改进银行家算法

设计文档

目录

1　需求分析

1.1　引言

1.1.1　编写目的

此需求说明书的编写仅用来让开发人员与用户对所开发系统的理解站在同一层面上。通过阅读此文档，开发人员可以获取到当前业务开发的具体需求以及需要实现的相应功能；用户可以确认开发人员是否对其所提出的业务有一个清楚的认识，并对即将开发的系统功能有一个前瞻性的了解。

1.1.2　项目说明

项目名称：改进银行家算法实现

开发人员：*****

1.1.3　背景

该项目由*****独立开发和维护，致力于完成*****。

1.2　项目概述

1.2.1　项目来源及背景

银行家算法在操作系统的避免死锁中一直扮演着较为重要的角色。

传统银行家算法对进程申请资源的处理思路如图 1 所示：

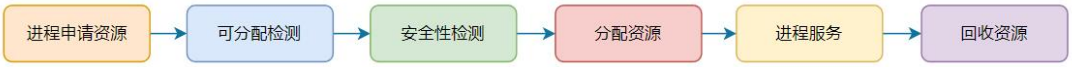


图 1 传统银行家算法思路

虽然该算法可以很好地避免死锁的产生，但由于承载系统没有一个良好的界面环境，导致用户操作起来非常麻烦；并且算法对安全性检测的要求较低，只需要判断是否含有安全序列即可，并没有对安全序列的资源利用效率考虑，这很可能让系统服务进程时进入效率低下的岔路序列中去；除外，由于算法在进程服务上并没有开展相应的多线程技术，这往往导致整个系统的运行速度十分缓慢。

改进银行家算法实现建立在传统银行家算法基础之上，在界面、安全性检测、进程服务等方面进行了改进。首先，此系统以良好的界面让用户在使用上不再感到繁琐，向导式开发也让操作更加清晰，对于开发人员来说，在后期的扩展、改进方面十分方便。在安全性检测方面，系统会根据资源利用效率对所有的安全序列进行排序，选用效率最高的序列，尽可能地提升系统的资源利用率。在进程服务方面，新的系统选用多线程服务，利用线程之间的并行极大地提高了系统的运行速度。

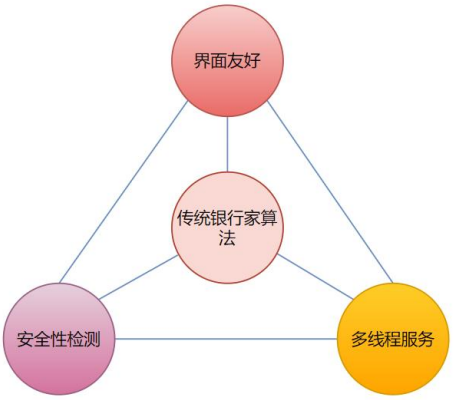


图 2 改进银行家算法

1.2.2 项目目标

（1）基于传统银行家的执行思路，在给定客户数目和资源类别数目的情况下系统能够自行生成必要数据并运行；

（2）明确各模块之间的调用关系及实现中的关键技术；

（3）在算法运行过程中系统可以开启多个安全线程，并确保数据间的同步关系。

1.2.3 项目任务

（1）界面友好；

（2）n 个客户，m 类资源（每个资源的上限随机生成）；

（3）已分配资源的初始值是随机生成的；

（4）需求资源的初始值是随机生成的；

（5）每个客户占用每类资源的时间是不同的；

（7）生成尽可能多的安全序列，并从资源利用效率方面给出这些安全序列的排序；

1.3 系统分析建模

1.3.1 功能模型——数据流图

数据流图以图形的方式刻画了数据流从输入到输出的传输变换过程，描述了整体系统中数据的流向及处理。系统中，用户输入的客户数目以及资源种类数目送至①（初始化资源分配表）中用于生成资源分配表；输入的资源请求送至②（接受请求向量）中用于接受数据。资源请求与资源分配表中的资源分配信息一起流向③（安全性检测）中用于生成安全序列，此时会对资源分配表进行修改。④（计算资源利用效率）通过模拟运行计算安全序列对应时间并改写安全序列信息表。⑤（安全序列排序）通过比对资源利用效率对安全序列进行排序。排序后的资源利用率最高的安全序列流向⑥(多线程服务)，结合资源分配信息开展多线程服

务，最后的服务结果送给用户。

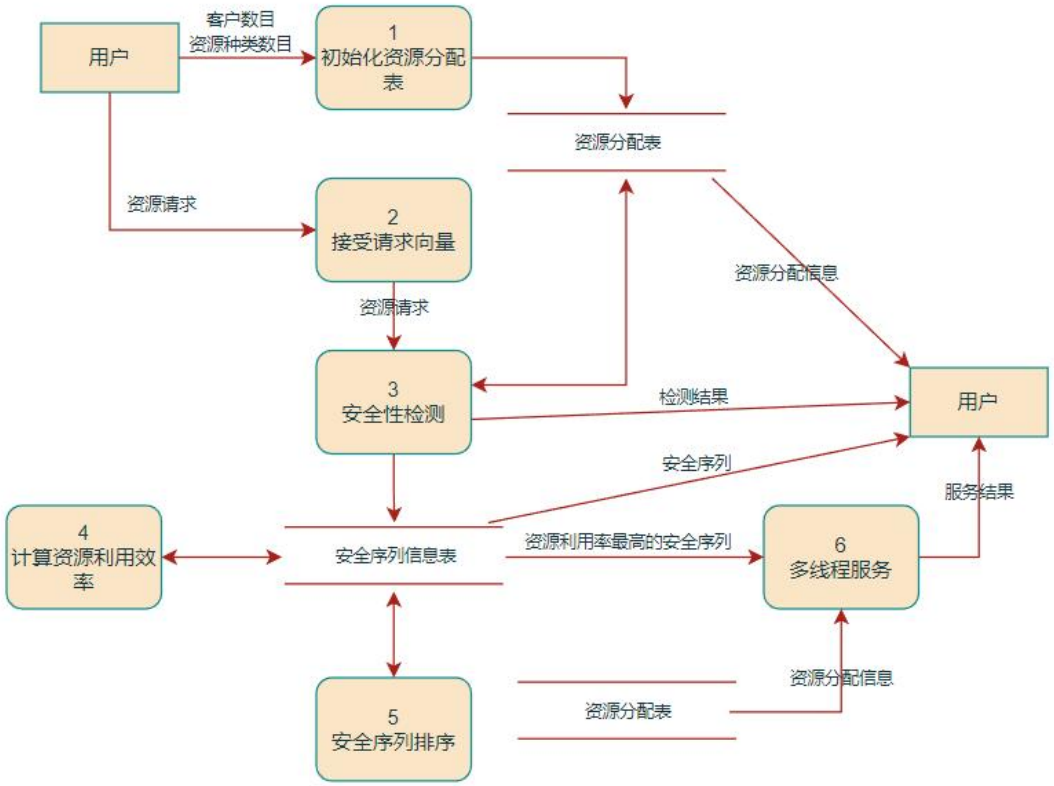


图 3 改进银行家算法的数据流图

1.3.2 行为模型——状态转换图

系统在运行时的状态转换如图 4，在用户输入客户数目与资源类别数目后，系统会自行开展初始化资源分配表并打印资源分配表。当进程开始申请资源，系统会首先对申请资源序列进行相关的可分配检测，判断当前可用资源序列是否满足进程需求以及进程申请的资源序列是否超出了自身运行需要。如果检测结果均为可以，则系统尝试对其进行资源分配，并做安全性检测。在安全性检测中，系统会利用深度搜索找寻所有安全序列。以上检测如果其中有一项失败都会直接打印错误结果并拒接此次分配。只有所有的检测通过的进程，系统才会为其分配资源。

通过安全性检测后，系统会对已经找寻到的安全序列计算其资源利用效率，之后按照资源利用效率对安全序列排序并打印。完成后，系统按照资源利用率最高的安全序列依次为各个进程创建安全线程，之后开展服务。在服务中不断打印服务状态，当服务完成后退出系统。

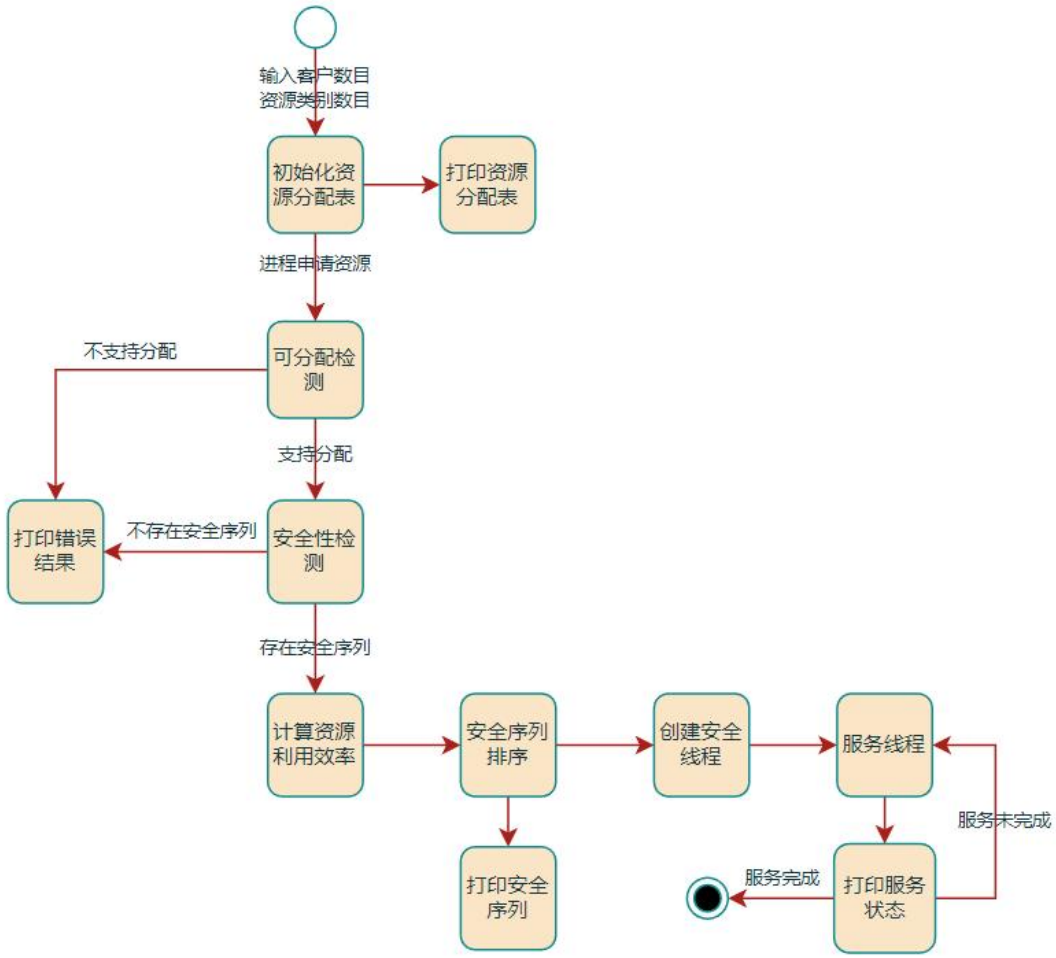


图 4 改进银行家算法的状态转换图

1.4 功能需求

1.4.1 功能编号和优先级

表 1 功能编号和优先级

功能编号	功能	优先级
1	系统初始化	中
2	可分配检测	中
3	安全性检测	高
4	计算资源利用效率	高
5	安全序列排序	中
6	创建安全线程	中
7	线程服务	高
8	内容显示	低

1.4.2 加工说明

表 2 各功能对应 IPO

功能	输入	处理	输出
系统初始化	客户数 n 和资源数 m	系统根据输入信息生成初始化数据	资源分配表
可分配检测	①资源请求序列 ②资源分配表	判断能否为进程分配资源	检测结果
安全性检测	①资源请求序列 ②资源分配表	尝试分配资源并进行安全性检测	①检测结果 ②安全序列
计算资源利用效率	安全序列	为每个安全序列计算资源利用率	含有资源利用率的安全序列
安全序列排序	含有资源利用率的安全序列	按照资源利用效率进行排序	排序后的安全序列
创建安全线程	资源利用率最高的安全序列	依照序列次序创建安全线程	安全线程
线程服务		开展线程服务	服务状态
内容显示	①资源分配表 ②检测结果 ③服务状态	打印内容	

1.4.3 HIPO 图

1.4.3.1 随机生成模块

编号	模块	调用	被调用	输入	输出	处理
1	随机生成模块	①资源上限生成 ②已分配资源生成 ③需求资源生成 ④资源占用时间生成			资源分配表	随机生成并初始化系统所需的各种数据。
1.1	资源上限生成		随机生成模块		每个资源的上限	随机生成每个资源的上限。
1.2	已分配资源生成		随机生成模块		已分配资源数目	随机生成已分配资源的初始值。

1.3	需求资源生成		随机生成模块		需求资源数目	随机生成需求资源的各个初始值。
1.4	资源占用时间生成		随机生成模块		每个客户占用每类资源的时间	随机生成每个客户占用每类资源的时间。

1.4.3.2 检测模块

编号	模块	调用	被调用	输入	输出	处理
2	检测模块	①可分配检测 ②安全性检测		资源分配表	①检测结果 ②排序后的安全序列	判断是否可以为进程分配资源并输出检测结果。如果可以分配则输出排序后的安全序列。
2.1	可分配检测		检测模块	资源分配表	检测结果	从进程的资源请求向量和可用资源序列上判断能否为该进程分配资源。
2.1	安全性检测	①深度搜索 ②安全序列排序	检测模块	资源分配表	①检测结果 ②安全序列	为进程尝试分配资源，并执行安全性检测，返回检测结果。
2.1.1	深度搜索		安全性检测	资源分配表	安全序列	按照资源分配情况和请求向量深度搜索所有安全序列
2.1.2	安全序列排序	计算资源利用效率	安全性检测	安全序列	排序后的安全序列	对安全性检测时所产生的安全序列，按照资源的利用效率进行排序。
2.1.2.1	计算资源利用效率		安全序列排序	安全序列	安全序列对应资源利用效率	按照安全序列指示进程顺序模拟运行，将模拟的时间作为其对应的资源利用效率输出。

1.4.3.3 多线程服务模块

编号	模块	调用	被调用	输入	输出	处理
3	多线程服务模块	①创建安全线程 ②线程服务		资源分配表	进程服务状态	对进程开展多线程服务。
3.1	创建安全线程		多线程服务模块	安全序列	多个安全线程	为进程创建多个安全线程。
3.2	线程服务		多线程服务模块	安全线程	进程服务状态	为线程提供服务。

