

《操作系统》**课程设计报告**

|  |  |
| --- | --- |
| **课程设计题目：** | **进程调度和同步算法的实现** |
| **专 业：** | 软件工程 |
| **年 级：** | 2017级 |
| **团队成员：** | YA1714180-许东明 |
| **（学号姓名）** | YA1714036-方俊翔 |
|  | YA1714037-汪鹏 |
|  | YA1714052-袁兴武 |
|  |  |
|  |  |
| **指导教师：** | 徐国明 |

**安徽大学互联网学院**

**2020年 3 月 11 日**

**进程调度与同步算法的实现**

**摘要:** 无论是在[批处理系统](https://baike.baidu.com/item/%E6%89%B9%E5%A4%84%E7%90%86%E7%B3%BB%E7%BB%9F/3702274" \t "https://baike.baidu.com/item/%E8%BF%9B%E7%A8%8B%E8%B0%83%E5%BA%A6/_blank)还是[分时系统](https://baike.baidu.com/item/%E5%88%86%E6%97%B6%E7%B3%BB%E7%BB%9F/9706663" \t "https://baike.baidu.com/item/%E8%BF%9B%E7%A8%8B%E8%B0%83%E5%BA%A6/_blank)中，用户进程数一般都多于[处理机](https://baike.baidu.com/item/%E5%A4%84%E7%90%86%E6%9C%BA/128842" \t "https://baike.baidu.com/item/%E8%BF%9B%E7%A8%8B%E8%B0%83%E5%BA%A6/_blank)数、这将导致它们互相争夺处理机。另外，系统进程也同样需要使用[处理机](https://baike.baidu.com/item/%E5%A4%84%E7%90%86%E6%9C%BA/128842" \t "https://baike.baidu.com/item/%E8%BF%9B%E7%A8%8B%E8%B0%83%E5%BA%A6/_blank)。这就要求进程调度程序按一定的策略，动态地把[处理机](https://baike.baidu.com/item/%E5%A4%84%E7%90%86%E6%9C%BA/128842" \t "https://baike.baidu.com/item/%E8%BF%9B%E7%A8%8B%E8%B0%83%E5%BA%A6/_blank)分配给处于就绪队列中的某一个[进程](https://baike.baidu.com/item/%E8%BF%9B%E7%A8%8B/382503" \t "https://baike.baidu.com/item/%E8%BF%9B%E7%A8%8B%E8%B0%83%E5%BA%A6/_blank)，以使之执行。进程同步是一个操作系统级别的概念,是在多道程序的环境下，存在着不同的制约关系，为了协调这种互相制约的关系，实现资源共享和进程协作，从而避免进程之间的冲突，引入了进程同步。

**关键词:**处理机，进程调度，进程同步，操作系统

**Abstract**

No matter in batch system or time-sharing system, the number of user processes is generally more than the number of processors, which will cause them to compete for each other's processors. In addition, the system process also needs to use a processor. This requires the process scheduler to dynamically allocate the processor to a process in the ready queue according to a certain strategy so that it can execute. Process synchronization is an operating system level concept. In the environment of multi-channel programs, there are different constraints. In order to coordinate these constraints, realize resource sharing and process collaboration, and avoid conflicts between processes, process synchronization is introduced.

**Key words:** processor, process scheduling, process synchronization, operating system

目录

[1 课题介绍 1](#_Toc16245)

[1.1 课题简介 1](#_Toc27522)

[1.1.1 课题背景 1](#_Toc126)

[1.1.2 课题目的 1](#_Toc22987)

[1.1.3 要求及内容 1](#_Toc6674)

[1.2 理论依据 1](#_Toc30178)

[1.2.1 先来先服务算法 1](#_Toc18677)

[1.2.2 短作业优先调度算法 3](#_Toc9956)

[1.2.3 优先级调度算法 4](#_Toc10130)

[1.2.4 时间片轮转算法 5](#_Toc13898)

[1.2.5 多级反馈队列算法 6](#_Toc30542)

[1.2.6 哲学家进餐问题 7](#_Toc20075)

[1.2.7 读者/写者问题 8](#_Toc5509)

[2 设计简介及设计方案论述 8](#_Toc18962)

[2.1 设计简介 8](#_Toc333)

[2.2 设计方案论述 8](#_Toc17829)

[2.3 软件开发相关环境 9](#_Toc16500)

[3 概要设计 9](#_Toc9909)

[3.1 功能模块设计 9](#_Toc9559)

[3.1.1 功能模块图 9](#_Toc13557)

[3.1.2 模块的功能 9](#_Toc17535)

[3.2 数据结构设计 11](#_Toc30688)

[3.2.1 进程调度模块 11](#_Toc8241)

[3.2.2 哲学家进餐模块 12](#_Toc26263)

[3.2.3 读者写者模块 15](#_Toc325)

[4 详细设计 17](#_Toc9160)

[4.1 进程调度管理模块 17](#_Toc28682)

[4.1.1 主要方法函数、参数 17](#_Toc3473)

[4.1.2 新建进程控制块PCB窗体中的点击创建按钮触发事件 17](#_Toc5380)

[4.1.3 主窗体MainFrame中阻塞菜单触发事件 18](#_Toc27073)

[4.1.4 主窗体MainFrame中点击删除菜单触发事件 19](#_Toc18620)

[4.1.5 主窗体MainFrame中点击暂停计时按钮触发事件 19](#_Toc30951)

[4.1.6 主窗体MainFrame中点击继续计时按钮触发事件 19](#_Toc4865)

[4.1.7 主窗体MainFrame中点击重置按钮触发事件 19](#_Toc20237)

[4.1.8 主窗体类MainFrame中调度下一个进程开始运行方法startRunPcb 20](#_Toc7503)

[4.1.9 主窗体类MainFrame中addPriorityPcb和updatePriorityTable方法 21](#_Toc1807)

[4.1.10 主窗体类MainFrame中加入各个进程Table的方法AddTable 21](#_Toc31019)

[4.1.11 主窗体类MainFrame中唤醒进程的方法wakeWaitPcb() 22](#_Toc19953)

[4.1.12 主窗体类MainFrame中将进程属性加入Vector中方法CreateVector 22](#_Toc25278)

