**目录**

[一、 问题描述与基本要求 4](#_Toc169888207)

[二、 需求分析 4](#_Toc169888208)

[2.1功能需求 4](#_Toc169888209)

[2.1.1消费者-生产者模型模拟 4](#_Toc169888210)

[2.1.2. 银行家算法 4](#_Toc169888211)

[2.1.3 调度算法模拟 5](#_Toc169888212)

[三、 项目计划与文档设计 5](#_Toc169888213)

[3.1.项目开发计划书 5](#_Toc169888214)

[3.2.需求设计分析书（兼可行性分析报告） 6](#_Toc169888215)

[3.3.软件设计说明书（概要设计与详细设计说明书） 7](#_Toc169888216)

[3.4.用户操作手册 7](#_Toc169888217)

[3.5.测试计划 7](#_Toc169888218)

[3.6.测试分析报告 8](#_Toc169888219)

[3.7.开发总结 9](#_Toc169888220)

[3.8.软件维护，软件修改报告 9](#_Toc169888221)

[四、概要设计 9](#_Toc169888222)

[4.1.程序流程图设计 9](#_Toc169888223)

[4.1.1 生产者消费者流程图 10](#_Toc169888224)

[4.1.2 银行家算法 11](#_Toc169888225)

[4.1.3 调度算法模拟计算 12](#_Toc169888226)

[4.2.用例图设计 12](#_Toc169888227)

[4.3.数据流设计 13](#_Toc169888228)

[4.4.数据库设计 14](#_Toc169888229)

[4.4.1.数据库表设计 14](#_Toc169888230)

[4.4.2.数据库E-R图设计 17](#_Toc169888231)

[五、详细设计 18](#_Toc169888232)

[5.1.整体GUI设计 18](#_Toc169888233)

[5.1.1 ProducterAndConsumer 18](#_Toc169888234)

[5.1.2 BankAlorithSinmulation、OSD2 18](#_Toc169888235)

[5.1.3 GUI、CPUScheduler 19](#_Toc169888236)

[5.2.多线程同步异步控制分析以及公有资源的互斥访问设计 21](#_Toc169888237)

[5.4.对输入表格数据的抽象处理 24](#_Toc169888238)

[5.5.详细算法设计 26](#_Toc169888239)

[六、总结与心得 27](#_Toc169888240)

# 问题描述与基本要求

在当前计算机操作系统的教学中，经常需要讲解和理解各种复杂的系统模型。然而，传统的教学方法往往依赖于PPT中简陋的动画来演示这些模型的过程，这种方式既不直观也缺乏通用性，难以有效地帮助学生理解操作系统的内部机制。本项目旨在通过实际计算机环境中的模拟计算，为计算机操作系统中的关键模型提供动态演示和分析工具，以增强教学效果和学生理解力。

消费者-生产者模型是操作系统中常见的并发控制问题，通过模拟多个生产者和消费者进程在共享缓冲区中的交互，可以演示互斥锁、信号量等同步机制的实现和作用。通过实际模拟这一模型，学生可以更直观地理解并发控制的核心概念和技术细节。

​银行家算法用于解决系统中的死锁避免问题，通过模拟不同资源请求和分配的场景，展示银行家算法如何通过资源分配策略来避免死锁的发生。通过这种动态模拟，学生可以观察到在不同条件下，算法如何做出决策，从而深入理解死锁避免的原理和实现方法。

​调度算法是操作系统的核心功能之一，负责在多个进程之间分配CPU资源。本项目将模拟包括先来先服务、短作业优先、最短剩余时间优先、轮转法、优先级调度和多级反馈队列在内的六种经典调度算法。通过设置不同参数，可以演示这些算法在实际运行中的调度过程和效果，帮助学生比较和分析不同算法的优缺点及其适用场景。

​通过上述模拟工具，本项目将为操作系统教学提供更直观、动态和互动的教学手段，使学生能够在真实的计算机环境中观察和分析各种操作系统模型的运行机制。模拟结果将提供准确、合理的输出，帮助学生理解不同参数设置对模型行为的影响，帮助学生理解不同参数设置对模型行为的影响，从而提升学习效果和实际操作能力。

# 需求分析

## 2.1功能需求

## 2.1.1消费者-生产者模型模拟

​消费者-生产者模型模拟功能的主要需求是模拟多个生产者和消费者进程在共享缓冲区中的交互过程，展示互斥锁和信号量在并发控制中的应用。输入数据包括生产者数量、消费者数量和缓冲区大小；输出数据则涵盖生产和消费的次数、缓冲区状态、等待时间和执行时间等。

​在用例描述中，用户输入生产者数量、消费者数量和缓冲区大小后，点击“开始模拟”按钮，系统将显示模拟过程并输出结果。用户可以调整参数重新运行模拟，并将结果保存到本地文件。

## 2.1.2. 银行家算法

​银行家算法模拟的主要需求是展示银行家算法在不同资源请求和分配场景中的工作原理和决策过程，避免系统死锁。输入数据包括进程数量、资源数量、各进程最大需求和当前持有资源数量；输出数据则为系统是否安全、资源分配状态和进程执行顺序。

​在用例描述中，用户输入各进程的资源需求和当前持有资源数量，点击“开始模拟”按钮，系统将显示模拟过程并输出结果。用户可以调整参数重新运行模拟，并将结果保存到本地文件。

## 2.1.3 调度算法模拟

​调度算法模拟的主要需求是模拟六种经典调度算法，展示不同调度策略的调度过程和效果。输入数据包括作业数量、作业到达时间、作业执行时间和优先级（如果适用）；输出数据则为各作业的开始时间、完成时间、等待时间和周转时间。

