合肥工業大學

系统软件综合设计报告
操作系统分册

设计题目	11.绘制资源分配图
学生姓名	孙淼
学　　号	2018211958
专业班级	计算机科学与技术 18-2 班
指导教师	田卫东　李　琳
完成日期	2021.7.4

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资源分配图相关概念，按照内核代码的实现原则，设计和实现一个能检查出进程死锁（自加）的绘制资源分配图程序，具体的，需要：建立资源分配图的数据结构描述；建立绘制资源分配图的例程，包括结点和有向边；可以删除、添加结点或有向边；可以将资源分配图存入文件，从文件中取出；可以检测死锁（自加）。

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资源分配图数据结构，对资源分配图数据结构进行“绘制”的各个功能，由一系列的函数组成；另一个部分是演示系统，调用“绘制”资源分配图的各个相应函数，以让其运行，同时提供系统的展示界面作为 GUI 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，

其中资源分配图数据结构是由我自己设计的，包括：

资源分配图数据结构由三小部分组成，一是描述资源类结点最大资源数的一维数组数据结构；二是建立描述资源到进程的分配的资源数的二维数组的数据结构；三是建立描述进程到资源的申请的资源数的二维数组的数据结构；

其中内核中函数具体包括：

1）初始化资源分配图函数，初始化资源分配图为一个死锁状态；

2）删除进程结点函数，删除某个进程向所有资源的申请和收到的所有资源的分配；

3）删除资源结点函数，删除某个资源向所有进程的分配和收到的所有进程的申请；

4）删除进程对资源的申请函数，删除某个进程对某个资源的申请；

5）删除资源对进程的分配函数，删除某个资源对某个进程的分配；

6）增加进程对资源的申请函数，增加某个进程对某个资源的申请；

7）增加资源对进程的分配函数，增加某个资源对某个进程的分配；

8）从文件取出数据结构加载函数，取出特定路径文件内的数据结构（资源分配图），加载到 GUI 界面上。

9）将数据结构存入到文件存入函数，将当前 GUI 界面上的数据结构（资源分配图）存入到特定路径文件内，相当于做个备份，随时可以取出加载。

10）检查死锁函数，通过我自己设计的一个循环算法，检查当前状态下的资源分配图是否存在死锁，若有，则资源分配图简化到卡住的那一步，配合 GUI 的函数，可以在 GUI 上显示 N,若没有，则资源分配图简化到进程和资源全部释放，并在 GUI 上显示 Y。

其中界面中函数具体包括：

1）刷新线条函数，结合多线程思想，设计了一个将 Qt 中不能被调用的画图函数的“调用”函数；

2）显示资源类已分配资源数函数，点击即可刷新显示各个资源类已经分配出去的资源数。

3）显示资源类总资源数函数，点击即可刷新显示各个资源类所有的资源数。

4）对内核中各个函数调用的控件的事件函数。

2. 开发环境

Windows 10,Intel(R) Core(TM) i7-8750H CPU @ 2.20GHz，

Visual Studio Community 2017，

qt-opensource-windows-x86-5.12.6。

3. 相关原理及算法

资源分配图是由顶点和边的结对组成的有向图：G = （V,E）式中 V 是顶点的集合，E 是边的集合。

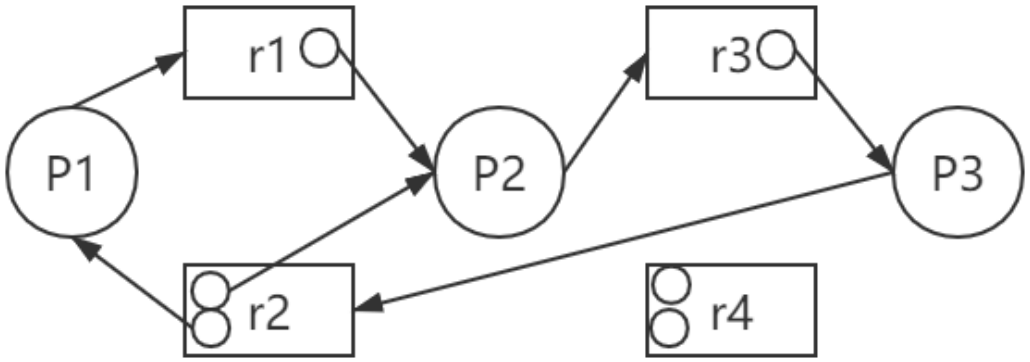
顶点集合分为两部分：P = {P1，P2，P3，…，Pn}，它由进程集合所有活动进程组成。R = {r1，r2，r3，…，rn}，它由进程集合所涉及的全部资源类型组成。边集合分为以下两种，申请边 Pi->rj，表示进程 Pi 申请一个单位的 rj 资源，但当前 Pi 在等待该资源，赋给边 rj->Pi：表示有一个单位的 rj 资源已分配给进程 Pi。

如下图所示为一个资源分配图，通常用圆圈表示进程，用方框表示资源（一个方框表示一类资源，其中的圆圈表示单个资源实体）。注意申请边要指向表示资源的方框，赋给边必须起于方框中的一个黑点，下图给出了下列的信息：

（1）P = {P1，P2，P3}

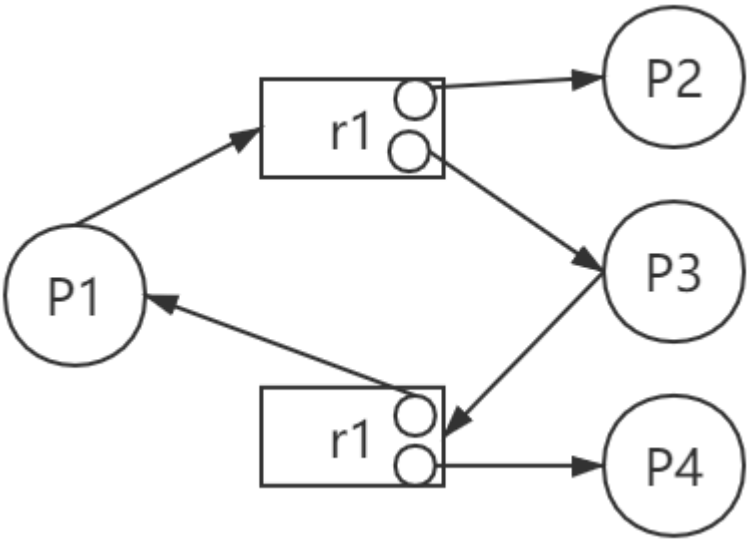
（2）R = {r1，r2，r3，r4}，其各类资源的个数分别为 1,2,1,3

（3）E = {P1->r1，P2->r3，r1->P2，r2->P2，r2->P1，r3->P3}，即进程 P1 占有了一个 r2 资源，且等待一个 r1 资源，进程 P2 占有 r1 和 r2 资源各一个，且等待一个 r3 资源。进程 P3 占有一个 r3 资源。



关于我自己加的检测死锁功能，相关原理和算法为：利用资源分配图可以直观，精确地描述死锁，对每种类型只有一个资源的系统（如只有一台扫描仪，一台 CD 刻录机，一台绘图仪）构造的资源分配图中，如果出现环路就说明存在死锁。在此环路中的每个进程都是死锁进程。如果没有出现环路，系统就没有发生死锁。

如果每类资源的实体不止一个，那么资源分配图中出现环路并不表明一定出现死锁。在这种情况下，资源分配图中存在环路是死锁存在的必要条件，但不是充分条件，在上图中，存在两个最小的环，P1->r1->P2->r3->P3->r2->P1 和 P2->r3->P3->r2->P2。此时系统发生死锁，进程 P1，P2 和 P3 都在环中，因而都是死锁进程。



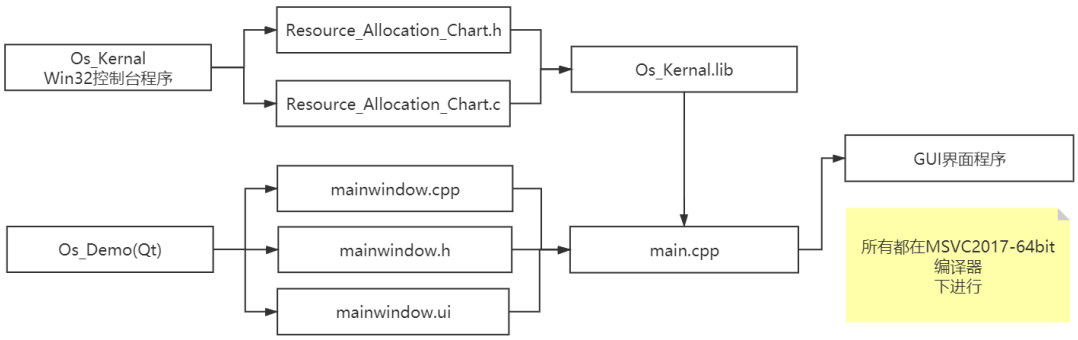
又如上图所示，其中也有一个环路，P1->r1->P3->r2->P1，然而没有出现死锁。因为进程 P4 能释放它占有的资源 r2，然后就可以将 r2 分给 P3，这样环路就打开了。总之，如果资源分配图中没有环路，那么系统就不会陷入死锁状态，如果存在环路，那么系统就有可能出现死锁。

4. 系统结构和主要的算法设计思路

我所设计的系统架构是：

由 Os_Kernal 文件夹中名为 Os_Kernal 的 Win32 控制台程序项目中的内核代码 Resource_Allocation_Chart.h 和 Resource_Allocation_Chart.c 的 C 语言文件生成 Os_Kernal.lib 等一系列文件。

建立 Qt 项目 Os_Demo，然后通过 Qt 中的 lib 库调用语句调用 lib，在 Qt 中的 mainwindow.cpp 编写各个控件调用内核函数的事件函数，在 mainwindow.h 中声明一些 GUI 代码需要的全局变量（后面将介绍全局变量设置的巧妙之处），在 mainwindow.ui 设置各个控件的相对位置等，最后在 main.cpp 中生成 MainWindow 类的对象 w，最终显示程序 GUI，系统架构的流程图如下所示：



注：由于 Qt 对于 Visual Studio 只支持 MSVC2017 编译器，所以第二部分开发环境如此。

我的设计中主要的算法思想，我们分功能介绍:

内核代码中各个函数的算法思想：**（算法介绍对应的代码见第六部分“程序实现——主要程序清单”）**

1） 初始化资源分配图函数，初始化资源分配图为一个死锁状态：

不想通过简单的赋值初始化资源分配图的死锁状态，所以我选择的方法是通过 for 循环中的循环计数变量 i 来对资源分配图的数据结构进行赋值。比如我想对资源类结点的最大资源数赋值为 1，2,3,4,5,6 对于资源到进程的资源分配矩

阵，进程到资源的资源申请矩阵也是同理，都是通过非直接赋值的方式给到。代码见第六部分 void init(Resourece_Allocation_Chart *chart)

2）删除进程结点函数，删除某个进程向所有资源的申请和收到的所有资源的分配；

对于我设计的资源分配图的数据结构而言，想要删除某个进程结点，在数据结构中的体现就是将资源到进程的分配矩阵，进程到资源的申请矩阵中，与进程相关的数据都置为 0 即可，也就是将资源到进程的分配矩阵的横轴（进程）的要删除的进程下标对应的那一行置为 0，将进程到资源的申请矩阵的纵轴（进程）的要删除的进程下标对应的那一行置为 0。

代码见第六部分 void Delete_Process_Node
(Resourece_Allocation_Chart*chart, int Process_index)

3）删除资源结点函数，删除某个资源向所有进程的分配和收到的所有进程的申请；

对于我设计的资源分配图的数据结构而言，想要删除某个资源结点，在数据结构中的体现就是将资源到进程的分配矩阵，进程到资源的申请矩阵中，与资源相关的数据都置为 0 即可，也就是将资源到进程的分配矩阵的纵轴（资源）的要删除的资源下标对应的那一行置为 0，将进程到资源的申请矩阵的横轴（资源）的要删除的资源下标对应的那一行置为 0。

代码见第六部分 void Delete_Resource_Node
(Resourece_Allocation_Chart*chart, int Resource_index)

4）删除进程对资源的申请函数，删除某个进程对某个资源的申请；

对于我设计的资源分配图的数据结构而言，想要删除某个进程对某个资源的申请，在数据结构中的体现就是只对进程到资源的申请矩阵的特定位置（i，j）处置为 0，由于无论是否存在这个申请，由于限定是 1 或 0，所以置为 0 一定可以，其中 i 是进程下标对应的在申请矩阵中的横轴位置；j 是资源下标对应的在申请矩阵中的纵轴位置。

代码见第六部分 void Delete_Request_Edge(Resourece_Allocation_Chart *chart, int Process_index, int Resource_index)

5）删除资源对进程的分配函数，删除某个资源对某个进程的分配；

对于我设计的资源分配图的数据结构而言，想要删除某个资源对某个进程的分配，在数据结构中的体现就是只对资源到进程的分配矩阵的特定位置（i，j）数据 -1，所以前提是已经有分配/分配给该进程的资源数 ＞0，其中 i 是资源下标对应的在申请矩阵中的纵轴位置；j 是进程下标对应的在申请矩阵中的横轴位置。

代码见第六部分 void Delete_Allocation_Edge(Resourece_Allocation_Chart *chart, int Resource_index, int Process_index)

6）增加进程对资源的申请函数，增加某个进程对某个资源的申请；

对于我设计的资源分配图的数据结构而言，想要增加某个进程对某个资源的申请，在数据结构中的体现就是只对进程到资源的申请矩阵的特定位置（i，j）数据 +1，所以前提是该进程没有对其他资源发起申请（也就是进程对资源的申请矩阵的进程所在的那一列都是 0），其中 i 是进程下标对应的在申请矩阵中的横轴位置；j 是资源下标对应的在申请矩阵中的纵轴位置。

代码见第六部分

void Add_Edge_Process_To_Resource(Resourece_Allocation_Chart *chart, int Process_index, int Resource_index)

7）增加资源对进程的分配函数，增加某个资源对某个进程的分配；

对于我设计的资源分配图的数据结构而言，想要增加某个资源对某个进程的分配，在数据结构中的体现就是只对资源到进程的分配矩阵的特定位置（i，j）数据 +1，所以前提是该资源类的资源数没有分配完/到达上限，（也就是资源对进程的分配矩阵的资源所在的那一列的和小于该类资源的最大资源数），其中 i 是资源下标对应的在申请矩阵中的纵轴位置；j 是进程下标对应的在申请矩阵中的横轴位置。

