
						2	1	30
						3	1	40
						4	1	30
						5	1	20
						6	1	10
						7	1	10
						8	1	20
						9	1	20
						10	1	10
						11	1	20
						12	1	10
						13	1	20
						14	1	20
						15	1	30
						16	1	20
						17	1	30
6	42	740	330	12	1	1	1	30
						2	1	30
						3	1	40
						4	1	30
						5	1	30
						6	1	30
						7	1	30
						8	1	10
						9	1	30
						10	1	20
						11	1	30
						12	1	20
7	63	860	160	7	1	1	1	20
						2	1	30
						3	1	30
						4	1	30
						5	1	10

						6	1	10
						7	1	30
8	29	1060	470	20	1	1	1	10
						2	1	30
						3	1	20
						4	1	10
						5	1	20
						6	1	30
						7	1	10
						8	1	30
						9	1	10
						10	1	10
						11	1	40
						12	1	40
						13	1	30
						14	1	30
						15	1	30
						16	1	30
						17	1	30
						18	1	30
						19	1	10
						20	1	20
9	73	1240	170	8	1	1	1	10
						2	1	30
						3	1	30
						4	1	30
						5	1	30
						6	1	10
						7	1	10
						8	1	20
10	60	1420	520	19	1	1	1	40
						2	1	10
						3	1	30
						4	1	40
						5	1	30
						6	1	30
						7	1	30
						8	1	20
						9	1	30
						10	1	20
						11	1	30
						12	1	10
						13	1	20

						14	1	30
						15	1	30
						16	1	40
						17	1	20
						18	1	30
						19	1	30

（三）附件 3：全 0 型指令与 1 型指令数据

ProID	Priority	InTimes	RunTimes	InstrucNum	PSW	Instruc_ID	Instruc_State	Instruc_Times
1	92	0	220	7	1	1	1	40
						2	0	30
						3	1	40
						4	0	40
						5	1	40
						6	1	10
						7	0	20
2	41	170	490	16	1	1	0	10
						2	1	40
						3	1	40
						4	1	40
						5	0	30
						6	0	20
						7	0	10
						8	1	40
						9	0	20
						10	0	20
						11	0	40
						12	0	40
						13	1	20
						14	1	40
						15	1	40
						16	1	40
3	58	300	160	6	1	1	0	10
						2	0	40
						3	0	10
						4	0	20
						5	0	40
						6	1	40
4	16	430	310	13	1	1	1	40
						2	1	20
						3	0	10
						4	1	10
						5	1	40

						6	1	40
						7	1	10
						8	1	20
						9	1	10
						10	1	30
						11	0	40
						12	0	30
						13	1	10
5	57	560	150	5	1	1	0	40
						2	0	10
						3	0	20
						4	1	40
						5	1	40
6	116	740	510	19	1	1	1	10
						2	1	20
						3	0	40
						4	0	20
						5	0	20
						6	1	20
						7	1	30
						8	1	40
						9	1	30
						10	1	20
						11	1	40
						12	0	30
						13	0	10
						14	1	40
						15	1	10
						16	0	30
						17	0	40
						18	1	20
						19	0	40
7	26	920	270	10	1	1	0	10
						2	0	10
						3	0	20
						4	0	40
						5	1	30
						6	1	20
						7	1	40
						8	1	40
						9	0	40
						10	1	20
8	3	1080	200	8	1	1	0	40

						2	1	30
						3	1	20
						4	0	10
						5	0	30
						6	1	10
						7	0	20
						8	0	40
9	101	1280	260	9	1	1	0	10
						2	1	10
						3	0	30
						4	0	40
						5	0	40
						6	1	10
						7	1	40
						8	1	40
						9	1	40
10	15	1480	320	11	1	1	0	40
						2	1	20
						3	0	40
						4	1	30
						5	1	30
						6	0	40
						7	0	10
						8	1	30
						9	1	30
						10	0	10
						11	1	40

（四）附件 4：混合指令数据

ProID	Priority	InTimes	RunTimes	InstrucNum	PSW	Instruc_ID	Instruc_State	Instruc_Times
1	119	0	290	9	1	1	2	50
						2	2	50
						3	0	30
						4	1	20
						5	1	30
						6	1	30
						7	1	10
						8	1	30
						9	1	40
2	21	200	820	20	1	1	2	50
						2	1	40
						3	2	50
						4	1	20

						5	2	50
						6	2	50
						7	2	50
						8	2	50
						9	0	30
						10	0	20
						11	1	40
						12	2	50
						13	2	50
						14	2	50
						15	0	30
						16	1	40
						17	1	10
						18	0	40
						19	2	50
						20	2	50
3	22	380	220	6	1	1	2	50
						2	0	40
						3	2	50
						4	1	30
						5	0	30
						6	0	20
4	52	580	560	14	1	1	2	50
						2	0	10
						3	2	50
						4	2	50
						5	2	50
						6	2	50
						7	1	40
						8	2	50
						9	0	10
						10	2	50
						11	2	50
						12	1	40
						13	0	10
						14	2	50
5	104	760	400	13	1	1	0	30
						2	0	10
						3	2	50
						4	1	30
						5	2	50
						6	1	30
						7	1	10

						8	1	30
						9	2	50
						10	0	40
						11	1	30
						12	0	30
						13	0	10
6	103	890	370	11	1	1	2	50
						2	0	40
						3	1	10
						4	1	40
						5	1	10
						6	0	30
						7	0	20
						8	1	40
						9	0	40
						10	2	50
						11	1	40
7	94	1060	310	9	1	1	0	20
						2	1	40
						3	2	50
						4	0	40
						5	0	30
						6	1	10
						7	1	30
						8	2	50
						9	1	40
8	85	1220	460	16	1	1	1	20
						2	0	10
						3	1	40
						4	1	30
						5	0	20
						6	1	40
						7	0	20
						8	2	50
						9	0	30
						10	1	10
						11	2	50
						12	0	10
						13	0	30
						14	0	10
						15	0	40
						16	2	50
9	28	1320	610	19	1	1	2	50

						2	2	50
						3	2	50
						4	0	20
						5	0	40
						6	0	20
						7	0	20
						8	0	20
						9	0	20
						10	2	50
						11	1	10
						12	1	40
						13	0	10
						14	2	50
						15	1	40
						16	2	50
						17	2	50
						18	0	10
						19	1	10
10	57	1490	200	6	1	1	1	30
						2	2	50
						3	2	50
						4	0	20
						5	0	10
						6	0	40

（五）附件 5：全 1 型指令输出结果

见同目录下“附件 5：全 1 型指令输出结果.txt”

（六）附件 6：全 0 型指令与 1 型指令输出结果

见同目录下“附件 6：全 0 型指令与 1 型指令输出结果.txt”

（七）附件 7：混合指令输出结果

见同目录下“附件 7：混合指令输出结果.txt”