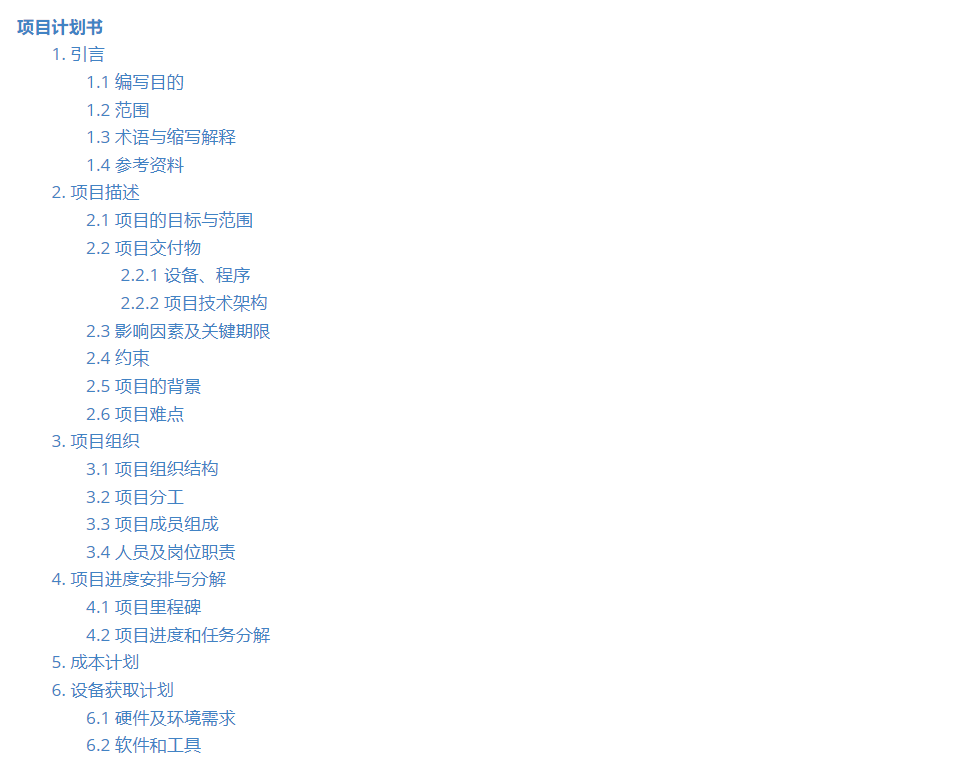
​在用例描述中，用户输入作业参数，选择调度算法，点击“开始模拟”按钮，系统将显示模拟过程并输出结果。用户可以调整参数重新运行模拟，并将结果保存到本地文件。