1.5 性能需求

1.5.1 数据精度

表 2 系统各字段及精度

字段	精度
客户数目 n	int 型
资源数目 m	int 型

1.5.2 时间特性

（1）响应时间：用户在任意的操作后，系统能在 1s 内做出反应

（2）数据处理时间：视系统运行状态决定

1.5.3 灵活性

当客户和资源数目发生改变时，整体系统的结构、操作流程、运行环境等不会发生改变，只会修改系统运行中的数据。

1.6 运行需求

1.6.1 软件接口

（1）操作系统：Microsoft Windows 10、11

（2）软件设备：Visual Studio 2022

1.6.2 硬件接口

（1）内存：512M 以上

（2）磁盘空间：40G 以上

（3）CPU：333Mhz 以上

（4）硬盘空间：1.5G 以上

1.7 其他需求

1.7.1 质量属性

（1）可用性：用户与开发人员很容易上手使用，用户界面符合操作习惯

（2）可靠性：在给定时间内系统可以大致上满足无错运行的要求

（3）可维护性：系统运行会将行为写进日志以供维护工作的进行

（4）安全性：定时对关键数据、关键程序进行备份，防止数据丢失

（5）可移植性：移动端之间可以互相移植

2 系统设计

2.1 系统设计原则

2.1.1 先进性

从系统软硬件设备的设计出发，在数据存储、传输、系统服务等方面沿用高新技术潮流。让系统在满足以期需求的情况下，具有一定前瞻性，能够在今后较长时间持续保持技术的先进性。

2.1.2 安全性

系统对接入用户采取严格的审核方式，并定时对关键数据、关键程序进行备份，防止数据丢失，确保系统能够长时间正常运行。

2.1.3 可用性

系统基于用户习惯，设计清晰、友好的人机交互界面。不仅操作简单、快捷、灵活易用，在用户误操作或系统出错时还能够给予恰当的提示，方便用户使用以及管理人员维护。

2.1.4 可扩展性

系统在设计上具有良好的输入输出接口，同时在模块设计上考虑到了规模的可能变化。系统在交付后，可以实现持续的定制功能开发以及规模变更等扩展操作。

2.1.5 可维护性

系统采用模块化设计，基于高级程序语言 c++实现，并涵盖良好的用户界面设计以及完备的出错提醒与处理机制。系统具有易操作、易维护、方便实用以及自检和故障诊断的功能。

2.1.6 经济性

系统从简单、实用的角度出发，基于“合适最好”的理念，在软硬件构造的选取上，采用了经济实用的技术和设备。在满足期望需求的基础上，从器件、人力、物力；后期升级、维护等方面尽可能地降低了产品的建设成本。

2.2 总体架构

2.2.1 层次图

层次图中将系统分为四大模块，分别为随机生成模块、检测模块、多线程服务模块和内容显示模块。各模块功能如下：

（1）随机生成模块：根据用户输入的客户数目和资源类别数目随机生成初始化资源分配表。主要包括已分配资源、客户需求资源以及每个客户占用每类资源的时间等；

（2）检测模块：根据资源分配表和用户发出的资源请求判断系统是否可以为其分配资源。模块通过调用可分配检测和安全性检测实现，其中可分配检测判断依据为请求向量是否小于可用资源向量和需求向量；安全性检测需要利用深度搜索查询所有的安全序列，并调用安全序列排序模块对其排序，排序根据为各安全序列对应的资源利用效率；

（3）多线程服务模块：这一模块是整个系统的“灵魂”。主要根据资源利用率最高的安全序列依次为各个进程创建安全线程并开展服务。模块通过调用创建安全线程和线程服务实现，其中创建安全线程部分需要注意线程之间的同步关系，线程服务部分需要注意对临界区资源的互斥访问；

（4）内容显示模块：主要作用为在系统运行中将后台数据显示在界面上，方便用户查阅以了解系统运行状态。主要要求为准确、实时。当后台数据发生变化时，特别是线程服务过程中服务状态的变动，该模块应及时捕获到变化并对其显示。

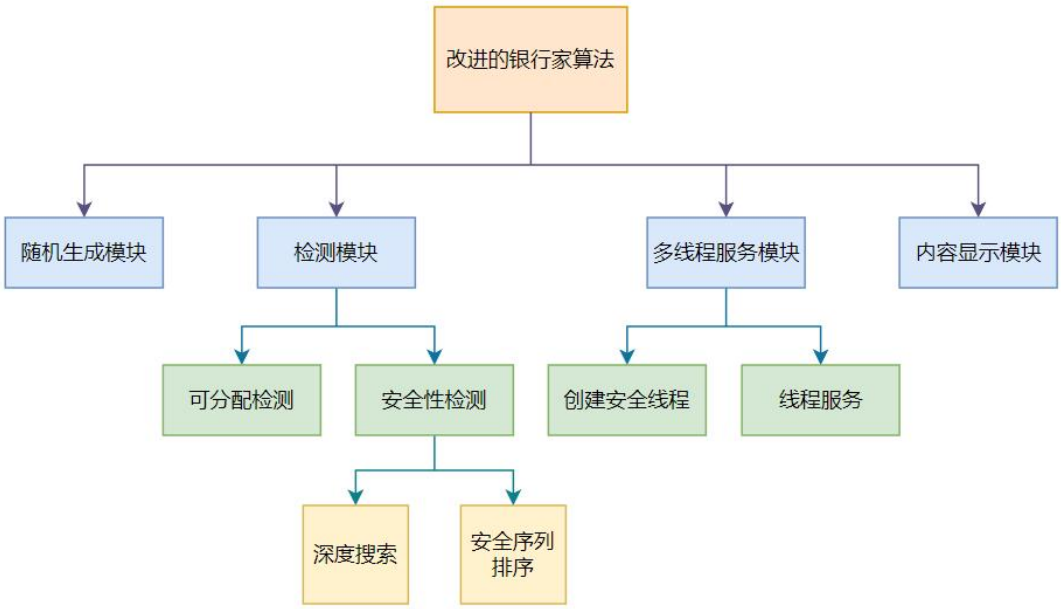


图 5 改进银行家算法的层次图

2.2.2 系统总体流程图

系统总体流程图通过控制流向直观地描述了系统的工作过程，包括系统的输入、处理和输出等。通过阅读总体流程图，开发人员可以对系统的结构有一个整体的把控，清楚各个处

理所处阶段以及不同分支与循环的导向，并更精确地掌握各个模块之间前后的调用次序，以更好地实现编码与后期维护工作。

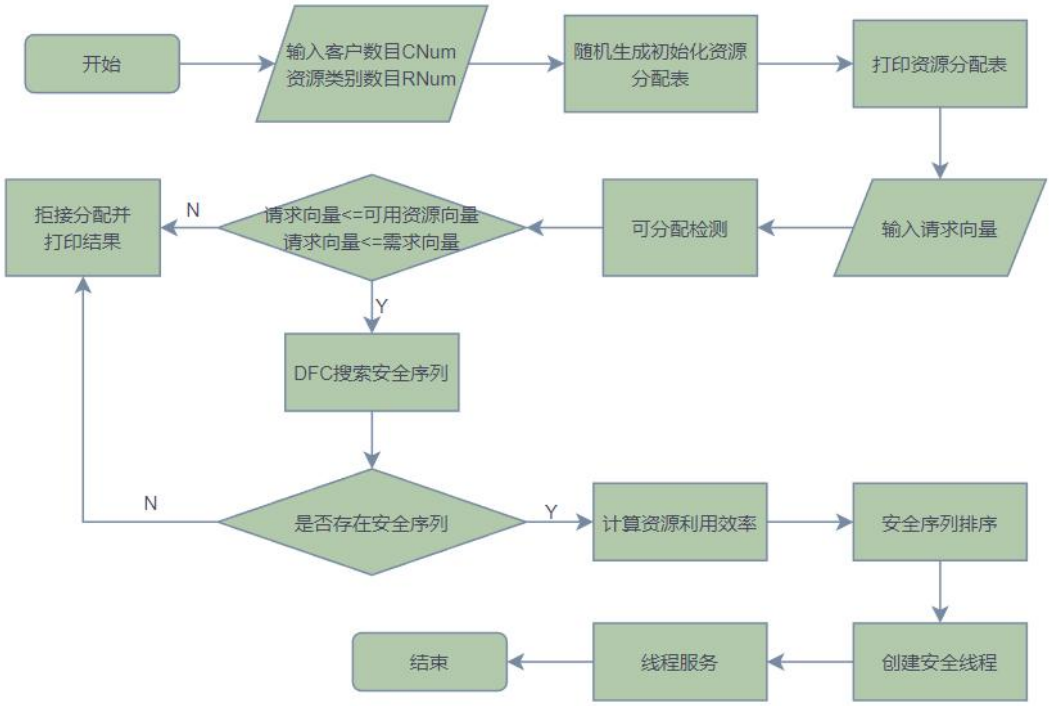


图 6 改进银行家算法的总体流程图

2.3 数据结构

```
//资源分配表(Resource allocation table)
typedef struct RAT {
    int CNum = 0;  // 客户数目
    vector<string> CName;  // 客户名称向量

    int RNum = 0;  // 资源类别数目
    vector<int> RUpper; //资源上限向量
    vector<string> RName;  // 资源名称向量

    vector<vector<int>> Allocation;  // 已分配资源矩阵
    vector<vector<int>> Need;  // 需求资源矩阵
    vector<vector<int>> Max;  // 最大需求矩阵
    vector<vector<int>> TOccupy; //资源占用时间矩阵

    vector<int> Available;  // 可利用资源数目向量
    vector<int> Request; // 资源请求向量

    vector<vector<int>> SafeList; // 安全序列
    vector<int> Mutex; // 互斥锁（保证进程按照顺序服务）

    vector<vector<int>> Serve_State; // 各进程已经服务的状态
    vector<bool> Finish; //各进程是否完成
    bool Allow_to_run = true; //线程是否被允许服务
}RAT;
```

（1） 资源分配表*(Resource allocation table)*

系统数据的存储位置，以结构体的方式存在，涵盖系统运行所需要的所有数据。包括客户数目、资源类别数目、已分配矩阵、需求矩阵等，为系统的正常运行提供最基本的保障。

（2）客户数目(*CNum*)

int 类型，存储客户数目。

（3）客户名称向量(*CName*)

vector<string>类型，长度为 CNum。存储客户的名称，名称设置为字母 C 开头加客户序号，序号从 0 开始，如 C0、C1……

（4）资源类别数目(*RNum*)

int 类型，存储资源类别数目。

（5）资源上限向量(*RUpper*)

vector<int>类型，长度为 RNum。依次存储不同资源对应的上限。

（6）资源名称向量(*RName*)

vector<string>类型，长度为 RNum。存储客户的名称，名称设置为字母 C 开头加客户序号，序号从 0 开始，如 C0、C1……

（7）已分配资源矩阵(*Allocation*)

vector<vector<int>>类型，大小为 CNum*RNum。Allocation[i][j]代表已经分配给第 i 个客户的第 j 类资源的数目。

（8）需求资源矩阵(*Need*)

vector<vector<int>>类型，大小为 CNum*RNum。Need[i][j]代表第 i 个客户还需要第 j 类资源的数目。

（9）最大需求矩阵(*Max*)

vector<vector<int>>类型，大小为 CNum*RNum。Max[i][j]代表第 i 个客户对第 j 类资源的最大需求。

（10）资源占用时间矩阵(*TOccupy*)

vector<vector<int>>类型，大小为 CNum*RNum。TOccupy[i][j]代表第 i 个客户占用第 j 类资源的时间。

（11）可利用资源数目向量(*Available*)

vector<int>类型，长度为 RNum。Available[i]代表第 i 类资源的剩余可分配数目。

（12）资源请求向量(*Request*)

vector<int>类型，长度为 RNum+1。Request[0]代表请求客户的所在序号，Request[i](i>0)

代表该客户对第 i-1 类资源的请求数目。

（13）安全序列(*SafeList*)

vector<vector<int>>类型，长度不定，维度为 CNum。SafeList[i]代表第 i 个安全序列，SafeList[i][j]代表第 i 个安全序列中第 i 个进程的进程号。

（14）互斥锁（保证进程按照顺序服务）(*Mutex*)

vector<int>类型，长度为 CNum+1，作用为保证进程按照顺序服务。当 Mutex[i]>=RNum 时，第 i 个进程才能开始服务。

（15）各进程已经服务的状态(*Serve_State*)

vector<vector<int>>类型，大小为 CNum*RNum。Serve_State[i][j]代表第 i 个客户中对第 j 类资源已经服务的时间。

（16）各进程是否完成(*Finish*)

vector<bool>类型，大小为 CNum。Finish[i]表示安全序列中第 i 个进程是否服务完成，true 代表服务完成，false 代表未服务完成。

（17）线程是否被允许服务(*Allow_to_run*)

bool 类型，代表当前系统是否可以服务线程，用于实现系统的暂停服务功能。当 Allow_to_run 为 true，系统可以服务；当为 false，系统不能服务。

```
mutex mutex2; // 临界区互斥
```

（18）临界区互斥(全局变量)(*mutex2*)

mutex 类型，用于实现系统中对临界区操作时不同线程之间的互斥。

2.4 模块分析与设计

2.4.1 随机生成模块

随机生成模块是程序运行的基础，用于生成系统所需的必要数据。这些数据在后期系统运行中需要不停周转流动，因此该模块应利用客户数目与资源类别数目的限制生成安全性良好的数据，比如已分配资源不能大于最大需求、需求资源应等于最大需求减去已分配资源等。

该模块实现方法如下：手动输入客户和资源类别的初始值，用随机数生成 *Allocation*、*Need*、*TOccupy*、*Available*，利用 *Allocation*+*Need* 计算最大需求 *Max*，利用 *Allocation*+*Available* 计算资源上限 *RUpper*。按照固定规则生成客户与资源名称并初始化 *Request*、*Serve_State* 为 0，*Finish* 为 *false*。

图 7 随机生成模块流程图

2.4.2 检测模块

检测模块是保证系统正常运行的关键，主要用于判断系统当前资源分配状态是否可以满足进程发出的资源请求，并对可满足的请求按照资源利用效率生成安全序列。对于此模块，如果不能对规范请求进行判断，系统在分配资源后势必会陷入意想不到的“危险”状态，并且极大可能发生死锁。该模块的实现与否直接关系到系统能否安全响应进程的请求。除此之外，模块生成的安全序列将被多线程服务模块所使用，若该模块未能很好完成，系统将不能继续后续操作。模块由可分配检测和安全性检测两大部分组成。

