合肥工業大學

操作系统课程设计报告

设计题目	高中低三级调度系统模型
学生姓名	曲艺
学　　号	2020218037
专业班级	计科 20-4 班
指导教师	田卫东
完成日期	2023.5.30

1. 课程设计任务、要求、目的

1.1 课程设计任务

依据操作系统课程所介绍的操作系统进程和作业的多种调度算法方案，按照内核代码的实现原则，设计和实现高中低三级调度系统模型的系统内核模块。

1.2 课程设计目的和要求

系统应该包含两个部分，一个部分是按内核代码原则设计的高中低三级调度系统模型，由一系列的函数组成；另一个部分是演示系统，调用高中低三级调度系统模型，以让其运行，同时提供系统的展示界面，可以是 GUI 或者字符界面，以展示系统的运行状态，显示系统的关键数据结构的内容。

具体包括：

①建立操作系统内核中的作业调度、进程调度、中级调度的三级队列调度模型；

②使用两种方式产生作业/进程：

a. 自动产生，

b. 手工输入；

③构建作业控制块和进程控制块等核心数据结构；

④计算并显示一批作业，从创建作业、创建进程，到作业个进程在诸队列上流转的完整过程。

⑤将一批作业/进程的执行情况存入磁盘文件，以后可以读出并重放；

⑥每类调度需要实现一种调度算法。

⑦时间的流逝可用下面几种方法模拟：

a. 按键盘，每按一次可认为过一个时间单位；

b. 响应 WM_TIMER；

2. 开发环境

操作系统：Windows 11 家庭中文版（版本号：21H2）

开发工具：Visual Studio 2022、Qt6

编程语言：C/C++

3. 相关原理及算法

（1）作业调度算法：

①先来先服务（FCFS）调度算法：按照作业到达的顺序进行调度，先到先服务。

②短作业优先（SJF）调度算法：选择估计运行时间最短的作业进行调度。

③优先级调度算法：为每个作业分配一个优先级，根据优先级进行调度。
④高响应比优先（HRRN）调度算法：根据作业等待时间和估计运行时间的比
⑤率进行调度，提高响应性能。

（2）进程调度算法：
①轮转法调度算法：按照固定时间片轮流分配给各个进程，实现时间片轮转。
②最短剩余时间优先（SRTF）调度算法：选择剩余执行时间最短的进程进行调度。
③多级反馈队列调度算法：根据进程的优先级和时间片大小将进程分配到不同的队列，实现多级调度。

（3）中级调度算法：
①抢占式优先级调度算法：根据作业的优先级进行调度，可抢占正在执行的进程。
②非抢占式优先级调度算法：根据作业的优先级进行调度，不可抢占正在执行的进程。

（4）作业控制块（JCB）和进程控制块（PCB）：
①作业控制块：包含作业的相关信息，如作业标识符、状态、优先级等。
②进程控制块：包含进程的相关信息，如进程标识符、状态、寄存器内容、内存分配信息等。

（5）作业和进程的创建与流转过程：
①作业创建：生成作业的控制块，并将其加入作业队列。
②进程创建：在作业被调度执行时，根据作业的要求生成相应的进程，并将其加入进程队列。
③进程流转：根据调度算法将进程从一个队列调度到另一个队列，直至完成执行。

4. 系统结构和主要的算法设计思路

4.1 系统架构

4.1.1 算法层面：

整个模拟系统分为三个主要部分。

第一部分是"作业的读取、加载和记录模块"，该模块的主要功能是将用户存放在文件中的作业加载到系统中，并生成内存中的 UserJob 队列。

第二部分是"作业的模拟执行模块"，该模块模拟操作系统对各种事件的

处理，包括用户的主动挂起请求和虚拟 CPU 的模拟执行。

第三部分是操作系统的三级调度模块，包括作业调度、作业调度和中间调度。这三种调度算法会被实现并嵌入第二部分的系统中，用于进行调度处理。

4.1.2 可视化层面：

使用 Qt 绘制可视化展示界面，将结构体的内容按照队列的形式绘制出来，以更加形象的方式展现操作系统的三级调度过程。

为用户提供：

a.手动添加作业功能

b.单步执行功能

c.挂起功能

d.回放功能

4.2 整体算法思想

4.2.1 整体的工作流程如下：

首先进入 VMEcect 函数，该函数运行一次就相当于操作系统经历了一个时间片，在这个函数里主要做三件事：

①判断模拟系统所处的模式——“执行”和“回放”

②运行对应的“前进”函数

③更新每一个队列的长度

（下面对“执行”模式进行讲解。）

进入 run 函数后，首先处理用户提出的挂起请求。通过调用相应的函数或按钮事件处理函数，处理挂起就绪队列和挂起阻塞队列的操作。

然后为就绪队列中的一个进程分配一个 CPU 时间片。这意味着将 CPU 的控制权交给该进程，让其执行一定的指令。这个过程可以通过调用相应的函数来模拟。

在 CPU 执行过程中，判断当前进程是否需要进行输入/输出操作（IO）。如果需要 IO，将该进程从就绪队列中移出，并将其送往阻塞队列。这个过程也可以通过调用相应的函数来模拟。

接下来，进行低级调度。低级调度的目标是从就绪队列中选择下一个要执行的进程。这个过程可以根据一定的调度算法（如轮转调度、优先级调度等）来进行。在这个过程中，可能需要调用相关的数据结构和算法来选择下一个进程。

经过上述过程后，如果就绪队列仍然没有被填满，才会开始进行中级调度。中级调度的目标是从就绪挂起队列、阻塞挂起队列和挂起队列中选择进程，将其移入就绪队列，以提高系统的资源利用率。

最后，如果在"就绪队列"、"就绪挂起队列"、"阻塞挂起队列"和"挂起队列"中的所有进程总数小于最大进程数（MAX_JOB），才会进行高级调度。高级调度的目标是根据一定的策略和算法，从所有队列中选择进程进行调度和分配资源，以保证整个系统的性能和公平性。

OS_View 类的 run 函数模拟了操作系统中的事件处理和 CPU 执行程序的过程，通过处理挂起请求、分配 CPU 时间片、处理 IO 操作、进行低级调度和高级调度等步骤来管理和调度进程，以实现操作系统的正常运行和资源管理。