# 项目计划与文档设计

此处电子版可直接查看，为避免文档过于冗长，这里补贴出全文，仅展示目录

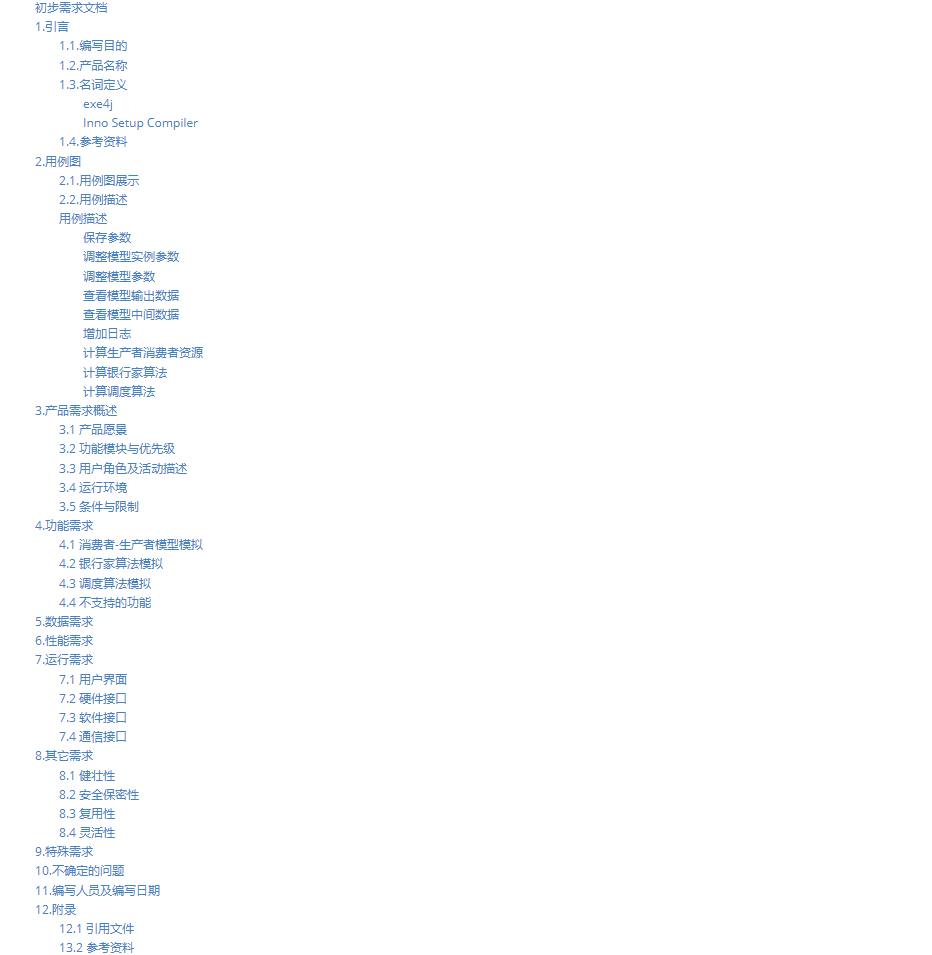
## 3.1.项目开发计划书





## 3.2.需求设计分析书（兼可行性分析报告）





## 3.3.软件设计说明书（概要设计与详细设计说明书）

软件设计说明书即本文档内容

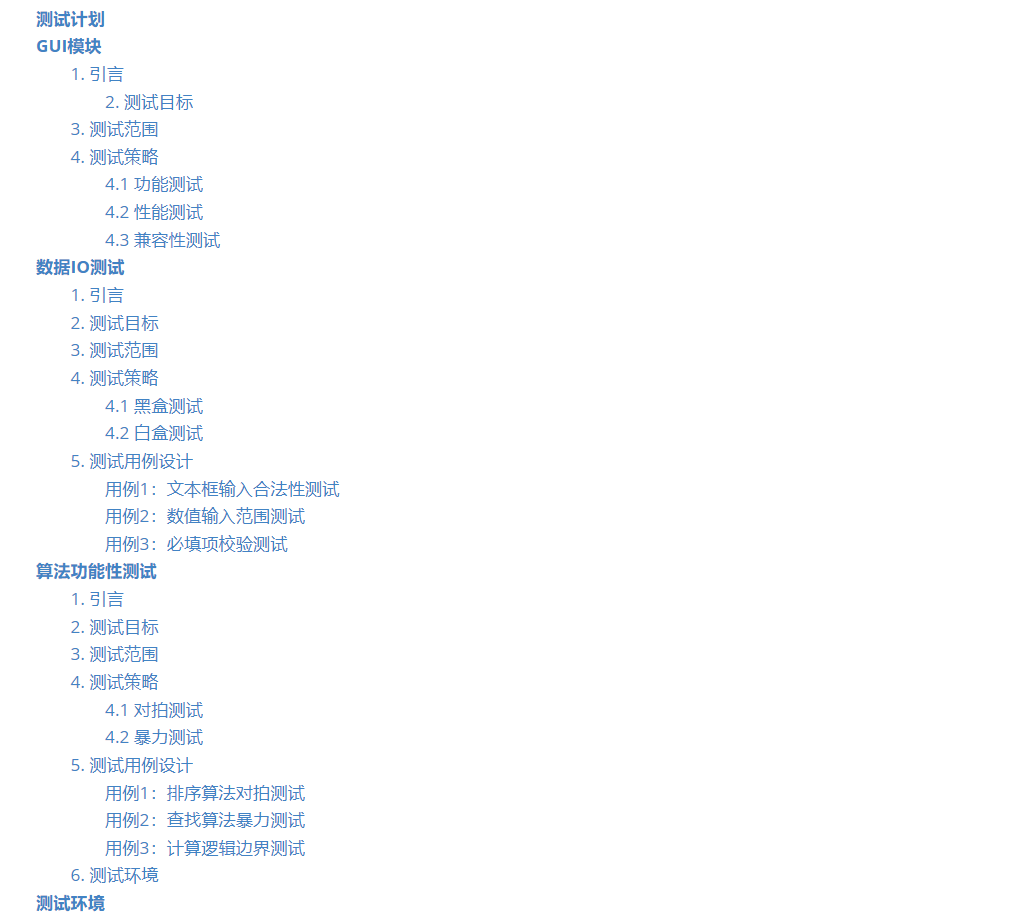
## 3.4.用户操作手册





## 3.5.测试计划





## 3.6.测试分析报告

对GUI部分的测试分析报告

1. 父子窗口依赖有待改善，在关闭父窗口后，子窗口不一定会完全关闭

2. 对窗口的反复开关测试中发现，在不同的窗口中会调用历史数据导致每次进程并非独立

对数据检查的报告

1. 对银行家算法中，输入的数据矩阵缺少对Allocation少于Max的检查
2. 在生产者消费者模型中缺少对资源上限的检查
3. 在调度算法中，缺少对浮点时间的支持

对功能性的检查

1. 缺少了动态的功能下显示

以上问题已在开发中解决，仍未解决的问题有动态的图像显示调度过程，详细更多测试问题请参考项目维护网站*47.120.18.226*。

## 3.7.开发总结

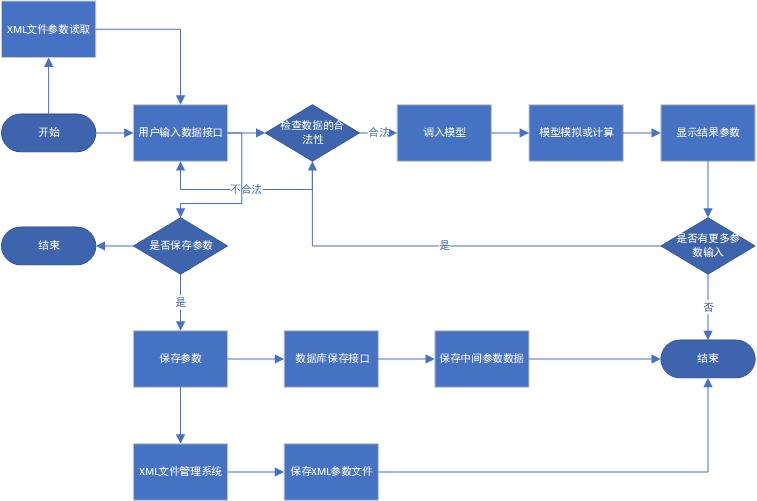
即本文档。

## 3.8.软件维护，软件修改报告

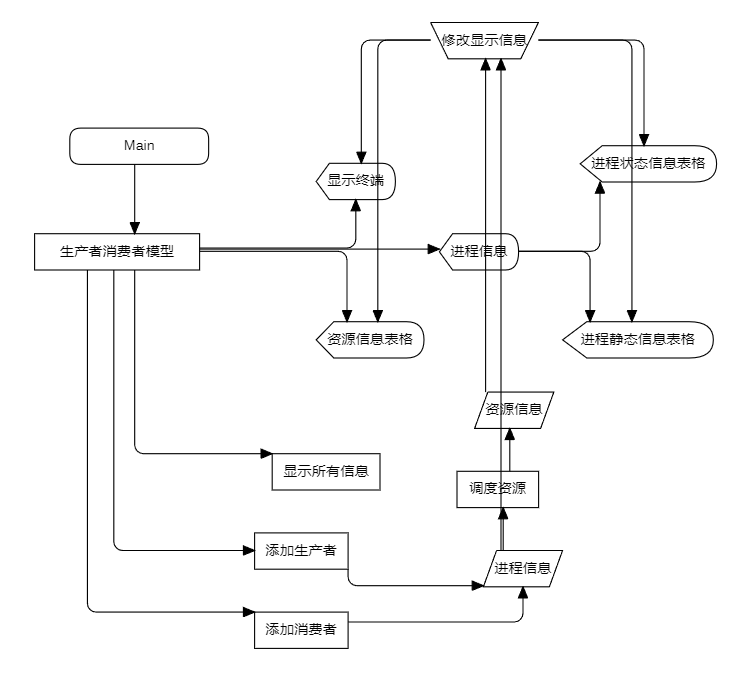
于个人网站*47.120.18.226*持续更新、维护，并在该网站更新网络使用日志。