代码见第六部分

void Add_Edge_Resource_To_Process(Resourece_Allocation_Chart *chart, int Resource_index, int Process_index)

8）从文件取出数据结构加载函数，取出特定路径文件内的数据结构（资源分配图），加载到 GUI 界面上。

对于我设计的资源分配图的数据结构而言，想要从文件中取出数据结构加载

到程序 GUI 中，方法就是读取文件中的三个矩阵（1 * 6,6 * 6,6 * 6），并且赋值给程序中创建的结构体 Resource_Allocation_Chart 实例中。

代码见第六部分 void Chart_Read_From_File(Resourece_Allocation_Chart *chart)

9）将数据结构存入到文件存入函数，将当前 GUI 界面上的数据结构（资源分配图）存入到特定路径文件内，相当于做个备份，随时可以取出加载。

对于我设计的资源分配图的数据结构而言，想要从程序中创建的结构体 Resource_Allocation_Chart 实例中取出数据结构存入到特定文件中，方法就是读取结构体实例中的三个矩阵（1 * 6,6 * 6,6 * 6），并且写入到文件中。

代码见第六部分 void Chart_Write_To_File(Resourece_Allocation_Chart *chart)

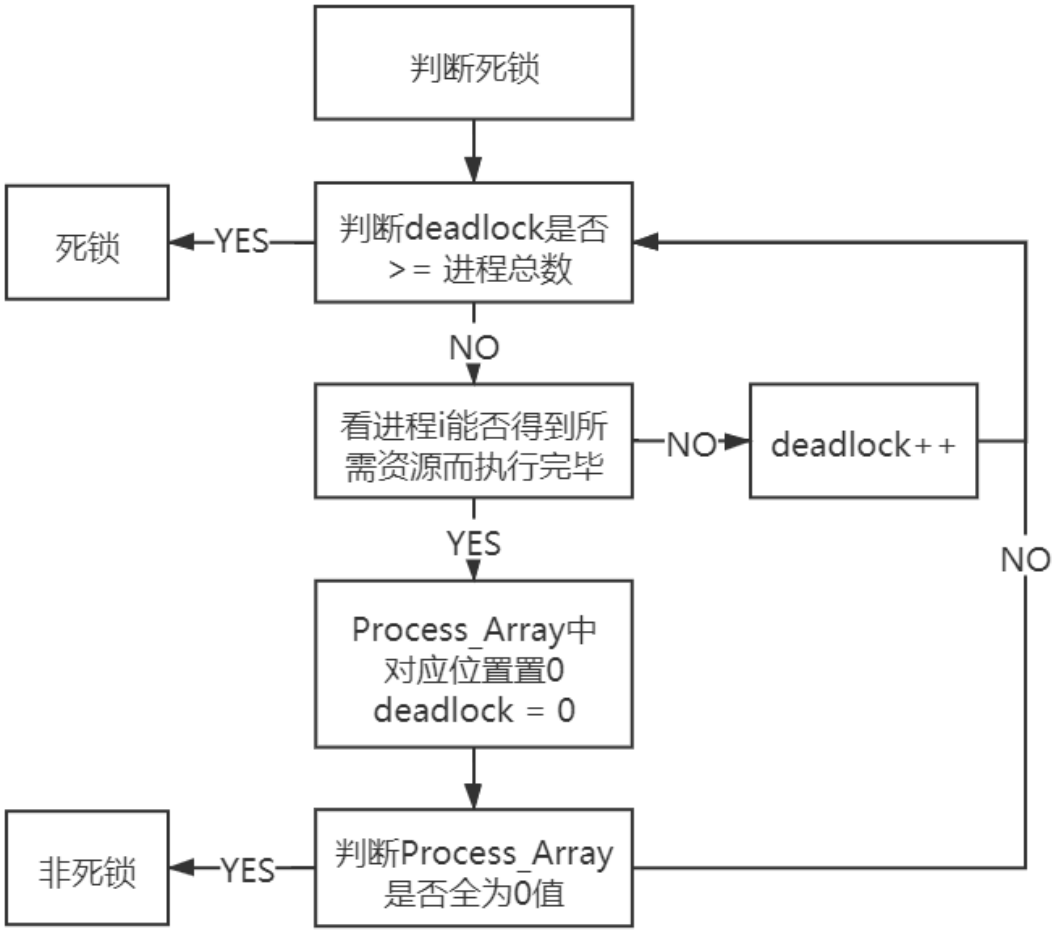
10）检查死锁函数，通过我自己设计的一个循环算法，检查当前状态下的资源分配图是否存在死锁，若有，则资源分配图简化到卡住的那一步。（配合 GUI 的函数，可以在 GUI 上显示 N,若没有，则资源分配图简化到进程和资源全部释放，并在 GUI 上显示 Y。）

对于我设计的资源分配图的数据结构而言，想要检查死锁，需要对这个功能函数设计两个出口，一个是有死锁，一个是没有死锁。

想要判断是死锁，那么必须是对每个进程而言，都是死锁，并且连续（比如一共六个进程，就要六个进程都**连续地**被检查为死锁状态），判断连续的方法是定义一个函数内的“全局变量”deadlock 并且初始化为 0，一旦有进程被释放，deadlock 置为 0；若是当前进程不能被释放，deadlock 就+1，每次经过一轮从进程 0 到进程 last 的循环，就判断 deadlock 是否大于等于进程总数（之所以是大于等于，是因为可能第一个死锁发现于程序中间，而不是开头，而为了节省一些代码，判断是放在开头的）。

想要判断不是死锁，方法是六个进程被运行完毕了，我的方法是通过新建一个函数内的局部变量 Process_Array 整型数组作为记录,分别记录每个进程为 1，如果某个进程获得了其需要的所有资源而被释放，就将其在数组中对应位置的数改为 0，在任何时刻，一旦发现 Process_Array 中全为 0，就代表进程全部被释放，为非死锁状态。

代码见第六部分 void Judge_DeadLock(Resourece_Allocation_Chart *chart)检查死锁的函数流程图如下所示，一目了然：

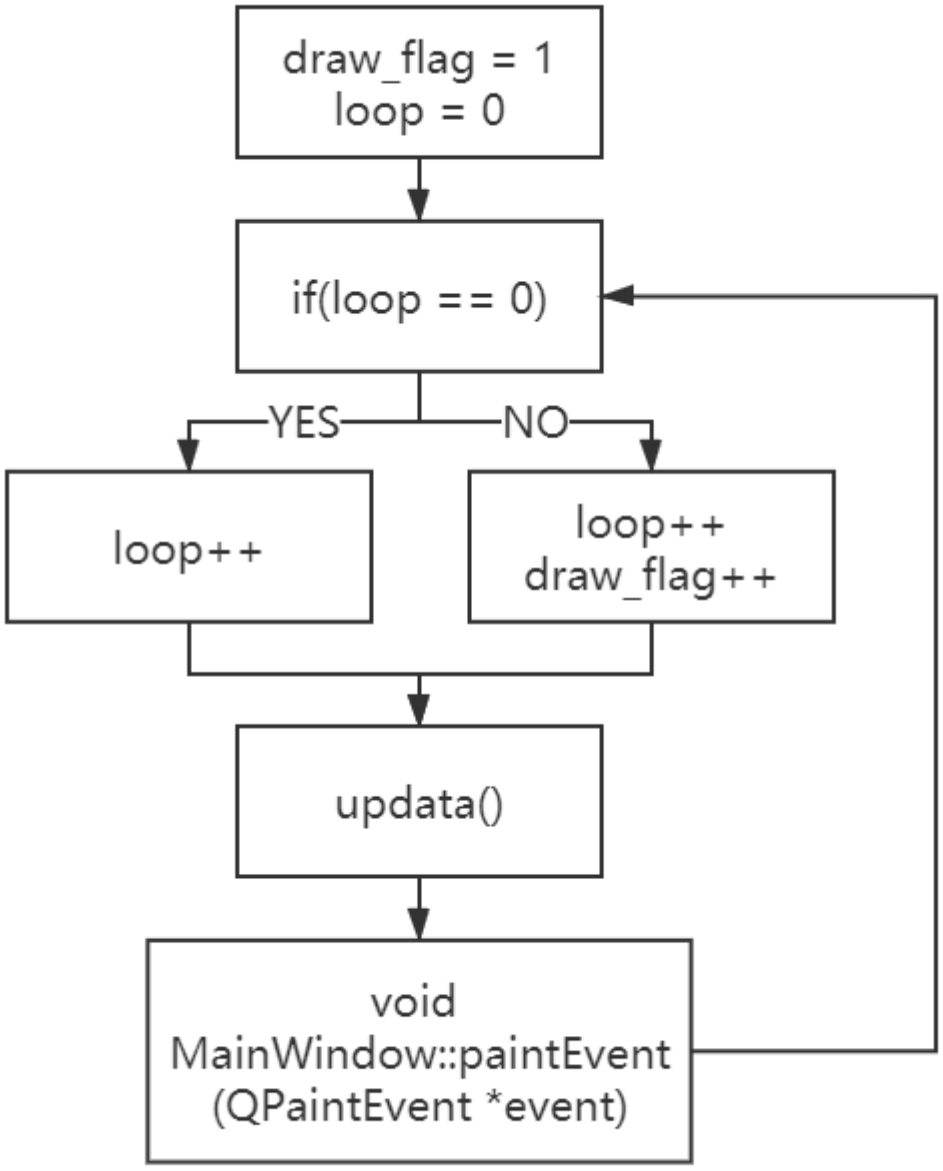


其中界面中函数具体包括：

1）刷新线条函数，结合多线程思想，设计了一个将 Qt 中不能被调用的画图函数的“调用”函数；

由于画线位置不同，Qt 中设计的画线函数是以要画的线段的开头点和末端点为参数，画出这个线段，所以首先要解决的就是如何画出多个线段，我选择的方法是通过一个局部变量 r_i * n，n 是各个结点之间的竖直距离，这样通过 r_i 就可以实现画出的线的距离的变化。**此外，**由于画线函数在 Qt 中被限定为一个 void MainWindow::paintEvent(QPaintEvent *event)函数，且无法被别的函数调用，那么想要实现变化画图，就用到了我在 Java 编程中常用的多线程思想，通过两个 Qt 中 mainwindow.h 的全局变量和 Qt 中的 update()函数来实现刷新，反复画图的功能。

这一部分的流程图如下所示：



具体来说，mainwindow.h 中的全局变量画图标志位如下所示：

```
1.  //画图标志位
2.      int draw_flag = 1;
3.      int loop = 0;
```

在调用画图函数按钮的事件函数中的关键代码就是：

```
1.  if(loop == 0){
2.      loop++;
3.      update();
4.  }
```

```
else{
    draw_flag++;
    loop++;
    update();
}
```

在画图函数就是：

```
if(loop == draw_flag){……}
```

这样在第一次打开 GUI 程序时，由于 drwa_flag = 1，loop = 0，两者不相等，所以画图函数不运行，不画图。

但是一旦按下画图按钮，事件函数被运行，由于 loop = 0，loop++(= 1)，draw_flag 不变，update()再次调用画图函数，此时 loop = draw_flag，开始画图。

此后，每次按下画图按钮，loop++，draw_flag++，都一样，都会继续画图。

2）显示资源类已分配资源数函数，点击即可刷新显示各个资源类已经分配出去的资源数。

即读取程序中的资源分配图结构体内的数据结构值，然后被总资源数减去，显示在控件上；

3）显示资源类总资源数函数，点击即可刷新显示各个资源类所有的资源数。

即读取程序中的资源分配图结构体内的数据结构值，显示在控件上；

4）对内核中各个函数调用的控件的事件函数。

即 C++中控件的事件函数；

5. 程序实现---主要数据结构

本实验中我使用到的数据结构——资源分配图，是我自己设计的，本数据结构由三部分组成，结构体代码如下所示：

```
typedef struct Resourece_Allocation_Chart
{
    int Resource_Node_Array[Resource_Node_Num]; /*资源结点类数组，数组内的值是该资源类的资源个数，个数不能超过 Resource_Max_Num*/
    int Resource_To_Process_Allocation_Edge[Resource_Max_Num][Process_Node_Num];  /*资源到进程的分配边，横着是资源，竖着是进程*/
    int Process_To_Resource_Request_Edge[Process_Node_Num][Resource_Node_Num];    /*进程到资源的分配边，横着是进程，竖着是资源*/

```

```
}Resourece_Allocation_Chart;
```

举个例子说明一下，比如资源分配图数据结构的三个矩阵如下所示：

```
/*
    资源类结点的最大资源数

    1 2 3 4 5 6

    资源->进程 矩阵初始化（横轴资源 竖轴进程）

    进程
资源1 0 0 0 0 0
    0 2 0 0 0 0
    0 0 3 0 0 0
    0 0 0 4 0 0
    0 0 0 0 5 0
    0 0 0 0 0 6

    进程->资源 矩阵初始化（横轴进程 竖轴资源）我认为同时只能申请一个 所以初始化矩阵是

    资源
进程0 1 0 0 0 0
    0 0 1 0 0 0
    0 0 0 1 0 0
    0 0 0 0 1 0
    0 0 0 0 0 1
    1 0 0 0 0 0
    */
```

那么对于资源类结点的最大资源数矩阵而言，就是说第一类资源的资源最大数量为 1，第二类资源的资源最大数量为 2，第三类资源的资源最大数量为 3，第四类资源的资源最大数量为 4，第五类资源的资源最大数量为 5，第六类资源的资源最大数量为 6。

对于资源到进程的分配矩阵而言，就是说资源 1 对进程 1 分配了一个资源，资源 2 对进程 2 分配了 2 个资源，资源 3 对进程 3 分配了 3 个资源，资源 4 对进程 4 分配了 4 个资源，资源 5 对进程 5 分配了 5 个资源，资源 6 对进程 6 分配了 6 个资源。

对于进程对资源的申请矩阵而言，就是进程 1 对资源 1 有一个资源的申请，进程

2 对资源 2 有一个资源的申请，进程 3 对资源 3 有一个资源的申请，进程 4 对资源 4 有一个资源的申请，进程 5 对资源 5 有一个资源的申请，进程 6 对资源 6 有一个资源的申请。