整体算法流程图如下：

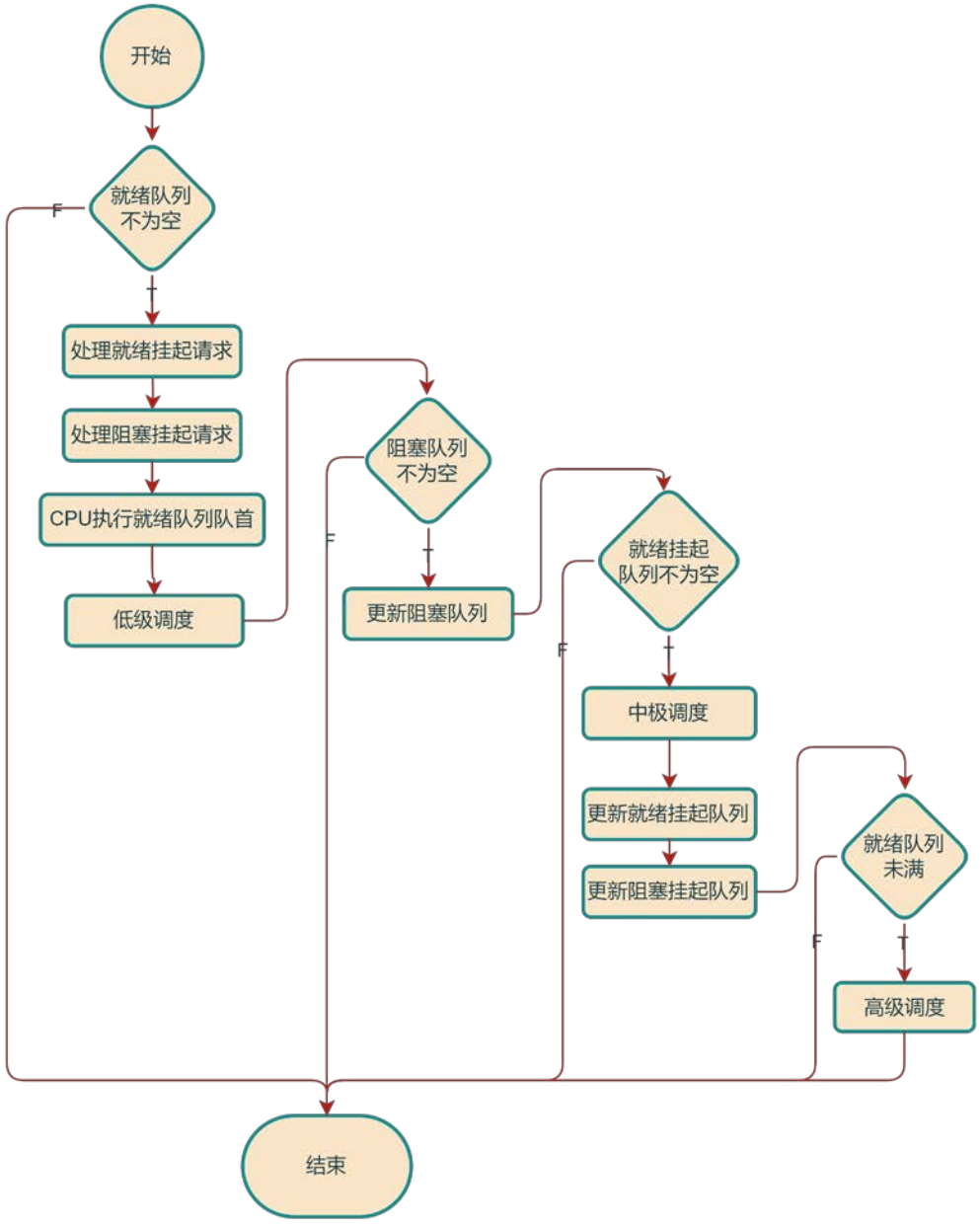

图 4-1 整体算法流程图

4.3 调度算法方面

4.3.1 高级调度——短作业优先。

在低级调度由作业生成进程的过程中，是将作业一次生成一个进程，然后将该进程放入就绪队列中。在低级调度时会遍历一遍后备队列，选取预计执行时间最短的作业，将其放到后备队列队首，然后对每一个作业一次生产一个进程，当内存中的作业少于 10 个时候进入添加。