[4.1.13 时钟计时器类CpuColockTimer的count方法 22](#_Toc28068)

[4.1.14 时钟计时器类CpuColockTimer的DispatchPcb调度方法 23](#_Toc10491)

[4.2 哲学家进餐模块 24](#_Toc31502)

[4.2.1 主要方法函数 24](#_Toc28670)

[4.2.2 记录型信号量 24](#_Toc10591)

[4.2.3 最多只有四个哲学家同时就餐 25](#_Toc26526)

[4.2.4 哲学家就餐分奇偶 25](#_Toc5359)

[4.2.5 只有取到两把筷子才能吃饭 26](#_Toc7997)

[4.3 读者写者问题模块 27](#_Toc10818)

[4.3.1 主要方法函数 27](#_Toc10582)

[5 测试计划 29](#_Toc12767)

[5.1 进程调度管理 29](#_Toc5224)

[5.1.1 先来先服务调度 29](#_Toc4118)

[5.1.2 短作业优先调度 30](#_Toc11294)

[5.1.3 优先级调度算法 30](#_Toc32765)

[5.1.4 时间片轮转算法 30](#_Toc19695)

[5.1.5 多级反馈队列算法 30](#_Toc2840)

[5.2 哲学家进餐 30](#_Toc9613)

[5.3 读者写者问题 30](#_Toc19346)

[6 运行结果截图 32](#_Toc23756)

[6.1 项目启动程序 32](#_Toc13552)

[6.2 进程调度管理 33](#_Toc2044)

[6.3 哲学家进餐 34](#_Toc16289)

[6.4 读者写者问题 35](#_Toc31938)

[7 团队分工 36](#_Toc8939)

[8 小结 37](#_Toc5312)

[9 参考文献 37](#_Toc19938)

[10 附录：主要代码 38](#_Toc19016)

[课程设计成绩评定表 49](#_Toc8865)

# 课题介绍

## 课题简介

### 课题背景

操作系统在计算机科学中是一门核心专业基础课，涉及较多硬件和软件知识。在计算机软、硬件课程的设置上，它起着承上启下的作用。其特点是概念多、较抽象、涉及的知识面广。

### 课题目的

通过学习，使学生了解计算机操作系统的发展特点；理解操作系统的基本概念、组成和各部分的功能；掌握计算机操作系统设计的基本原理和方法；理解常见操作系统的结构原理并能熟练使用。

### 要求及内容

本次课程设计需要完成作业/进程调度算法的模拟与实现和进程同步算法的模拟与实现两个课题，作业/进程调度算法的模拟与实现这个课题需要了解操作系统中作业/进程的基本概念和常用的作业/进程调度算法，然后通过先来先服务、短作业优先、优先级、时间片轮转、多级队列反馈等算法实现作业的调度。进程同步算法这个课题需要了解临界区、进程同步、进程互斥等概念，掌握用信号量和PV操作实现进程互斥的方法。最后通过进程同步和进程互斥实现哲学家进餐问题算法，读者/写者问题的读者优先算法、写者优先算法和无优先算法。

## 理论依据

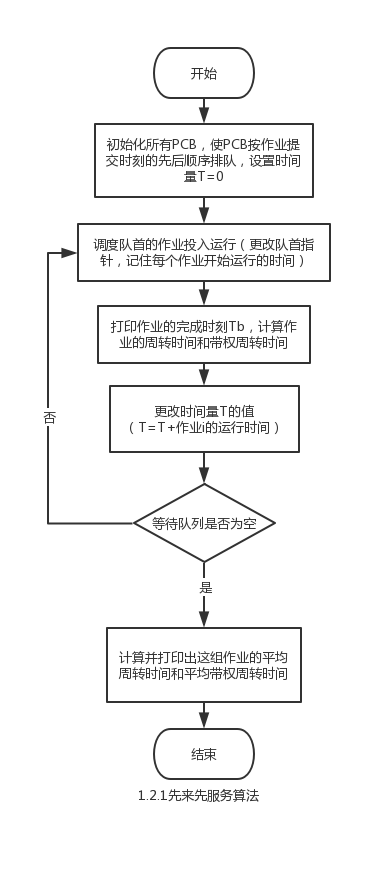
### 先来先服务算法

最简单的调度算法，既可以用于作业调度 ，也可以用于程序调度，当作业调度中采用该算法时，系统将按照作业到达的先后次序来进行调度，优先从后备队列中，选择一个或多个位于队列头部的作业，把他们调入内存，分配所需资源、创建进程，然后放入“就绪队列”,直到该进程运行到完成或发生某事件堵塞后，进程调度程序才将处理机分配给其他进程。 先来先服务调度算法：系统按照作业到达的先后次序来进行调度，或者说它优先考虑在系统中等待时间最长的作业，而不管该作业所需执行时间的长短，从后备作业队列中选择几个最先进入该队列的作业，将它们调入内存，为它们分配资源和创建进程。然后把它放入就绪队列。当进程调度中才有FCFS算法时，每次调度是从就绪的进程队列中选择一个最先进入该队列的进程，为之分配处理机，使之投入运行。该进程一直运行到完成或发生某事件而阻塞后，进程调度程序才将处理机分配给其他进程。