# 四、概要设计

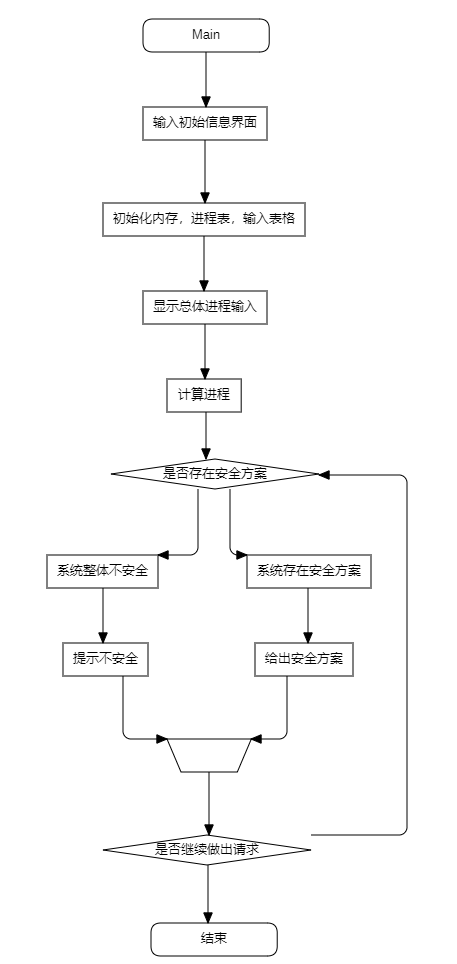
## 4.1.程序流程图设计



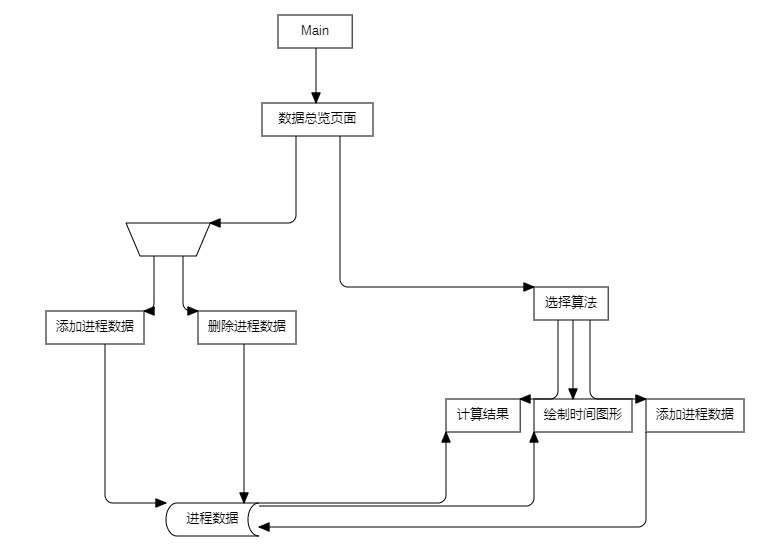
### 4.1.1 生产者消费者流程图



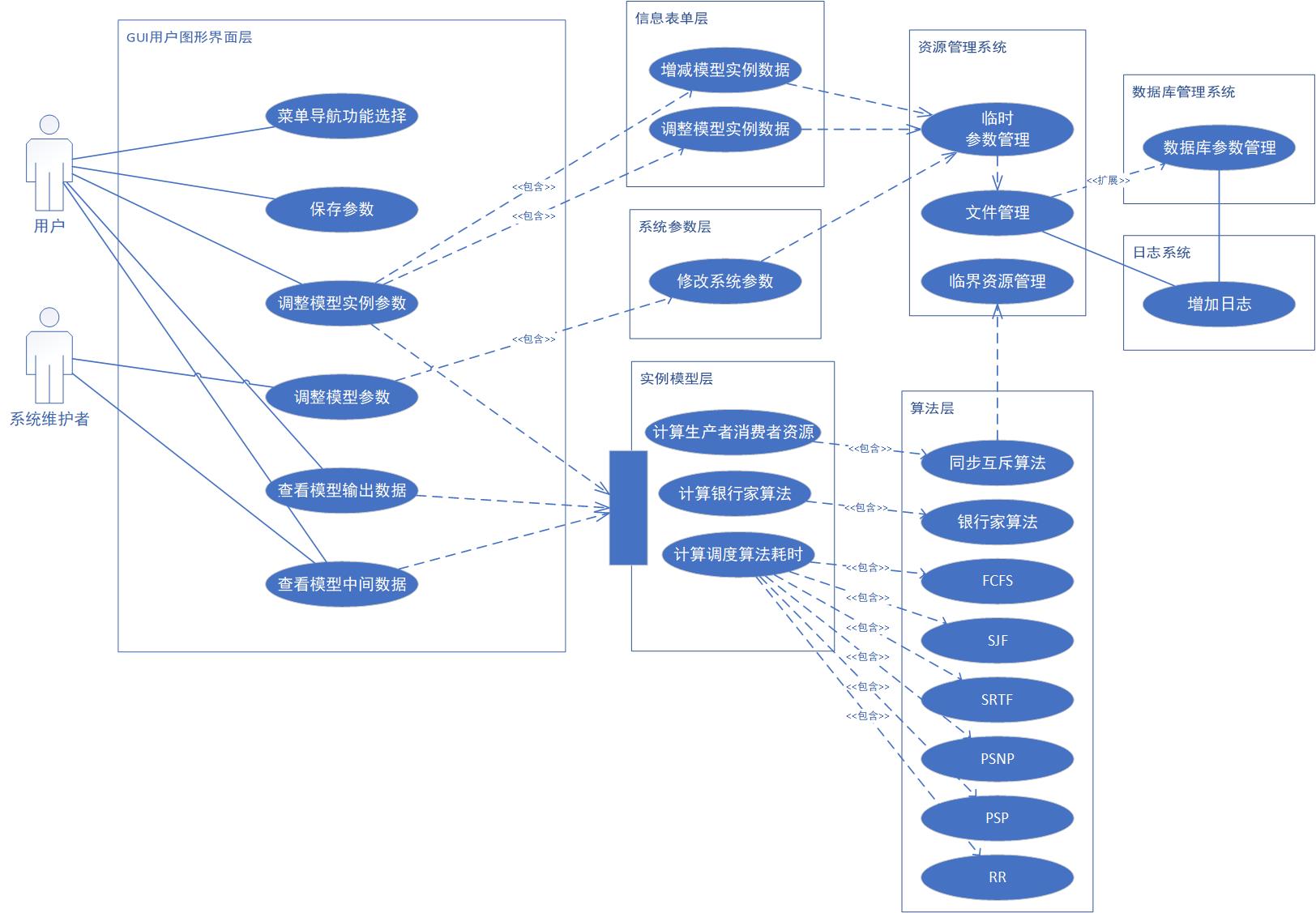
### 4.1.2 银行家算法

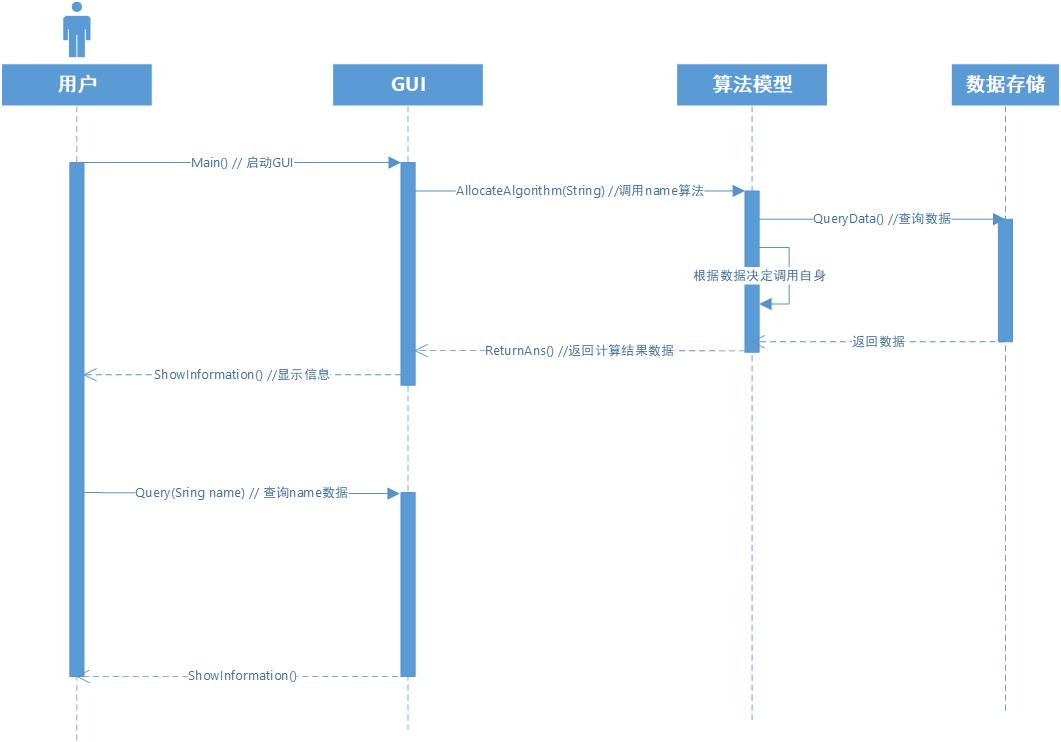


### 4.1.3 调度算法模拟计算

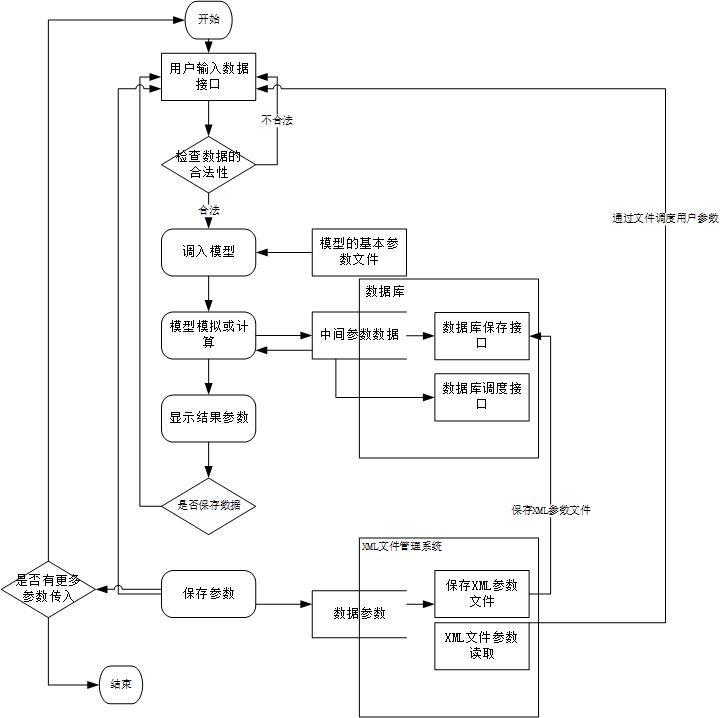


## 4.2.用例图设计





## 4.3.数据流设计

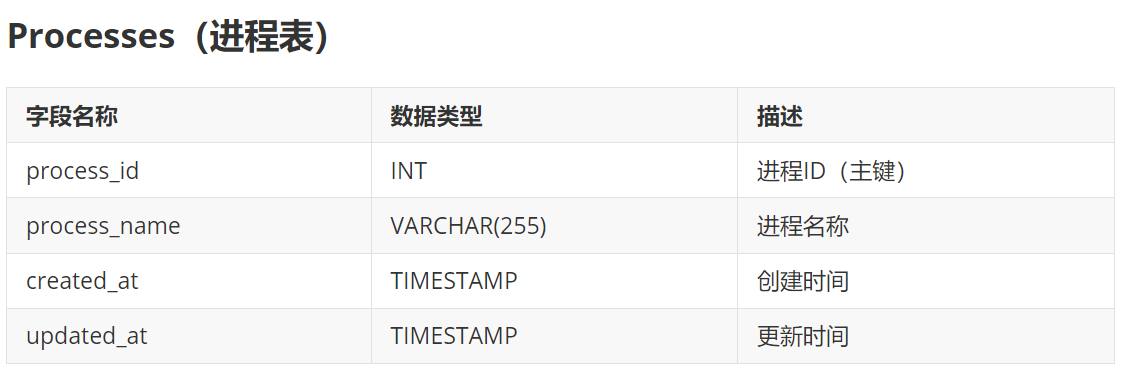


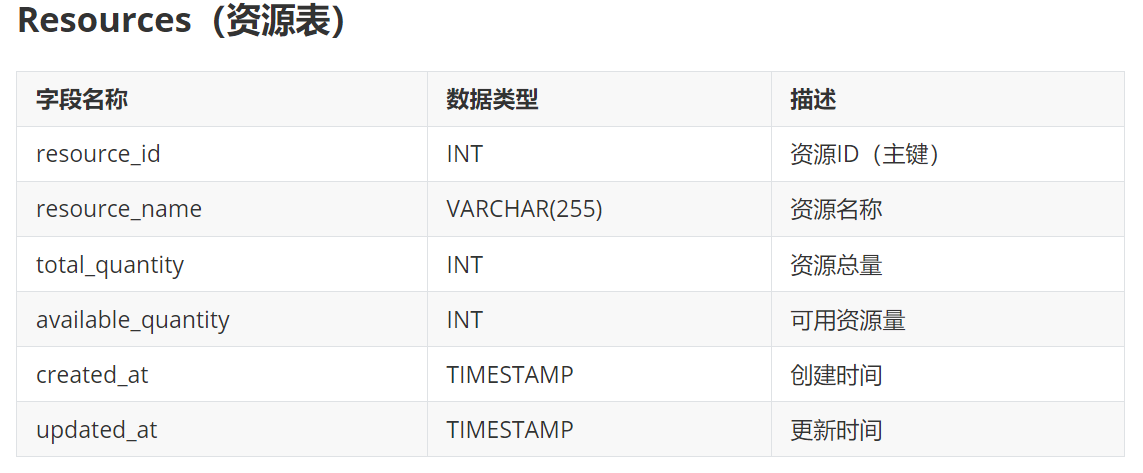
## 4.4.数据库设计

### 4.4.1.数据库表设计









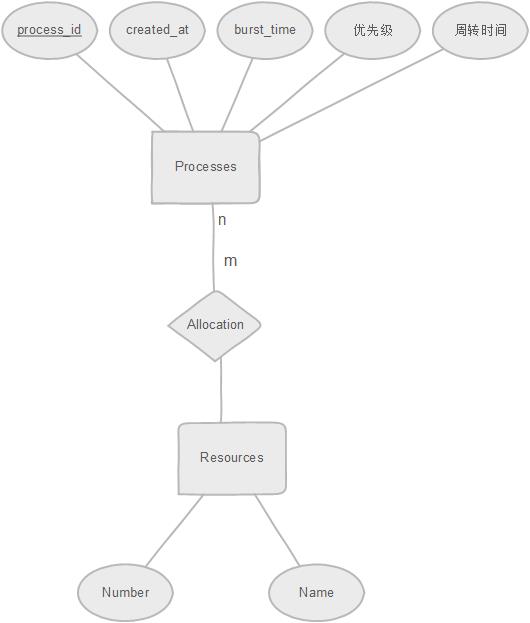