6. 程序实现---主要程序清单

我所设计系统的核心程序、关键函数的程序清单如下，其中内核中函数如下：

1）初始化资源分配图函数，初始化资源分配图为一个死锁状态；

```
1.  void init(Resourece_Allocation_Chart *chart) /*资源分配图初始化*/
2.  {
3.      /*
4.      资源类结点的最大资源数
5.
6.      1 2 3 4 5 6
7.
8.      资源->进程 矩阵初始化（横轴资源 竖轴进程）
9.
10.     进程
11. 资源1 0 0 0 0 0
12.     0 2 0 0 0 0
13.     0 0 3 0 0 0
14.     0 0 0 4 0 0
15.     0 0 0 0 5 0
16.     0 0 0 0 0 6
17.
18.     进程->资源 矩阵初始化（横轴进程 竖轴资源）我认为同时只能申请一个 所以初始化矩阵是
19.
20.     资源
21. 进程0 1 0 0 0 0
22.     0 0 1 0 0 0
23.     0 0 0 1 0 0
24.     0 0 0 0 1 0
25.     0 0 0 0 0 1
26.     1 0 0 0 0 0
27.     */
28.     //Resourece_Allocation_Chart chart;
29.     //但是这个没用上，感觉用处不大
30.     for (int i = 0; i < Resource_Node_Num; i++) {//数组除了初始化不能直接赋值的，你要给这个数组赋值就得循环一个个赋值了。
31.         (*chart).Resource_Node_Array[i] = 0;
32.         if (i + 1 <= Resource_Max_Num)
```

```
            (*chart).Resource_Node_Array[i] = i + 1;//{1, 2, 3, 4, 5, 6}

    }
    for (int i = 0; i < Resource_Node_Num; i++) {
        for (int j = 0; j < Process_Node_Num; j++) {
            (*chart).Resource_To_Process_Allocation_Edge[i][j] = 0;
            if (i == j)
                (*chart).Resource_To_Process_Allocation_Edge[i][j] = i + 1;

        }
    }
    for (int i = 0; i < Process_Node_Num; i++) {
        for (int j = 0; j < Resource_Node_Num; j++) {
            (*chart).Process_To_Resource_Request_Edge[i][j] = 0;
            if (j == (i + 1) % 6)
                (*chart).Process_To_Resource_Request_Edge[i][j] = 1;
        }
    }
}
```

2）删除进程结点函数，删除某个进程向所有资源的申请和收到的所有资源的分配；

```
void Delete_Process_Node(Resourece_Allocation_Chart*chart, int Process_index) {
    //对资源->进程图 操作
    for (int m = 0; m < Resource_Node_Num; m++) {
        (*chart).Resource_To_Process_Allocation_Edge[m][Process_index - 1] = 0;
    }
    //对进程->资源图 操作
    for (int n = 0; n < Process_Node_Num; n++) {
        (*chart).Process_To_Resource_Request_Edge[Process_index - 1][n] = 0;

    }
}
```

3）删除资源结点函数，删除某个资源向所有进程的分配和收到的所有进程的申请；

```
void Delete_Resource_Node(Resourece_Allocation_Chart*chart, int Resource_index) {
    //用 sizeof 试试 好玩
```

```
    int rows = sizeof((*chart).Resource_To_Process_Allocation_Edge) / sizeof((*chart).Resource_To_Process_Allocation_Edge[0]);
    int cols = sizeof((*chart).Resource_To_Process_Allocation_Edge[1]) / sizeof((*chart).Resource_To_Process_Allocation_Edge[1][1]);
    //对资源->进程图 操作
    for (int j = 0; j < cols; j++) {
        (*chart).Resource_To_Process_Allocation_Edge[Resource_index - 1][j] = 0;
    }
    //对进程->资源图 操作
    for (int i = 0; i < rows; i++) {
        (*chart).Process_To_Resource_Request_Edge[i][Resource_index - 1] = 0;
    }
}
```

4）删除进程对资源的申请函数，删除某个进程对某个资源的申请；

```
void Delete_Request_Edge(Resourece_Allocation_Chart *chart, int Process_index, int Resource_index) {//进程 向 资源 发出请求
                            //对资源->进程图 操作
                            //if (chart.Process_To_Resource_Request_Edge[Process_index - 1][Resource_index - 1] > 0)
                            //  chart.Process_To_Resource_Request_Edge[Process_index - 1][Resource_index - 1] -= 1;

                            //对进程->资源图 操作
    //for (int i = 0; i < Resource_Node_Num; i++) {
        (*chart).Process_To_Resource_Request_Edge[Process_index - 1][Resource_index - 1] = 0;
    //}
    //if ((*chart).Process_To_Resource_Request_Edge[Resource_index - 1][Process_index - 1] > 0;
}
```

5）删除资源对进程的分配函数，删除某个资源对某个进程的分配；

```
void Delete_Allocation_Edge(Resourece_Allocation_Chart *chart, int Resource_index, int Process_index) {//资源 分配 给 进程
```

```
2.
                                    //对资源->进程图 操作
3.      if ((*chart).Resource_To_Process_Allocation_Edge[Resource_index - 1][Pro
    cess_index - 1] > 0)
4.          (*chart).Resource_To_Process_Allocation_Edge[Resource_index - 1][Pro
    cess_index - 1] -= 1;
5.
6.      //对进程->资源图 操作
7.      //if (chart.Process_To_Resource_Request_Edge[Process_index - 1][Resource
    _index - 1] > 0)
8.      //  chart.Process_To_Resource_Request_Edge[Process_index - 1][Resource_i
    ndex - 1] -= 1;
9.
10. }
```

6）增加进程对资源的申请函数，增加某个进程对某个资源的申请；

```
1.  void Add_Edge_Process_To_Resource(Resourece_Allocation_Chart *chart, int Pro
    cess_index, int Resource_index) {//一次只允许加一条边
2.      int temp = 0;
3.      for (int i = 0; i < Resource_Node_Num; i++) {
4.          temp += (*chart).Process_To_Resource_Request_Edge[Process_index - 1]
    [i];
5.      }
6.      if (temp == 0)
7.          (*chart).Process_To_Resource_Request_Edge[Process_index - 1][Resourc
    e_index - 1] = 1;
8.      else
9.          ;
10. }
```

7）增加资源对进程的分配函数，增加某个资源对某个进程的分配；

```
1.  void Add_Edge_Resource_To_Process(Resourece_Allocation_Chart *chart, int Res
    ource_index, int Process_index) {//一次只允许加一条边 前提是没有超出资源结点的能
    力范围
2.      int temp = 0;
3.      for (int i = 0; i < Process_Node_Num; i++) {
4.          temp += (*chart).Resource_To_Process_Allocation_Edge[Resource_index
    - 1][i];
5.          //if ((*chart).Resource_To_Process_Allocation_Edge[Resource_index -
    1][Process_index - 1] + (*chart).Process_To_Resource_Request_Edge[Process_in
    dex - 1][Resourece_index - 1] < (*chart).Resource_Node_Array[Resourece_index
     - 1])
```

```
        //  (*chart).Resource_To_Process_Allocation_Edge[Resource_index - 1][Process_index - 1] += 1;
        //else
        //  ;
    }
    if (temp < (*chart).Resource_Node_Array[Resource_index - 1])
        (*chart).Resource_To_Process_Allocation_Edge[Resource_index - 1][Process_index - 1] += 1;
    else
        ;
}
```

8）从文件取出数据结构加载函数，取出特定路径文件内的数据结构（资源分配图），加载到 GUI 界面上。

```
void Chart_Read_From_File(Resourece_Allocation_Chart *chart) {
    //注意最后要把这些 print 去掉 因为指导书要求不能有

    FILE *stream;
    errno_t err;
    err = fopen_s(&stream, "D:\\data.txt", "r");
    //使用 w+ 如果你的路径文件不存在 自动建个新的 如果该文件存在，其内容将被销毁可读可写 不会乱码
                                                /*
                                                if (err == 0) {
                                                printf("was opened");
                                                }
                                                else
                                                printf("not opened");
                                                */
    for (int i = 0; i < 3; i++) {
        if (i == 0) {
            for (int j = 0; j < Resource_Max_Num; j++) {
                /*
                需要注意区别 fscanf_s 的第三个参数是地址 与 fprintf 不同
                区别如下
                fprintf(fp,"%d,",ss[i]);
                fscanf(fp,"%d,",&new[i]);
                */
                fscanf_s(stream, "%d ", &chart->Resource_Node_Array[j]);
                if (j == Process_Node_Num - 1)
                    fscanf_s(stream, "\n");
            }
        }
```

```
        if (i == 1) {
            for (int m = 0; m < Resource_Max_Num; m++) {
                for (int n = 0; n < Process_Node_Num; n++) {
                    fscanf_s(stream, "%d ", &chart->Resource_To_Process_Allocation_Edge[m][n]);
                    if (n == Process_Node_Num - 1)
                        fscanf_s(stream, "\n");
                }
            }
        }
        if (i == 2) {
            for (int k = 0; k < Process_Node_Num; k++) {
                for (int l = 0; l < Resource_Max_Num; l++) {
                    fscanf_s(stream, "%d ", &chart->Process_To_Resource_Request_Edge[k][l]);
                    if (l == Resource_Max_Num - 1)
                        fscanf_s(stream, "\n");
                }
            }
        }
    }
    //for (i = 0; i<N; i++, pb++) {
    //  fscanf(fp, "%s %d %d %f\n", pb->name, &pb->num, &pb->age, &pb->score);
    //}
    err = fclose(stream);
    /*
    if (err == 0) {
    printf("was not close");
    }
    else
    printf("was close");
    */
}
```

9）将数据结构存入到文件存入函数，将当前 GUI 界面上的数据结构（资源分配图）存入到特定路径文件内，相当于做个备份，随时可以取出加载。

```
void Chart_Write_To_File(Resourece_Allocation_Chart *chart) {
    //注意最后要把这些 print 去掉 因为指导书要求不能有

    FILE *stream;
    errno_t err;
```

```
    err = fopen_s(&stream, "D:\\data.txt", "w+");//使用 w+ 如果你的路径文件不存在 自动建个新的 可读可写 不会乱码
                                                 /*
                                                 if (err == 0) {
                                                 printf("was opened");
                                                 }
                                                 else
                                                 printf("not opened");
                                                 */
    for (int i = 0; i < 3; i++) {
        if (i == 0) {
            for (int j = 0; j < Resource_Max_Num; j++) {
                fprintf(stream, "%d ", chart->Resource_Node_Array[j]);
                if (j == Process_Node_Num - 1)
                    fprintf(stream, "\n");
            }
        }
        if (i == 1) {
            for (int m = 0; m < Resource_Max_Num; m++) {
                for (int n = 0; n < Process_Node_Num; n++) {
                    fprintf(stream, "%d ", chart->Resource_To_Process_Allocation_Edge[m][n]);
                    if (n == Process_Node_Num - 1)
                        fprintf(stream, "\n");
                }
            }
        }
        if (i == 2) {
            for (int k = 0; k < Process_Node_Num; k++) {
                for (int l = 0; l < Resource_Max_Num; l++) {
                    fprintf(stream, "%d ", chart->Process_To_Resource_Request_Edge[k][l]);
                    if (l == Resource_Max_Num - 1)
                        fprintf(stream, "\n");
                }
            }
        }
    }
    //fwrite(&student[i], sizeof(student), 1, fp);
    //fprintf(stream, "%d\n", chart->Resource_Node_Array);//, pa->num, pa->age, pa->score %d %f%s
    err = fclose(stream);
    /*
    if (err == 0) {
```

```
    printf("was not close");
    }
    else
    printf("was close");
    */
}
```