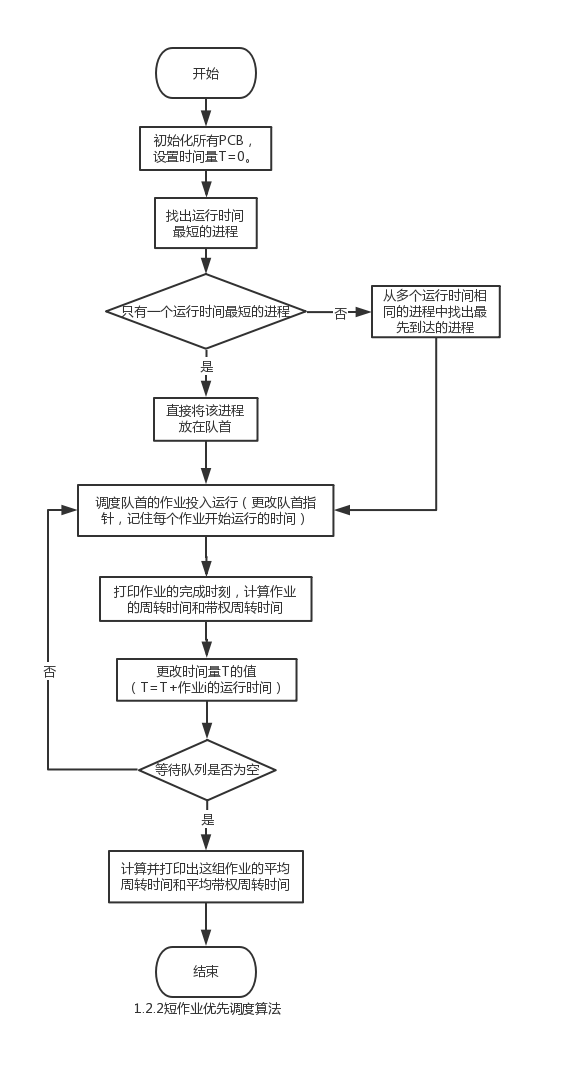
CPU繁忙型作业是指该类作业需要大量的CPU时间进行计算，而很少请求I/O。通常的科学计算便属于CPU繁忙型作业。

I/O繁忙型作业是指CPU进行处理时需频繁地请求I/O。目前的大多数事务处理都属于I/O繁忙型作业。

先到先服务很容易理解。先来的程序先执行，执行完毕后让出CPU给接下来到来的程序。这种策略可以使用FIFO的队列来容易实现。这样的调度策略实现简单，但是它的平均等待时间一般是较长的。更为糟糕的是，当正在执行的程序是一个有许多I/O操作的进程，那么这将导致CPU在空闲的时间里无法运行后面等待的进程，造成了CPU空闲。FCFS策略是非抢占式的，一旦CPU分配给某个进程，那么直到该进程结束之前，CPU都是属于该进程的。从CPU利用率这个指标来评估FCFS，它并不是一个很好的调度策略。



### 短作业优先调度算法

短作业优先（SJF, Shortest Job First）又称为“短进程优先”[SPN](https://baike.baidu.com/item/SPN" \t "_blank)(Shortest Process Next)；这是对FCFS算法的改进，其目标是减少平均周转时间。短作业优先算法的优点有：比FCFS改善平均[周转时间](https://baike.baidu.com/item/%E5%91%A8%E8%BD%AC%E6%97%B6%E9%97%B4" \t "_blank)和平均[带权周转时间](https://baike.baidu.com/item/%E5%B8%A6%E6%9D%83%E5%91%A8%E8%BD%AC%E6%97%B6%E9%97%B4" \t "_blank)，缩短作业的等待时间；提高系统的[吞吐量](https://baike.baidu.com/item/%E5%90%9E%E5%90%90%E9%87%8F" \t "_blank)；缺点有，对长作业非常不利，可能长时间得不到执行；未能依据作业的紧迫程度来划分执行的优先级；难以准确估计作业（进程）的执行时间，从而影响调度性能。短作业优先调度算法：SJF算法是以作业的长短来计算优先级，作业越短，其优先级越高。作业的长短是以作业所要求的运行时间来衡量的。SJF算法可以分别用于作业调度和进程调度。在把短作业优先调度算法用于作业调度时，它将从外存的作业后备队列中选择若干个估计运行时间最短的作业，优先将它们调入内存运行。

该算法对长作业不利，更严重的是，如果有一长作业(进程)进入系统的后备队列(就绪队列)，由于调度程序总是优先调度那些(即使是后进来的)短作业(进程)，将导致长作业(进程)长期不被调度。

该算法完全未考虑作业的紧迫程度，因而不能保证紧迫性作业(进程)会被及时处理。

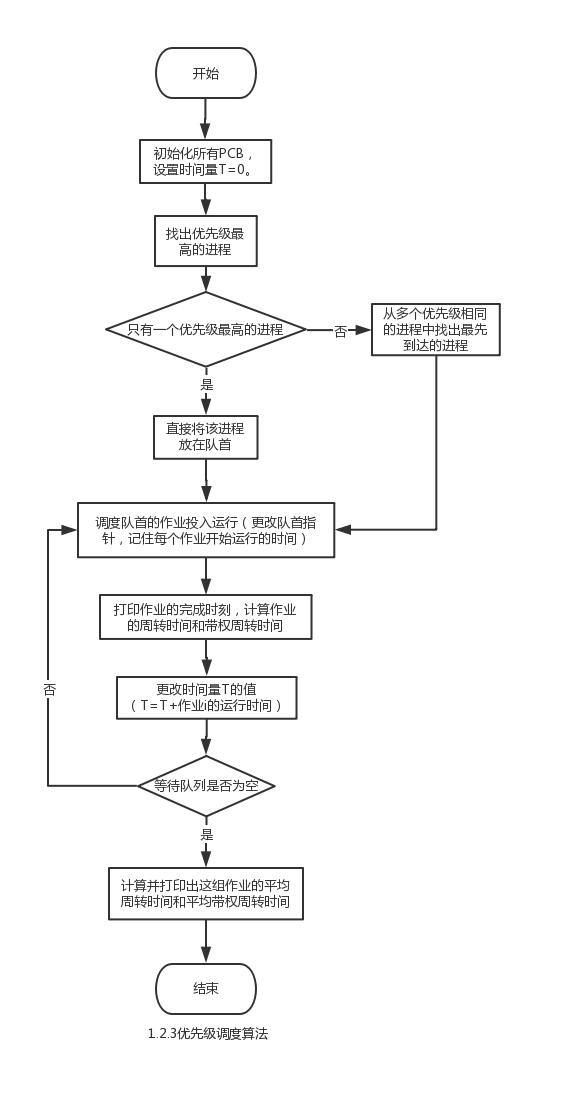
由于作业(进程)的长短只是根据用户所提供的估计执行时间而定的，而用户又可能会有意或无意地缩短其作业的估计运行时间，致使该算法不一定能真正做到短作业优先调度。