### 4.4.2.数据库E-R图设计



# 五、详细设计

## 5.1.整体GUI设计

### 5.1.1 ProducterAndConsumer

这个Java类ProducterAndConsumer实现了一个图形用户界面（GUI），用于管理生产者-消费者模型的模拟。



### 5.1.2 BankAlorithSinmulation、OSD2

BankAlorithSinmulation，这个Java类实现了一个银行家算法的模拟程序的图形用户界面（GUI）。



OSD2，这个类实现了一个图形用户界面（GUI），用于接收用户输入的资源分配数据，并对银行家算法进行模拟。





### 5.1.3 GUI、CPUScheduler

GUI这个类实现了一个简单的调度算法模拟的图形用户界面（GUI）。



CPUScheduler 类是一个抽象类，用于定义通用的调度算法逻辑和接口。以下是它的详细分析：

类和成员变量

* 类声明: CPUScheduler 是一个抽象类，旨在为各种具体的 CPU 调度算法提供一个基类。
* 成员变量:
  + rows: 用于存储所有进程的列表。
  + timeline: 用于存储调度过程中产生的事件列表，表示进程的执行时间段。
  + timeQuantum: 时间片，主要用于时间片轮转（RR）调度算法。

构造方法

* CPUScheduler(): 初始化 rows 和 timeline 为空的 ArrayList，并将 timeQuantum 设置为 1。

公共方法

* add(Row row): 向 rows 列表中添加一个新的进程。
* setTimeQuantum(int timeQuantum): 设置时间片的大小。
* getTimeQuantum(): 获取当前设置的时间片大小。
* getAverageWaitingTime(): 计算并返回所有进程的平均等待时间。
* getAverageTurnAroundTime(): 计算并返回所有进程的平均周转时间。
* getEvent(Row row): 根据给定的进程行，返回相应的事件。如果找不到对应的事件，返回 null。
* getRow(String process): 根据进程名返回对应的进程行。如果找不到对应的进程，返回 null。
* getRows(): 返回所有进程的列表。
* getTimeline(): 返回调度过程中产生的事件时间线。