本程序实现的高级（作业）调度算法流程图如下：

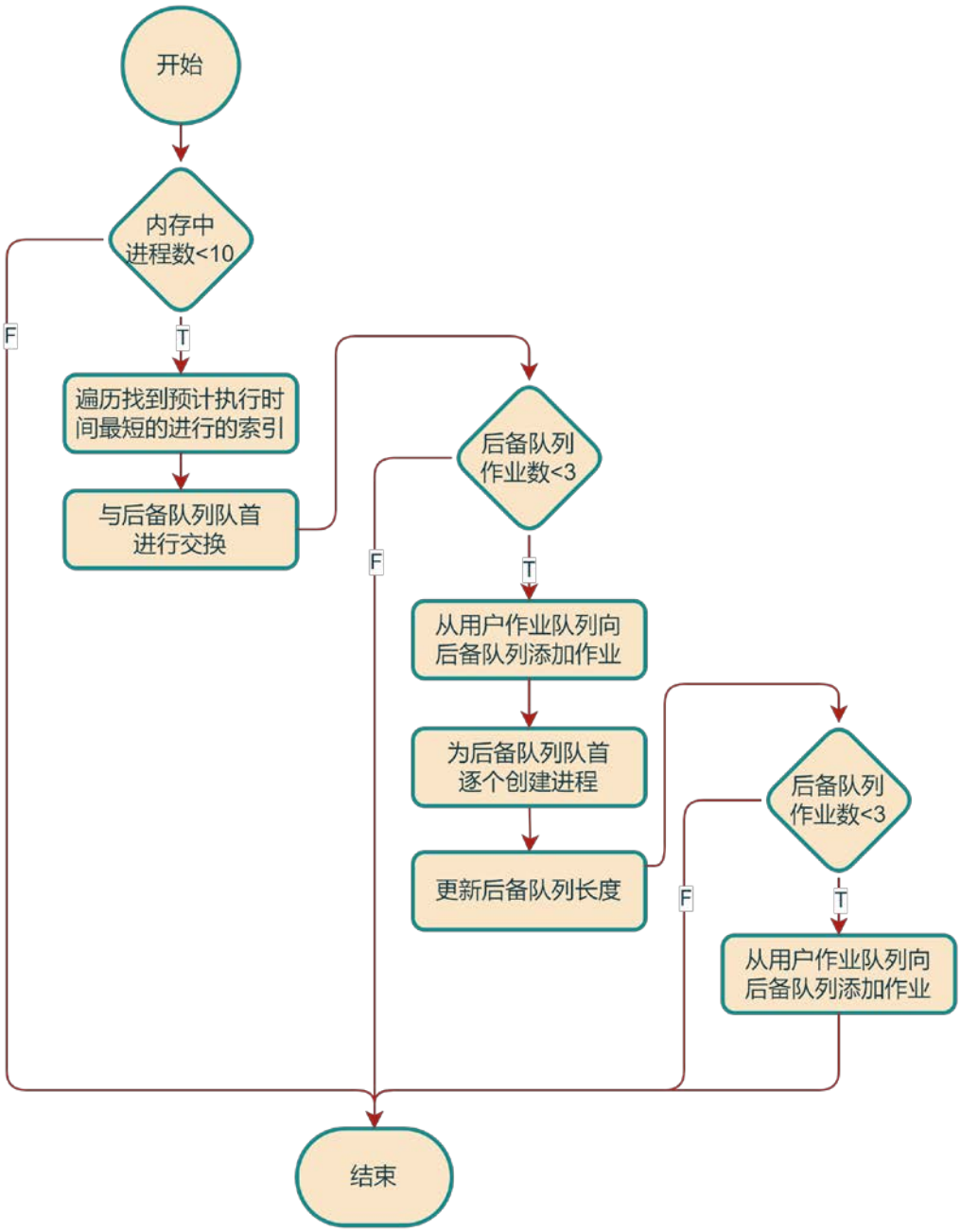

图 4-2 高级调度：短作业优先算法流程图

作业调度的伪代码如下：

Procedure 1 Job Scheduling

Input: Job queue $JobQueue$, Ready queue $readyQueue$, User job $UserJob$

1: **if** $(readyQueueLen + blockQueueLen + readySuspendedQueueLen + blockedSuspendedQueueLen) < 10$ **then**
2: **for** $i \leftarrow 0$ **to** $JobQueueLen - 2$ **do**
3: $min \leftarrow i$
4: **for** $j \leftarrow i + 1$ **to** $JobQueueLen - 1$ **do**
5: **if** $JobQueue[min].time\_spent > JobQueue[j].time\_spent$ **then**
6: $min \leftarrow j$
7: **end if**
8: **end for**
9: **if** $min \neq i$ **then**
10: $tmp \leftarrow JobQueue[i]$
11: $JobQueue[i] \leftarrow JobQueue[min]$
12: $JobQueue[min] \leftarrow tmp$
13: **end if**
14: **end for**
15: **if** $JobQueueLen \leq 3$ **and** $UserJob[0].valid =$ **true then**
16: **updateJobQueue**($JobQueue$, $UserJob$)
17: **end if**
18: **for** $i \leftarrow 0$ **to** $JobQueueLen - 1$ **do**
19: **while** $JobQueue[i].taken \neq 0$ **do**
20: $totalPcb \leftarrow totalPcb + 1$
21: $index \leftarrow readyQueueLen$
22: $readyQueue[index].PID \leftarrow totalPcb$
23: $readyQueue[index].need\_time \leftarrow (JobQueue[i].time\_spent/JobQueue[i].taken)+2$
24: $readyQueue[index].io \leftarrow JobQueue[i].io$
25: $readyQueue[index].io\_when \leftarrow readyQueue[index].need\_time/2$
26: $readyQueue[index].io\_finish \leftarrow (JobQueue[i].io =$ **TRUE**?**FALSE** : **TRUE**)
27: $readyQueue[index].priority \leftarrow JobQueue[i].priority$
28: $readyQueue[index].suspend\_time \leftarrow 0$
29: $readyQueue[index].wait\_time \leftarrow 0$
30: $readyQueueLen \leftarrow readyQueueLen + 1$
31: $JobQueue[i].time\_spent \leftarrow JobQueue[i].time\_spent - (JobQueue[i].time\_spent/JobQueue[i].taken)$
32: $JobQueue[i].time\_spent \leftarrow (JobQueue[i].time\_spent < 0?0 : JobQueue[i].time\_spent)$
33: $JobQueue[i].taken \leftarrow JobQueue[i].taken - 1$
34: **if** $JobQueue[i].taken \leq 0$ **then**
35: **for** $j \leftarrow 0$ **to** $JobQueueLen - 2$ **do**
36: $JobQueue[j] \leftarrow JobQueue[j + 1]$
37: **end for**
38: $JobQueueLen \leftarrow JobQueueLen - 1$
39: **end if**
40: **if** $JobQueueLen = 0$ **then**
41: **updateJobQueue**($JobQueue$, $UserJob$)
42: **end if**
43: **if** $readyQueueLen = MAX\_readyQueue$ **then**
44: **updateJobQueue**($JobQueue$, $UserJob$)
45: **return**
46: **end if**
47: **end while**
48: **end for**
49: **end if**

4.3.2 中级调度——先来先服务。

中级调度主要是对于“就绪挂起队列”、“阻塞挂起队列”进行的调度，每当这两个挂起队列队首的进程挂起时间结束，
并且下一级队列有空位时，就会将队首的进程放到目标队列中去。

中级调度算法流程图：

图 4-3 中级调度：先来先服务算法流程图

中级调度算法伪代码：

Procedure 2 Middle Scheduling

Input: Ready queue $readyQueue$, Ready suspended queue $readySuspendedQueue$

1: $readyQueue[readyQueueLen] \leftarrow readySuspendedQueue[0]$
2: $readyQueueLen \leftarrow readyQueueLen + 1$
3: **for** $j \leftarrow 0$ **to** $readySuspendedQueueLen - 2$ **do**
4: $\quad readySuspendedQueue[j] \leftarrow readySuspendedQueue[j+1]$
5: **end for**
6: $readySuspendedQueueLen \leftarrow readySuspendedQueueLen - 1$

4.3.3 低级调度——高优先级调度。

低级调度是对于“就绪队列”进行调度，为了能够实现高优先级调度，我在进程的结构体中加上了等待时间 WT 和需要执行的时间 NT，优先级 $PR = \frac{WT + NT}{NT}$，通过遍历并且更新就绪队列，能够找到优先级最高的进程，然后将其放到队首，为它分配 CPU，当它获得了 CPU 处理之后，就将