最短作业调度是将后续具有最短处理时间的进程先放到CPU上运行，如果就绪队列中有同样长度的进程，那么它们之间是采用FCFS调度的。最短下一个CPU区间，需要操作系统知道接下来是那个进程的CPU区间最短。SJF就是调度这个最短CPU区间的进程。SJF算法具有最短的平均等待时间，它是最佳的调度算法。但是SJF面对的难题是恐怖的，那就是操作系统是如何获知后面就绪队列中哪一个进程具有最短的CPU区间。对于一个批处理系统而言，这不是问题，因为用户会设定进程执行时间。但是现代的操作系统是多任务的交互式系统，操作无法获知下一个CPU区间的长度，我们只能去近似SJF，而不能做到SJF。近似的方法就是去估计，预测它的值。可以认为下一个CPU区间的长度和以前的相似。下一个CPU区间通常可以预测为以前CPU区间的测量长度的指数平均。

### 优先级调度算法

SJF算法可以看做是时间优先级的一种优先级调度算法。在现代的操作系统中，每一个进程都会有一个优先级与其相关。具有最高优先级的进程会被分配到CPU。具有相同优先级的进程按照FCFS算法调度。优先级可以通过内部或者外部方式来定义。优先级调度可以是可抢占的或者非抢占的。

优先级调度算法的一个主要问题是无穷阻塞问题（饥饿）。优先级调度会使得低优先级的进程可能在无穷等待CPU的到来。通常这种情形发生的时候，可能就会导致很严重的后果。老化是一种解决该问题的方案，老化以逐渐增加在系统中等待很长时间的进程的优先级。



### qt_temp时间片轮转算法

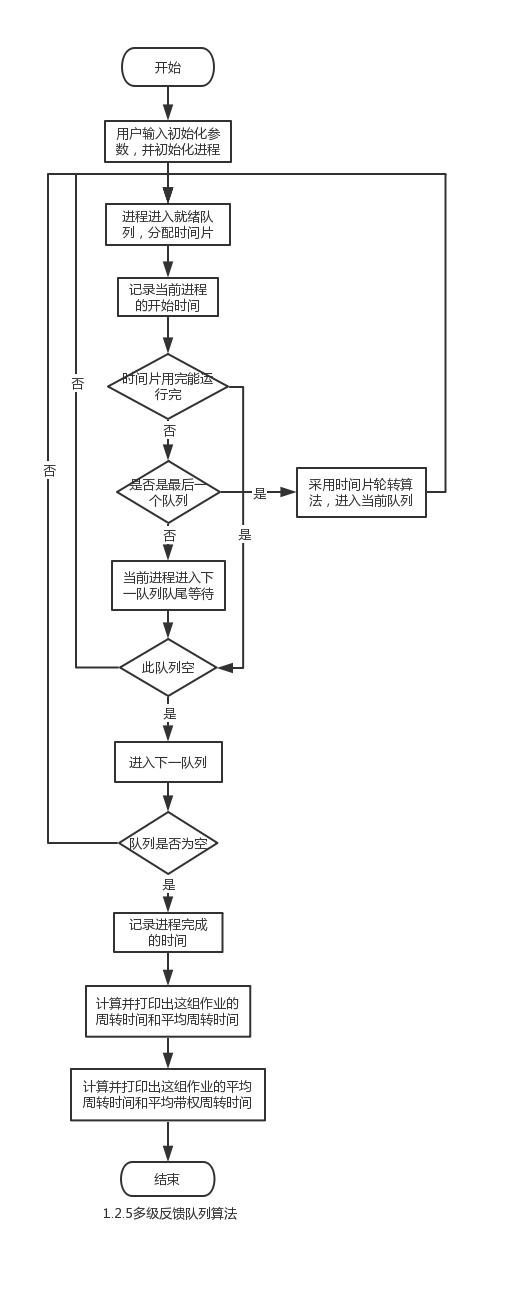
在早期的时间片轮转法中，系统将所有的就绪进程按先来先服务的原则排成一个队列，每次调度时，把CPU分配给队首进程，并令其执行一个时间片。时间片的大小从几ms到几百ms。当执行的时间片用完时，由一个计时器发出时钟中断请求，调度程序便据此信号来停止该进程的执行，并将它送往就绪队列的末尾；然后，再把处理机分配给就绪队列中新的队首进程，同时也让它执行一个时间片。这样就可以保证就绪队列中的所有进程在一给定的时间内均能获得一时间片的处理机执行时间。换言之，系统能在给定的时间内响应所有用户的请求。

时间片轮转法是专门为分时系统设计的。它是现代的桌面系统，服务器系统广泛采用的一种调度策略。它定义一个较小的时间单元，称为时间片。CPU调度程序为每个进程分配不超过一个时间片间隔的CPU时间。时间片轮转算法的关键在于时间片大小的选择。

RR算法的性能基本取决于时间片设计的大小是否合理。时间片设计的过大，将会导致响应很慢。这时候的RR算法可能就接近于FCFS算法。同样，如果时间片设计的很小，那么RR算法可以看做是处理器共享的。但是这样会使得每个进程的处理速度都下降到1/n（假设n个进程），上下文切换对于RR算法的影响也是非常大，频发的在进程之间切换可能会导致开销的时间占比很大，相应的进程实际的执行时间就会被压缩。所以时间片的大小选择是RR算法的核心问题。