抽象方法

* process(): 抽象方法，需要在具体的调度算法类中实现。这个方法的实现应该包含具体调度算法的逻辑。

CPUScheduler是六个算法类的父类对进程调度的公共方法进行了实现。

## 5.2.多线程同步异步控制分析以及公有资源的互斥访问设计

1. List<String> checkResourse(){
2. List<String> res = **new** ArrayList<>();
3. **for** (Pair<String, Integer> a: ProduceItems
4. ) {
5. **if**(ResourceManager.getInstance().getResourceNumberByName(a.getKey()) + a.getValue() > ResourceManager.getULIMIT()){
6. res.add(a.getKey());
7. }
8. }
9. **return** res;
10. }
12. @Override
13. **public** **void** run() {
14. ProducterAndConsumer.getInstance().getTerminal().append("生产者"+name+"开始生产了.....\n");
15. ProducterAndConsumer.getInstance().getRunningTableModel().addRow(**new** String[]{String.valueOf(ID), name});
16. ProducterAndConsumer.getInstance().getOtableModel().addRow(**new** String[]{String.valueOf(ID), name});
17. DefaultTableModel run = ProducterAndConsumer.getInstance().getRunningTableModel();
18. DefaultTableModel wait = ProducterAndConsumer.getInstance().getWaitTableModel();
19. //        run.addRow(new String[]{String.valueOf(ID), name});
20. **new** Timer().schedule(**new** TimerTask() {
21. @Override
22. **public** **void** run() {
23. List<String> lockRes = **null**;
24. **int** f = 1;
25. **do**{
26. **try** {
27. P();
28. System.out.println("生产者"+ name +"锁同步");
29. } **catch** (InterruptedException e) {
30. **throw** **new** RuntimeException(e);
31. }
32. lockRes = checkResourse();
33. **if**(f == 1 && !lockRes.isEmpty()) {
34. **for** (**int** i = 0; i < run.getRowCount(); i++) {
35. **if** (run.getValueAt(i, 1) == name) run.removeRow(i);
36. }
37. ProducterAndConsumer.getInstance().getTerminal().append("生产者者" + name + "因资源"+lockRes.toString()+",过多挂起\n");
38. wait.addRow(**new** String[]{String.valueOf(ID), lockRes.toString()});
39. f=0;
40. }
41. **if**(!lockRes.isEmpty()) {
42. V();
43. System.out.println("生产者者" + name + "挂起解同步\n");
45. **try** {
46. sleep(10000);
47. } **catch** (InterruptedException e) {
48. **throw** **new** RuntimeException(e);
49. }
50. }
51. }**while**(!lockRes.isEmpty());
53. **if**(f == 0) {
54. ProducterAndConsumer.getInstance().getTerminal().append("生产者" + name + "已经允许恢复生产\n");
55. **for** (**int** i = 0; i < wait.getRowCount(); i++) {
56. **if** (wait.getValueAt(i, 0).equals(String.valueOf(ID))) wait.removeRow(i);
57. }
58. run.addRow(**new** String[]{String.valueOf(ID), name});
59. f=1;
60. }
62. **for** (Pair<String, Integer> a: ProduceItems
63. )
64. ResourceManager.getInstance().updateResourceItem(a.getKey(),a.getValue());
65. V();
66. System.out.println("生产者" + name + "解同步");
67. //                System.out.println(multex);
69. **try** {
70. sleep(1000);
71. } **catch** (InterruptedException e) {
72. **throw** **new** RuntimeException(e);
73. }
74. }
75. }, 1000, 1000);
76. }

以Producter类对共有资源的调度为例为例。

具体来说，通过 ResourceManager.getInstance().getResourceNumberByName 和 ResourceManager.getLIMIT 方法检查资源是否超过限制，如果超过则将资源添加到 res 列表中。在 run 方法中，调用 checkResourse 方法获取需要锁定的资源列表 lockRes。如果 lockRes 非空，则表示有资源需要锁定。

互斥机制的实现首先体现在资源的锁定与释放上。代码通过检查 lockRes 是否为空，决定是否执行资源锁定操作。如果 lockRes 非空且不为空列表，生产者将尝试执行 IO() 方法并打印锁定信息，这里隐含着资源的互斥使用，因为 IO() 可能涉及对共享资源的操作。在捕获 InterruptedException 异常后，生产者通过抛出新的 RuntimeException 确保在异常情况下的资源释放和线程中断处理。

资源状态管理在生产者类中也有充分体现。在生产者生产资源的过程中，通过 addRow 方法将新生产的资源添加到 run 表中，并在 wait 表中更新资源状态。这确保了系统中的资源状态始终与实际情况一致，方便其他线程进行资源检查和使用。

在锁的检查与获取方面，生产者在处理完资源后，通过 removeRow 方法从 run 表中移除处理完的资源，并从 wait 表中移除锁定的资源。这确保了资源在使用后被正确地释放，以供其他生产者或消费者使用。同时，代码通过 try-catch 块捕获并处理 InterruptedException 异常，确保在生产过程中如果发生中断，资源状态能够被正确地维护和更新，避免资源泄漏和死锁情况。

此外，生产者在资源锁定操作后，通过 sleep(10000) 方法使线程进入睡眠状态，模拟资源生产过程中等待其他条件满足的过程。这段时间内，生产者线程不占用CPU资源，但保持对资源的锁定，确保其他线程无法同时访问这些资源。总之，生产者类通过 checkResourse 方法实现资源的检查和锁定，通过 run 方法实现资源的生产和状态管理，并通过同步机制（如 Timer 和 schedule）和互斥机制（如资源锁定和释放）确保了生产过程中的资源安全和高效利用。

## 5.4.对输入表格数据的抽象处理

1. **public** **abstract** **class** CPUScheduler
2. {
3. **private** **final** List<Row> rows;
4. **private** **final** List<Event> timeline;
5. **private** **int** timeQuantum;
7. **public** CPUScheduler()
8. {
9. rows = **new** ArrayList();
10. timeline = **new** ArrayList();
11. timeQuantum = 1;
12. }
14. **public** **boolean** add(Row row)
15. {
16. **return** rows.add(row);
17. }
19. **public** **void** setTimeQuantum(**int** timeQuantum)
20. {
21. **this**.timeQuantum = timeQuantum;
22. }
24. **public** **int** getTimeQuantum()
25. {
26. **return** timeQuantum;
27. }
29. **public** **double** getAverageWaitingTime()
30. {
31. **double** avg = 0.0;
33. **for** (Row row : rows)
34. {
35. avg += row.getWaitingTime();
36. }
38. **return** avg / rows.size();
39. }
41. **public** **double** getAverageTurnAroundTime()
42. {
43. **double** avg = 0.0;
45. **for** (Row row : rows)
46. {
47. avg += row.getTurnaroundTime();
48. }
50. **return** avg / rows.size();
51. }
53. **public** **double** getAverageTurnAroundTimeWithPower()
54. {
55. **double** avg = 0.0;
56. **for**(Row row: rows){
57. avg += row.getTurnaroundTime() / row.getBurstTime();
58. }
59. **return** avg / rows.size();
60. }
62. **public** Event getEvent(Row row)
63. {
64. **for** (Event event : timeline)
65. {
66. **if** (row.getProcessName().equals(event.getProcessName()))
67. {
68. **return** event;
69. }
70. }
72. **return** **null**;
73. }
75. **public** Row getRow(String process)
76. {
77. **for** (Row row : rows)
78. {
79. **if** (row.getProcessName().equals(process))
80. {
81. **return** row;
82. }
83. }
85. **return** **null**;
86. }
88. **public** List<Row> getRows()
89. {
90. **return** rows;
91. }
93. **public** List<Event> getTimeline()
94. {
95. **return** timeline;
96. }
98. **public** **abstract** **void** process();
99. }