10）检查死锁函数，通过我自己设计的一个循环算法，检查当前状态下的资源分配图是否存在死锁，若有，则资源分配图简化到卡住的那一步，并在 GUI 上显示 N,若没有，则资源分配图简化到进程和资源全部释放，并在 GUI 上显示 Y。

```
void Judge_DeadLock(Resourece_Allocation_Chart *chart) {
    int Process_Array[6] = { 1,1,1,1,1,1 };
    int break_flag = 1;
    int process_0_request_resource_index = 0;
    int resource_allocated = 0;
    int deadlock = 0;
    while (break_flag == 1) {//i = 0
        if (deadlock >= 6)
            break;
        /*六部分
        进程 1
        */
        for (process_0_request_resource_index = 0; process_0_request_resource_index < Resource_Node_Num; process_0_request_resource_index++) {
            if ((*chart).Process_To_Resource_Request_Edge[0][process_0_request_resource_index] == 1) {// j 是进程 0 正在申请的资源的数组下标
                break;
            }
        }
        resource_allocated = 0;
        for (int i = 0; i < Process_Node_Num; i++) {
            resource_allocated += (*chart).Resource_To_Process_Allocation_Edge[process_0_request_resource_index][i];
        }
        if ((*chart).Resource_Node_Array[process_0_request_resource_index] > resource_allocated) {//该资源类还有剩余的资源数 进程 0 可以完成
            //该进程结点可以释放 删除两个表里与其有关的
            for (int m = 0; m < Resource_Node_Num; m++) {
                (*chart).Resource_To_Process_Allocation_Edge[m][0] = 0;
            }
            for (int n = 0; n < Process_Node_Num; n++) {
                (*chart).Process_To_Resource_Request_Edge[0][n] = 0;
```

```
            }
            Process_Array[0] = 0;
            deadlock = 0;
        }
        for (int i = 0; i <= Process_Node_Num; i++) {
            if (i == 6)
                break_flag = 0;
            if (Process_Array[i] == 1) {
                deadlock++;
                break;
            }
        }
        if (break_flag == 0)
            break;

        /*
        Process 2
        */
        for (process_0_request_resource_index = 0; process_0_request_resourc
e_index < Resource_Node_Num; process_0_request_resource_index++) {
            if ((*chart).Process_To_Resource_Request_Edge[1][process_0_reque
st_resource_index] == 1) {// j 是进程 0 正在申请的资源的数组下标
                break;
            }
        }
        resource_allocated = 0;
        for (int i = 0; i < Process_Node_Num; i++) {
            resource_allocated += (*chart).Resource_To_Process_Allocation_Ed
ge[process_0_request_resource_index][i];
        }
        if ((*chart).Resource_Node_Array[process_0_request_resource_index] >
 resource_allocated) {//该资源类还有剩余的资源数 进程 0 可以完成
            //该进程结点可以释放 删除两个表里与其有关的
            for (int m = 0; m < Resource_Node_Num; m++) {
                (*chart).Resource_To_Process_Allocation_Edge[m][1] = 0;
            }
            for (int n = 0; n < Process_Node_Num; n++) {
                (*chart).Process_To_Resource_Request_Edge[1][n] = 0;
            }
            Process_Array[1] = 0;
            deadlock = 0;
        }
        for (int i = 0; i <= Process_Node_Num; i++) {
            if (i == 6)
```

```
                break_flag = 0;
            if (Process_Array[i] == 1) {
                deadlock++;
                break;
            }

        }
        if (break_flag == 0)
            break;

        /*
        Process 3
        */
        for (process_0_request_resource_index = 0; process_0_request_resource_index < Resource_Node_Num; process_0_request_resource_index++) {
            if ((*chart).Process_To_Resource_Request_Edge[2][process_0_request_resource_index] == 1) {// j 是进程 0 正在申请的资源的数组下标
                break;
            }
        }
        resource_allocated = 0;
        for (int i = 0; i < Process_Node_Num; i++) {
            resource_allocated += (*chart).Resource_To_Process_Allocation_Edge[process_0_request_resource_index][i];
        }
        if ((*chart).Resource_Node_Array[process_0_request_resource_index] > resource_allocated) {//该资源类还有剩余的资源数 进程 0 可以完成
            //该进程结点可以释放 删除两个表里与其有关的
            for (int m = 0; m < Resource_Node_Num; m++) {
                (*chart).Resource_To_Process_Allocation_Edge[m][2] = 0;
            }
            for (int n = 0; n < Process_Node_Num; n++) {
                (*chart).Process_To_Resource_Request_Edge[2][n] = 0;
            }
            Process_Array[2] = 0;
             deadlock = 0;
         }
         for (int i = 0; i <= Process_Node_Num; i++) {
            if (i == 6)
                break_flag = 0;
            if (Process_Array[i] == 1) {
                deadlock++;
                break;
            }
```

```
109.
110.         }
111.         if (break_flag == 0)
112.             break;
113.
114.         /*
115.         Process 4
116.         */
117.         for (process_0_request_resource_index = 0; process_0_request_resour
    ce_index < Resource_Node_Num; process_0_request_resource_index++) {
118.             if ((*chart).Process_To_Resource_Request_Edge[3][process_0_requ
    est_resource_index] == 1) {// j 是进程 0 正在申请的资源的数组下标
119.                 break;
120.             }
121.         }
122.         resource_allocated = 0;
123.         for (int i = 0; i < Process_Node_Num; i++) {
124.             resource_allocated += (*chart).Resource_To_Process_Allocation_E
    dge[process_0_request_resource_index][i];
125.         }
126.         if ((*chart).Resource_Node_Array[process_0_request_resource_index]
    > resource_allocated) {//该资源类还有剩余的资源数 进程 0 可以完成
127.             //该进程结点可以释放 删除两个表里与其有关的
128.             for (int m = 0; m < Resource_Node_Num; m++) {
129.                 (*chart).Resource_To_Process_Allocation_Edge[m][3] = 0;
130.             }
131.             for (int n = 0; n < Process_Node_Num; n++) {
132.                 (*chart).Process_To_Resource_Request_Edge[3][n] = 0;
133.             }
134.             Process_Array[3] = 0;
135.             deadlock = 0;
136.         }
137.         for (int i = 0; i <= Process_Node_Num; i++) {
138.             if (i == 6)
139.                 break_flag = 0;
140.             if (Process_Array[i] == 1) {
141.                 deadlock++;
142.                 break;
143.             }
144.
145.         }
146.         if (break_flag == 0)
147.             break;
148.
```

```
149.         /*
150.         Process 5
151.         */
152.         for (process_0_request_resource_index = 0; process_0_request_resour
    ce_index < Resource_Node_Num; process_0_request_resource_index++) {
153.             if ((*chart).Process_To_Resource_Request_Edge[4][process_0_requ
    est_resource_index] == 1) {// j 是进程 0 正在申请的资源的数组下标
154.                 break;
155.             }
156.         }
157.         resource_allocated = 0;
158.         for (int i = 0; i < Process_Node_Num; i++) {
159.             resource_allocated += (*chart).Resource_To_Process_Allocation_E
    dge[process_0_request_resource_index][i];
160.         }
161.         if ((*chart).Resource_Node_Array[process_0_request_resource_index]
    > resource_allocated) {//该资源类还有剩余的资源数 进程 0 可以完成
162.             //该进程结点可以释放 删除两个表里与其有关的
163.             for (int m = 0; m < Resource_Node_Num; m++) {
164.                 (*chart).Resource_To_Process_Allocation_Edge[m][4] = 0;
165.             }
166.             for (int n = 0; n < Process_Node_Num; n++) {
167.                 (*chart).Process_To_Resource_Request_Edge[4][n] = 0;
168.             }
169.             Process_Array[4] = 0;
170.             deadlock = 0;
171.         }
172.         for (int i = 0; i <= Process_Node_Num; i++) {
173.             if (i == 6)
174.                 break_flag = 0;
175.             if (Process_Array[i] == 1) {
176.                 deadlock++;
177.                 break;
178.             }
179.
180.         }
181.         if (break_flag == 0)
182.             break;
183.
184.         /*
185.         Process 6
186.         */
187.         for (process_0_request_resource_index = 0; process_0_request_resour
    ce_index < Resource_Node_Num; process_0_request_resource_index++) {
```

```
            if ((*chart).Process_To_Resource_Request_Edge[5][process_0_request_resource_index] == 1) {// j 是进程 0 正在申请的资源的数组下标
                break;
            }
        }
        resource_allocated = 0;
        for (int i = 0; i < Process_Node_Num; i++) {
            resource_allocated += (*chart).Resource_To_Process_Allocation_Edge[process_0_request_resource_index][i];
        }
        if ((*chart).Resource_Node_Array[process_0_request_resource_index] > resource_allocated) {//该资源类还有剩余的资源数 进程 0 可以完成
            //该进程结点可以释放 删除两个表里与其有关的
            for (int m = 0; m < Resource_Node_Num; m++) {
                (*chart).Resource_To_Process_Allocation_Edge[m][5] = 0;
            }
            for (int n = 0; n < Process_Node_Num; n++) {
                (*chart).Process_To_Resource_Request_Edge[5][n] = 0;
            }
            Process_Array[5] = 0;
            deadlock = 0;
        }
        for (int i = 0; i <= Process_Node_Num; i++) {
            if (i == 6)
                break_flag = 0;
            if (Process_Array[i] == 1) {
                deadlock++;
                break;
            }

        }
        if (break_flag == 0)
            break;
    }
}
```

我所设计系统的核心程序、关键函数的程序清单如下，其中 GUI 调用中函数如下：

1）刷新线条函数（分为按钮和画线函数），结合多线程思想，设计了一个将 Qt 中不能被调用的画图函数的“调用”函数，首先是画线函数：

```
void MainWindow::paintEvent(QPaintEvent *event)//画线进程 1 到资源 1 的申请红线
{
    while(true){
```

```
4.          QPainter paint(this);
5.          paint.setPen(QPen(Qt::red,5));
6.          paint.drawLine(530, 33,500, 33);
7.          paint.drawLine(525, 28,530, 33);
8.          paint.drawLine(525, 38,530, 33);
9.          paint.setPen(QPen(Qt::blue,5));
10.         paint.drawLine(530, 63,500, 63);
11.         paint.drawLine(505, 68,500, 63);
12.         paint.drawLine(505, 57,500, 63);
13.         break;
14.
15.     }
16.     //在此图形上下文的坐标系中，使用当前颜色在点 (x1, y1) 和 (x2, y2) 之间画一条线。
17.     /*
18.      * 640,120 是资源 1 的点
19.      *
20.      * 480,480 是进程 6 的点
21.      * 480, 410 是进程 5 的点
22.      * 说明上下距离是 70
23.      */
24.
25.     //学习 java 多线程的时候，学到的思想，使用外部公共变量来精确控制函数内部的代码块的调用
26.     //注意 Qt 中每次 update 都会重新画 之前的线条也就清除了
27.     //init(&chart);
28.     //进程向资源的申请用红线
29.     QPainter paint(this);
30.     paint.setPen(QPen(Qt::red,5));
31.         if(loop == draw_flag){
32.         for(int r_0 = 0; r_0 < 6; r_0++){
33.             if((&chart)->Process_To_Resource_Request_Edge[0][r_0] == 1){
34.                 paint.drawLine(530,115 + 70 * r_0,370, 115);
35.                 break;//进程只能向一个资源申请
36.             }
37.         }
38.         //画进程 2 到资源 i 的线
39.         for(int r_1 = 0; r_1 < 6; r_1++){
40.             if((&chart)->Process_To_Resource_Request_Edge[1][r_1] == 1){
41.                 paint.drawLine(530,115 + 70 * r_1,370, 185);
42.                 //QPainter paint(this);
43.                 //paint.setPen(Qt::red);
44.                 break;//进程只能向一个资源申请
45.             }
```

```
        }
        for(int r_2 = 0; r_2 < 6; r_2++){
            if((&chart)->Process_To_Resource_Request_Edge[2][r_2] == 1){
                //QPainter paint(this);
                //paint.setPen(Qt::red);
                paint.drawLine(530,115 + 70 * r_2,370, 255);
                break;//进程只能向一个资源申请
            }
        }
        for(int r_3 = 0; r_3 < 6; r_3++){
            if((&chart)->Process_To_Resource_Request_Edge[3][r_3] == 1){
                //QPainter paint(this);
                //paint.setPen(Qt::red);
                paint.drawLine(530,115 + 70 * r_3,370, 325);
                break;//进程只能向一个资源申请
            }
        }
        for(int r_4 = 0; r_4 < 6; r_4++){
            if((&chart)->Process_To_Resource_Request_Edge[4][r_4] == 1){
                //QPainter paint(this);
                //paint.setPen(Qt::red);
                paint.drawLine(530,115 + 70 * r_4,370, 395);
                break;//进程只能向一个资源申请
            }
        }
        for(int r_5 = 0; r_5 < 6; r_5++){
            if((&chart)->Process_To_Resource_Request_Edge[5][r_5] == 1){
                //QPainter paint(this);
                //paint.setPen(Qt::red);
                paint.drawLine(530,115 + 70 * r_5,370, 465);
                break;//进程只能向一个资源申请
            }
        }

        //画资源->进程的分配线条
        paint.setPen(QPen(Qt::blue,5));
        r_6_sum = 0;//该类资源分配出去的资源数
        for(int r_6 = 0 ; r_6 < 6; r_6++){
            if((&chart)->Resource_To_Process_Allocation_Edge[0][r_6] != 0){

                r_6_sum += (&chart)->Resource_To_Process_Allocation_Edge[0][
r_6];
                paint.drawLine(370, 120 + 70 * r_6, 530, 120);
                continue;//进程只能向一个资源申请
```

```
            }
            //在方框中显示该类资源分配出去的资源数
            if(r_6 == 5){
                QString qstr_r_6 = QString::number(r_6_sum,10);
                ui->textEdit_29->setText(qstr_r_6);
            }
        }
        r_7_sum = 0;
        for(int r_7 = 0; r_7 < 6; r_7++){
            if((&chart)->Resource_To_Process_Allocation_Edge[1][r_7] != 0){

                r_7_sum += (&chart)->Resource_To_Process_Allocation_Edge[1][
    r_7];
                paint.drawLine(370, 120 + 70 * r_7, 530, 190);
                 continue;//进程只能向一个资源申请
            }
            //在方框中显示该类资源分配出去的资源数
            if(r_7 == 5){
                QString qstr_r_7 = QString::number(r_7_sum,10);
                ui->textEdit_30->setText(qstr_r_7);
            }
         }
         r_8_sum = 0;
         for(int r_8 = 0; r_8 < 6; r_8++){
            if((&chart)->Resource_To_Process_Allocation_Edge[2][r_8] != 0){

                r_8_sum += (&chart)->Resource_To_Process_Allocation_Edge[2]
    [r_8];
                paint.drawLine(370, 120 + 70 * r_8, 530, 260);
                continue;//进程只能向一个资源申请
            }
            //在方框中显示该类资源分配出去的资源数
            if(r_8 == 5){
                QString qstr_r_8 = QString::number(r_8_sum,10);
                ui->textEdit_31->setText(qstr_r_8);
            }
         }

         r_9_sum = 0;
         for(int r_9 = 0; r_9 < 6; r_9++){
            if((&chart)->Resource_To_Process_Allocation_Edge[3][r_9] != 0){

                r_9_sum += (&chart)->Resource_To_Process_Allocation_Edge[3]
    [r_9];
```

```
                paint.drawLine(370, 120 + 70 * r_9, 530, 330);
                continue;//进程只能向一个资源申请
            }
            //在方框中显示该类资源分配出去的资源数
            if(r_9 == 5){
                QString qstr_r_9 = QString::number(r_9_sum,10);
                ui->textEdit_32->setText(qstr_r_9);
            }
        }
        r_10_sum = 0;
        for(int r_10 = 0; r_10 < 6; r_10++){
            if((&chart)->Resource_To_Process_Allocation_Edge[4][r_10] != 0)
{
                r_10_sum += (&chart)->Resource_To_Process_Allocation_Edge[4
][r_10];
                paint.drawLine(370, 120 + 70 * r_10, 530, 400);
                continue;//进程只能向一个资源申请
            }
            //在方框中显示该类资源分配出去的资源数
            if(r_10 == 5){
                QString qstr_r_10 = QString::number(r_10_sum,10);
                ui->textEdit_33->setText(qstr_r_10);
            }
        }
        r_11_sum = 0;
        for(int r_11 = 0; r_11 < 6; r_11++){
            if((&chart)->Resource_To_Process_Allocation_Edge[5][r_11] != 0)
{
                r_11_sum += (&chart)->Resource_To_Process_Allocation_Edge[5
][r_11];
                paint.drawLine(370, 120 + 70 * r_11, 530, 470);
                continue;//进程只能向一个资源申请
            }
            //在方框中显示该类资源分配出去的资源数
            if(r_11 == 5){
                QString qstr_r_11 = QString::number(r_11_sum,10);
                ui->textEdit_34->setText(qstr_r_11);
            }
        }
    }
}
```