WT 置为 0，降低自己的优先级，让其他进程有机会获得处理机。

算法流程图如下：

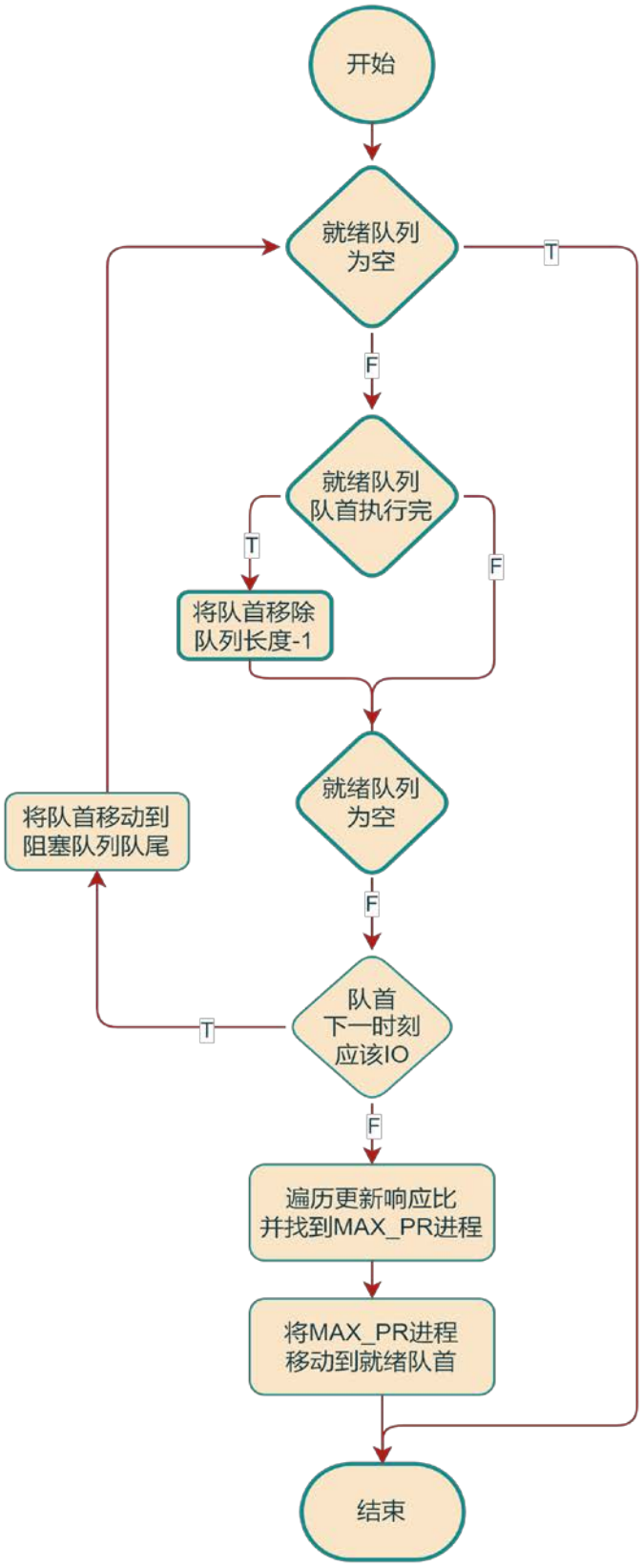

图 4-5 低级调度：高优先权调度算法流程图

算法伪代码如下：

Procedure 3 Process Scheduling

Input: Ready queue *readyQueue*, Block queue *blockQueue*

```
1: if readyQueueLen ≠ 0 then
2:   if readyQueue[0].need_time = 0 then
3:     removeFirstProcessScheduling(readyQueue, readyQueueLen)
4:   end if
5:   if readyQueueLen = 0 then
6:     return
7:   end if
8:   if readyQueue[0].io_finish = false and readyQueue[0].io_when = 0
     then
9:     blockQueue[blockQueueLen] ← readyQueue[0]
10:    blockQueue[blockQueueLen].io_exe ← 0
11:    blockQueueLen ← blockQueueLen + 1
12:    removeFirstProcessScheduling(readyQueue, readyQueueLen)
13:  else if readyQueue[0].io_finish = true and
     readyQueue[0].need_time = 0 then
14:    removeFirstProcessScheduling(readyQueue, readyQueueLen)
15:  end if
16:  for i ← 0 to readyQueueLen do
17:    readyQueue[i].wait_time ← readyQueue[i].wait_time + 1
18:    readyQueue[i].priority ← ⌊(readyQueue[i].wait_time+readyQueue[i].need_time) / readyQueue[i].need_time⌋
19:  end for
20:  max ← 0
21:  for i ← 0 to readyQueueLen do
22:    if readyQueue[max].priority < readyQueue[i].priority then
23:      max ← i
24:    end if
25:  end for
26:  tmp ← readyQueue[0]
27:  readyQueue[0] ← readyQueue[max]
28:  readyQueue[max] ← tmp
29: end if
```

5. 程序实现---主要数据结构

整个程序主要的三个数据结构是“UserJob——用户提交的作业”、“JCB——作业控制块”、“PCB——进程控制块”，这三个部分是整个程序的基础。

UserJob 用来存储从文件中加载的用户作业，JCB 用来存储进入后备队列中的作业，而 PCB 作为进程主要的结构体，在“就绪”、“阻塞”、“挂起”队列中都起着重要的作用，其中的“等待时间”和“执行时间”决定了“优先级”，而根据优先级可以进行高优先级调度等其他的调度算法。

对于 UserJob，根据用户提交时作业具备的属性设计了：

①“UserName”属性用于记录是哪个用户提交的作业

②“JID”用来区分和标识不同的作业

③“start_time”用来记录作业 d 提交时间

④“taken”用来标识该作业需要的进程数

⑤“time_spent”标识这个作业执行完预计需要花费的时间
⑥“io”用来标识该作业需不需要使用 IO 设备
⑦“priority”用来记录该作业的优先级
⑧“valid”用来标识该作业是否有效

对于作业控制块（JCB），根据作业的特性，设计了以下属性：
① "UserName" 属性用于记录提交该作业的用户的名字。
② "JID" 用来区分和标识不同的作业。
③ "taken" 标识该作业需要的进程数。
④ "time_spent" 表示该作业执行完预计需要花费的时间。
⑤ "io" 用来标识该作业是否需要使用 I/O 设备。
⑥ "io_time" 表示该作业进行 I/O 请求所需的时间片个数。
⑦ "priority" 表示该作业的优先级。

对于进程控制块（PCB），根据进程的特性，设计了以下属性：
① "PID" 表示进程的标识符。
② "io" 用来标识该进程是否需要使用 I/O 设备。
③ "io_when" 表示该进程在执行到第几个时间片时需要进行 I/O。
④ "need_time" 表示该进程需要的时间片数。
⑤ "io_exe" 表示已经执行的 I/O 时间。
⑥ "io_finish" 表示该进程的 I/O 是否完成，完成为真（TRUE），未完成为假（FALSE）。
⑦ "suspend_time" 表示进程已经挂起的时间。
⑧ "wait_time" 表示进程已经等待的时间。
⑨ "priority" 表示进程的优先级。

6. 程序实现---主要程序清单

6.1 三级调度算法核心程序

表 6-1 主要函数清单

函数名	函数参数 1	函数参数 2	函数参数 3	函数作用
readJob	JOB* UserJob			文件读取
JobScheduling	JCB* JobQueue	PCB* readyQueue	JOB* UserJob	高级调度（作业调度）
manaulAddJobToUserJob	JOB* UserJob			手动从控制台向用户提交的作业 UserJob 中添加作业
middleScheduling	PCB* readyQueue	PCB* readySuspend		中级调度:先来先服务

		edQueue		
processScheduling	PCB* readyQueue	PCB* blockQueue		低级调度:高优先权调度
removeFirstProcess Scheduling	PCB* pcbQueue	Int* len		将队列第一个移除
run	JOB* UserJob	JCB* JobQueue	PCB* readyQueue	PCB* blockQueue
JobScheduling	JCB* JobQueue	PCB* readyQueue	JOB* UserJob	高级调度：短作业优先
updateJobQueue	JCB* JobQueue	JOB* UserJob		往后备队列中添加新的作业

6.1.1 作业调度——JobScheduling

主要作用是将后备队列中的作业在合适的情况下产生子进程，然后将产生的子进程添加到就绪队列中，另外还需要负责从用户提交的作业中及时获取作业，保证后备队列空闲区域不能超过 30%，这样既能够保证较高的利用率，又能减少作业调度的次数。

短作业优先核心代码：

```
//最短时间优先：将预计时间最短的作业放到第一个位置
for (int i = 0; i < JobQueueLen-1; i++)
{
    int min = i; //假设第一个最小
    for (int j = i+1; j < JobQueueLen; j++)
    {
        if (JobQueue[min].time_spent > JobQueue[j].time_spent)
        {
            min = j;
        }
    }
    //交换位置
    if (min != i)
    {
        JCB tmp = JobQueue[i];
        JobQueue[i] = JobQueue[min];
        JobQueue[min] = tmp;
    }
}
```

6.1.2 中级调度——middleScheduling

主要作用是将“就绪挂起队列”中已将挂起时间足够的进程及时的放回就绪队列中让其执行，。

6.1.3 进程调度——processScheduling

主要作用是在每一次 CPU 执行结束之后更新就绪队列的内容，由于每次都是队列队首获得处理机，所以每次总是从队列队首开始处理，清除掉执行完的进程，将下一个周期需要 IO 的进程放到阻塞队列中进行 IO，这个持续

的处理，知道遇到第一个没有执行完并且不需要 IO 的进程就停止，然后将就绪队列中所有的进程的优先级进行更新，将优先级最高的进程放到队列队首，下一个周期队首将获得 CPU 的使用权。

进程调度核心代码：

```
//----------------------低级调度：高响应比优先调度算法 start-------------------------
//计算响应比
for (int i = 0; i < readyQueueLen; i++)
{
    readyQueue[i].wait_time += 1; //等待时间+1
    //响应比=（等待时间+要求服务时间）/要求服务时间
    readyQueue[i].priority = (int)((readyQueue[i].wait_time + readyQueue[i].need_time) /
                                                                readyQueue[i].need_time);
}
//找到最大响应比
int max = 0;
for (int i = 0; i < readyQueueLen; i++)
{
    if (readyQueue[max].priority < readyQueue[i].priority)
    {
        max = i;
    }
}
//将最大响应比的进程放到队首
PCB tmp = readyQueue[0];
readyQueue[0] = readyQueue[max];
readyQueue[max] = tmp;
//----------------------低级调度：高响应比优先调度算法 end-------------------------
```

6.2 调度可视化程序

调度可视化部分使用了 Qt6 调用预先生成好的.lib 文件完成的，并且我还完成了使用 MinGw 编译的 Linux 等系统适用的.a 静态连接库文件，与 Qt 项目进行配合，能够实现很好的代码不同平台的移植。