2.4.2.1 可分配检测

可分配检测和安全性检测共同组成检测模块。该模块是检测的第一步，主要用于判断进程发出的资源请求是否符合规格，以便从起点杜绝一切不规范的资源请求，是保证系统安全

分配资源的首要步骤。

该模块实现方法如下：判断系统资源请求向量是否满足 *Request<=Available* 且 *Request<=Need*。如果满足，则通过检测；如果不满足，则未通过检测。返回结果包含两个字符串，ans[0]保存检测结果，ans[1]保存显示字符串（用于界面显示）。

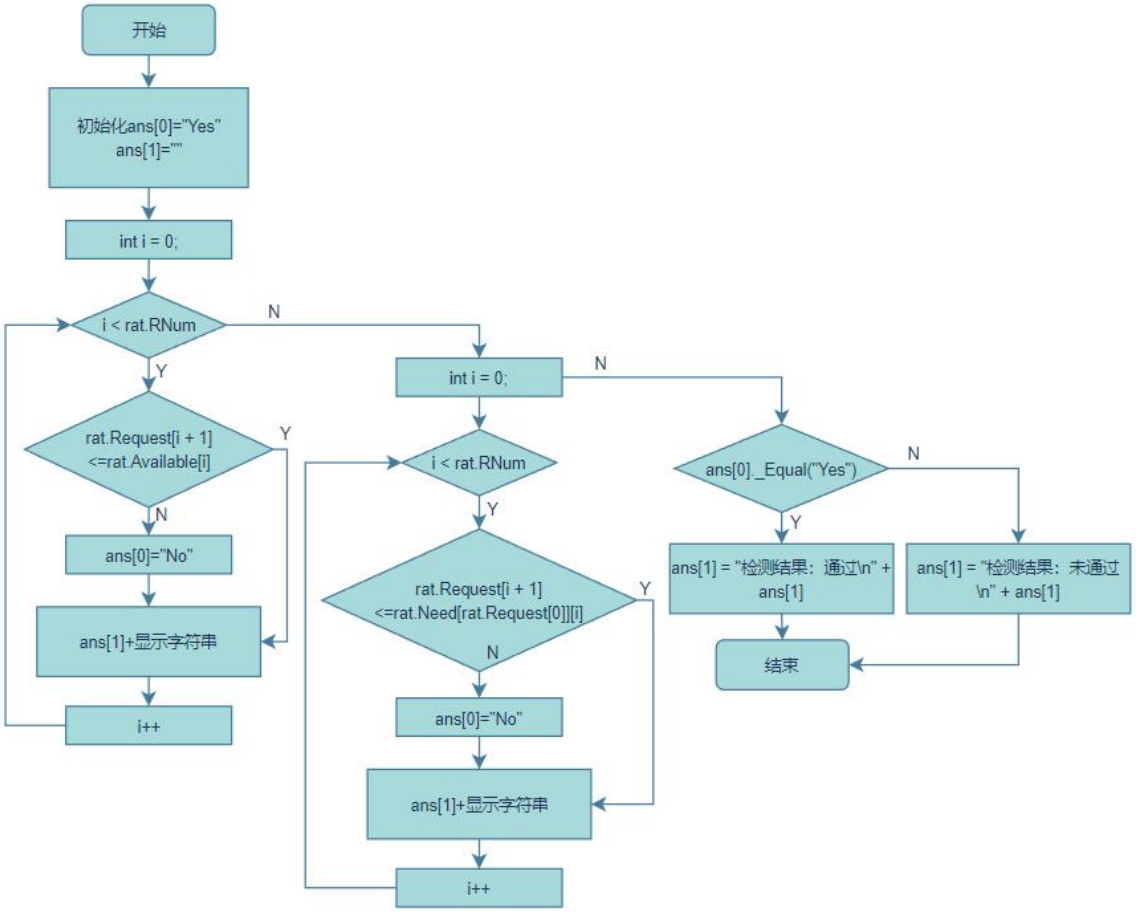


图 8 可分配检测模块流程图

2.4.2.2 安全性检测

安全性检测在检测模块中占据着核心的位置。可分配检测只能将明显不和规格的资源请求拒之门外，对那些看似可以分配但没有一个安全序列可以进行服务的请求不能加以判断，这很容易导致死锁的出现。安全性检测就是为此而设计的，通过尝试分配，安全性检测可以为当前资源请求查询安全序列。如果可以查询得到，即可说明该请求不会引发系统出现死锁，此时系统才会真正对其分配资源。此外，当系统判断存在安全序列，及时根据资源利用效率对其排序，方便在多线程服务模块系统可以更好地开展服务。

该模块实现方法如下：首先建立一个测试资源分配表 *rat_test*，在 *rat_test* 上进行资源的尝试分配，更新 *rat_test.Allocation* 、 *rat_test.Need* 和 *rat_test.Available* 为

rat_test.Allocation+rat_test.Request 、 *rat_test.Need-rat_test.Request* 和 *rat_test.Available-rat_test.Request*。之后利用深度搜索找寻所有安全序列，如果存在安全序列则认定此次分配安全，系统执行确认分配，更新 *rat.Allocation*、*rat.Need* 和 *rat.Available* 为 *rat.Allocation+rat.Request*、*rat.Need-rat.Request* 和 *rat.Available-rat.Request*。最后对所有安全序列按照资源利用效率排序后结束。如果没有找到安全序列，则判断此次分配不安全，系统不会对资源请求分配资源，直接结束。

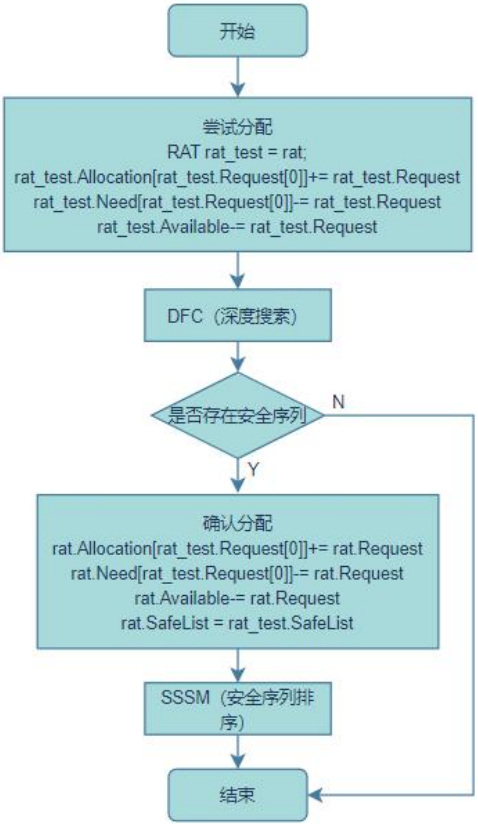


图 9 安全性检测模块流程图

2.4.2.2.1 DFC 深度搜索

该模块用于在安全性检测中找寻安全序列。是安全性检测模块的关键步骤，通过该算法，系统能够获取到当前资源请求下所有的安全序列，不仅是判断是否能够为其分配资源的准则之一，同时还为后期的多线程服务定下基础。

安全序列的判断如下：首先从进程队列中选取一个进程，查看当前资源分配表能够支持为其服务(*rat_test.Need<rat_test.Available&&Finish_test==false*)，如果可以，将此进程保存至安全序列，为其分配资源并找寻下一个进程，同时令其 *Finish_test=true*；如果不可以，继续判断下一个进程。重复上面过程，直到所有进程 *Finish_test==true*，即安全序列长度等于进程数目，此时判定该序列为一个安全序列，保存至 *rat_test.SaftList* 中。上述过程，如若判断

资源分配表不能为任何一个进程服务，直接退出搜索，认定没有安全序列。

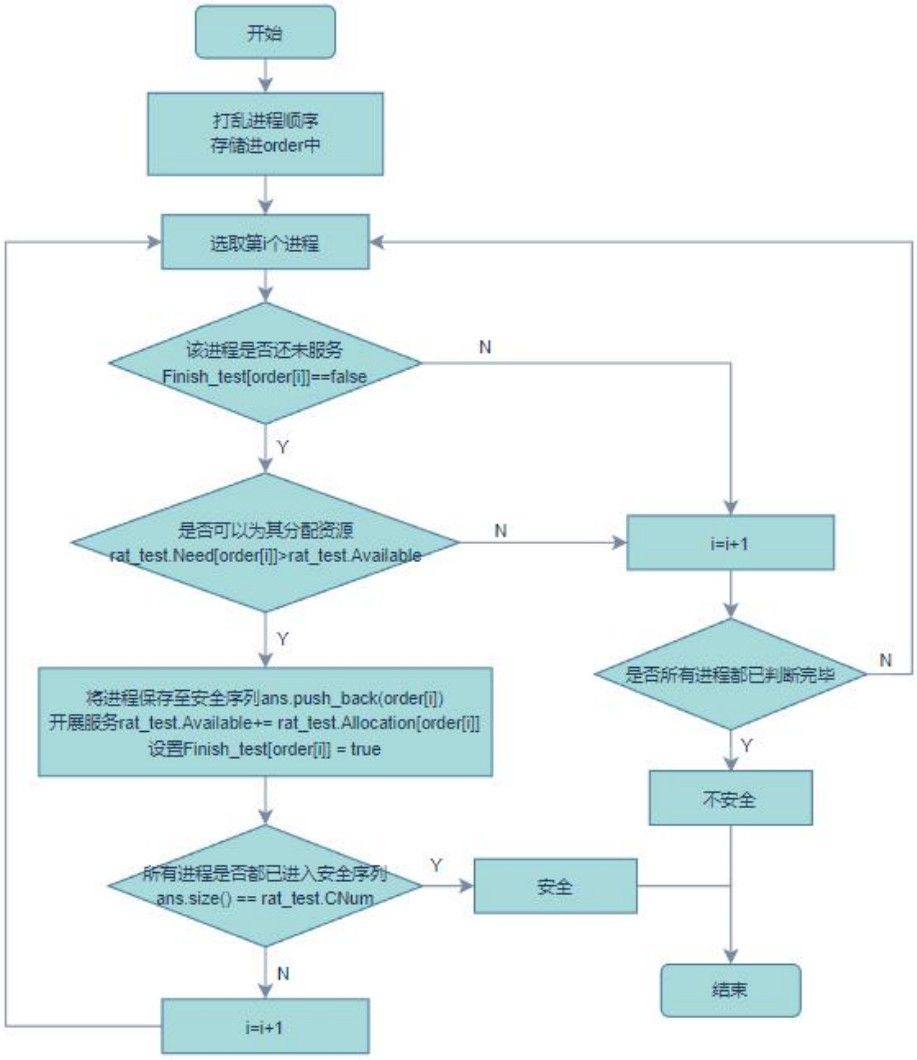


图 10 DFC 深度搜索流程图

图 9 为安全性算法流程图，深度搜索建立在此基础之上：当一个进程存入安全序列后，由该进程出发，遍历完旗下的所有进程。遍历完成后，继续对下一进程重复上述过程，直到所有进程排列组合都判断一遍。由于深度搜索的时间复杂度过高，为了系统能够及时响应用户，这里设置最多找寻 100 个安全序列。同时，为了增加随机性，每次遍历进程之前，都会对进程序列打乱，力求所有进程排列都有相同的可能参与判断。

2.4.2.2.2 安全序列排序

安全序列排序是提高系统运行效率的重要手段，在判断当前资源请求安全的情况下，按照不同安全序列开展服务往往对应着不同的运行效率。对于资源利用率高的安全序列，执行下来必然会减轻系统负担，同时提高系统的响应度。在安全序列排序中关键是不同安全序列的资源利用效率怎么计算。

在该系统中，资源利用效率以安全序列的模拟运行时间表示，时间越短，效率越高。计算方法如下：首先将第一个进程的所有使用资源保存，内容包括占用时间、资源类别以及占用数目。之后对这些资源占用块按照占用时间由大到小进行排序。然后由后向前弹出占用块，每次弹出便对下一进程是否可以服务进行判断，如果不能服务则继续弹出；如果可以服务则按照上述规则将其所有资源占用和原有占用块一起保存并排序，注意占用时间应当加上已经服务的时间。重复上述过程，直到所有进程都已进入服务，占用块队列中第一个资源所占用时间即是此安全序列模拟运行的时间。

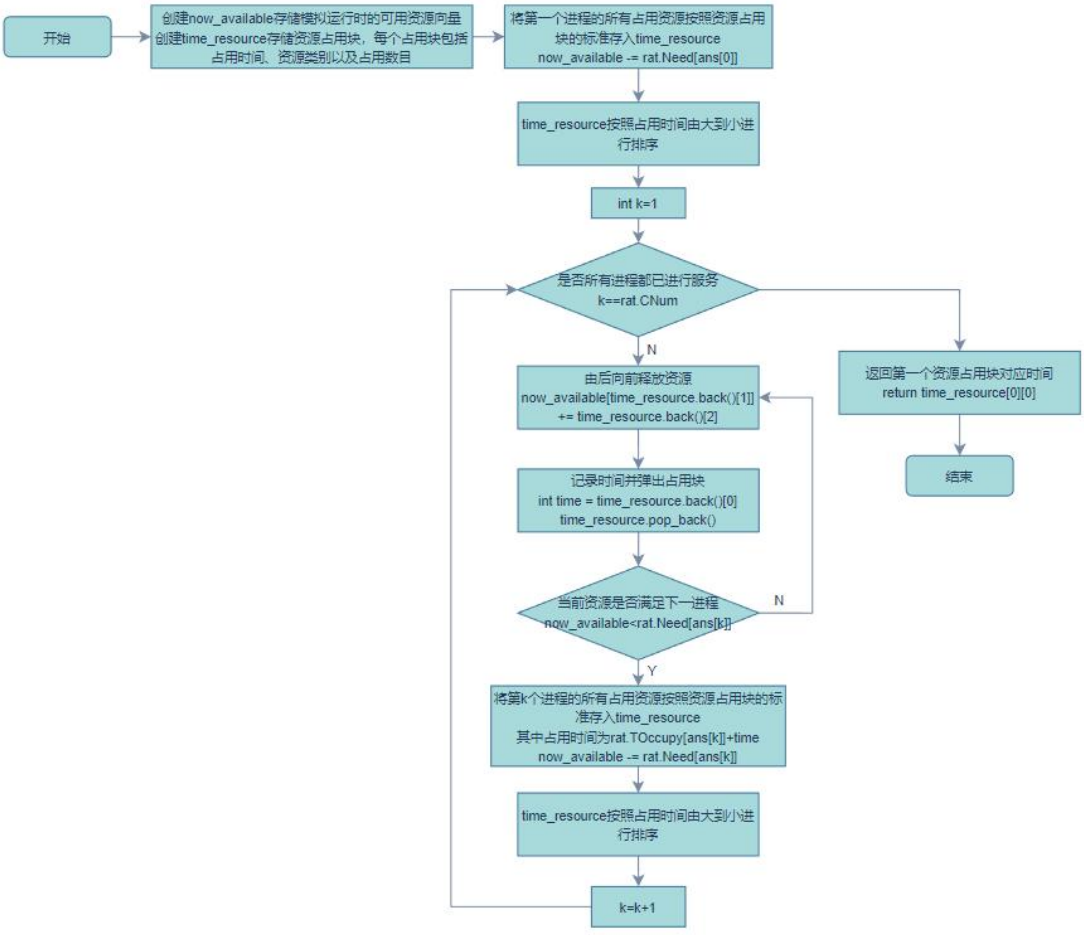


图 11 资源利用效率计算流程图

2.4.3 多线程服务模块

多线程服务模块是整个系统的灵魂所在。通过创建多个安全线程，系统可以对资源的利用率达到最大化，运行效率也会水涨船高。在多线程服务模块中，主要需要注意的是线程的同步创建时机与线程服务时对临界资源的互斥使用。如果没有协调好同步和互斥的关系，势必会导致系统运行发生故障，甚至产生结果与期望不符的问题。该模块由创建安全线程和线程服务共同组成。