然后是画线按钮函数：

```
void MainWindow::on_pushButton_10_clicked()//刷新线条 按钮
{
    /*
     * 之所以这样 是因为 void MainWindow::paintEvent(QPaintEvent *event)好像有记忆功能 用过的 flag 就不能用了（没有 update 作用）
     * 即使用 0 1 2 1 2 1 2 这样的循环也不行
     *
     */

    //进程点的隐藏
    for(int r_0 = 0; r_0 < 6; r_0++){
        if((&chart)->Process_To_Resource_Request_Edge[0][r_0] == 0 && (&chart)->Resource_To_Process_Allocation_Edge[r_0][0] == 0){
            if(r_0 == 5){
                ui->textEdit->setVisible(0);
            }
            continue;
        }
        ui->textEdit->setVisible(1);
        break;
    }

    for(int r_1 = 0; r_1 < 6; r_1++){
        if((&chart)->Process_To_Resource_Request_Edge[1][r_1] == 0 && (&chart)->Resource_To_Process_Allocation_Edge[r_1][1] == 0){
            if(r_1 == 5){
                ui->textEdit_2->setVisible(0);
            }
            continue;
        }
        ui->textEdit_2->setVisible(1);
        break;
    }

    for(int r_2 = 0; r_2 < 6; r_2++){
        if((&chart)->Process_To_Resource_Request_Edge[2][r_2] == 0 && (&chart)->Resource_To_Process_Allocation_Edge[r_2][2] == 0){
            if(r_2 == 5){
                ui->textEdit_3->setVisible(0);
            }
            continue;
        }
        ui->textEdit_3->setVisible(1);
        break;
```

```
    }

    for(int r_3 = 0; r_3 < 6; r_3++){
        if((&chart)->Process_To_Resource_Request_Edge[3][r_3] == 0 && (&chart)->Resource_To_Process_Allocation_Edge[r_3][3] == 0){
            if(r_3 == 5){
                ui->textEdit_4->setVisible(0);
            }
            continue;
        }
        ui->textEdit_4->setVisible(1);
        break;
    }

    for(int r_4 = 0; r_4 < 6; r_4++){
        if((&chart)->Process_To_Resource_Request_Edge[4][r_4] == 0 && (&chart)->Resource_To_Process_Allocation_Edge[r_4][4] == 0){
            if(r_4 == 5){
                ui->textEdit_5->setVisible(0);
            }
            continue;
        }
        ui->textEdit_5->setVisible(1);
        break;
    }

    for(int r_5 = 0; r_5 < 6; r_5++){
        if((&chart)->Process_To_Resource_Request_Edge[5][r_5] == 0 && (&chart)->Resource_To_Process_Allocation_Edge[r_5][5] == 0){
            if(r_5 == 5){
                ui->textEdit_6->setVisible(0);
            }
            continue;
        }
        ui->textEdit_6->setVisible(1);
        break;
    }

    //资源点的隐藏
    for(int r_0 = 0; r_0 < 6; r_0++){//资源点 1
        if((&chart)->Process_To_Resource_Request_Edge[r_0][0] == 0 && (&chart)->Resource_To_Process_Allocation_Edge[0][r_0] == 0){
            if(r_0 == 5){
                ui->textEdit_8->setVisible(0);
```

```
            }
            continue;
        }
        ui->textEdit_8->setVisible(1);
        break;
    }

    for(int r_1 = 0; r_1 < 6; r_1++){
        if((&chart)->Process_To_Resource_Request_Edge[r_1][1] == 0 && (&char
t)->Resource_To_Process_Allocation_Edge[1][r_1] == 0){
            if(r_1 == 5){
                ui->textEdit_9->setVisible(0);
            }
            continue;
        }
        ui->textEdit_9->setVisible(1);
        break;
    }

    for(int r_2 = 0; r_2 < 6; r_2++){
         if((&chart)->Process_To_Resource_Request_Edge[r_2][2] == 0 && (&cha
rt)->Resource_To_Process_Allocation_Edge[2][r_2] == 0){
             if(r_2 == 5){
                 ui->textEdit_10->setVisible(0);
             }
             continue;
         }
         ui->textEdit_10->setVisible(1);
         break;
     }

     for(int r_3 = 0; r_3 < 6; r_3++){
         if((&chart)->Process_To_Resource_Request_Edge[r_3][3] == 0 && (&cha
rt)->Resource_To_Process_Allocation_Edge[3][r_3] == 0){
             if(r_3 == 5){
                 ui->textEdit_11->setVisible(0);
             }
             continue;
         }
         ui->textEdit_11->setVisible(1);
         break;
     }

     for(int r_4 = 0; r_4 < 6; r_4++){
```

```
122.         if((&chart)->Process_To_Resource_Request_Edge[r_4][4] == 0 && (&cha
    rt)->Resource_To_Process_Allocation_Edge[4][r_4] == 0){
123.             if(r_4 == 5){
124.                 ui->textEdit_12->setVisible(0);
125.             }
126.             continue;
127.         }
128.         ui->textEdit_12->setVisible(1);
129.         break;
130.     }
131.
132.     for(int r_5 = 0; r_5 < 6; r_5++){
133.         if((&chart)->Process_To_Resource_Request_Edge[r_5][5] == 0 && (&cha
    rt)->Resource_To_Process_Allocation_Edge[5][r_5] == 0){
134.             if(r_5 == 5){
135.                 ui->textEdit_7->setVisible(0);
136.             }
137.             continue;
138.         }
139.         ui->textEdit_7->setVisible(1);
140.         break;
141.     }
142.
143.     if(loop == 0){
144.         loop++;
145.         update();
146.     }
147.     else{
148.         draw_flag++;
149.         loop++;
150.         update();
151.     }
152. }
```

2）显示资源类已分配资源数函数，点击即可刷新显示各个资源类已经分配出去的资源数。

```
1.  void MainWindow::on_pushButton_12_clicked()
2.  {
3.      QString qstr_19 = QString::number(r_6_sum,10);
4.      ui->textEdit_29->setText(qstr_19);
5.
6.      QString qstr_20 = QString::number(r_7_sum,10);
7.      ui->textEdit_30->setText(qstr_20);
```

```
8.
9.      QString qstr_21 = QString::number(r_8_sum,10);
10.     ui->textEdit_31->setText(qstr_21);
11.
12.     QString qstr_22 = QString::number(r_9_sum,10);
13.     ui->textEdit_32->setText(qstr_22);
14.
15.     QString qstr_23 = QString::number(r_10_sum,10);
16.     ui->textEdit_33->setText(qstr_23);
17.
18.     QString qstr_24 = QString::number(r_11_sum,10);
19.     ui->textEdit_34->setText(qstr_24);
20. }
```

3）显示资源类总资源数函数，点击即可刷新显示各个资源类所有的资源数。

```
1.  void MainWindow::on_pushButton_clicked()//显示资源数按钮
2.  {
3.      //每个资源类内的资源个数为 qstr 从 13 开始
4.      QString qstr_13 = QString::number((&chart)->Resource_Node_Array[0],10);//
    /Resource_Node_Array[]
5.      ui->textEdit_13->setText(qstr_13);
6.      QString qstr_14 = QString::number((&chart)->Resource_Node_Array[1],10);//
    /Resource_Node_Array[]
7.      ui->textEdit_14->setText(qstr_14);
8.      QString qstr_15 = QString::number((&chart)->Resource_Node_Array[2],10);//
    /Resource_Node_Array[]
9.      ui->textEdit_15->setText(qstr_15);
10.     QString qstr_16 = QString::number((&chart)->Resource_Node_Array[3],10);//
    /Resource_Node_Array[]
11.     ui->textEdit_16->setText(qstr_16);
12.     QString qstr_17 = QString::number((&chart)->Resource_Node_Array[4],10);//
    /Resource_Node_Array[]
13.     ui->textEdit_17->setText(qstr_17);
14.     QString qstr_18 = QString::number((&chart)->Resource_Node_Array[5],10);//
    /Resource_Node_Array[]
15.     ui->textEdit_18->setText(qstr_18);
16. }
```

4）对内核中各个函数调用的控件的事件函数。

```
1.  void MainWindow::on_pushButton_9_clicked()//存入到文件 按钮
2.  {
3.      Chart_Write_To_File(&chart);
```

```
}

void MainWindow::on_pushButton_8_clicked()//从文件读取 按钮
{
    Chart_Read_From_File(&chart);
}

void MainWindow::on_pushButton_11_clicked()//初始化按钮
{
        init(&chart);

}

void MainWindow::on_pushButton_2_clicked()//删除进程
{
    QString qstr_delete_P = ui->textEdit_20->toPlainText();
    int temp = qstr_delete_P.toInt();
    Delete_Process_Node(&chart, temp);
}

void MainWindow::on_pushButton_3_clicked()//删除资源
{
    QString qstr_delete_R = ui->textEdit_21->toPlainText();
    int temp = qstr_delete_R.toInt();
    Delete_Resource_Node(&chart, temp);
}

void MainWindow::on_pushButton_4_clicked()//删除进程对资源的请求
{
    QString qstr_P = ui->textEdit_19->toPlainText();
    int temp_P = qstr_P.toInt();

    QString qstr_R = ui->textEdit_25->toPlainText();
    int temp_R = qstr_R.toInt();

    Delete_Request_Edge(&chart, temp_P, temp_R);
}

void MainWindow::on_pushButton_5_clicked()//删除资源到进程的分配(一次只删一个资源而不是一个类）
{
    QString qstr_P = ui->textEdit_26->toPlainText();
    int temp_P = qstr_P.toInt();

```

```
47.     QString qstr_R = ui->textEdit_22->toPlainText();
48.     int temp_R = qstr_R.toInt();
49.
50.     Delete_Allocation_Edge(&chart, temp_R, temp_P);
51. }
52.
53. void MainWindow::on_pushButton_6_clicked()//增加进程对资源的请求（前提是该进程对其他资源没有请求才可以）
54. {
55.     QString qstr_P = ui->textEdit_23->toPlainText();
56.     int temp_P = qstr_P.toInt();
57.
58.     QString qstr_R = ui->textEdit_27->toPlainText();
59.     int temp_R = qstr_R.toInt();
60.
61.     Add_Edge_Process_To_Resource(&chart, temp_P, temp_R);
62. }
63.
64. void MainWindow::on_pushButton_7_clicked()//增加资源对进程的分配
65. {
66.     QString qstr_P = ui->textEdit_28->toPlainText();
67.     int temp_P = qstr_P.toInt();
68.
69.     QString qstr_R = ui->textEdit_24->toPlainText();
70.     int temp_R = qstr_R.toInt();
71.
72.     Add_Edge_Resource_To_Process(&chart, temp_R, temp_P);
73. }
74. void MainWindow::on_pushButton_13_clicked()
75. {
76.     int sum = 0;
77.     Judge_DeadLock(&chart);
78.     for(int i = 0; i < Resource_Node_Num; i++){
79.         for(int j = 0; j < Process_Node_Num; j++){
80.             sum += (&chart)->Resource_To_Process_Allocation_Edge[i][j];
81.         }
82.     }
83.     for(int i = 0; i < Process_Node_Num; i++){
84.         for(int j = 0; j < Resource_Node_Num; j++){
85.             sum += (&chart)->Process_To_Resource_Request_Edge[i][j];
86.         }
87.     }
88.     if(sum == 0){
```

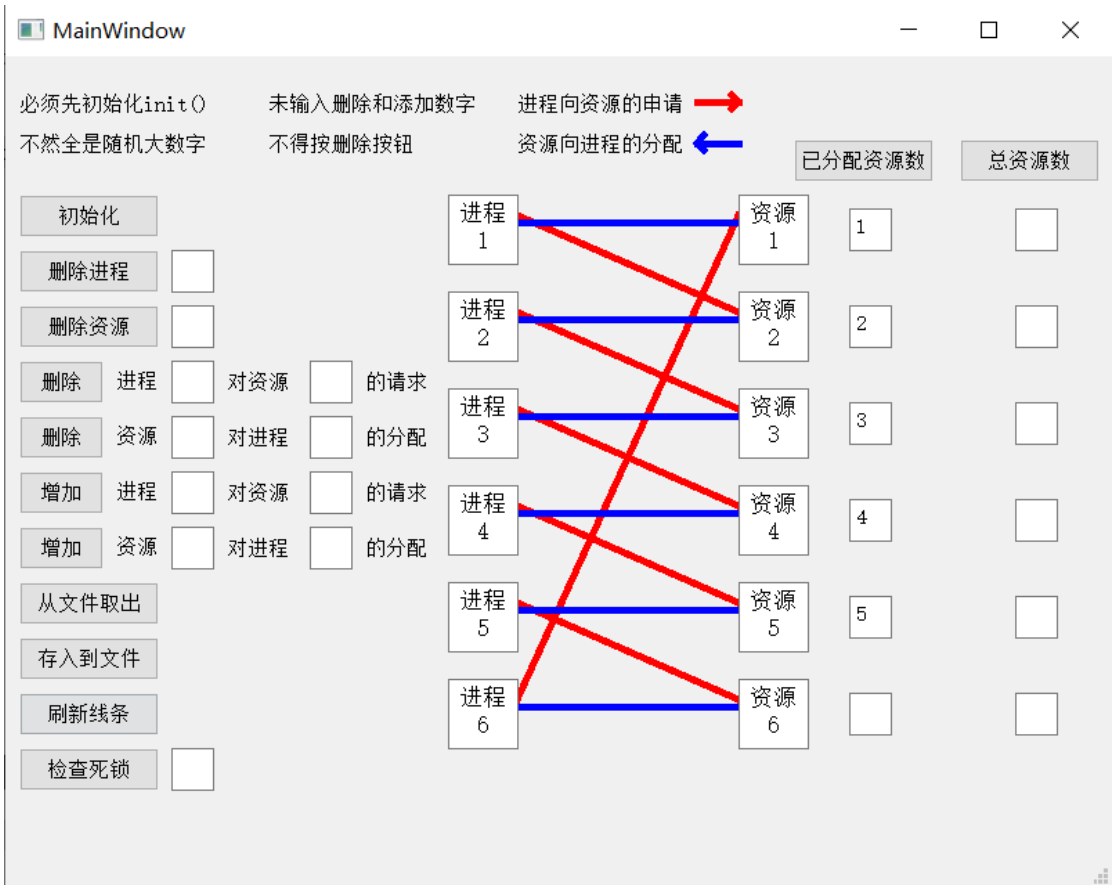
```
89.         ui->textEdit_35->clear();//先清空显示窗口再显示，可以达到实时刷新显示的目
    的
90.         ui->textEdit_35->setText(QString::fromStdString("N"));
91.         //ui->textEdit_35->insertPlainText("无");//这里别用 append()是添加一个
    新行显示
92.     }
93.     else{
94.         ui->textEdit_35->clear();
95.         ui->textEdit_35->setText(QString::fromStdString("Y"));
96.     }
97. }
```