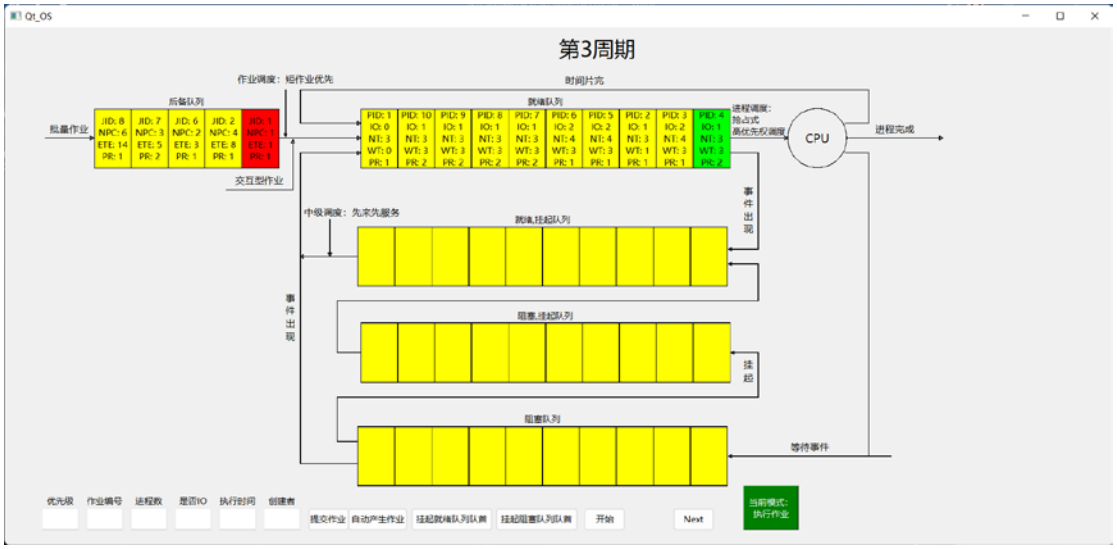

图 6-1 可视化效果

6.2.1 可视化逻辑的类——OS_View

该部分定义了时间触发器，用来模拟 CPU_time,每隔一定的时间就会自动执行一次 VMExect，由此实现了自动执行，另外 OS_View 还定义了“用户提交作业”、“自动生成作业”、“单步执行”、“模式选择”这些功能选择按钮，其中的函数实现是调用了预先编译好的.lib 静态连接库。

主要成员函数：

表 6-2 OS_View 主要函数清单

成员函数	中文注释
OS_View(QWidget *parent = nullptr);	构造函数
~OS_View();	析构函数
void paintEvent(QPaintEvent* event) override;	重写的绘图事件处理函数
void drawCpuGraphics(QPainter& painter, const QPoint& center, int radius);	绘制 CPU 图形
void drawLine(QPainter& painter, const QPoint& startPoint, const QPoint& endPoint, QString labelText, bool withArrow, int arrowPosition=RIGHT);	绘制线条
void VMExect();	模拟执行，一次执行一个 CPU_time
void recordData();	记录数据到文件
void replay();	重放函数
void readRecord(int exNum);	从文件中读取记录
void updateModeButtonColor();	更新按钮颜色

6.2.2 可视化数据展示类——MatrixRect

这个是一个矩形数组的类，这个类能够读取到和它绑定在一起的队列，并且将队列中的内容在每一个矩形内实时的绘制出来，让我们能够更加清楚的看到系统三级调度的过程。

成员函数：

表 6-3 MatrixRect 成员函数

成员函数	中文注释
MatrixRect(QWidget *parent = nullptr);	构造函数
void setRectangleColor(const QColor& color);	设置矩形颜色
void setMatrixPosition(const QPoint& position);	设置矩阵位置
void setMatrixCount(int count);	设置矩形个数
void updateData();	更新数据
void setQueueName(QString queueName);	设置队列名字
void paintEvent(QPaintEvent* event) override;	重写的绘图事件处理函数

成员变量：

表 6-4 MatrixRect 成员变量

变量类型	变量名	中文注释
int	Kind	默认自己的队列是非法类型
int	DIREC	默认在头在右边
int	maxLen	队列最多任务数量
int	nowLen	队列最少任务数量
JCB*	jcbQueue	进程队列+作业队列
PCB*	pcbQueue	进程控制块队列
int	QueueKind	队列类型
int	rectWidth	每个矩形的宽度
int	rectHeight	矩形的高度
QColor	m_rectangleColor	矩形颜色
QVector<int>	Num	矩形显示依赖数组
QPoint	m_matrixPosition	矩阵位置
int	m_matrixCount	矩形的个数
QString	queueName	队列的名字

7. 程序运行的主要界面和结果截图

（1）模拟过程的演示

下面首先展示程序执行过程中“就绪”、“阻塞”、“挂起”三种状态的转换过程，其中程序的阻塞是当进程中 IO=0 时自动阻塞，等待 IO 完成，而“挂起”操作则是用用户点击下方“挂起就绪队列队首”、“挂起阻塞队列队首”这两个按钮实现主动挂起，而程序的 IO 时间设定为两个周期，挂起需要等待的周期也是两个周期，才能够进入下一个队列。接下来是这个过程的展示：