2.4.3.1 创建安全线程

创建安全线程是多线程服务中的第一步，这一模块中，需要注意进程之间的同步关系。只有上一个进程已经开展服务后，才能为下一个进程创建对应的安全线程，否则进程之间的运行便与安全序列不符，很有可能产生死锁问题。

安全线程的创建方法如下：首先判断上一进程是否已经开展服务，注意第一个进程默认可以直接运行。如果未开始服务则循环等待，直到已经服务为止；如果已经开展服务，则继续判断当前资源是否满足服务需求，如果不满足则循环等待，直到满足为止；如果满足则为此进程创建安全线程。安全线程的创建按照资源划分，即为每一种不同的资源创建一个线程进行服务。

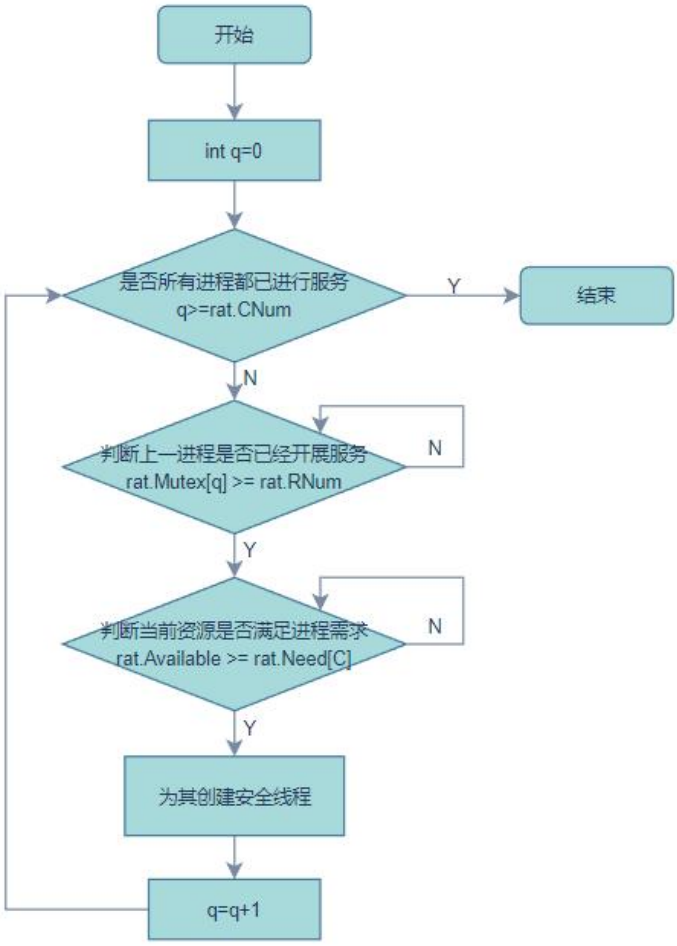


图 12 创建安全线程流程图

2.4.3.2 线程服务

线程服务是多线程服务中的核心，这一部分需要尤其关注临界数据的互斥操作。若不同线程对同一块资源进行修改，需要保证同一时刻仅能有一个线程对其访问。另外，这一模块同时影响着创建安全线程，只有上一个进程的所有线程进入服务状态，下一个

进程才能开始服务。

线程服务的实现方法如下：首先为该线程分配资源，之后标识此线程已经开始服务（当一个进程的所有线程进入服务，下一个进程才能开展服务）。后续进入服务，在服务的同时，不断修改服务状态（此操作方便显示）。服务完成后即刻回收资源。

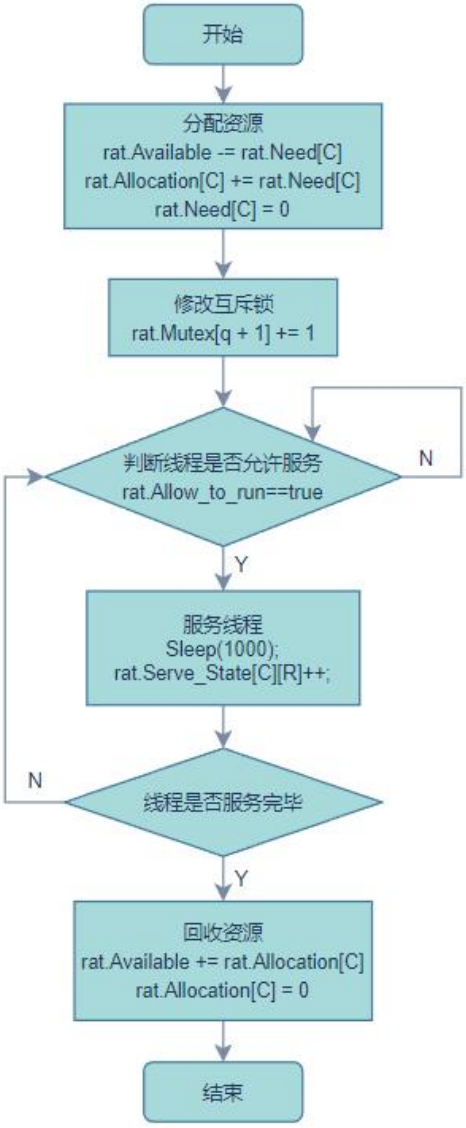


图 13 线程服务流程图

2.4.4 内容显示模块

内容显示是系统中贯穿始终的模块。通过调用该模块，系统可以将资源的分配状态、资源请求的检测结果、进程的服务状态等信息直接显示在界面上方便用户查阅。该模块的设计虽不会影响系统后台的运行，但直接关系到用户的使用体验，一个好的内容显示模块应当将信息准确、清晰、有条理地进行显示。

该模块的实现比较分散，当为每一种信息设计一显示函数，系统通过调用显示函数将信

息打印到屏幕之上。

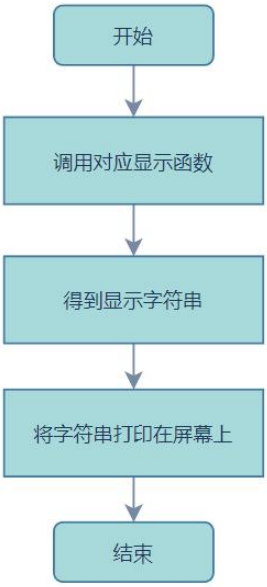


图 14 内容显示模块流程图

3 代码实现

3.1 方法类

```
class method {
public:
    method(int a = 0, int b = 0);
    ~method();

    void SetCNum(int); //设置客户数目
    void SetRNum(int); //设置资源类别数目
    void SetAllowToRun(bool); //设置是否允许服务
    void Clear(void); //清楚内容
    void SetRequest(int, int, int); //设置请求向量
    void RandomSetRequest(void); //随机设置请求向量

    void RGM(void); //随机生成模块(Randomly generated module)
    vector<string> ADM(void); //可分配检测模块(Assignable detection module)

    void SDM(void); //安全性检测模块(Security detection module)
    void DFS(RAT&, vector<int>, vector<bool>); //深搜找寻安全序列

    void SSSM(void); //安全序列排序模块(Security sequence sorting module)
    int CRUE(vector<int>); //计算资源利用效率(Computing resource utilization efficiency)

    void MSM(void); //多线程服务模块(Multithread service module)
    static void Serve(method *, int,int); //线程服务
    static void CreateThread(method*); //创造线程

    RAT Get_Rat(void); //获取资源分配表
    string Get_Need_Show(void); //显示需求矩阵
    string Get_TOccupy_Show(void); //显示占用时间

private:
    RAT rat;
};
```

系统基于面向对象的方式实现，所有模块的功能被封装进一个总体的方法类中。类中的成员变量为一个资源分配表，成员方法为系统操作时所需要的各种功能，其中已包含系统设

计中所提到的所有模块。系统运行时，通过创建类的实例并调用其中的成员方法来实现所需要求。

3.2 设置方法

3.2.1 设置客户数目 void SetCNum(int);

```
void method::SetCNum(int a)
{
    rat.CNum = a;
}
```

3.2.2 设置资源类别数目 void SetRNum(int);

```
void method::SetRNum(int b)
{
    rat.RNum = b;
}
```

3.2.3 设置是否允许服务 void SetAllowToRun(bool);

```
void method::SetAllowToRun(bool flag)
{
    rat.Allow_to_run = flag;
}
```

3.2.4 设置请求向量 void SetRequest(int, int, int);

```
void method::SetRequest(int a, int b, int c)
{
    rat.Request[0] = a;
    rat.Request[b + 1] = c;
}
```

该方法中，传递参数 a 为请求进程，b 为请求资源，c 为请求数目。通过直接设置，更新类中的资源分配表。

3.2.5 随机设置请求向量 void RandomSetRequest(void);

```
void method::RandomSetRequest(void)
{
    srand((unsigned)time(NULL));
    int a = 0;
    int b = rat.CNum - 1;
    rat.Request[0] = (rand() % (b - a + 1)) + a;
    for (int i = 1; i <= rat.RNum; i++) {
        a = 0;
        b = 7;
        if (rat.CNum >= 7) {
            b = 5;
        }
        rat.Request[i] = (rand() % (b - a + 1)) + a;
    }
}
```

该方法中，首先随机确定请求进程。之后判断总的客户数目，如果客户数目大于等于 7 个，则从 0~5 中为每类资源设置一请求；如果客户数目小于 7 个，则从 0~7 中为每类资源设置一请求。

3.2 随机生成模块

3.2.1 随机生成模块(Randomly generated module) void RGM(void);

```
// 随机生成模块(Randomly generated module)
void method::RGM(void)
{
    //客户名称生成
    for (int i = 0; i < rat.CNum; i++) {
        rat.CName.push_back("C" + to_string(i));
    }
    //资源名称生成
    for (int i = 0; i < rat.RNum; i++) {
        rat.RName.push_back("R" + to_string(i));
    }
    srand((unsigned)time(NULL));
    //随机数范围设计[a, b]
    int a = 10;
    int b = 20;
    //已分配资源生成[10, 20]
    for (int i = 0; i < rat.CNum; i++) {
        vector<int> q;
        for (int j = 0; j < rat.RNum; j++) {
            a = 10;
            b = 20;
            int k = (rand() % (b - a + 1)) + a;
            q.push_back(k);
        }
        rat.Allocation.push_back(q);
    }
    //需求资源生成[10, 20]
    for (int i = 0; i < rat.CNum; i++) {
        vector<int> q;
        for (int j = 0; j < rat.RNum; j++) {
            a = 10;
            b = 20;
            int k = (rand() % (b - a + 1)) + a;
            q.push_back(k);
        }
        rat.Need.push_back(q);
    }
    //最大需求矩阵生成
    for (int i = 0; i < rat.CNum; i++) {
        vector<int> q;
        for (int j = 0; j < rat.RNum; j++) {
            int k = rat.Allocation[i][j] + rat.Need[i][j];
            q.push_back(k);
        }
        rat.Max.push_back(q);
    }
```

该模块按照流程图指示，首先生成客户名称以及资源名称，均为字符加下标组成的字符串。之后，模块设置随机数的上下限，用于后续数据生成。首先是已分配资源，大小是 CNum*RNum。先创建一个空的容器，之后通过循环插入 RNum 个数目表示该进程已经分配资源的个数。再通过另一循环，为所有的进程创建好分配矩阵。需求矩阵的创建方式与上雷同，只需要将分配矩阵 *Allocation* 改成 *Need* 即可。

最大需求矩阵由 *Allocation* 和 *Need* 相加求出，通过双层循环实现。

```
    //可利用资源数目生成[15, 30]
    a = 15;
    b = 30;
    for (int i = 0; i < rat.RNum; i++) {
        int j = (rand() % (b - a + 1)) + a;
        rat.Available.push_back(j);
    }
    //资源上限生成
    for (int i = 0; i < rat.RNum; i++) {
        int sum = rat.Available[i];
        for (int j = 0; j < rat.CNum; j++) {
            sum = sum + rat.Allocation[j][i];
        }
        rat.RUpper.push_back(sum);
    }
    //资源占用时间生成[3, 10]
    for (int i = 0; i < rat.CNum; i++) {
        vector<int> q;
        for (int j = 0; j < rat.RNum; j++) {
            a = 3;
            b = 10;
            int k = (rand() % (b - a + 1)) + a;
            q.push_back(k);
        }
        rat.TOccupy.push_back(q);
    }
    //默认请求进程为C0，资源请求都为0
    for (int i = 0; i <= rat.RNum; i++) {
        rat.Request.push_back(0);
    }
    //服务状态为0（未开始服务）
    for (int i = 0; i < rat.CNum; i++) {
        vector<int> q;
        for (int j = 0; j < rat.RNum; j++) {
            q.push_back(0);
        }
        rat.Serve_State.push_back(q);
    }
    //默认所有进程都未完成
    for (int i = 0; i < rat.CNum; i++) {
        rat.Finish.push_back(false);
    }
}
```

后续生成可利用资源数目，此时上下限设置为[15,30]，利用单层循环加随机数即可实现。各资源上限的生成由 *Available* 和 *Allocation* 相加求出。资源占用时间利用双层循环，外层为各个进程，内层为各个资源，生成随机数上下限为[3,10]。最后，设置默认请求进程为 C0，各个资源请求为 0；每个进程的每个资源服务状态为 0；所有进程都未完成。

3.3 检测模块

3.3.1 可分配检测模块(Assignable detection module) vector<string> ADM(void);