以CPUScheduler  为例，该类将按表格输入的数据进行封装，初始化时间片轮转，通过持有数据引用的方式对合法数据进行进程化抽象，进行算法判断。同时提供接口查询数据。该类对输入数据的抽象使得后续的算法设计简单化。

## 5.5.详细算法设计

FCFS（先来先服务）：

* 概念：按照作业到达的顺序进行调度，即先到先服务。当一个作业进入就绪队列时，如果CPU处于空闲状态，则该作业立即被调度执行；否则，作业需要等待前面所有作业执行完成后才能执行。
* 平均周转时间（Average Turnaround Time）：所有作业完成执行所需时间的平均值。
* 带权周转时间（Weighted Turnaround Time）：平均周转时间除以作业的执行时间。

SJF（短作业优先）：

* 概念：按照作业的执行时间长度进行调度，即执行时间最短的作业优先执行。当一个作业进入就绪队列时，如果CPU处于空闲状态，且该作业的执行时间比当前正在执行的作业更短，则该作业被立即调度执行；否则，作业需要等待前面执行时间更短的作业执行完成后才能执行。
* 平均周转时间：所有作业完成执行所需时间的平均值。
* 带权周转时间：平均周转时间除以作业的执行时间。

SRTF（最短剩余时间优先）：

* 概念：是SJF的一种变体，在执行过程中，不断检查就绪队列中作业的剩余执行时间，如果有作业的剩余执行时间比当前正在执行的作业更短，则抢占当前作业执行更短的作业。
* 平均周转时间：所有作业完成执行所需时间的平均值。
* 带权周转时间：平均周转时间除以作业的执行时间。

PSP（优先级调度 - 抢占式）：

* 概念：每个作业都有一个优先级，优先级高的作业先执行。当一个作业进入就绪队列时，如果CPU处于空闲状态，或者该作业的优先级高于当前正在执行的作业，则抢占当前作业执行优先级更高的作业。
* 平均周转时间：所有作业完成执行所需时间的平均值。
* 带权周转时间：平均周转时间除以作业的执行时间。

PSNP（优先级调度 - 非抢占式）：

* 概念：与PSP类似，但不允许作业在执行过程中被抢占，即一旦作业开始执行，则直到执行完成或被阻塞才会释放CPU。
* 平均周转时间：所有作业完成执行所需时间的平均值。
* 带权周转时间：平均周转时间除以作业的执行时间。

RR（时间片轮转）：

* 概念：每个作业被分配一个固定长度的时间片（时间片轮转），作业在该时间片内执行，如果执行完毕则被移出就绪队列；如果时间片结束但作业仍未执行完毕，则该作业被暂时挂起，放到就绪队列末尾，等待下一轮时间片。这样轮流进行，直到所有作业执行完成。
* 平均周转时间：所有作业完成执行所需时间的平均值。
* 带权周转时间：平均周转时间除以作业的执行时间。

总体而言，这些调度算法在不同场景下有各自的优缺点。FCFS简单易实现，但可能导致平均等待时间较长；SJF和SRTF可以减少平均等待时间，但可能导致长作业被饿死；PSP和PSNP允许根据优先级调度，但可能导致低优先级作业长时间等待；RR适用于时间片短且变化不大的情况，但可能导致上下文切换频繁。

在实现的过程中，抢占式算法的实现采用了观察者的理念动态的计算在不同调度算法下的“优先权值”，从而实现了调度算法中的抢占式。

# 六、总结与心得

很可惜这次并没有找到合适的共同开发的队友丧失了这次合作的经历十分遗憾。或许这是因为我转专业学生的缘故，或许是因为我自身的原因。但这次的实验仍然让我学到了许多，我一个人完成了整体的软件文档写作，也对软件管理，维护，更新有了更深入的了解。在提交代码到开源社区，真正参与到开源社区的维护中后，我更加认为软件开发并非一个人的游戏，在互联网上，开源社区里也一直有着开放，包容，友善，共享的互联网精神。Git技术的发扬光大更拓展了这一精神。

在合作的过程中，我尤其体会到标准、文档，这些在理工科“人轻事微”的事物实际上并不是那么简单和微不足道。事实上，他们十分重要，正如我给自己起的队名“DDL驱动模型”，在个人的开发中以及合作中，我们常常会遇到这样的困境，不知从何做起，不知谁来负责，最后一拖再拖，直到DDL前才恍然醒悟。因此，一个“第一驱动力”便显得格外重要，软件工程很多时候便是充当了这么一个角色，一个分化DDL的角色。

软件工程是一门涉及软件开发、维护和管理的学科，其目标是通过系统化的方法来提高软件的质量、效率和可靠性。在实验中，我们主要探讨了软件工程中的几个关键方面：

首先，软件需求分析是软件开发的关键起点。在需求分析阶段，我们学习了如何识别、理解和管理用户的需求和期望。这包括需求的获取、分析、规格化以及需求验证的过程。通过合理的需求分析，可以有效地降低后续开发过程中的变更和错误，提升软件开发的成功率和用户满意度。

其次，软件设计是将需求转化为可执行方案的关键步骤。我们研究了不同的设计模式和架构风格，如面向对象设计、分层架构等。良好的软件设计不仅仅是代码的组织，更是系统的结构化表达，有助于提高系统的可维护性和扩展性，减少潜在的设计缺陷。

接着，软件测试是确保软件质量的重要手段。我们学习了测试策略的制定、测试用例的设计和执行，以及缺陷管理的方法。通过充分的测试覆盖和高质量的测试用例，可以有效地发现和修复软件中的问题，确保软件在交付前具备较高的稳定性和可靠性。

最后，软件项目管理是协调和管理软件开发过程中资源、进度和风险的关键活动。我们探讨了敏捷开发、迭代开发等项目管理方法论，并了解了项目计划、进度跟踪、团队协作等方面的实际操作。良好的项目管理能够帮助团队有效地控制开发过程，及时应对变更和风险，确保项目按时交付并达到预期的质量标准。

总结而言，通过实验我们深入了解了软件工程中各个阶段的重要性和相互关联性。每个阶段的有效执行和良好的协同作用是保证软件开发成功的关键因素，也是提升软件质量和开发效率的重要保障。