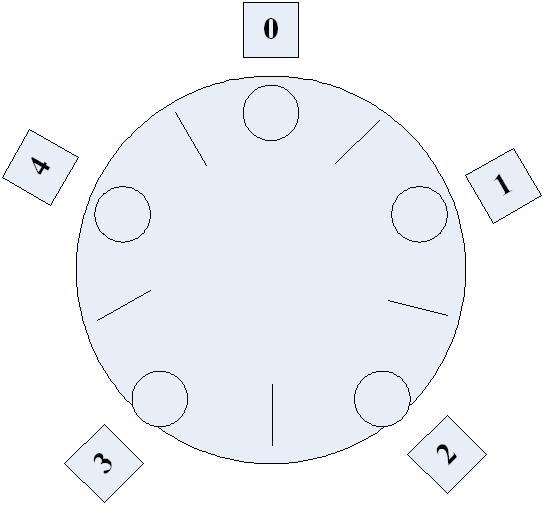
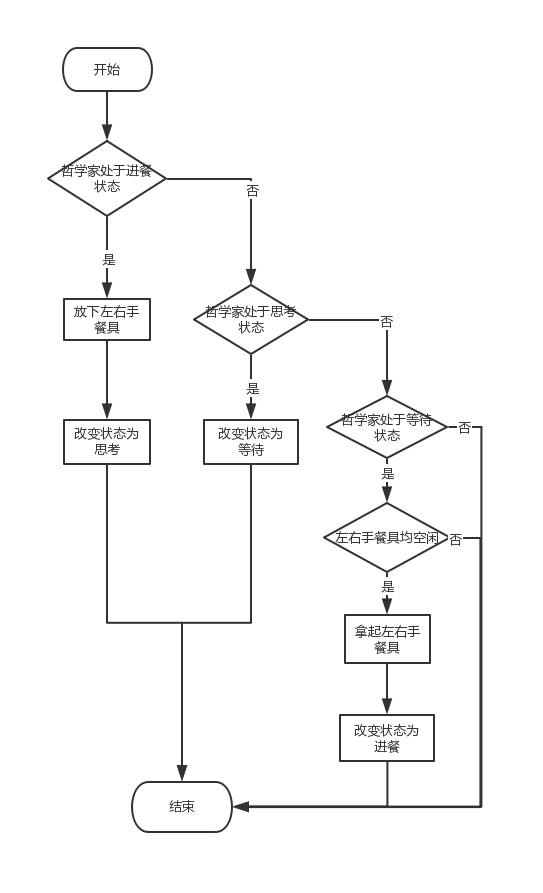
### 多级反馈队列算法

对于多级队列调度算法，进程会被永久的分配至某个队列。这样不够灵活。多级反馈队列调度允许进程在队列之间移动。它依据不同CPU区间特点来划分进程。如果进程使用的CPU时间过多，那么它将会被移到更低优先级的队列。目的是将I/O约束和交互式进程留在较高的优先级。以增加系统的响应。当然了，当它在低优先级等待的时间过长了，老化算法就会让它提升优先级。这样，进程的优先级就会是一个动态的变化。



### 哲学家进餐问题

有五个哲学家，他们的生活方式是交替地进行思考和进餐。哲学家们共用一张园桌，分别坐在周围五张椅子上。在圆桌上五支筷子，平时一个哲学家进行思考，饥饿时便试图取用其左右最靠近他的筷子，只有在他拿到两支筷子时才能进餐。进程毕，放下筷子又继续思考。



### 读者/写者问题

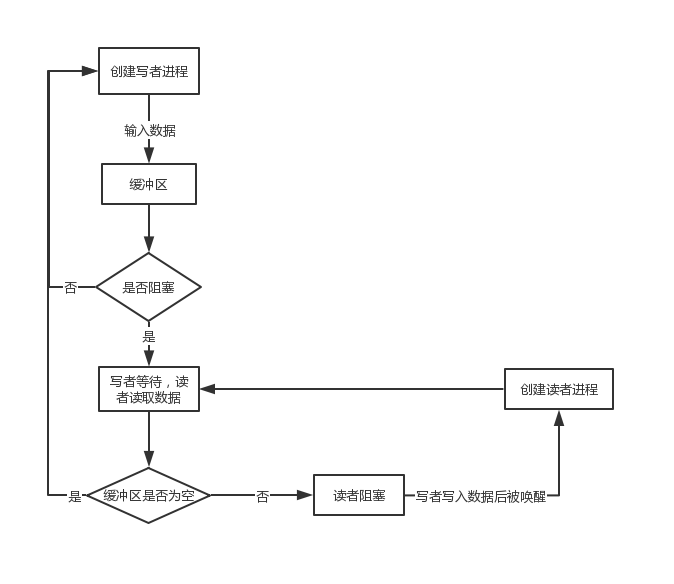
有读者和写者两组并发进程，共享一个文件，当两个或以上的读进程同时访问共享数据时不会产生副作用，但若某个写进程和其他进程（读进程或写进程）同时访问共享数据时则可能导致数据不一致的错误。

因此要求：

①允许多个读者可以同时对文件执行读操作；

②只允许一个写者往文件中写信息；

③任一写者在完成写操作之前不允许其他读者或写者工作；

④写者执行写操作前，应让已有的读者和写者全部退出。

# 设计简介及设计方案论述

## 设计简介

本程序是完成作业/进程调度算法的模拟与实现和进程同步算法的模拟与实现。在程序出现菜单界面之后用户输入此程序所需要的各种参数，然后选择要实现的算法，此程序就可以将通过算法计算出的结果通过界面呈现给用户。该程序可以实现进程调度中的先来先服务、短作业优先、优先级、时间片轮转、多级队列反馈等算法，实现进程同步中的哲学家进餐问题算法，读者/写者问题的读者优先算法、写者优先算法和无优先算法。

## 设计方案论述

我们通过把这个项目拆分成4个子模块来进行项目开发，分别为操作系统课程设计启动程序、进程调度管理模块、哲学家进餐模块、读者写者模块；然后对每个模块原理进行分析产生大体项目流程框架，然后我们使用java可视化组件开发，设计好一个比较美观的UI界面。