可分配检测模块不仅涉及到对资源请求规格的判断，同时也包含了显示字符串。该模块返回内容为长度为 2 的字符串，第 0 位为检测结果，数值为 Yes 和 No。其中 Yes 代表检测通过，No 代表检测未通过。第 1 位为显示字符串，用于界面显示，包含检测结果的文字表述以及各个资源的比对情况。

```
// 可分配检测模块(Assignable detection module)
vector<string> method::ADM(void)
{
    vector<string> ans(2); //0:是否可以分配(Yes or No), 1:显示字符串
    ans[0] = "Yes";
    ans[1] = "Available:\t";
    for (int i = 0; i < rat.RNum; i++) {
        string str = "";
        if (rat.Request[i + 1] < rat.Available[i]) {
            str = "<";
        }
        else if (rat.Request[i + 1] == rat.Available[i]) {
            str = "=";
        }
        else {
            str = ">";
            ans[0] = "No";
        }
        ans[1] = ans[1] + "R" + to_string(i) + ":" + to_string(rat.Request[i + 1]) + str + to_string(rat.Available[i]) + "\t";
    }
```

方法中首先假设 ans[0]为 Yes，即检测通过。之后对请求向量 Request 和 Available 之间的大小进行判断。每次判断的结果依照字符串的形式填充进 ans[1]中，当判断某个资源的请求大于该资源的空闲数值时，ans[0]改为 No。ans[0]一旦变为 No，便不能再返回为 Yes。

```
    ans[1] += "\nNeed:\t";
    for (int i = 0; i < rat.RNum; i++) {
        string str = "";
        if (rat.Request[i + 1] < rat.Need[rat.Request[0]][i]) {
            str = "<";
        }
        else if (rat.Request[i + 1] == rat.Need[rat.Request[0]][i]) {
            str = "=";
        }
        else {
            str = ">";
            ans[0] = "No";
        }
        ans[1] = ans[1] + "R" + to_string(i) + ":" + to_string(rat.Request[i + 1]) + str + to_string(rat.Need[rat.Request[0]][i]) + "\t";
    }
    if (ans[0]._Equal("Yes")) {
        ans[1] = "检测结果: 通过\n" + ans[1];
    }
    else {
        ans[1] = "检测结果: 未通过\n" + ans[1];
    }
    return ans;
}
```

在对请求向量 Request 和 Available 判断完成后，算法会继续对 Need 进行判断。判断思路和上述相同，每次判断的结果会以字符串形式存入 ans[1]中。当某个资源的请求大于其所需要的资源时，ans[0]改为 No，表示检测不予通过。

在上述的两个判断完成之后，算法对显示字符串 ans[1]进行更新，主要是填充检测结果的文字表示。如果 ans[0]为 Yes，即检测通过，则填充“检测结果：通过”在字符串的头部；否则填充“检测结果：未通过”。

3.3.2 安全性检测模块(Security detection module) void SDM(void);

安全性检测的总体流程设计到清空过时缓存、尝试分配、找寻所有安全序列以及确认分配、安全序列排序五大步骤。其中找寻安全序列以及安全序列排序已经封装为 DFC 和 SSSM 两个成员方法，由于方法中修改的内容同为资源分配表，所以在算法实现过程中直接调用二者即可。

```
// 安全性检测模块(Security detection module)
void method::SDM(void)
{
    // 清空过时缓存
    rat.SafeList.clear();
    // 尝试分配
    RAT rat_test = rat;
    for (int i = 0; i < rat_test.RNum; i++) {
        rat_test.Allocation[rat_test.Request[0]][i] += rat_test.Request[i + 1];
        rat_test.Need[rat_test.Request[0]][i] -= rat_test.Request[i + 1];
        rat_test.Available[i] -= rat_test.Request[i + 1];
    }
    vector<bool> Finish_test(rat.CNum, false);
    vector<int> ans;
    // 找寻所有安全序列
    DFS(rat_test, ans, Finish_test);
    // 确认分配
    if (!rat_test.SafeList.empty()) {
        for (int i = 0; i < rat.RNum; i++) {
            rat.Allocation[rat.Request[0]][i] += rat.Request[i + 1];
            rat.Need[rat.Request[0]][i] -= rat.Request[i + 1];
            rat.Available[i] -= rat.Request[i + 1];
        }
        rat.SafeList = rat_test.SafeList;
    }
    // 安全序列排序
    SSSM();
}
```

首先清空过时缓存，这一部分对应与后续的 DFC 深度搜索。由于搜索过程中安全序列是利用 push_back 操作存入 SafeList 中，如果在进行安全性检测之前未清空缓存，势必导致安全序列存储过时数据，影响系统运行。

在清空缓存之后，建立测试资源分配表 rat_test，该测试表仅用于安全性检测，检测完成后自动销毁。对 rat_test 进行资源分配，具体为更改 Allocation、Need 和 Available。修改之后创建 Finish_test，用于记录每个进程是否完成服务。该变量同样仅用于测试，检测完成后自动销毁。

建立 ans 容器用于存储安全序列，之后调用 DFC 成员方法找寻所有安全序列。执行完成之后通过判断有无安全序列，即 rat_test.SafeList 是否为空来判断该请求是否通过了安全性检测。如果存在安全序列则确认分配，此时更新真实资源分配表 rat 中的 Allocation、Need 和 Available。更新完成之后调用成员方法 SSSM 来进行安全序列排序。

3.3.2.1 深搜找寻安全序列 void DFS(RAT&, vector<int>, vector<bool>);

该算法用于找寻所有安全序列，其中输入内容包括三项，第一项为资源分配表的引用，后续操作会对资源分配表进行直接修改，主要为导入安全序列；第二项为一个 int 型容器，用于在算法执行暂时存储各个安全序列；第三项为一个 bool 型容器，用于存储各个进程是否服务完成，服务完成为 true，未服务为 false。

```
void method::DFS(RAT& rat_test, vector<int> ans, vector<bool> Finish_test)
{
    if (rat_test.SafeList.size() == 100) { //最多存储100个安全序列
        return;
    }
    if (ans.size() == rat_test.CNum) { //找寻到一个安全序列
        rat_test.SafeList.push_back(ans);
        return;
    }
```

首先查看算法的退出条件。该算法有两个出口，一个是搜寻到 100 个安全序列之后，算法自动退出。如此设置是降低算法的时间复杂度，防止系统因过度搜寻而反应缓慢，产生不好的用户体验。第二个出口是对安全序列的判断，如果找寻到一个安全序列，则直接退出算法，原因是后续已经没有任何进程需要进行判断。

```
    vector<int> order; //打乱访问
    for (int i = 0; i < rat_test.CNum; i++) {
        order.push_back(i);
    }
    shuffle(order.begin(), order.end(), std::mt19937{ std::random_device{}() });
    for (int i = 0; i < rat_test.CNum; i++) {
        if (!Finish_test[order[i]]) { //当前进程未完成
            bool flag = true;
            for (int j = 0; j < rat_test.RNum; j++) { //判断是否可以对其服务
                if (rat_test.Need[order[i]][j] > rat_test.Available[j]) {
                    flag = false;
                    break;
                }
            }
            if (flag) {
                for (int j = 0; j < rat_test.RNum; j++) { //开展服务，更新可利用资源向量
                    rat_test.Available[j] += rat_test.Allocation[order[i]][j];
                }
                Finish_test[order[i]] = true; //更新进程状态
                ans.push_back(order[i]); //保存进程

                DFS(rat_test, ans, Finish_test); //深搜

                ans.pop_back(); //回退进程
                Finish_test[order[i]] = false; //回退进程状态
                for (int j = 0; j < rat_test.RNum; j++) { //回退所做操作
                    rat_test.Available[j] -= rat_test.Allocation[order[i]][j];
                }
            }
        }
    }
}
```

由于安全序列最多只搜寻 100 个，对于序列很多的资源请求，如果都从第一个进程开始找寻显然不太公平。为了消除这一点，算法中首先对进程队列进行了打乱，之后按照被打乱的队列搜寻，力求每种组合被找到的概率都是一样的。

在对进程队列打乱之后，算法会按照打乱的顺序依次扫描每个进程。在扫描的同时，算法会判断当前进程是否完成，判断标准是 Finish_test[order[i]]是否为 true。如果当前进程未完成，则继续下一步判断，判断当前的空闲资源是否能够满足进程需求，如果可以满足则继续进行下一步。

下一步为开展服务，根据该进程的已分配资源调整空闲资源向量，即从该进程将资源进行回收。在更新状态后，将进程的服务状态 Finish_test[order[i]]改为 true，防止重复访问。之后，将该进程保存进 ans 中，调用 DFC 继续寻找下一层。从下一层退出后，反向执行上述操作，意为该进程未开展服务。

3.3.2.2 **安全序列排序模块**(Security sequence sorting module) void SSSM(void);

```
// 安全序列排序模块(Security sequence sorting module)
bool compare_SSSM(vector<int> a, vector<int> b) {
    return a.back() < b.back();
}
void method::SSSM(void)
{
    // 计算资源利用效率
    for (int i = 0; i < rat.SafeList.size(); i++) {
        rat.SafeList[i].push_back(CRUE(rat.SafeList[i]));
    }
    // 按照资源利用效率进行排序
    sort(rat.SafeList.begin(), rat.SafeList.end(), compare_SSSM);
}
```

该模块算法结构较为简单，主要由两部分组成。第一部分是计算资源利用效率，这一部分是该算法的关键内容。在进行资源利用效率计算后，每个安全序列的模拟运行时间会被保存在该序列末尾，即每个序列实际长度为 CNum+1。之后利用 sort 函数和设计的比较函数即可实现排序功能。排序中，每一项比较的键为序列的模拟运行时间，即 a.back()和 b.back()。

3.3.2.3 **计算资源利用效率**(Computing resource utilization efficiency) int CRUE(vector<int>);

```
// 计算资源利用效率(Computing resource utilization efficiency)
bool compare_CRUE(vector<int> a, vector<int> b) {
    return a[0] > b[0];
}
int method::CRUE(vector<int> ans)
{
    vector<int> now_available(rat.Available);
    vector<vector<int>> time_resource; //0:占用时间, 1:占用资源, 2:占用数目
    for (int i = 0; i < rat.RNum; i++) {
        vector<int> c;
        c.push_back(rat.TOccupy[ans[0]][i]);
        c.push_back(i);
        c.push_back(rat.Allocation[ans[0]][i] + rat.Need[ans[0]][i]);
        now_available[i] -= rat.Need[ans[0]][i];
        time_resource.push_back(c);
    }
    // 由时间先后逐渐释放资源（占用时间越短，排序越靠后，从后向前释放）
    sort(time_resource.begin(), time_resource.end(), compare_CRUE);
```

该模块是安全序列排序中的关键，其实现与否直接影响到安全序列能够有效地开展排序。在这一部分中，算法首先创建两个容器 now_available 和 time_resource。前者用于记录每个变化时刻空闲的资源向量，后者用于保存各个资源占用块。资源占用块是进程对每个资源占

用的描述，由一个 3 维 int 向量组成，第 0 位保存占用时间，第 1 位保存占用的资源，第 2 位保存该资源的占用数目。

首先将第一个进程的各个资源以资源占用块的形式存入 time_resource，之后对其按照资源的占用时间由大到小排序，排序方式和上述提到的安全序列排序方法同理：利用 sort 加设计比较函数。

```
    int k = 1;
    if (k == rat.CNum) {
        return time_resource[0][0];
    }
    while (true) {
        // 释放资源
        now_available[time_resource.back()[1]] += time_resource.back()[2];
        // 记录时间
        int time = time_resource.back()[0];
        // 弹出占用块
        time_resource.pop_back();
        // 判断是否满足下一进程
        bool flag = true;
        for (int i = 0; i < rat.RNum; i++) {
            if (now_available[i] < rat.Need[ans[k]][i]) {
                flag = false;
                break;
            }
        }
        if (flag) {
            // 添加占用块
            for (int i = 0; i < rat.RNum; i++) {
                vector<int> c;
                c.push_back(rat.TOccupy[ans[k]][i] + time);
                c.push_back(i);
                c.push_back(rat.Allocation[ans[k]][i] + rat.Need[ans[k]][i]);
                now_available[i] -= rat.Need[ans[k]][i];
                time_resource.push_back(c);
            }
            // 排序
            sort(time_resource.begin(), time_resource.end(), compare_CRUE);
            k = k + 1;
            // 所有进程全都进入服务后，返回最长时间
            if (k == rat.CNum) {
                return time_resource[0][0];
            }
        }
    }
}
```

在对第一个进程导入占用块后，正式开始模拟运行。如果只有一个进程，则直接返回首位占用块对应时间即可。如果不只一个进程，则不断由后向前弹出占用块，直到空闲资源满足下一进程服务。此时将下一进程的资源同样按照占用块的标准进行存储，注意此时应该在占用时间部分加上刚刚弹出的占用块所存时间。排序，记录已经开始服务的进程个数，当所

有进程都已经开始服务时，首位占用块所对应时间即是这一安全序列的模拟运行时间。

3.4 多线程服务模块(Multithread service module) void MSM(void);

```
// 多线程服务模块(Multithread service module)
void method::MSM(void)
{
    //互斥锁（用于操作同步）
    for (int i = 0; i <= rat.CNum; i++) {
        if (i == 0) {
            rat.Mutex.push_back(rat.RNum); // 所有资源已就绪
        }
        else {
            rat.Mutex.push_back(0);
        }
    }
    thread t(CreateThread, this);
    t.detach();
}
```

在多线程服务模块，首先初始化互斥锁。该锁用于操作同步，具体是令进程按照安全序列开展服务，即上一个进程未开展服务时，下一个进程不能服务。第 q 个进程能否服务的判断条件是 rat.Mutex[q] >= rat.RNum（含义是上一个进程的所有线程都已经开始服务），由于第一个进程肯定是可以开始服务的，所以令 rat.Mutex[0]=rat.RNum，后续的所有数值都设置为 0。注意互斥锁的长度要比 CNum 多一位，这是因为当服务至最后一个进程时，同样会执行 rat.Mutex[q+1]++的操作，如果长度和 CNum 一样，便会发生越界。

完成互斥锁的同步后，进入创建安全线程的步骤，该步骤利用一个线程执行。

3.4.1 创建安全线程 static void Serve(method *, int,int);

```
void method::CreateThread(method* W) {
    //创建线程
    for (int q = 0; q < W->rat.CNum; q++) {
        //判断上一进程是否已经开始服务
        while (W->rat.Mutex[q] < W->rat.RNum);
        //判断当前资源是否满足需求
        int C = W->rat.SafeList[0][q];
        while (true) {
            bool flag = true;
            for (int R = 0; R < W->rat.RNum; R++) {
                if (W->rat.Available[R] < W->rat.Need[C][R]) {
                    flag = false;
                    break;
                }
            }
            if (flag)
                break;
        }
        for (int R = 0; R < W->rat.RNum; R++) {
            thread t1(W->Serve, W, q, R);
            t1.detach();
        }
    }
}
```