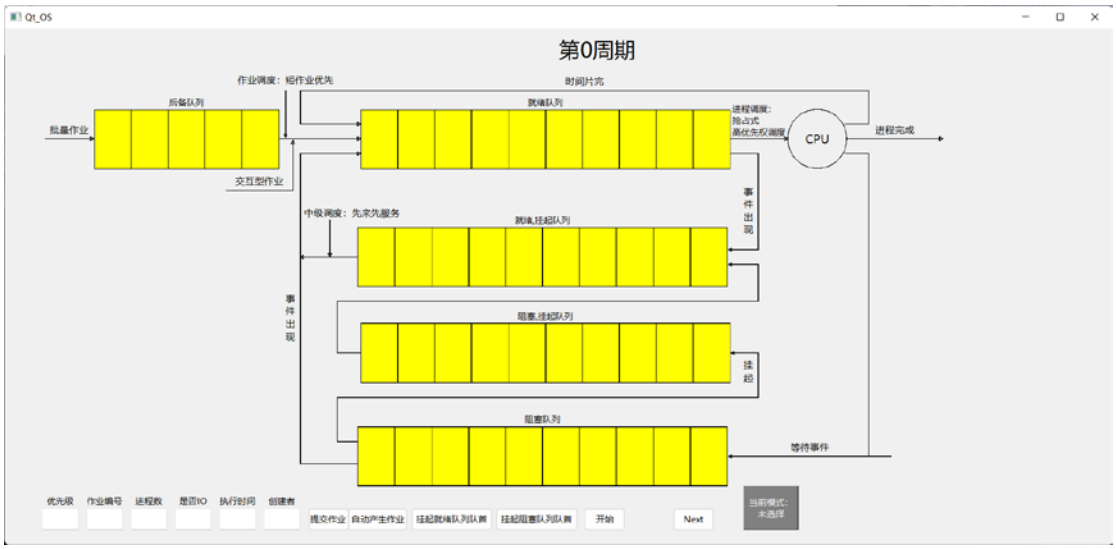

图 6-2 初始界面

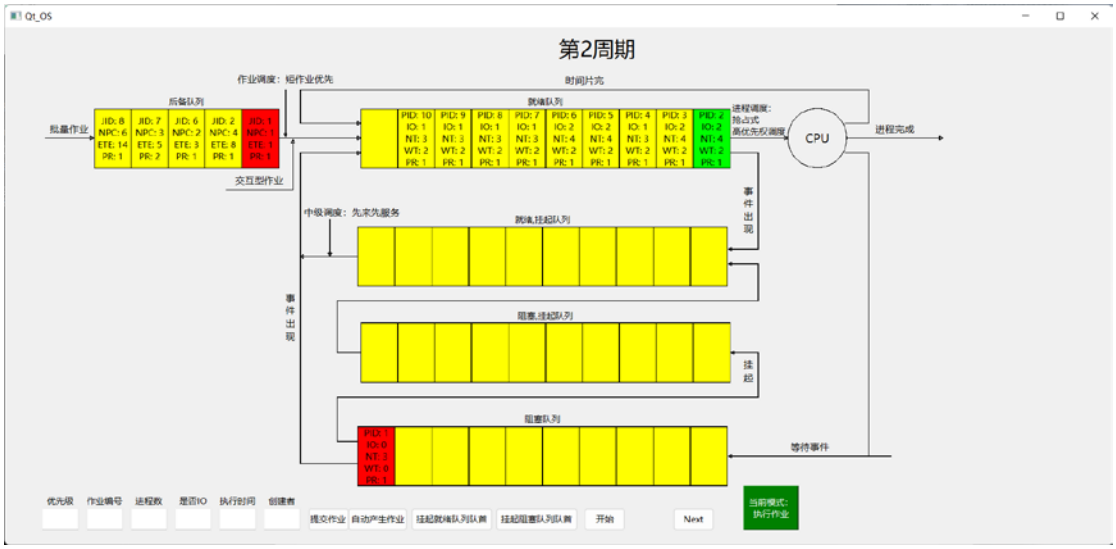

图 6-3 执行中状态

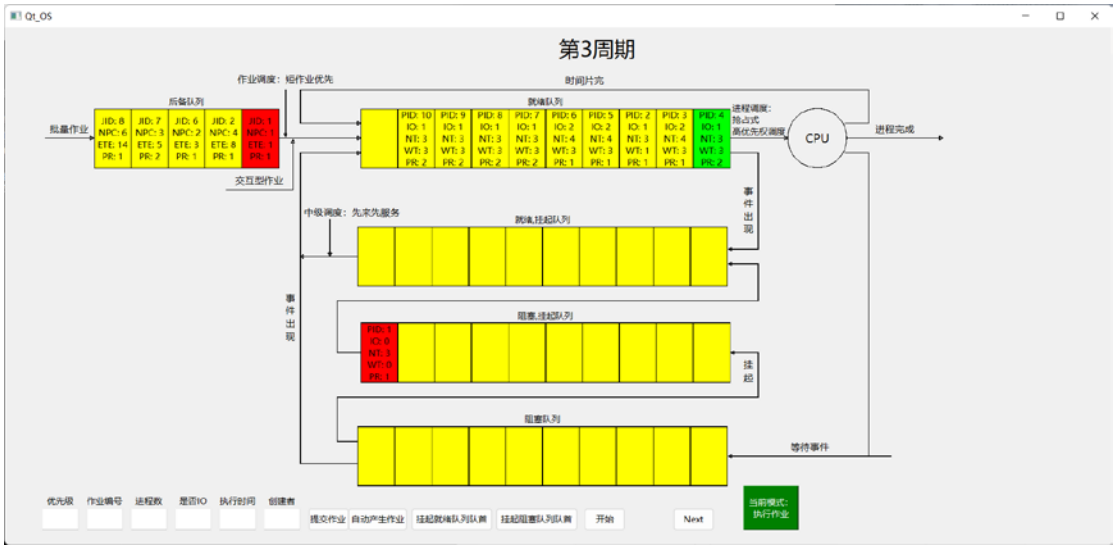

图 6-4 挂起 “阻塞队列队首 PID=1”

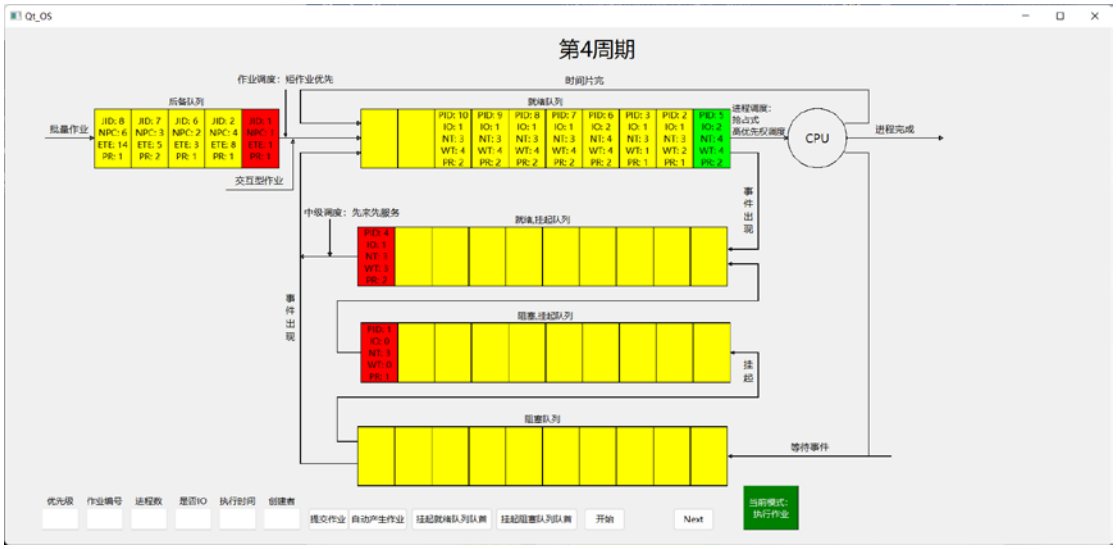

图 6-5 挂起“就绪队列队首 PID=4”

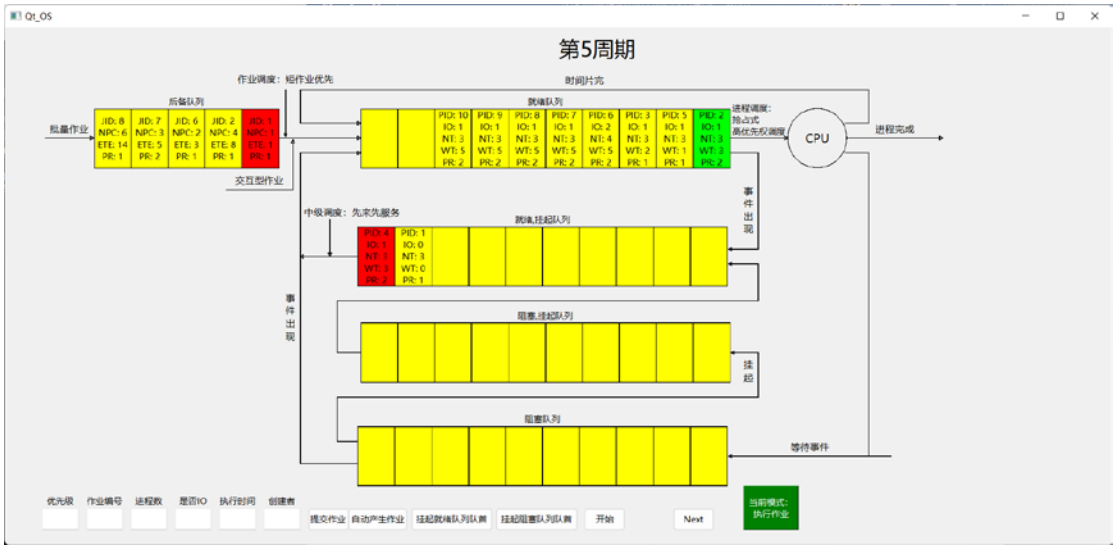

图 6-6 “阻塞挂队列队首 PID=1”进入“就绪挂起队列”