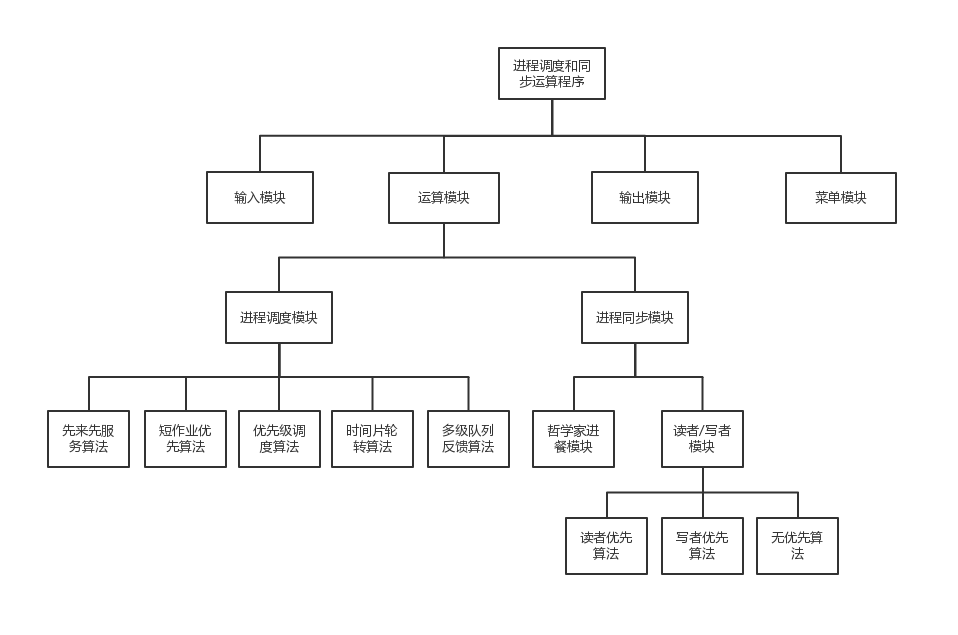
## 软件开发相关环境

* 操作系统：Windows 10 64bit
* JDK环境：Java SE Development Kit（JDK）Version 8
* 开发工具：Eclipse3.4
* 运行环境：Java SE Runtime Environment（JRE）Version 8

# 概要设计

## 功能模块设计

### 功能模块图



### 模块的功能

1. 进程调度管理模块
2. 实现先来先服务算法模块

按进入就绪队列的先后作为进程执行的顺序的思想进行调度。

1. 实现时间片轮转法模块

将CPU的处理时间分成固定大小的时间片。如果一个进程在被调度选中之后用完了系统规定的时间片，但未完成要求的任务，则它自行释放自己所占用的CPU而派到就绪队列1的末尾，等待下依次调度。同时，进程调度程序又去调度当前就绪队列中的第一个进程或作业。时间片可以自己设定，默认为5；

1. 实现多级反馈轮转法模块

首先设置两个就绪队列，并为各个队列赋予不同的优先权。第一个队列的优先权最高，第二次之。一个新进程进入内存后，首先将它放入第一个队列末尾，按FCFS的原则调度。该进程如果在一个时间片内没有执行完毕，则转入第二个队列的末尾，同样地再按照FCFS被调度执行，在最后的队列中则按照轮转发进行调度执行，直到进程运行结束。只有当优先权高的第一个队列为空时，才能调度优先权次高队列中的进程。

1. 实现静态优先级法模块

就绪作业队列中挑选优先级高、且资源能够得到满足的进程进入内存投入运行。

1. 实现短作业优先算法

SJF算法可以分别用于作业调度和进程调度。在把短作业优先调度算法用于作业调度时，它将从外存的作业后备队列中选择若干个估计运行时间最短的作业，优先将它们调入内存运行。

1. 实现阻塞进程模块

在执行进程表中先选中表中的一行，右击弹出子菜单，点击阻塞，则将正在执行的进程调入等待进程表中。

1. 实现删除进程模块

在任意进程表中先选中一行，右击弹出子菜单，点击删除，则将该进程从进程队列中删除掉。

1. 暂停cpu计时器模块

点击暂停按钮则cpu暂停执行进程和计时器计时。

1. 恢复cpu计时器模块

点击继续按钮则cpu继续执行进程和计时器计时。

1. 创建进程模块

点击创建按钮则填写新进程的信息并加入到就绪进程队列中。

1. 重置系统模块

点击重置按钮则重新运行程序，计时器从零开始计时，进程队列置空。

1. cpu计时器模块

主要用于计时和调度进程运行，每过一秒判断当前执行态的进程是否用完了时间片和是否执行完成。

1. 哲学家进餐
2. 选择执行方式模块

该模块有自动执行与手动执行两种类型。

1. 记录型信号量模块

选择记录型信号量，每个哲学家每次就餐总是先拿起左边的筷子再拿起右边的筷子，只有两把筷子同时拿起两把筷子才可以就餐，当5个哲学家同时都拿起左边筷子时可能会出现死锁。

1. AND信号量机制模块

选择使用AND信号量机制，利用AND 型信号量机制实现：根据课程讲述，在一个原语中，将一段代码同时需要的多个临界资源，要么全部分配给它，要么一个都不分配，因此不会出现死锁的情形。

1. 奇左偶右算法模块

选择奇左偶右算法，规定奇数号的哲学家先拿起他左边的筷子,然后再去拿他右边的筷子;而偶数号 的哲学家则相反.按此规定,将是1,2号哲学家竞争1号筷子,3,4号哲学家竞争3号筷子.即 五个哲学家都竞争奇数号筷子,获得后,再去竞争偶数号筷子,最后总会有一个哲学家能获 得两支筷子而进餐。

1. 最多4个哲学家就餐模块

选择最多4个哲学家就餐，每次最多只有四个哲学家申请筷子，当第五个饥饿时需等待至少其他四个中的一个吃完才可以就餐。

1. 手动 允许死锁

只对筷子进行信号量限制，五位哲学家可以同时竞争筷子，当五位哲学家都去左手边或者右手边的筷子时会产生死锁。

1. 手动 两只筷子同时取

此方法是在哲学家去左边筷子之前先判断右边筷子是否被用，如果被用则无法获得筷子。当哲学家拿起筷子做筷子时，右边筷子也被同时征用，那么哲学家将放下左边筷子。反之亦然。

1. 读者写者问题
2. 该模块首先选择是读者优先还是写作优先；
3. 选择读者优先后，根据用户添加读操作或者写操作；
4. 点击开始功能按键，程序进行一系列操作模拟；

## 数据结构设计

### 进程调度模块

1. 进程结构类PCB

定义了进程各个状态的记录信息，结构如图，功能如表

|  |  |
| --- | --- |
| **属性** | **功能及作用** |
| pid | 进程标识号 |
| mem | 占用内存量 |
| Pritory | 优先级 |
| Status | 状态（等待，执行，就绪） |
| lastTime | 进程运行持续时间 |
| IuputTime | 进程创建时间 |
| endTime | 结束时间 |
| executeTime | 已执行时间 |