7. 程序运行的主要界面和结果截图

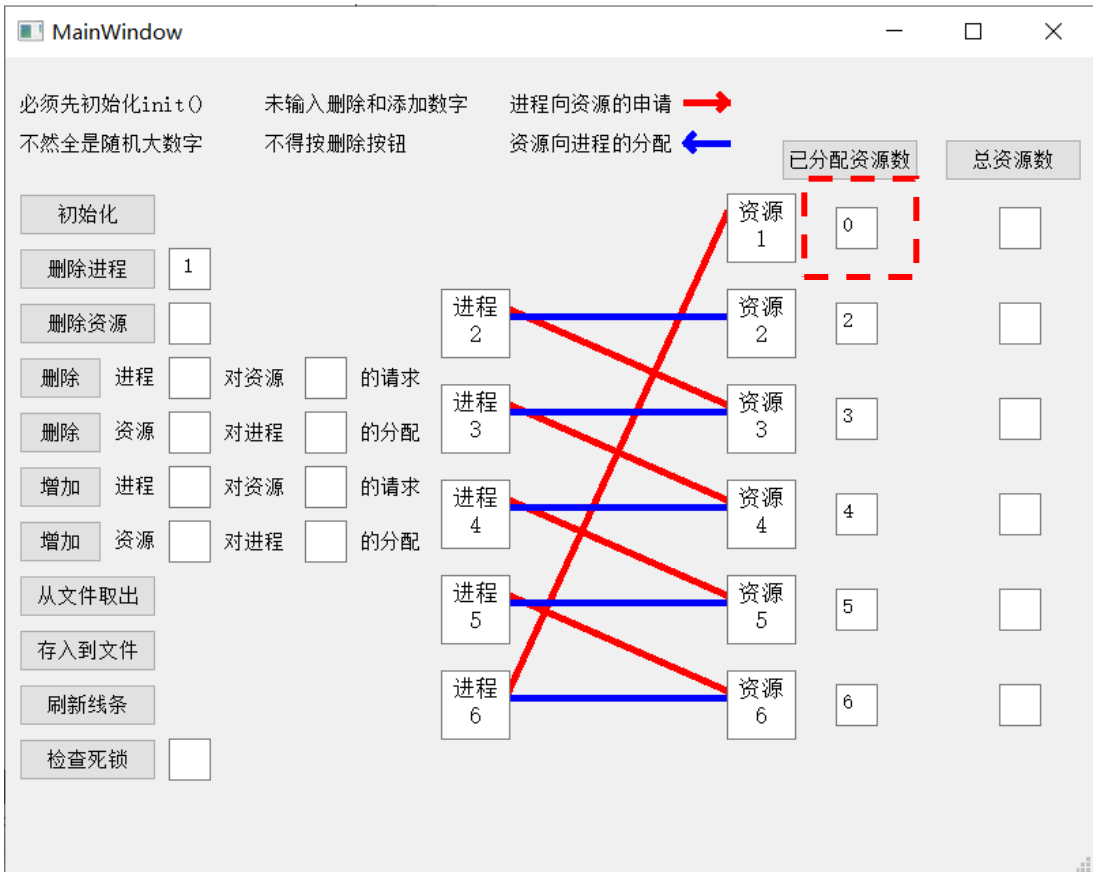
我将按照自己设计的算例进程运行和截图：

算例如下：

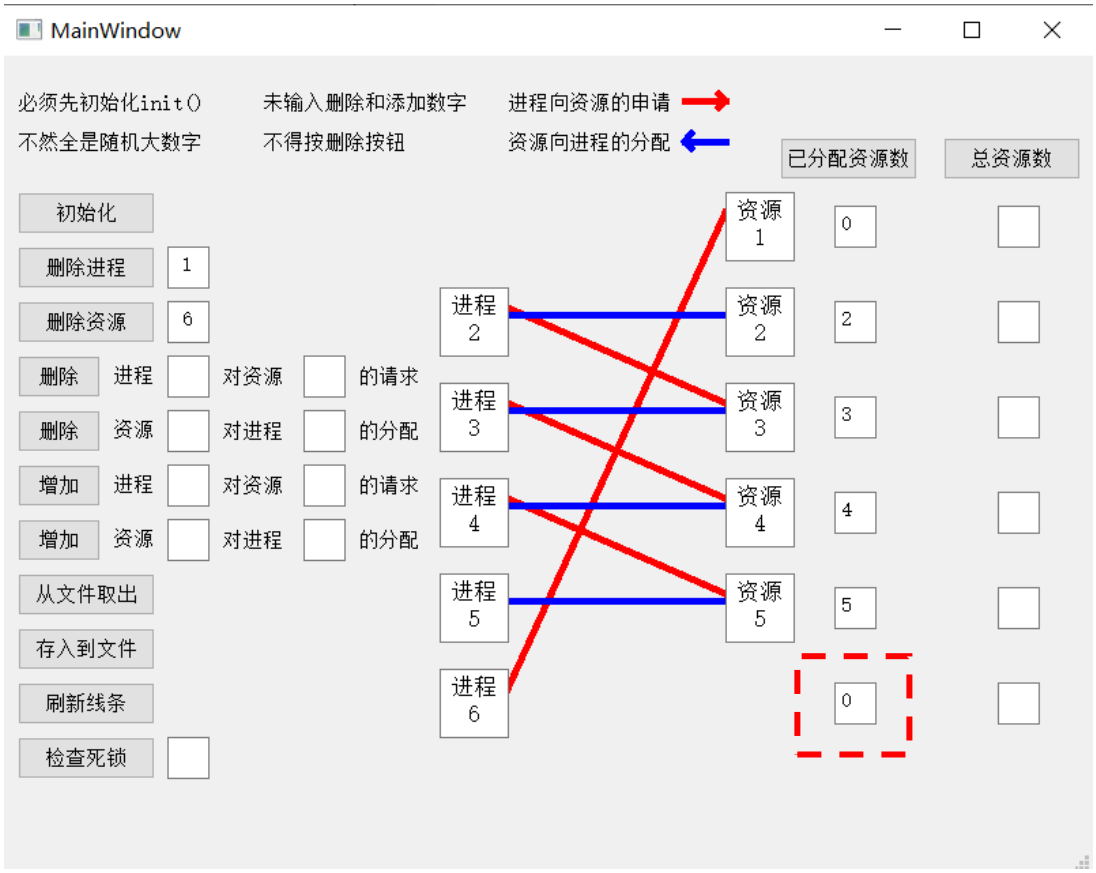
1）首先运行，按下初始化按钮，然后刷新线条，看看资源分配图如何，初始化是一个我设计的死锁状态；



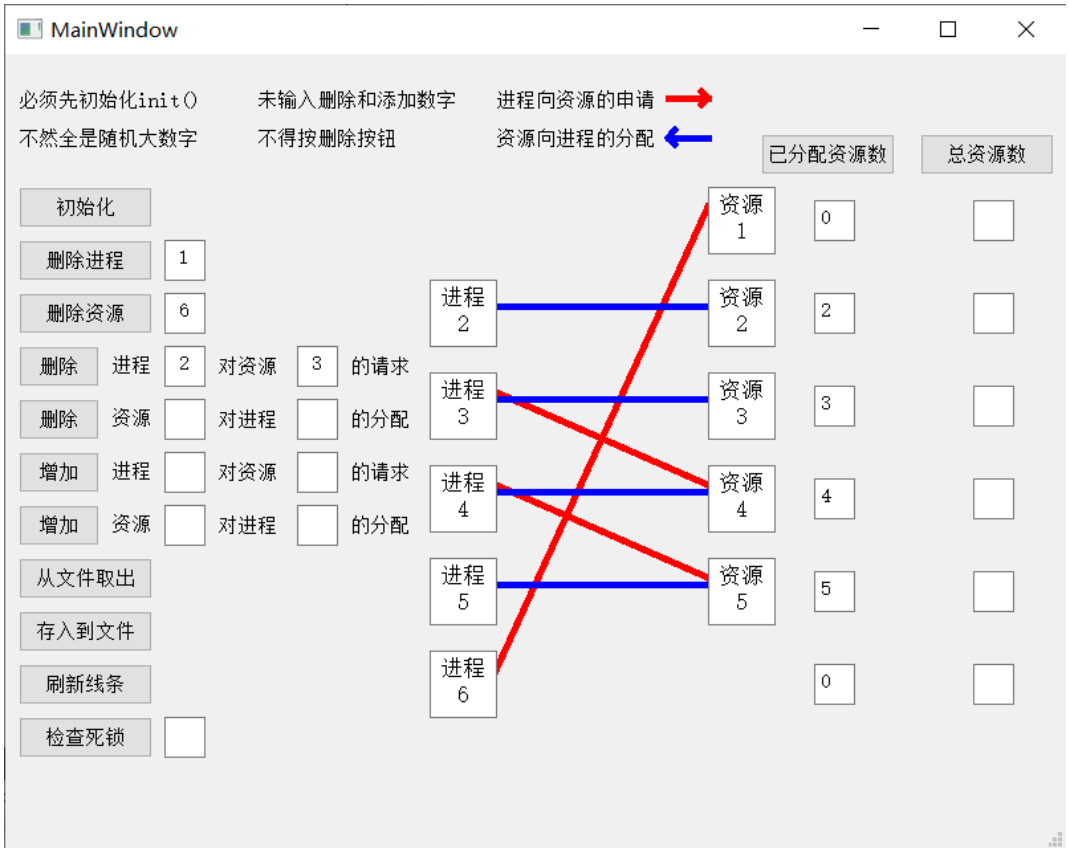
2）删除进程 1，然后刷新线条，进程 1 和与其有关的线条都没了，资源 1 的已分配资源被回收，变成 0（如红框所示）；