创建安全线程部分由于需要对方法类的传参，意味着在类的成员方法中传递类，想要实现这个效果，只有将被调类的指针传递进入。另外，该方法需要声明为静态函数，这是因为线程的创建需要在编译时确定地址，而普通的类函数，编译时不能确定地址，需要创建类的对象才能获取，声明成 static 函数后，地址在编译时即可确定。

解决好函数的创建和传参后，创建安全线程即可以实现了。首先对每一个进程进行遍历，按照遍历的顺序创建线程。由于线程一经创建，其运行顺序不受主线程的控制，为了避免第二个进程的线程先于前一个进程的线程开展服务，即抢占前一个进程的所需资源，这里必须保证上一个进程的所有线程进入服务后，下一个进程的线程才能创建并服务。为了实现这一点，就需要刚开始时所初始化的互斥锁 rat.Mutex，其数目代表着上一进程已经开展服务的线程个数，只有 rat.Mutex[q] >= rat.RNum，这一进程才能开展服务。

当进程有了这一服务条件，还不能立即创建，需要再对当前空闲资源的状态进行判断。只有 Available > Need 时，即资源能够支持服务时，才能为进程创建安全线程，开始服务。线程的创建采用了资源分类的方法，即为每一个进程的每一种资源创建一个线程。

线程创建完成后立即调用线程服务函数，开展服务。

3.4.2 线程服务 static void Serve(method *, int,int);

```
mutex mutex2; // 临界区互斥
void method::Serve(method* W, int q/*进程在安全序列中的下标*/, int R)
{
    //获取进程号
    int C = W->rat.SafeList[0][q];
    //分配资源
    W->rat.Available[R] -= W->rat.Need[C][R];
    W->rat.Allocation[C][R] += W->rat.Need[C][R];
    W->rat.Need[C][R] = 0;
    //资源已经获取完成
    mutex2.lock();
    W->rat.Mutex[q + 1] += 1;
    mutex2.unlock();
    //服务当前线程
    for (int i = 0; i < W->rat.TOccupy[C][R]; i++) {
        while (!W->rat.Allow_to_run);
        Sleep(1000);
        W->rat.Serve_State[C][R]++;
    }
    //回收资源
    W->rat.Available[R] += W->rat.Allocation[C][R];
    W->rat.Allocation[C][R] = 0;
}
```

线程服务函数中需要传递有三个参数，第一个与创建安全线程一样，都是类的指针，实现方法同理；第二个和第三个分别为当前进程在安全序列中的下标和此进程的第几类资源。

算法实现中，首先根据进程在安全序列中的下标获取到该进程号。之后为此进程的分配该种资源，由于对一类资源同一时刻只会有一个线程需要分配资源（这一部分在线程创建中已经调节好，即上一进程的所有线程未开始服务时，下一进程不能开始服务），所有不需要互斥操作。在资源获取完成后，互斥锁 Mutex+1，表明已经有一类资源开始服务。在这一步中，由于同一进程的各个线程可能同时对 Mutex 进行修改，所以对此操作需要互斥，不可以让不同线程同时操作，利用临界区互斥即可完成。

在分配好资源且更新好互斥锁后，即可为该线程开始服务。服务过程中利用 Sleep(1000) 模拟服务一秒，总共需要服务时长等于该进程对该资源的占用时间。服务时同步更新已经服务状态，方便在系统界面上进行显示。

服务完成后，立即回收资源。与分配资源同样，由于对一类资源同一时刻只会有一个线程需要回收资源，所以不需要使用互斥。资源回收后，该线程服务完毕。

3.4 内容显示模块

该模块没有一个完整的封装函数，每种需要显示的信息会单独设计一个显示函数。返回内容均为字符串，如下：

```
string Get_Serve_State_Show(void); //显示服务状态
string Get_SafeList_Show(void); //显示安全序列
string Get_Request_Show(void); //显示请求向量
string Get_Available_Show(void); //显示可利用资源向量
string Get_Allocated_Show(void); //显示分配矩阵
string Get_Need_Show(void); //显示需求矩阵
string Get_TOccupy_Show(void); //显示占用时间
```

3.4.1 显示服务状态 string Get_Serve_State_Show(void);

```
string method::Get_Serve_State_Show(void)
{
    string ans = "";
    for (int i = 0; i < rat.CNum; i++) {
        int c = rat.SafeList[0][i];
        int sum1 = 0;
        int sum2 = 0;
        ans = ans + rat.CName[c] + ":\t";
        for (int j = 0; j < rat.RNum; j++) {
            ans = ans + "R" + to_string(j) + ":" + to_string(rat.TOccupy[c][j]) + "-" + to_string(rat.Serve_State[c][j]) + "\t";
            sum1 += rat.TOccupy[c][j];
            sum2 += rat.Serve_State[c][j];
        }
        float pro = sum2 * 100.0 / sum1;
        string str = to_string(round(pro * 100) / 100);
        ans = ans + str.substr(0, str.find(".") + 3) + "%\n\n";
        if (sum2 == sum1) {
            rat.Finish[i] = true;
        }
    }
    return ans;
}
```

通过遍历每一个进程的每一种资源的服务状态来实现，显示格式为“客户名称:\t 资源名称:占用时间-已经服务的时间\t……服务进度（百分比）”。在对服务进度进行计算时，会保存小数点后两位，所有内容均转化为字符串输出。

3.4.2 显示安全序列 string Get_SafeList_Show(void);

```
string method::Get_SafeList_Show(void)
{
    string ans;
    ans = "";
    for (int i = 0; i < rat.SafeList.size(); i++) {
        ans = ans + to_string(i + 1) + ":\t";
        int j = 0;
        for (; j < rat.SafeList[i].size() - 1; j++) {
            ans = ans + "C" + to_string(rat.SafeList[i][j]) + "->";
        }
        ans = ans + "Time: " + to_string(rat.SafeList[i][j]) + "\n";
    }
    if (!rat.SafeList.empty()) {
        ans = "检测结果: 通过\n" + ans;
    }
    else {
        ans = "检测结果: 未通过\n" + ans;
    }
    return ans;
}
```

安全序列的显示通过遍历所有安全序列实现。每一个安全序列的显示格式为“序号:\t 进程名称->进程名称->……Time:模拟运行时间”。在将每一个安全序列扫描一遍后，会将最终安全性检测的结过放在最前面，如果安全序列非空（检测通过），即加上“检测结果：通过”，否则加上“检测结果：未通过”。

3.4.3 显示请求向量 string Get_Request_Show(void);

```
string method::Get_Request_Show(void)
{
    string ans = "";
    for (int i = 0; i <= rat.RNum; i++) {
        if (i == 0) {
            ans = ans + "C" + to_string(rat.Request[i]) + ":\t";
        }
        else {
            ans = ans + rat.RName[i - 1] + "-" + to_string(rat.Request[i]) + "\t";
        }
    }
    return ans;
}
```

请求向量中第一位数据为进程号，后续为各个资源的请求数值。显示格式为“进程名称：\t 资源名称-请求数值\t……”。

3.4.4 显示可利用资源向量 string Get_Available_Show(void);

```
string method::Get_Available_Show(void)
{
    string ans = "";
    for (int i = 0; i < rat.RNum; i++) {
        ans = ans + rat.RName[i] + "-" + to_string(rat.Available[i]) + "\t";
    }
    return ans;
}
```

Available 的显示利用遍历实现，显示格式为“资源名称-空闲资源数目\t……”。

3.4.5 **显示分配矩阵** string Get_Allocated_Show(void);

```
string method::Get_Allocated_Show(void)
{
    string ans = "";
    for (int i = 0; i < rat.CNum; i++) {
        ans = ans + rat.CName[i] + ":\t";
        for (int j = 0; j < rat.RNum; j++) {
            ans = ans + rat.RName[j] + "-" + to_string(rat.Allocation[i][j]) + "\t";
        }
        ans += "\n";
    }
    return ans;
}
```

分配矩阵的显示利用双层循环实现，通过对每个进程的每类资源访问汇总成字符串，显示格式为“进程名:\t 资源名称-分配资源数目\t……\n 进程名:\t 资源名称-分配资源数目……”。

3.4.6 **显示需求矩阵** string Get_Need_Show(void);

```
string method::Get_Need_Show(void)
{
    string ans = "";
    for (int i = 0; i < rat.CNum; i++) {
        ans = ans + rat.CName[i] + ":\t";
        for (int j = 0; j < rat.RNum; j++) {
            ans = ans + rat.RName[j] + "-" + to_string(rat.Need[i][j]) + "\t";
        }
        ans += "\n";
    }
    return ans;
}
```

需求矩阵的显示和分配矩阵同理，都是利用双层循环实现，显示格式为“进程名:\t 资源名称-分配资源数目\t……\n 进程名:\t 资源名称-需求资源数目……”。

3.4.7 **显示占用时间** string Get_TOccupy_Show(void);

```
string method::Get_TOccupy_Show(void)
{
    string ans = "";
    for (int i = 0; i < rat.CNum; i++) {
        ans = ans + rat.CName[i] + ":\t";
        for (int j = 0; j < rat.RNum; j++) {
            ans = ans + rat.RName[j] + "-" + to_string(rat.TOccupy[i][j]) + "\t";
        }
        ans += "\n";
    }
    return ans;
}
```

同分配矩阵与需求矩阵的显示方式，占用时间的显示同样依靠双层循环。显示格式为“进程名:\t 资源名称-分配资源数目\t……\n 进程名:\t 资源名称-资源占用时间……”。

3.5 **其他方法**

3.5.1 **获取资源分配表** RAT Get_Rat(void);

```
RAT method::Get_Rat(void)
{
    return rat;
}
```

3.5.2 清除内容 void Clear(void);

```
void method::Clear(void)
{
    rat.CNum = 0;
    rat.CName.clear();
    rat.RNum = 0;
    rat.RName.clear();
    rat.Allocation.clear();
    rat.Need.clear();
    rat.Max.clear();
    rat.Available.clear();
    rat.Request.clear();
    rat.RUpper.clear();
    rat.TOccupy.clear();
    rat.SafeList.clear();
    rat.Mutex.clear();
    rat.Serve_State.clear();
    rat.Finish.clear();
    rat.Allow_to_run = true;
}
```

4 软件介绍

4.1 界面介绍

整个系统由四大界面组成，分别为 *Improved_banker_algorithm*、*MainWindow1*、*MainWindow2*、*MainWindow3*。

4.1.1 界面一 *Improved_banker_algorithm*

该界面主要用于为用户介绍此系统的功能与用法，在此界面中，用户仅能输入客户数目与资源类别数目。并且输入数据必须在控制范围内，其他数据系统不会读入，这保证了系统的数据安全。

界面中，点击退出，则直接退出系统；点击确认，进入下一界面。

4.1.2 界面二 *MainWindow1*

在界面一中点击确认后进入此界面。系统会自行调用随机生成模块生成数据，并调用显示模块对资源请求向量、可利用资源向量、已分配资源、需求资源、资源占用时间等进行显示。

在这一界面中，用户可以为进程设置资源请求向量。设置方法为调整好请求进程与请求资源和数目后，点击确认。每次只可设置一个进程对一个资源的请求，如果资源较多，设置繁琐，可以直接点击左上角随机按钮，系统会随机生成一个请求向量，用户可以在该向量上继续做出相应修改。

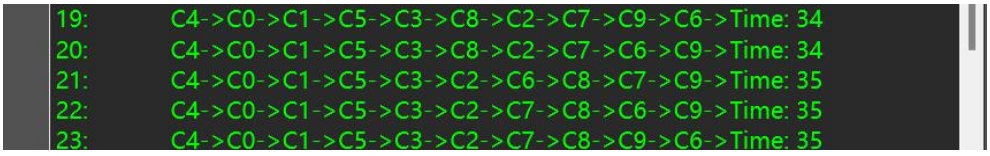
以上为点击随机后，系统自行生成的请求向量。

如果用户对先前设置的客户数目与资源类别数目不够满意，可以点击返回，系统会返回

界面一，此时用户可以重新设置各个数目。如果用户对当前资源分配表与资源请求向量满意，可以点击测试，系统会进入下一界面，开始服务前的测试。

4.1.3 界面三 *MainWindow2*

在界面二中点击检测进入该界面。系统会根据已经确认的资源分配表调用检测模块，执行可分配检测和安全性检测，并利用内容显示模块将检测的结果显示在界面上。安全性检测中，安全序列的显示是已经排好序后的安全序列，如下图：



模拟运行时间短的在前，时间长的在后，如果此时点击服务，系统会依照第一个安全序列开展进程的服务。

以下是此界面的其余几种状态，主要包括未通过可分配检测的和能够通过可分配检测但未通过安全性检测的。注意当请求未通过可分配检测，系统会认定此分配不安全，不会对请求做出安全性检测的判断。