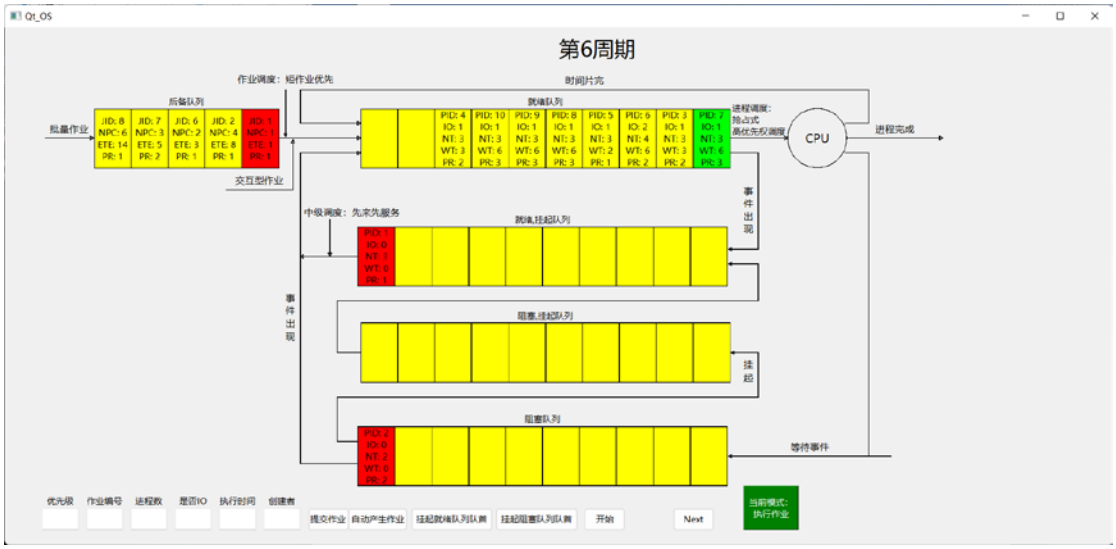

图 6-7 “就绪队列队首 PID=4”回到就绪队列队尾

（2）回放功能

程序在执行作业的时候，会将每一个周期的状态存放到以时间命名的文件中。这个文件包含了作业执行过程中的关键信息，方便用户进行后续的分析和调试。

用户可以点击右下角的“模式切换”按钮进行模式的切换，切换到回放模式可以选择文件，然后将该次执行的过程完整的重放一遍，重放的过程同样支持自动执行和点击单步执行。

通过提供模式切换和回放功能，我们的程序不仅能够有效执行作业，还能为用户提供更多的操作选择和全面的执行信息。无论是简单回顾还是深入分析，都能得到所需的结果。

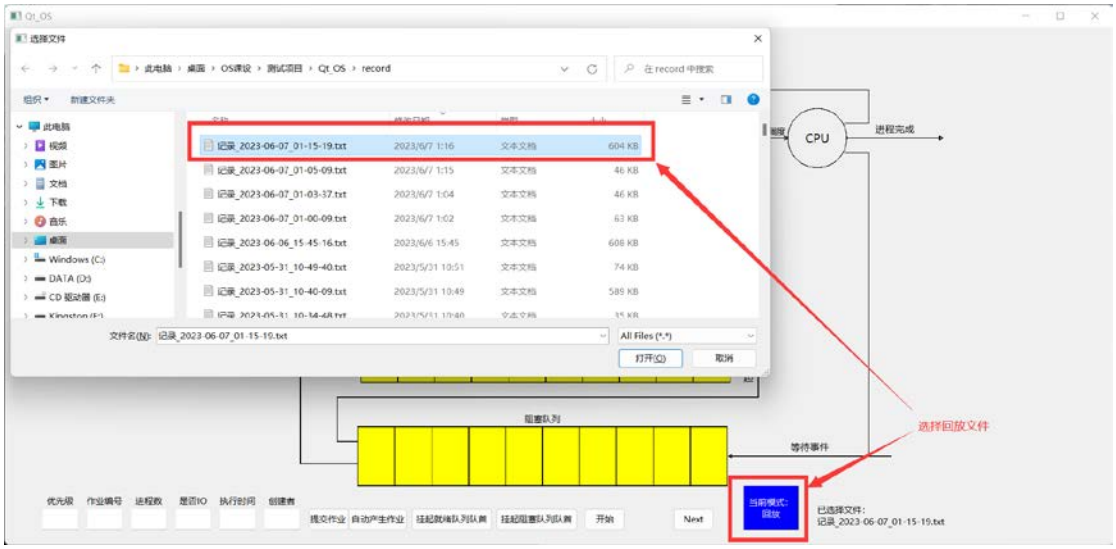

图 6-8 回放选择文件界面

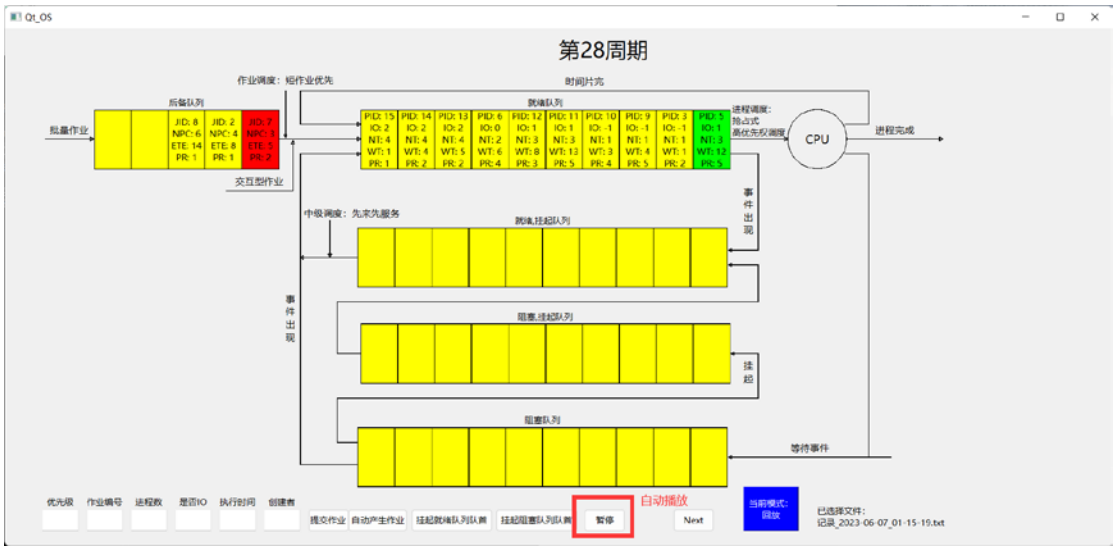

图 6-9 回放过程中

手动添加作业功能：

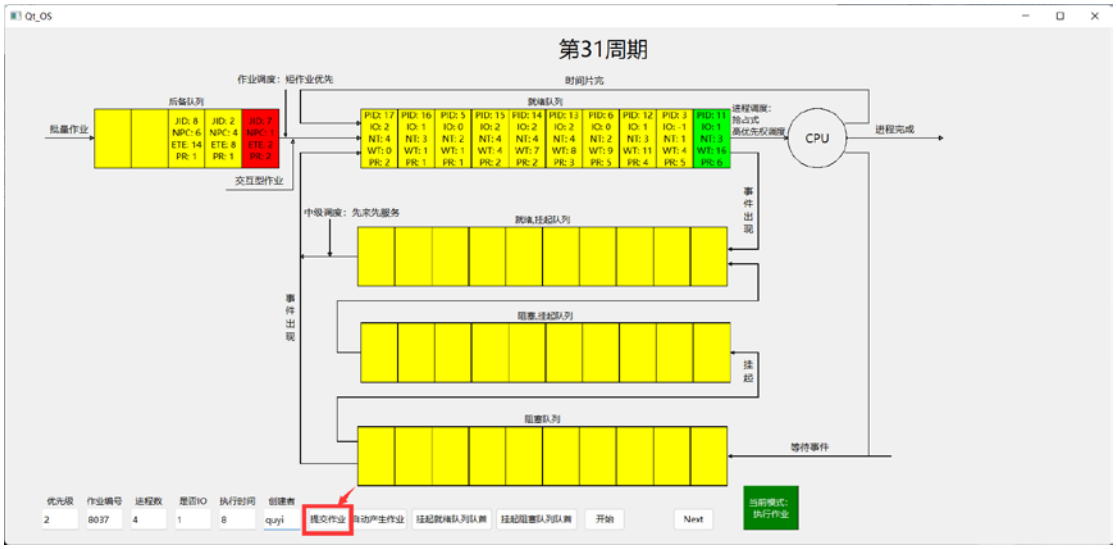

图 6-10 提交手动创建的作业 JID=8037

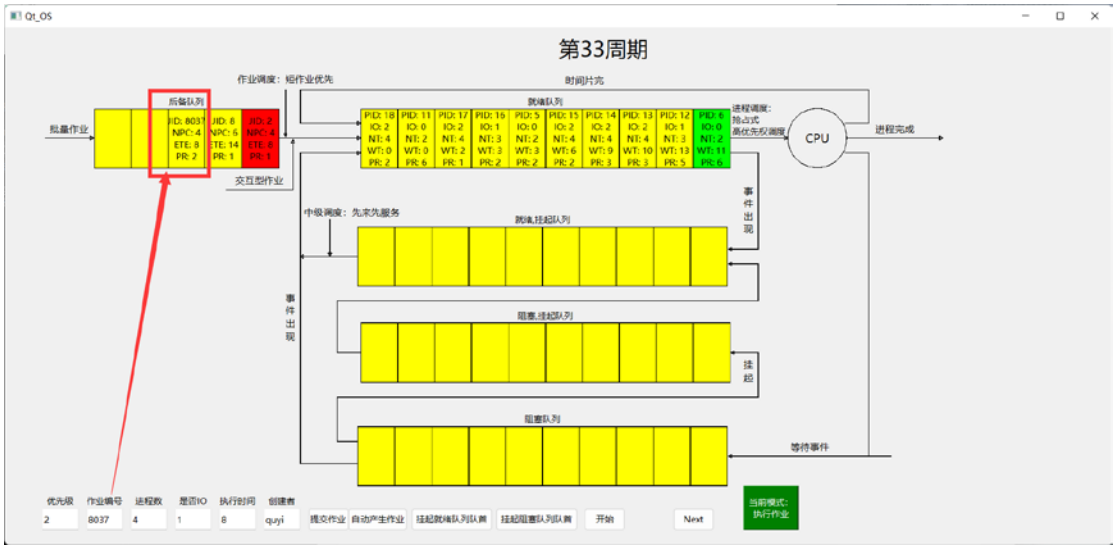

图 6-11 添加的作业进入到后备队列中

自动产生作业功能：

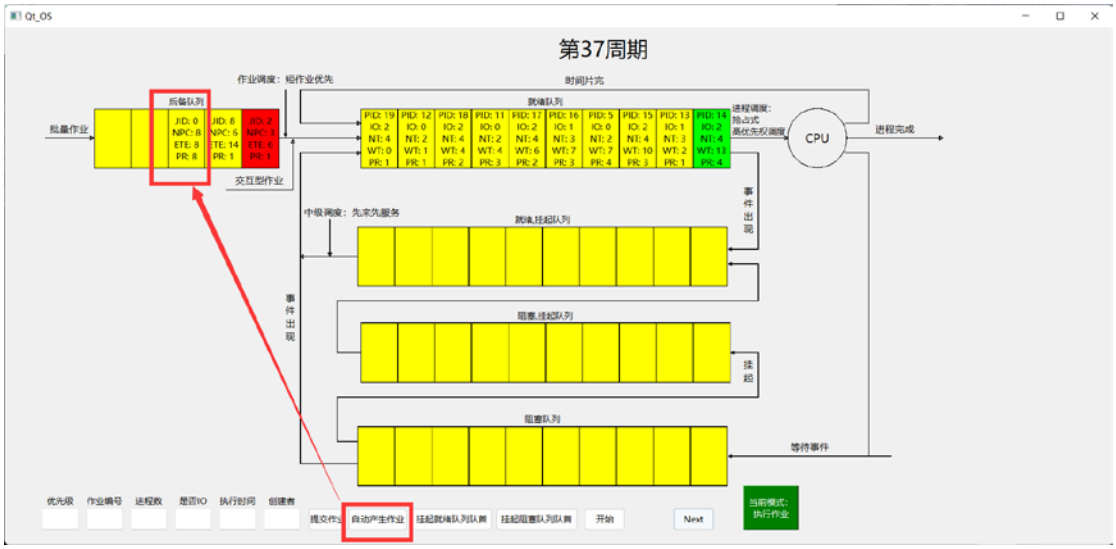

图 6-12 自动生成作业 JID=0

8. 总结和感想体会

通过本次课程设计，加深了我对操作系统中作业和进程调度的理解，从刚开始的不知所措，到逐渐有了眉目，再到之后的基本实现功能，以及最后的完善和优化，在这个过程中我对三级调度体系有了更加深刻的理解。