1. PCBFrame类

用来输入数据，创建PCB对象，对根据输入参数进行判断是否满足创建条件，结构如图

|  |  |
| --- | --- |
| **主要成员变量或方法** | **功能** |
| pidCount | 进程标识号计数器，每创建完一个进程，自动加1 |
| priorityComboBox | 用于设置进程的优先级大小 |
| PCBFrame() | 类PCBFrame的构造函数 |
| notInNewPcb(Vector<PCB>NewPcb,int pid) | 用来判断新建进程的标识号是否已存在 |
| showMsg(int type) | 根据参数type不同的值显示不同的提示信息 |

1. 时钟类CpuColockTimer

用于定时执行某个函数，进行进程的执行，并计时，结构如图

|  |  |
| --- | --- |
| **主要属性或方法** | **功能** |
| Timer | Java的时钟，用于定时触发事件 |
| Time | 用作基准时间 |
| timeSlice | 时间片，用于时间片轮转，多级反馈轮转法 |
| pauseCountTime | 用于控制计时器是否计时，false则计时，true 则停止计时 |
| DispatchPcb(int type) | 根据参数type不同的值调用不同的进程调度算法 |

1. MainFrame类

定义了整个程序运行的窗体和所需的属性，结果如图

|  |  |
| --- | --- |
| **主要属性和方法** | **功能** |
| timeSliceComboBox | 设置cpu时间片大小 |
| Vector<PCB> newPcb | 新建进程队列 |
| Vector<PCB> runPcb | 执行进程队列 |
| Vector<PCB> readyPcb1 | 就绪进程队列1 |
| Vector<PCB> readyPcb2 | 就绪进程队列2 |
| Vector<PCB> waitPcb | 等待进程队列 |
| JTable runTable | 执行进程Table，显示执行进程信息 |
| JTable readyTable1 | 就绪进程Table，显示就绪进程信息 |
| JTable readyTable2 | 就绪进程Table，显示就绪进程信息 |
| JTable waitTable | 等待进程Table，显示等待进程信息 |
| JLabel timeLabel | 显示计时时间 |
| CpuColockTimer CCT | cpu时钟计时器 |
| MainFrame() | 类MainFrame构造函数 |
| startRunPcb() | 开始运行下一个就绪或等待进程， |
| addPriorityPcb (Vector<PCB> readyPcb1, PCB pcb) | 根据进程优先级有序地插入到就绪队列1中，末尾的进程优先级最小 |
| addReadyTable(PCB p,int type) | 将进程加入就绪进程队列中 |
| addRunTable(PCB p) | 将进程加入执行进程队列中 |
| addWaitTable(PCB p) | 将进程加入等待进程队列中 |
| wakeWaitPcb() | 唤醒等待进程队列到就绪进程队列 |
| updatePriorityTable() | 更新就绪进程Table，先清空Table，再将有序的就绪进程队列加入Table中 |
| createVector(PCB p) | 将进程p 所有属性加入Vector向量v中 |
| resetOs() | 重置系统 |

### 哲学家进餐模块

1. 筷子类chopsticks

定义了进程各个状态的记录信息，结构如图，功能如表

|  |  |
| --- | --- |
| **主要属性和方法** | **功能及作用** |
| i | 筷子标号 |
| empty | 为空时图片 |
| P() | P操作 |
| V() | V操作 |

1. Eat类

手动--允许死锁

|  |  |
| --- | --- |
| **主要属性和方法** | **功能及作用** |
| Eat(String) | 基本界面和监听器 |
| changef(JButton) | 设置按钮控件不可用 |
| changet(JButton) | 设置按钮控件可用 |

1. Eat2类

手动--同时拿起两把筷子

|  |  |
| --- | --- |
| **主要属性和方法** | **功能及作用** |
| Eat2(String) | 基本界面和监听器 |
| changef(JButton) | 设置按钮控件不可用 |
| changet(JButton) | 设置按钮控件可用 |

1. interface1类

最多允许四个哲学家同时进餐

|  |  |
| --- | --- |
| **主要属性和方法** | **功能及作用** |
| interface1() | 基本界面和监听器 |
| c[] | chopsticks实体类数组（筷子进程） |
| p[] | philoDemo1实体类数组（哲学家进程） |
| cp[] | 五支筷子的图片组 |
| empty | 空的图 |
| tp | 思考中的图 |
| sp | 饥饿中的图 |
| ep | 吃饭中的图 |

1. philoDemo1类

最多允许四个哲学家同时进餐——哲学家进程

|  |  |
| --- | --- |
| **主要属性和方法** | **功能及作用** |
| philoDemo1(int,JLabel,chopsticks,JLabel,chopsticks,JLabel,ImageIcon,ImageIcon,JLabel,JLabel,JTextArea) | 基本界面元素替换 |
| toString() | 字符串处理 |
| P() | 类级别锁,P操作 |
| V() | 类级别锁,V操作 |
| think() | 思考 |
| eat() | 吃饭 |
| run() | 运行线程 |

1. Interface2类

奇数号先拿左边筷子，偶数号先拿右边筷子

|  |  |
| --- | --- |
| **主要属性和方法** | **功能及作用** |
| Interface2() | 基本界面和监听器 |
| c[] | chopsticks实体类数组（筷子进程） |
| p[] | philoDemo1实体类数组（哲学家进程） |
| cp[] | 五支筷子的图片组 |
| empty | 空的图 |
| tp | 思考中的图 |
| sp | 饥饿中的图 |
| ep | 吃饭中的图 |

1. philoDemo2类

奇数号先拿左边筷子，偶数号先拿右边筷子——哲学家进程

|  |  |
| --- | --- |
| **主要属性和方法** | **功能及作用** |
| philoDemo2(int,JLabel,chopsticks,JLabel,chopsticks,JLabel,ImageIcon,ImageIcon,JLabel,JLabel,JTextArea) | 基本界面元素替换 |
| toString() | 字符串处理 |
| P() | 类级别锁,P操作 |
| V() | 类级别锁,V操作 |
| think() | 思考 |
| eat() | 吃饭 |
| run() | 运行线程 |