服务按钮只有在两种测试都通过的基础上才可以点击，如果界面中有一种测试未通过，点击都无效，此时用户可以通过按下返回按钮返回到界面二重新设置请求向量。

4.1.4 界面四 *MainWindow3*

此界面是系统的最后一个界面，一旦进入服务界面，便代表系统已经做好了为进程开展服务的准备，此时不能再返回之前的几个界面。

点击开始后，系统会为进程开展服务：

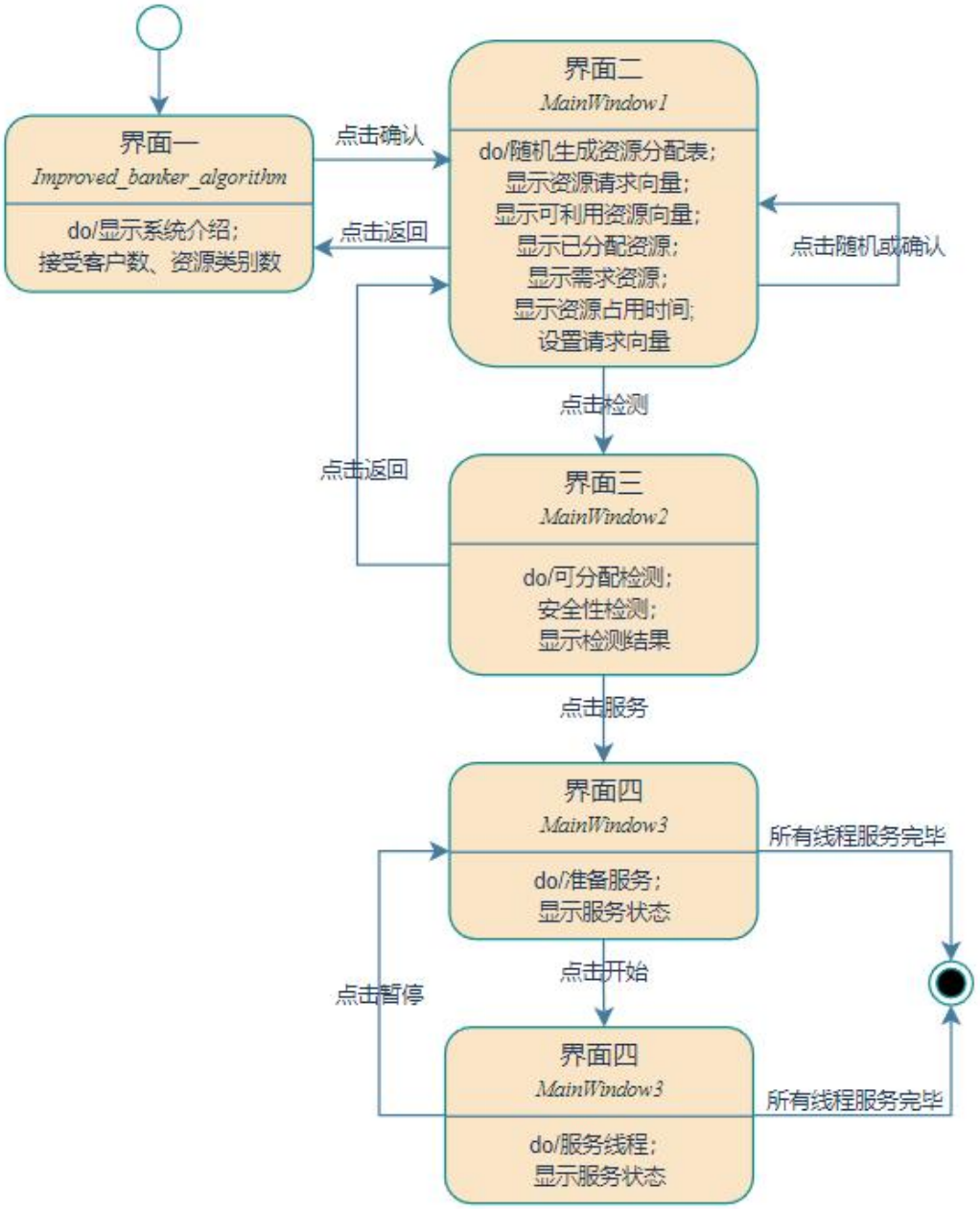
服务过程中，系统会每隔一秒刷新一次页面，更新当前的进程服务状态，并在左上角显示系统已经运行的时间。上图中可以看出，第一个进程的各个线程已经均被服务了 4 秒，整个进程已经被服务了百分之 53.33。

服务过程中可以按下暂停，此时系统会停止服务，但是系统的运行时间仍会上涨。上图是按下暂停后的界面，此界面不仅展示了暂停的效果，同时也是多线程服务的最好体现。可以看出，进程 6、7、5 都已经服务完毕，而剩余资源已经可以同时满足进程 2、3 的服务，二者同时服务，故服务时间相同，均为 6 秒。服务中，可以看到，进程 2 的 R4、R6 和进程 3 的 R3、R5 两个线程都已经服务完毕，由于每当线程服务完成时资源会自行回收，所以 C4 和 C1 也可以开展服务。在该系统中，不仅线程之间可以并行，当资源充足，进程之间也可以并行。这极大地提高了运行效率。

当所有进程服务完成后，会显示如上状态，可以看到，所有进程的服务都已经达到了百分之百。此时系统停止服务，运行时间不再上涨。开始暂停按钮虽然可以点击，但是不再发挥作用。

4.2 软件使用流程

软件的总体使用流程如下，前三个界面可以互相跳转，以供用户调整到自己满意的资源分配表、资源请求向量和安全序列等内容。在进入服务之前，用户可以不断更改内容，直到达到自身需求为止。当用户确认服务，即进入第四个界面之后，便不能再对数据进行调整。注意进入界面四的前提是资源请求向量通过可分配检测和安全性检测，即资源分配不会引发死锁等一系列安全问题。界面四是和界面一、二、三独立的一个界面。进入服务状态后，用户可以通过点击开始和暂停控制系统服务，不过系统运行时间不会被控制，即使暂停服务，总体的时间也会不断上升。当系统服务完成后，所有信息不再变化，系统停止，用户可以查看服务状态，或者直接退出系统。



4.3 界面关键代码

4.3.1 界面间传参

由于整个系统背后的数据由资源分配表承载，当两个界面之前切换时，必然会牵扯到数据的传递。这里我采用的是父界面发送信号配合子界面接受信号实现，字界面中为槽函数。这里由界面二[*MainWindow1*]切换到界面三[*MainWindow2*]为例，实现代码如下：

父界面（界面二中）发送信号：

```
signals:
    void sendExit(void);
    void sendStart(method);
```

连接函数：

```
MainWindow1::MainWindow1(QWidget* parent)
    : QMainWindow(parent)
{
    ui.setupUi(this);
    w2 = new MainWindow2();
    connect(this, SIGNAL(sendStart(method)), w2, SLOT(receiveStart(method)));
```

开始槽函数：

```
void MainWindow1::Start(void)
{
    w2->show();
    emit sendStart(Q);
    this->close();
}
```

子界面（界面三中）接受槽函数:

```
private slots://相应功能槽函数
    void receiveStart(method);
    void Start3(void); // 进入3
```

槽函数实现：

```
void MainWindow2::receiveStart(method Q)
{
    Q2 = Q;
    ui.textBrowser->setText(QString::fromLocal8Bit(QByteArray::fromStdString(Q2.ADM()[1])));
    if (Q2.ADM()[0]._Equal("Yes")) {
```

通过父界面发送信号，子界面接受，便可以实现父子界面之间的传参。当子界面退出时，同样利用信号，此时由子界面发送退出信号，父界面接受。接受后在槽函数中关闭子界面，打开父界面。

4.3.2 服务状态实时更新

在界面四中，系统会对进程开展服务，服务过程中全部进程的服务状态会实时更新。这一功能可以让用户随时了解到最新的服务状态，实现这一功能需要利用到定时器。通过在界

面中设置一个定时器，每当定时器到达指定时间便刷新一次显示内容，便可以实现上述效果。

实现代码如下：

创建定时器并连接刷新槽函数：

```
MainWindow3::MainWindow3(QWidget* parent)
    : QMainWindow(parent)
{
    ui.setupUi(this);
    timer = new QTimer(this);
    connect(timer, SIGNAL(timeout()), this, SLOT(Recording()));
}
```

打开定时器：

```
void MainWindow3::Begin() {
    if (state == 0) {
        state = 1;
        ui.pushButton_2->setText(QString::fromLocal8Bit(QByteArray::fromStdString("暂停")));
        Q3.MSM();
        timer->start(1000);
    }
```

刷新槽函数：

```
void MainWindow3::Recording(void)
{
    string str = "Time: " + to_string(Time) + "\n" + Q3.Get_Serve_State_Show();
    ui.textBrowser->setText(QString::fromLocal8Bit(QByteArray::fromStdString(str)));
    Time++;
    bool flag = true;
    for (int i = 0; i < Q3.Get_Rat().Finish.size(); i++) {
        if (!Q3.Get_Rat().Finish[i]) {
            flag = false;
            break;
        }
    }
    if (flag) {
        timer->stop();
    }
}
```

在 *Recording* 槽函数中，不仅有服务状态的刷新，还有系统运行时间的递增。另外，如果系统判断当前所有的进程都已经服务完毕，会即刻关闭定时器，停止运行。

4.3.3 暂停开始服务

在系统服务中，用户可以随时暂停或者开始服务。这个效果的实现利用到了资源分配表中的 *Allow_to_run* 这一成员变量，同时搭配界面四中的槽函数。在函数中，用户每次点击按钮，系统会对当前状态进行判断，其中有三个状态：

状态 0：系统还未开始服务，此时点击按钮系统会调用多线程服务模块开展服务，并打开定时器，切换状态为 1；

状态 1：系统处于服务状态，此时点击按钮系统会暂停服务并切换到状态 2，同时设置 *Allow_to_run* 为 *false*；

状态 2：系统处于暂停状态，此时点击按钮系统会暂停服务并切换到状态 1，同时设置 *Allow_to_run* 为 *true*；

实现代码如下：

界面四中的槽函数：

```
void MainWindow3::Begin() {
    if (state == 0) {
        state = 1;
        ui.pushButton_2->setText(QString::fromLocal8Bit(QByteArray::fromStdString("暂停")));
        Q3.MSM();
        timer->start(1000);
    }
    else if(state == 1) { //服务状态
        state = 2;
        ui.pushButton_2->setText(QString::fromLocal8Bit(QByteArray::fromStdString("开始")));
        Q3.SetAllowToRun(false);
    }
    else { //暂停状态
        state = 1;
        ui.pushButton_2->setText(QString::fromLocal8Bit(QByteArray::fromStdString("暂停")));
        Q3.SetAllowToRun(true);
    }
}
```

线程服务函数：

```
//服务当前线程
for (int i = 0; i < W->rat.TOccupy[C][R]; i++) {
    while (!W->rat.Allow_to_run);
    Sleep(1000);
    W->rat.Serve_State[C][R]++;
}
```

在线程服务函数中，如果 *Allow_to_run* 为 *false*，系统会停止服务直到 *Allow_to_run* 为 *true*。

4.4 软件主要特点

4.4.1 用户界面良好

该系统的设计完全从用户的角度出发，力求用户在使用系统时可以根据向导式设计快速上手，保姆式步骤可以让用户很少有对下一步怎么操作的疑惑。另外，在界面中根据数据调查选用了黑色与绿色为主旋律，一方面降低用户使用系统时的视觉疲劳感，另一方面，绿色的数据显示也会更加显眼，让用户每时每刻都能以最快的速度捕获到数据。

4.4.2 操作安全性极高

系统界面的设计尽可能已经考虑到所有的错误输入和错误操作，并将其排除在使用之外。如：在输入部分，系统已经在界面中设置了只能输入规定范围内的正整数，避免了用户输入负数、浮点数、字符串等错误数据时可能导致的系统崩溃。另外，系统所有的按钮都已经经过多次重复测试，无论在任何时刻点击任何按钮都能够到达可控界面，用户在使用时不必有

任何鼓励，随心探索即可。

4.4.3 响应速度优良

系统对操作的响应速度很快，即使是安全性检测，也能保证最长不超过 2 秒的响应时间，用户在使用时会有极佳的体验感。

4.4.4 模拟运行求解资源利用效率

系统在安全性检测中会利用各个安全序列的资源利用效率对序列排序，在对资源利用效率的计算上，系统选用了最符合实际的模拟运行。通过按照安全序列所对应的进程顺序依次对进程模拟运行，可以准确估量出按照该序列服务时系统所花费的时间。如此排序，系统可以选出最优良的安全序列，服务效率也会大大提高。

4.4.5 公平选取各个安全序列

系统在利用深度搜索找寻安全序列时，设置了最多存储 100 个安全序列的限制。为了避免每次搜索到的都是按照固定进程顺序组合的安全序列，在 DFC 之前，系统会对进程顺序进行随机打乱，之后进入下一层后会再次打乱，再进下一层再打乱……利用多层打乱让每一种安全序列被搜索到的概率相同。

4.4.6 多线程服务

系统在对进程开展服务时选用了为每类资源创建一个线程的方式。通过 thread 的实现，线程之间可以并行服务，多个线程同时服务，系统能够以极高的效率完成服务请求，整体运行的效率也会水涨船高。

4.4.7 允许进程并行

如果资源充足，系统并不限制必须上一个进程服务完毕下一个进程才能服务。只要当空闲资源满足需求并且上一进程已经开始服务，下一个进程可以即可开始服务，不用等待结束。这样的设置更符合实际情况，因为在多线程的基础上，并不是所有的资源都会同时释放，当一类资源释放完成，空闲资源很可能已经满足了下一个进程的需求。此时让多个进程同时服务，可以再次提高系统的运行效率。

4.4.8 实时显示服务状态

在界面四，系统开展对进程的服务时，系统会在每一秒的结束实时刷新进程的服务状态。通过查看窗口，用户可以随时了解系统当前各个进程的服务情况，包括各个资源的占用时间、整体的进度百分比等等。同时系统会将已经运行的时间显示在界面中，方便用户查阅。

4.4.9 允许控制线程服务

系统在对各个线程服务的过程中，允许用户对服务进行操控。用户可以通过点击开始和

暂停随时控制系统中对各个线程的服务能力。当暂停时，系统不会再对各个线程开展服务，此时线程状态不会丢失，会在原地持续等待，直到用户允许系统提供服务为止。注意，虽然暂停时系统会停止对线程服务，但是系统其实还在运行中，总的运行时间仍然会不停上涨，只有当所有线程服务完毕，系统才会停止，此时时间不再变化。

5 软件测试

软件测试验证软件是否符合规格和需求，发现并修复缺陷，确保软件质量和可靠性，提高用户体验，降低维护成本。软件测试可以确保软件质量和可靠性的关键步骤，帮助开发人员提高软件质量，减少维护成本，提高用户满意度。

5.1 任务完成度

任务	是否完成
界面友好	完成
n 个客户，m 类资源（每个资源的上限随机生成）	完成
已分配资源的初始值是随机生成的	完成
需求资源的初始值是随机生成的	完成
每个客户占用每类资源的时间是不同的	完成
生成尽可能多的安全序列，并从资源利用效率方面给出这些安全序列的排序	完成
在算法实现过程中开启多个安全线程	完成
在线程共享数据过程中，实现数据的实时同步	完成

5.2 各模块测试

5.2.1 随机生成模块

测试思路：采用“打印调试”的方法，这是一种简单而有效的调试技术。通过打印随机生成模块所生成的数据，开发人员可以快速检测是否存在问题，从而减少错误排查的时间。这种方法可以用于各种类型的软件，特别是需要大量数据生成的应用程序，如数据库系统和数据分析软件。打印调试可以帮助开发人员更快地找到软件的错误和缺陷，提高软件的质量和可靠性。此外，打印调试还可以为开发人员提供更深入的了解和掌握软件代码，从而更好地进行调试和优化。

测试界面如下：

测试界面中设定了客户数目为 5，资源类别数目为 3；除此数据，系统还对客户数为 10，资源类别为 5；客户数为 1，资源类别数为 1；客户数为 14，资源类别数为 14 等多种数据进行测试。测试结果如下：

测试用例	预期结果	测试结果
客户：5　资源：3	数据正常	数据正常
客户：10　资源：5	数据正常	数据正常
客户：1　资源：1	数据正常	数据正常
客户：14　资源：14	数据正常	数据正常
客户：20　资源：6	数据正常	数据正常
客户：7　资源：8	数据正常	数据正常