我从这个课程设计中学到了很多之前都没有接触的东西，首先我按照课程设计的要求使用 C 语言完整的完成了课程设计，然后将已完成的代码进行拆分、编译，最终生成了.lib 文件，为演示程序提供接口。在实现静态链接库的过程中，我发现官方文档往往是解决这类技术问题最高效的方法，而对于 CSDN 这样的平台往往解释的不够清楚，通过这个课程设计也培养了我学习新技术应该有的正确方法。

最课程设计基本任务完成之后，我的课程设计程序依然是一个黑色的命

令行形式，为了能够更加形象，更加直观的展示操作系统的三级调度过程，我学习了 Qt 编程，使用 Qt 提供的可视化方案，制作出了第二版课程设计，也就是现在呈现出来的效果，在这个过程中也促使我学习了新的技术，直观的展示往往更容易被使用者所接受和理解。

综上所述，我在本次课程设计中一方面是加深了对三级调度系统的理解，另一方面是掌握了新的可视化技术，这对我之后完成项目的成果展示有着很大的作用，相信我会在之后的学习过程中不断深化理解，提高自己的能力，更上一层楼。

参考文献

[1]Microsoft. (2021). Walkthrough: Creating and Using a Static Library (C++) [Web document]. https://learn.microsoft.com/zh-cn/cpp/build/walkthrough-creating-and-using-a-static-library-cpp?view=msvc-170

[2] The Qt Company. (2021). Qt Documentation [Web document]. https://doc.qt.io/

[3]汤子瀛. (2017). 操作系统（第 4 版）. 北京: 人民邮电出版社.

[4]王爱英, 张守攻. (2019). 操作系统教程. 北京: 高等教育出版社.

[5]杨渝民. (2020). 操作系统概论（第 4 版）. 北京: 清华大学出版社.