3）删除资源 6：资源 6 和与其有关的线条都没了，资源 6 的已分配资源被回收，变成 0（如红框所示）；

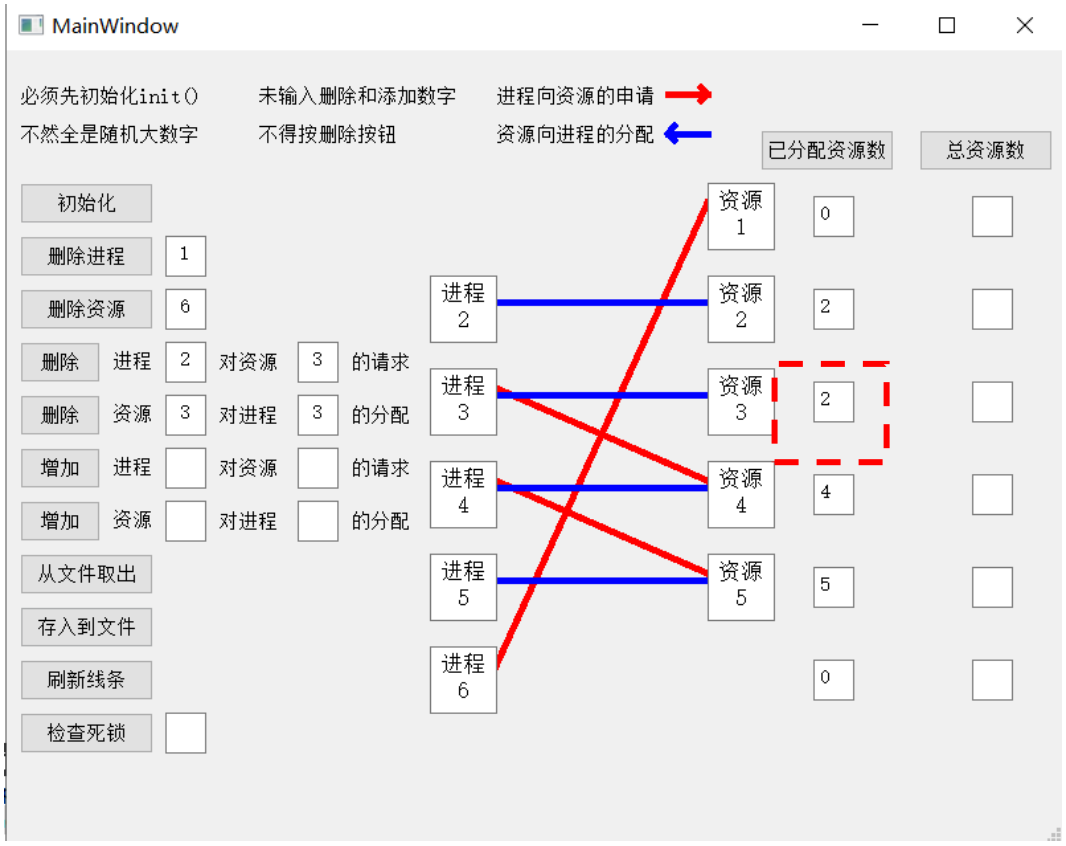


4）删除进程 2 对资源 3 的请求：

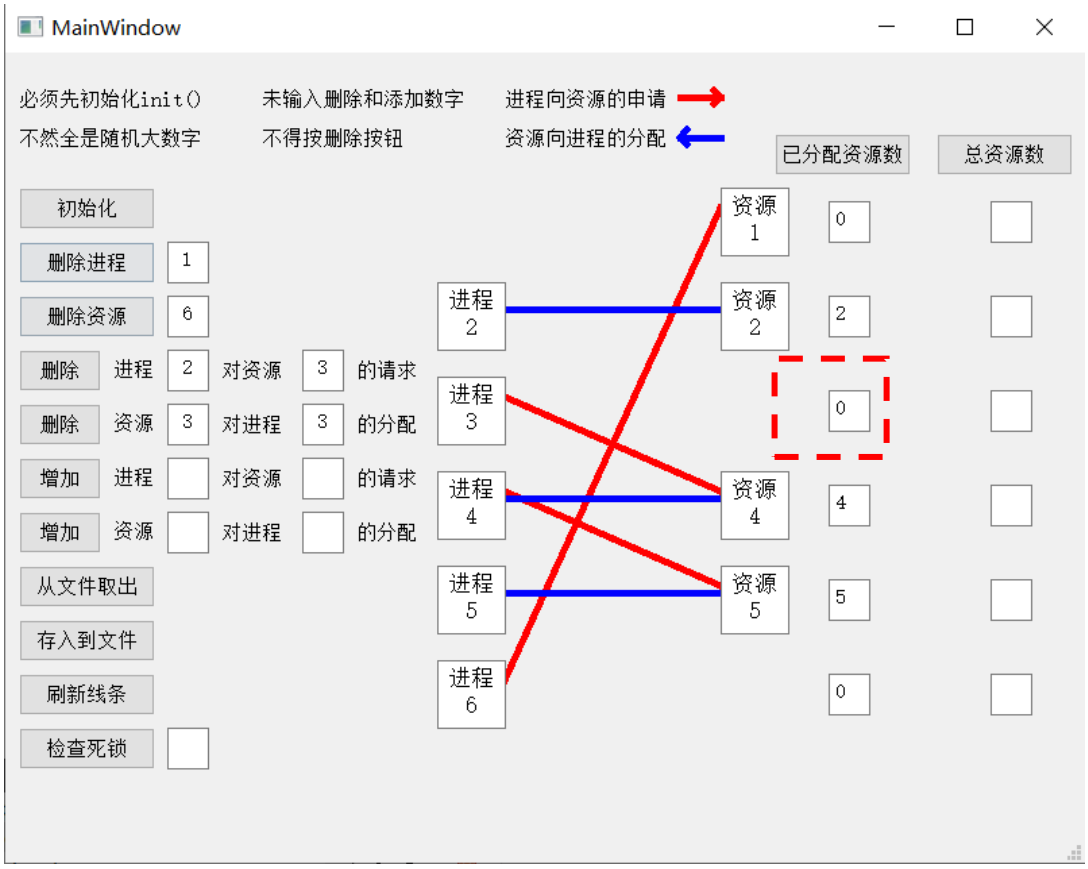


5）删除资源 3 对进程 3 的分配，可以看到删除一次没有变化，

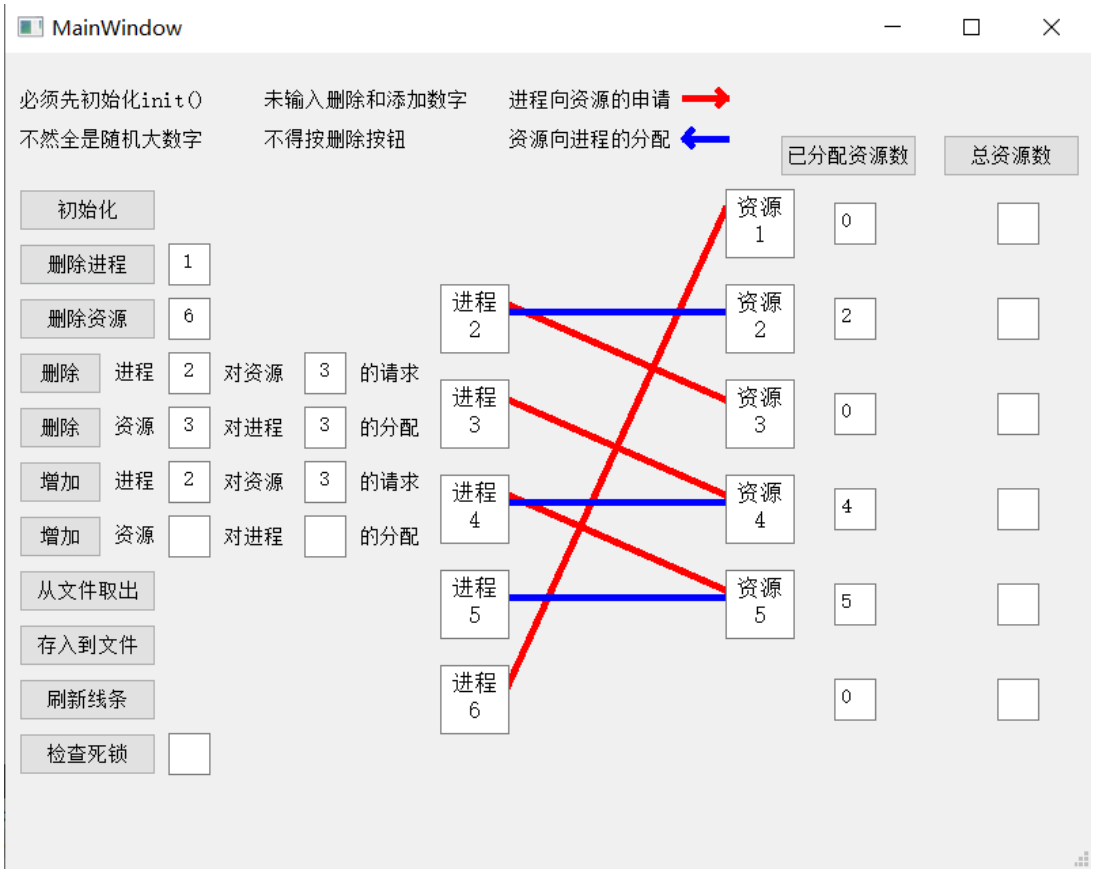
但是后面的已分配资源数变成 2（如红框所示）了，因为一共分配了三个；



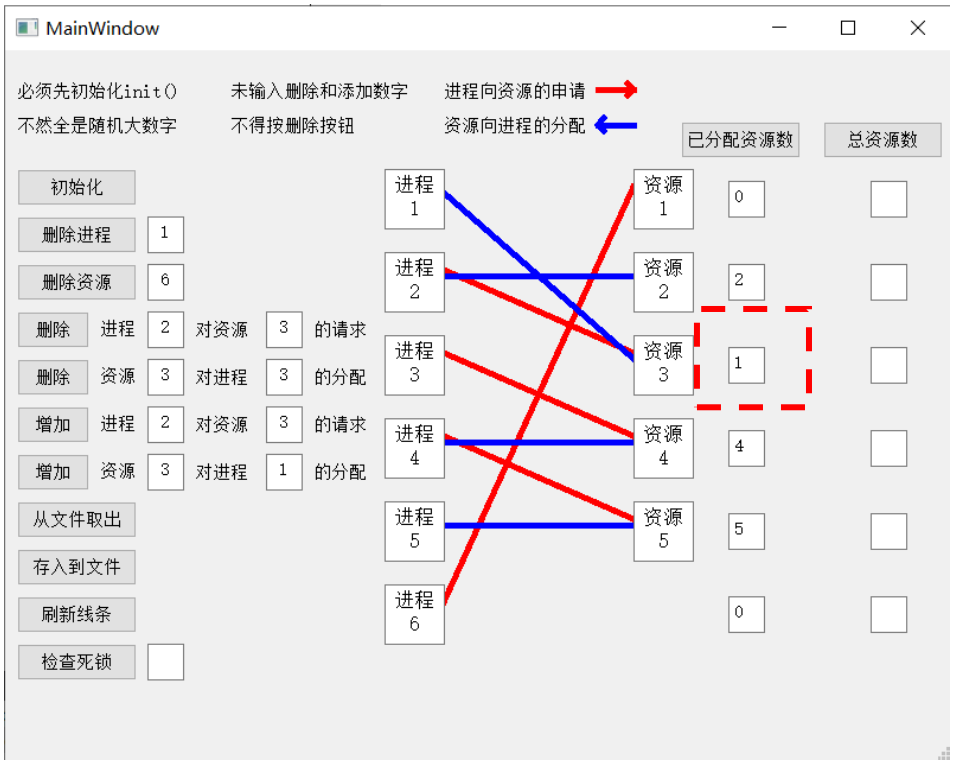
一共连续删除三次之后，再刷新就可以看到资源 3 的分配释放完毕，节点消失。



6）增加进程 2 对资源 3 的请求，刷新，可以看到成功；



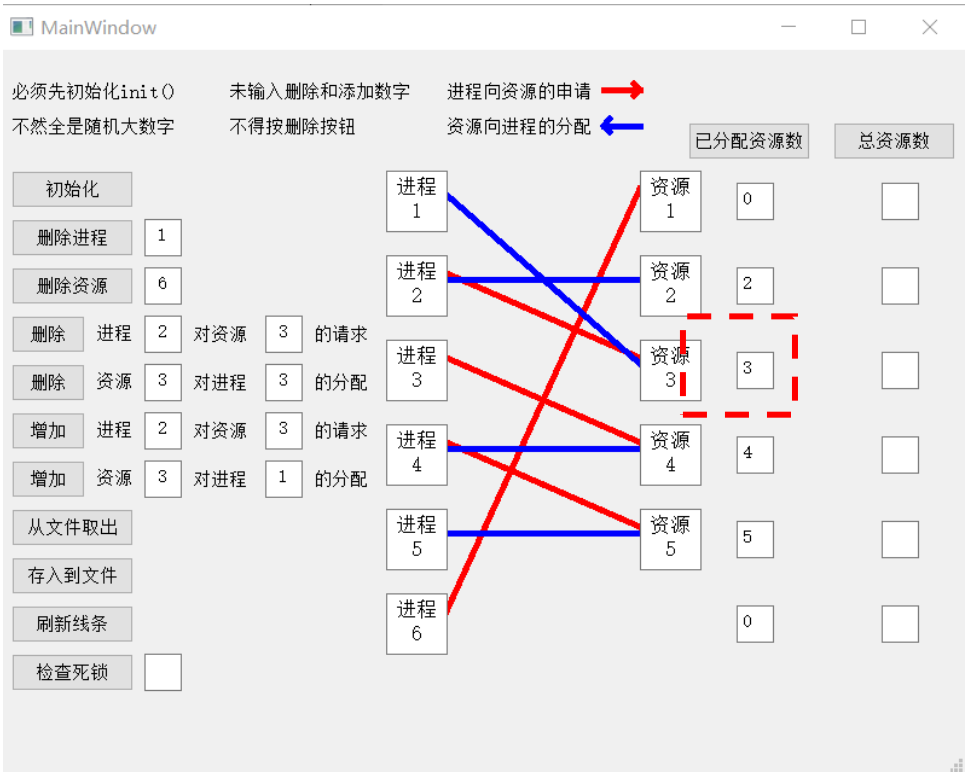
7）增加资源 3 对进程 1 的分配，成功，且资源 3 的已分配资源数+1（如红框所示）；



增加资源 3 对进程 2 的分配，成功，且资源 3 的已经分配资源数+1

增加资源 3 对进程 3 的分配，成功，且资源 3 的已经分配资源数+1，此时，资源 3 的已分配资源数来到 3，到达上限（如红框所示）

再增加资源 3 对进程 4 的分配，就不生效（在验收时已经体现）



8）到这里，先不要重新初始化，先看一下 Data.txt 文件

然后我们按下存入到文件，再次打开看一下文件

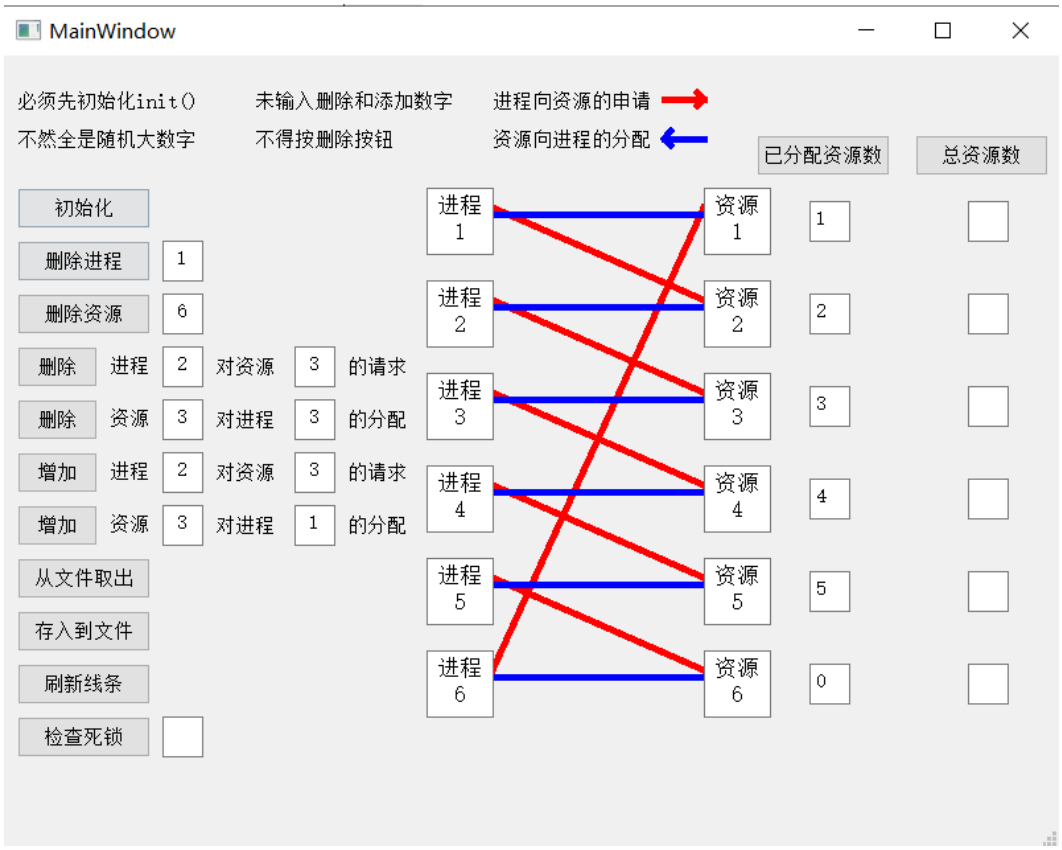
那么现在初始化一下，存入到文件，然后再看一下，可以看到有变化的，且与当前对应的程序内的数据结构一致。