经过充分的测试和比对，随机生成模块在生成数据方面表现良好，并且没有发现任何错误或缺陷。这是一个积极的结果，表明随机生成模块是可靠的，可以在各种应用程序中使用。

5.2.2　检测模块

对于检测模块，测试的目的是确保其能够正确地检测和识别各种可能的输入和情况，并生成正确的检测结果。为此，测试人员需要对各种不同类别的请求向量进行设置，以覆盖所有可能的检测结果。这样可以验证检测模块的有效性和可靠性，确保其能够正常运行并生成正确的结果。通过充分的测试和验证，可以确保检测模块能够满足用户的需求和期望，提高软件的质量和可靠性，减少维护成本，增强用户的信心和满意度。

检测时使用的资源分配表：

检测界面：

测试界面中设定了请求客户为 C0，资源请求为“R0-3 R1-7 R2-2 R3-7 R4-1”；这个测试

例子是针对特定的请求客户和资源请求进行设计的。通过对这个测试用例的执行，可以验证系统对于这种情况的处理符合预期。除此数据，测试还设计了其他多种样例，覆盖所有可能情况。测试结果如下：

测试用例	预期结果		测试结果
	可分配检测	安全性检测	
C9 R0-5 R1-2 R2-3 R3-2 R4-4 R5-5	通过	不通过	符合
C4 R0-22 R1-5 R2-5 R3-8 R4-9 R5-9	不通过	不检测	符合
C7 R0-5 R1-4 R2-2 R3-0 R4-1 R5-1	通过	不通过	符合
C3 R0-1 R1-4 R2-0 R3-1 R4-3 R5-2	通过	通过	符合
C1 R0-1 R1-0 R2-1 R3-0 R4-3 R5-3	通过	通过	符合

在进行多次测试用例输入后，我对检查模块进行了全面覆盖，覆盖了所有可能出现的情况。最终的测试结果与预期的一致，经过测试的模块已被证明符合要求，因此认为此次测试已经通过。

5.2.3 多线程服务模块

多线程服务模块在整个系统中扮演着核心角色，为了确保该模块的可靠性和稳定性，我对其进行了全面的测试。测试主要包括三个方面：第一，测试线程服务是否能够同时处理多个请求，不会出现阻塞或死锁；第二，测试各进程之间是否同步，能够互相通信和共享数据；第三，测试临界区是否互斥，能够避免多个线程同时访问同一个资源而导致的冲突。

测试界面如下：

测试结果表明每个进程的各个线程能够同时处理请求，没有出现阻塞或死锁现象。另外，我还测试了各进程之间是否同步，结果证明各个进程之间能够同步运行，并且下一个进程不会在上一个进程服务之前开始，保证了整个系统的正常运行。

除了上述测试之外，还设计了其余多种测试用例，如下：

测试用例	预期结果	测试结果
C5->C4->C8->C2->C1->C3->C9->C6->C7->C0	服务正常	服务正常
C5->C4->C8->C2->C1->C9->C7->C3->C6->C0	服务正常	服务正常
C5->C4->C8->C2->C1->C6->C9->C7->C0->C3	服务正常	服务正常
C3->C1->C7->C6->C2->C9->C8->C5->C4->C0	服务正常	服务正常
C8->C3->C1->C5->C9->C6->C0->C7->C4->C2	服务正常	服务正常
C7->C1->C0->C2->C5->C6->C9->C8->C4->C3	服务正常	服务正常

可以看出，多线程在服务上无任何问题，所有序列均通过检测。

下面测试真实服务时间和模拟运行时间是否匹配：

模拟运行时间：

真实服务时间：

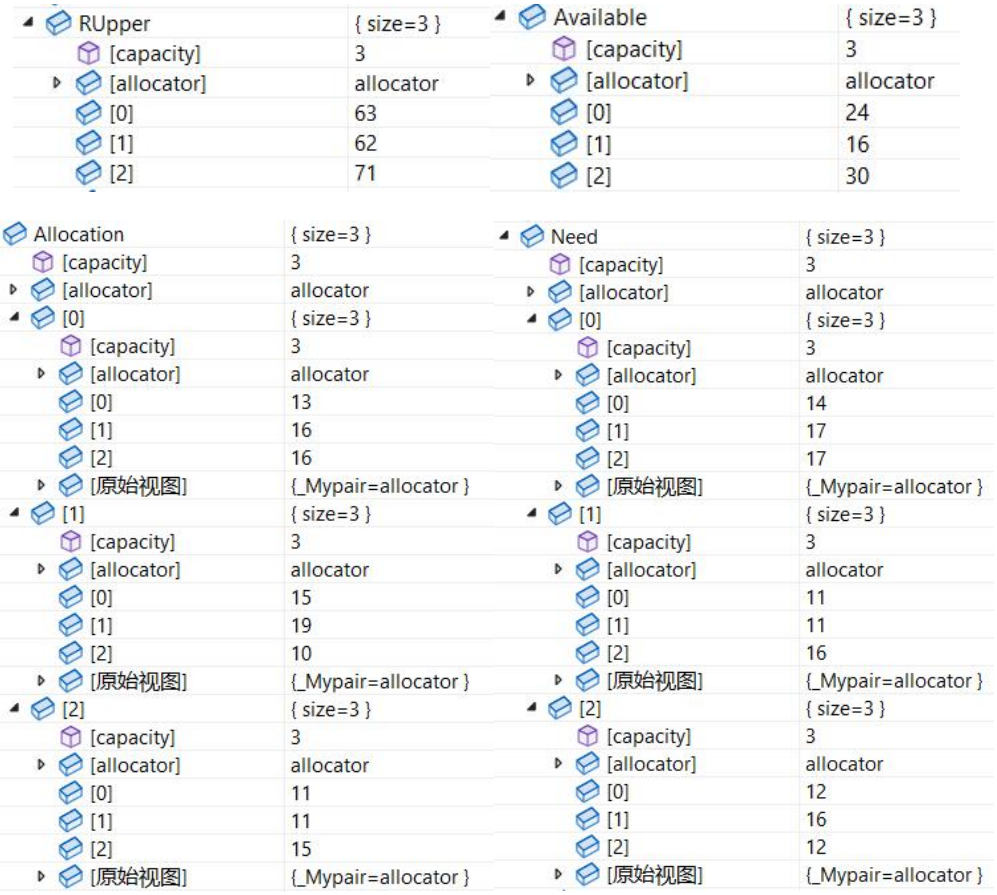
经过测试，可以发现多线程服务的真实服务时间与模拟运行时间一致，这表明所设计的多线程服务实现是正确的。同时，这也验证了本系统对资源利用效率的计算方式是准确无误的。这些测试结果为我们进一步优化系统性能提供了指导和参考。

5.2.4 内容显示模块

内容显示模块是整个软件系统中至关重要的一部分，其承担着向用户展示各种信息和数据的任务。由于其贯穿系统前后，且在软件使用过程中无时无刻不在向用户展示信息，因此对其进行充分的测试和验证显得尤为重要。如果信息显示错误，用户将会受到误导并可能导致错误的操作，严重影响用户体验和软件的可靠性。

测试方法如下：通过在内容显示后设置断点，可以暂停程序运行，进而比对界面显示的内容和系统资源分配表中承载的数据是否一致。这种测试方法可以有效地检测到可能存在的界面显示异常或数据处理错误，从而提高软件的稳定性和可靠性。同时，断点测试也是一种比较直观和简单的测试方法，容易实施和使用，适合在软件开发的不同阶段使用。

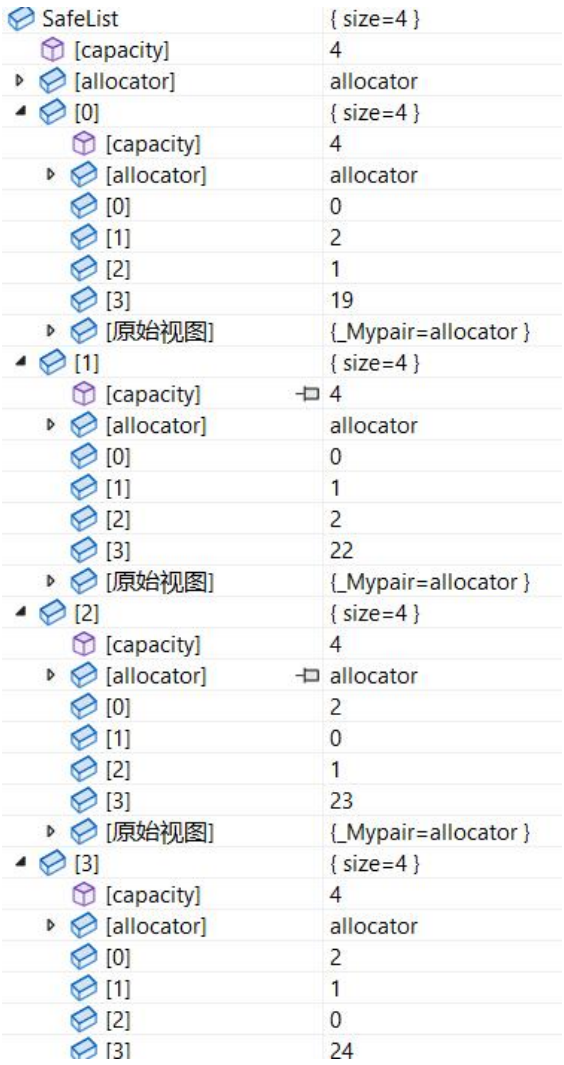
测试界面如下：



Max	{ size=3 }
[capacity]	3
[allocator]	allocator
[0]	{ size=3 }
[capacity]	3
[allocator]	allocator
[0]	27
[1]	33
[2]	33
[原始视图]	{_Mypair=allocator }
[1]	{ size=3 }
[capacity]	3
[allocator]	allocator
[0]	26
[1]	30
[2]	26
[原始视图]	{_Mypair=allocator }
[2]	{ size=3 }
[capacity]	3
[allocator]	allocator
[0]	23
[1]	27
[2]	27
[原始视图]	{_Mypair=allocator }

TOccupy	{ size=3 }
[capacity]	3
[allocator]	allocator
[0]	{ size=3 }
[capacity]	3
[allocator]	allocator
[0]	6
[1]	6
[2]	5
[原始视图]	{_Mypair=allocator }
[1]	{ size=3 }
[capacity]	3
[allocator]	allocator
[0]	7
[1]	7
[2]	3
[原始视图]	{_Mypair=allocator }
[2]	{ size=3 }
[capacity]	3
[allocator]	allocator
[0]	4
[1]	7
[2]	7

Request	{ size=4 }
[capacity]	4
[allocator]	allocator
[0]	0
[1]	0
[2]	0
[3]	0



上述测试说明界面显示结果和系统后台资源分配表数据吻合，证明了内容显示模块的正确性。除了上述测试样例，我还设计了其他用例，通过多次检测力求开展更好的验证。测试

用例如下：

测试用例	预期结果	测试结果
客户：4　资源：4	内容显示正确	内容显示正确
客户：8　资源：7	内容显示正确	内容显示正确
客户：4　资源：9	内容显示正确	内容显示正确
客户：1　资源：1	内容显示正确	内容显示正确
客户：20　资源：14	内容显示正确	内容显示正确
客户：17　资源：13	内容显示正确	内容显示正确

经过多个样例测试，最终验证了所有内容界面都与后台数据一致。这意味着内容显示模块通过了测试，能够正常地显示和处理数据，确保了软件的稳定性和可靠性。

5.3　公开测试

公开测试是指向公众开放的软件测试，即开放测试给广大用户自由试用，以获得用户的反馈和意见，进一步完善和改进软件。在公开测试中，通常会向用户提供测试版软件，让用户在实际使用中发现和报告软件的缺陷、漏洞、性能问题和用户体验等方面的问题，从而帮助开发团队发现和解决这些问题，提高软件的质量和稳定性。

在这次的改进银行家算法开发中，我邀请了三位同学对我的软件进行公开测试，通过他们的使用来进一步优化软件。

同学提出的问题：

（1）软件在处理大量数据时运行缓慢，导致用户体验不佳。

（2）软件界面的布局不够清晰，用户难以快速找到需要的功能。

（3）软件在某些情况下出现崩溃，导致数据丢失和操作无法完成。

我的改进：

（1）为了提高软件的性能，我对银行家算法的实现方式进行了优化，并进行了多次性能测试。同时，我还加入了进度条等功能来提示用户当前任务的进度，让用户能够更直观地感受到软件的运行情况。

（2）我根据用户反馈，重新设计了软件的界面布局，将常用功能置于更显眼的位置，并加入了搜索功能，让用户更方便地找到所需功能。同时，我还改善了字体、颜色等方面的细节，让界面更加美观。

（3）为了解决软件崩溃的问题，我进行了多次代码审查，并加入了错误处理机制，使得软件在出现异常情况时能够优雅地退出，并自动保存用户数据。同时，我还进行了多次稳定性测试，确保软件在各种情况下都能够稳定运行。

在三位同学的使用过程中，我发现了软件存在一些潜在的问题，例如运行缓慢、界面部分不够友好等，通过他们的反馈和建议，我对这些问题进行了改进和优化，以确保软件在交付用户时是一个足够健壮的版本。经过这次公开测试，我对软件开发和改进的方向有了更加清晰的认识，也更加重视用户的使用体验和反馈。

6 改进方向

6.1 提高软件的运行效率

本系统在对安全序列的计算中使用了深度搜索，这是一种很方便的搜索方法，不会漏掉任何一种安全序列。同时为了防止运行超时，搜索设置了最多 100 条；并且在公平搜索的实现上采用了打乱序列。然而，这种方式在 100 条的限制下仍然有可能进入到单路途中，不能找寻多种安全序列。

后期改进方向可以更换更高运行速率的设备，这样可以将 100 条的限制提高，让更多的安全序列进入考虑。另一方面，可以通过改进程序，比如更换红黑叔等更有效率的数据结构或者尝试利用广度搜索等。

6.2 提高软件的可扩展性和灵活性

在这次实验课中，我们的软件是基于一个固定的数据集和资源分配表进行运行的。但实际上，除此还需要考虑软件的可扩展性和灵活性，使其能够适应更加复杂和多样化的应用场景。

因此，我们可以考虑使用一些设计模式和架构模式，使得软件的代码更加模块化和可扩展，从而更容易地添加新的功能和算法。同时，我们还可以考虑使用一些动态配置文件，以便在运行时修改资源分配表和数据集，从而适应不同的需求。

6.3 提高软件的空间利用率

本次实验中，算法的实现基于大量的一维和二维容器，后期可以考虑通过优化数据结构，减少程序运行时所需的内存空间，从而提高软件的空间占用效率。可以考虑使用更为紧凑的数据结构，如位图、哈希表等。此外，可以通过压缩算法对数据进行压缩，进一步减少空间占用。

另外，可以搭配一些优化算法进行使用，这可以减少软件对空间的需求，提高空间占用效率。比如通过改进排序算法、搜索算法等，减少算法运行时所需的空间占用。