以上可以说明“存入到文件”功能是成功的。

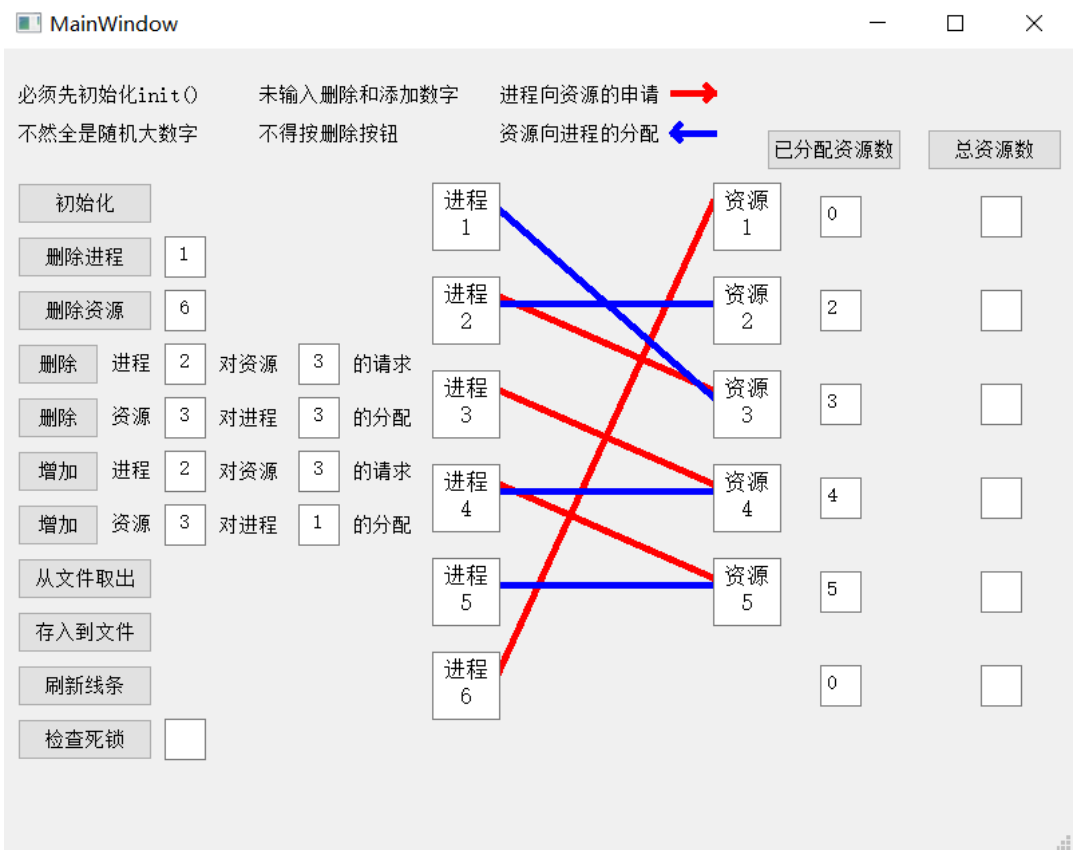
下面继续验证“从文件取出”功能的正确性。

首先初始化一下，然后刷新线条。

可以看到界面如下：



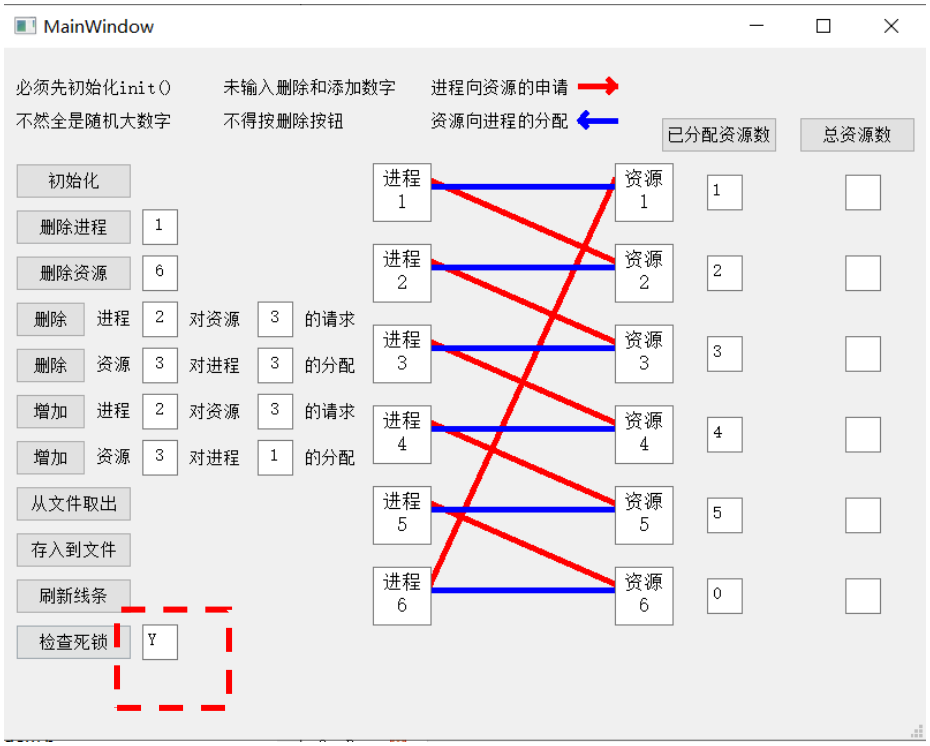
然后点击从文件取出，然后再刷新线条：



相当于把之前的存入文件的备份取出，说明“从文件取出”功能成功。

10）最后是检查死锁问题；初始化，刷新，

然后检查死锁，是 Y（代表 Yes，如红框所示），确实是死锁状态，没有一个进程得到释放。



然后删除进程 1，刷新，再次检查死锁，变成 N（代表 No，如红框所示），也就是没有死锁，

刷新，所有进程、资源全部释放，完成！

注：以上所有算例演示皆在验收时已演示给李老师。

8. 总结和感想体会

本实验所有代码没有一处借鉴，全为自己自学 Qt 等来实现，所以略显粗糙，但是收获比借鉴代码得到的收获要多得多。由于我选择自己设计数据结构，所以需要对死锁和资源分配图有更深刻的理解，这也促进我再次复习了一下相关的知识，体会很多，“温故而知新，可以为师矣”。

由于我准备出国读研，不需要复习高数等基础课程，省下的时间一直在锻炼编程能力和项目经验，所以对于本课设，我遇到的难度并不在于代码方面，而是 lib 的生成以及使用问题，比如 github 上只有最高版本到卸载 vs2015 的工具，所以我早期一直是使用 vs2015，但是想要设计界面，就不得不借助于 Qt，而 Qt 中编译器与 vs 兼容的只有 MSVC2017，就需要下载 vs2017，这里卡了我很久，最终安装 vs2017 才解决了，对于一直不知其所以然的 lib 部分也有了自己的理解，由于我是班上最早完成这个课设 lib 要求部分的，也帮助班上很多同学解决了一些 lib 方面的问题，比如选择什么项目才能生成解决方案得到 lib，如何理解头文件 h 和源文件 c 之间的引用关系，用 Qt 设计界面时在 vs 上有哪些需要注意等等。在这个帮助别人成长的过程，我也对实验有了更深的理解。

最大的收获是对于课内学习的 lib 的使用和功能有了实操下的理解，对其他《计算机操作系统》课内学习到的知识也有了真正实践的经验。

下面主要列举实验过程中遇到的比较难解决的海量问题和对应的解决办法，这些问题我将以自问自答的形式列出（这些问题记录在我的 Google 浏览记录中，所以很好回溯）：

1） 什么是例程？

答：例程的作用类似于函数，但含义更为丰富一些。例程是某个系统对外提供的功能接口或服务的集合。比如操作系统的 API、服务等就是例程；Delphi 或 C++ Builder 提供的标准函数和库函数等也是例程。我们编写一个 DLL 的时候，里面的输出函数就是这个 DLL 的例程。

2） 想要用 c 语言的 lib 为库，设计带有界面的程序，如何做？

答：MFC 或者将 lib 库加入兼容 C++的预编译，然后就可以在 Qt 中设计界面执行。

3） 有向图的数据结构有哪些巧妙的设计？

答：对于无向图，一般用邻接矩阵、邻接表作为其数据结构。在此基础上，我认为有向图就可以用两个矩阵来实现。（资源到进程的边用一个矩阵，进程到资源的边用一个矩阵）

4） C 语言中怎么实现类？（由于我们是直接学习的 C++，对 C 只有很浅的了解）

答：类也就是 class 是 C++里面的概念，类是因为面向对象而产生的，在 C 这个面向过程的语言中，没有类存在的必要。C 里也没有 class 这个关键字的。从某些方面可以认为 class 是 struct 的扩展和升华。

5） 关于 c 语言生成 lib？

答：由于我用的是 VS2017，与之前版本有所不同，所以新建一个 Windows 桌面向导，然后选择静态链接库 lib，取消预编译头。点击完成。添加新建项 Os_Kernal.c 和 Os_Kernal.h,编写好之后，选择 x64，然后 Debug，生成解决方案，在项目文件夹的 Debug 文件夹中即有.lib 文件。

6） c 语言中如何将结构体传递给函数，并且在函数内部改变结构体的值？

答：c 语言中将结构体对象指针作为函数的参数实现对结构体成员的修改。

7） fopen 与 fopen_s 的区别？

答：

fopen 原型：

```
1.  FILE * fopen(const char * path,const char * mode);
```

接收两个实参；返回值：文件顺利打开后，指向该流的文件指针就会被返回。如果文件打开失败则返回 NULL，并把错误代码存在 errno 中。

示例程序源码：

```
1.  FILE *cfPtr;
2.
3.                          if((cfPtr = fopen("test.dat","w")) == NULL)   //若
    cfPtr = NULL，即文件未成功打开，函数返回 0，否则返回 1
4.
5.                                  return 0;
6.
7.                          else   return 1;
```

fopen_s 原型：

```
errno_t fopen_s( FILE** pFile, const char *filename, const char *mode );
errno_t _wfopen_s( FILE** pFile, const wchar_t *filename, const wchar_t *mode );
```

pFile：文件指针将接收到打开的文件指针指向的指针。

infilename：文件名。

inmode：允许的访问类型。

返回值：如果成功返回 0，失败则返回相应的错误代码

8） 文件操作 fwrite 写 txt 文件乱码怎么办？

答：

fwrite 是根据二进制写入的函数

fprintf 则是按格式写入的函数。

但是要注意： fprintf fscanf 一对，fwrite fread 一对，最好对应他们的另一半。交叉使用的话不保险，还有就是文件的使用方式既然使用文本文件的函数 那么打开文件时也要对应的操作，这里推荐使用 w+，好处是如果你没有你输入文件名这个文件，他会帮你建一个新的，而且可读可写。不会乱码。

9） Qt 报错：warning: ignoring #pragma comment [-Wunknown-pragmas]

答：Qt 中导入 win 库的时候 不能使用#pragma comment(lib,...) 这是 MSVC 专有的表达式，在 mingw 中，需要在 Qt 的 pro 文件中加入 LIBS += -lxxx 即可，比如：msvc 中：

minGw:

```
#include <Shlwapi.h>
#pragma comment(lib, "shlwapi")
```

pro 文件添加：

```
LIBS += -lshlwapi
```

添加头文件：

```
#include <Shlwapi.h>
```

备注：mingw 使用 msvc 的方式，会发出 warning: ignoring #pragma comment [-Wunknown-pragmas]

10）C++如何使用 C 的 lib？

答：

```
#ifdef __cplusplus
extern "C" {
    void init(Resourece_Allocation_Chart *chart);
    void Chart_Read_From_File(Resourece_Allocation_Chart *chart);
    void Add_Edge_Resource_To_Process(Resourece_Allocation_Chart *chart, int Resourece_index, int Process_index);
    void Add_Edge_Process_To_Resource(Resourece_Allocation_Chart *chart, int Process_index, int Resource_index);
    void Delete_Resource_Node(Resourece_Allocation_Chart*chart, int Resource_index);
    void Delete_Process_Node(Resourece_Allocation_Chart*chart, int Process_index);
    void Delete_Allocation_Edge(Resourece_Allocation_Chart *chart, int Resource_index, int Process_index);
    void Delete_Request_Edge(Resourece_Allocation_Chart *chart, int Resource_index, int Process_index);
    void Chart_Write_To_File(Resourece_Allocation_Chart *chart);
    void Judge_DeadLock(Resourece_Allocation_Chart *chart);
    }
#endif
#endif
```

11）Qt creator 编译错误 ： cannot find file .pro qt

答：QT Creator 对大于带有两个空格的目录和中文命名的目录不支持。

12）Qt 中 kits 无 MSVC？

答：QT 默认安装下，无法使用 windows 的 CDB 调试器，无法调试 MSVC 编译的程序。显示界面上 MSVC 的编译器的图标带有黄色感叹号；需要下载 Visual Studio 2017。

参考文献

[1]郑然，庞丽萍. 计算机操作系统实验指导[M]. 人民邮电出版社:, 201408.125.

[2]杨宗德，吕光宏，刘雍. Linux 高级程序设计[M]. 人民邮电出版社:,

201211.498.

[3]禹振,苏小红,邱景.使用锁分配图动态检测混合死锁[J].计算机研究与发展,2017,54(07):1557-1568.

[4]潘敏学,李倩,李宣东.死锁检测工具的能力分析与综合应用[J].计算机科学与探索,2010,4(02):153-164.

附录

本实验难度 3，推荐完成人数 1-2 人，我在实验基本要求的基础上，自己**精心设计了资源分配图的数据结构，完成了紧凑、节省空间的数据结构设计要求**；并且**独立设计了图形界面，展示了一目了然的实验结果**；并且**设计了更能说明方法的算例，展示了我设计在内核中的所有函数方法**；检测死锁的算法我在课程内已经掌握熟练，考虑到增加些课设的难度，我额外增加了检测死锁的功能，在特定的情况下，**我自己设计了一种检测死锁的方法**。所有代码的比较有技巧的部分，我都在源码中注释出来了。

本程序涉及的代码皆已在我的 github 上传。

https://github.com/sunmiao0301/HFUT_CS/tree/main/%E5%A4%A7%E5%AD%A6%E4%B8%89%E5%B9%B4%E7%BA%A7%E4%B8%8B%E5%AD%A6%E6%9C%9F/%E7%B3%BB%E7%BB%9F%E8%BD%AF%E4%BB%B6%E7%BB%BC%E5%90%88%E8%AE%BE%E8%AE%A1-%E6%93%8D%E4%BD%9C%E7%B3%BB%E7%BB%9F%EF%BC%88%E7%8E%B0%E6%94%B9%E5%90%8D%E4%B8%BA%E6%93%8D%E4%BD%9C%E7%B3%BB%E7%BB%9F%E8%AF%BE%E7%A8%8B%E8%AE%BE%E8%AE%A1%EF%BC%89