1. Interface3类

要么同时拿起，要么不拿

|  |  |
| --- | --- |
| **主要属性和方法** | **功能及作用** |
| Interface3() | 基本界面和监听器 |
| c[] | chopsticks实体类数组（筷子进程） |
| p[] | philoDemo1实体类数组（哲学家进程） |
| cp[] | 五支筷子的图片组 |
| empty | 空的图 |
| tp | 思考中的图 |
| sp | 饥饿中的图 |
| ep | 吃饭中的图 |

1. philoDemo3类

要么同时拿起，要么不拿——哲学家进程

|  |  |
| --- | --- |
| **主要属性和方法** | **功能及作用** |
| philoDemo3(int,JLabel,chopsticks,JLabel,chopsticks,JLabel,ImageIcon,ImageIcon,JLabel,JLabel,JTextArea) | 基本界面元素替换 |
| toString() | 字符串处理 |
| P() | 类级别锁,P操作 |
| V() | 类级别锁,V操作 |
| think() | 思考 |
| eat() | 吃饭 |
| run() | 运行线程 |

1. Interface4类

记录型型号量（允许死锁）

|  |  |
| --- | --- |
| **主要属性和方法** | **功能及作用** |
| Interface4() | 基本界面和监听器 |
| c[] | chopsticks实体类数组（筷子进程） |
| p[] | philoDemo1实体类数组（哲学家进程） |
| cp[] | 五支筷子的图片组 |
| empty | 空的图 |
| tp | 思考中的图 |
| sp | 饥饿中的图 |
| ep | 吃饭中的图 |

1. philoDemo4类

记录型型号量（允许死锁）——哲学家进程

|  |  |
| --- | --- |
| **主要属性和方法** | **功能及作用** |
| philoDemo4(int,JLabel,chopsticks,JLabel,chopsticks,JLabel,ImageIcon,ImageIcon,JLabel,JLabel,JTextArea) | 基本界面元素替换 |
| toString() | 字符串处理 |
| P() | 类级别锁,P操作 |
| V() | 类级别锁,V操作 |
| think() | 思考 |
| eat() | 吃饭 |
| run() | 运行线程 |

1. Menu

开始、功能选择界面

|  |  |
| --- | --- |
| **主要属性和方法** | **功能及作用** |
| m0 | interface1类实体，最多允许4哲学家同时进餐 |
| m1 | Interface2类实体，奇左偶右 |
| m2 | Interface3类实体，AND信号量机制 |
| m3 | Interface4类实体，记录型型号量 |
| eat | Eat类实体，手动--允许死锁 |
| Eat2 | Eat2类实体，手动--两只筷子同时取 |
| Menu() | 基本界面和监听器 |

### 读者写者模块

1. MainJFrame类

基本界面

|  |  |
| --- | --- |
| **主要属性和方法** | **功能及作用** |
| MainJFrame() | 调用initComponents方法和其他初始化设置 |
| initComponents() | 基本界面和监听器 |
| count | 记录数量 |
| choose | 记录当前状态，读者优先OR写者优先 |

1. ReadWriteLock类

读写锁

|  |  |
| --- | --- |
| **主要属性和方法** | **功能及作用** |
| aa | ResultJFrame类实体化 |
| readingThreads | 读者线程数 |
| writingThreads | 写者线程数 |
| waitingThreads | 等待线程数 |
| rreadlock(String) | 读线程等待 读优先 |
| wreadlock(String) | 读线程等待 写优先 |
| readunlock(String) | 读线程释放 |
| writelock(String) | 写线程等待 |
| writeunlock(String) | 写线程释放 |

1. myThread类

线程

|  |  |
| --- | --- |
| **主要属性和方法** | **功能及作用** |
| Lock | ReadWriteLock类实体化 |
| readcount | 读者数 |
| writecount | 写者数 |
| myThread(String,String,boolean) | 初始化 |
| run() | 运行 |

1. ResultJFrame类

结果窗口

|  |  |
| --- | --- |
| **主要属性和方法** | **功能及作用** |
| ResultJFrame() | 调用initComponents方法和其他初始化设置 |
| initComponents() | 基本界面和监听器 |
| start() | 显示ResultJFrame窗口 |
| fill(String) | 填充记录文本 |
| zero() | 清空文本 |

# 详细设计

## 进程调度管理模块

### 主要方法函数、参数

1. **int** pid=0; //进程标识号
2. **int** mem=0; //空间大小
3. **int** priority=0;//优先级
4. String status="";//当前状态
5. **int** lastTime=0;//持续时间
6. **int** inputTime=0;//提交时间
7. **int** endTime=0;//完成时间
8. **int** executeTime=0;//已执行时间
9. **public** **int** getPid();                        //获得进程标识号
10. **public** **void** setPid(**int** pid);                //设置进程标识号
11. **public** **int** getPriority();                   //获得优先级
12. **public** **void** setPriority(**int** priority);      //设置优先级
13. **public** **int** getLastTime();                   //获得持续时间
14. **public** **void** setLastTime(**int** lastTime);      //设置持续时间
15. **public** **int** getInputTime();                  //获得提交时间
16. **public** **void** setInputTime(**int** inputTime);    //设置提交时间
17. **public** **int** getEndTime();                    //获得完成时间
18. **public** **void** setEndTime(**int** endTime);        //设置完成时间
19. **public** **int** getExecuteTime();                //获得已执行时间
20. **public** **void** setExecuteTime(**int** executeTime);//设置已执行时间
21. **public** String getStatus();                  //获得当前状态
22. **public** **void** setStatus(String status);       //设置当前状态

### 新建进程控制块PCB窗体中的点击创建按钮触发事件

1. //监听点击创建按钮事件
2. createButton.addActionListener(**new** ActionListener(){
3. **public** **void** actionPerformed(ActionEvent e) {
4. PCB pcb=**new** PCB();
5. pcb.setPid(Integer.parseInt(pidTextField.getText()));
6. pcb.setPriority(Integer.parseInt(priorityComboBox.getSelectedItem().toString()));
7. pcb.setMem(Integer.parseInt(memTextField.getText()));
8. pcb.setLastTime(Integer.parseInt(lastTimeTextField.getText()));
9. pcb.setExecuteTime(0);
10. **if**(yesRadioButton.isSelected())
11. {
12. pcb.setPrinter(Integer.parseInt(PrinterNumTextField.getText()));
13. pcb.setPrinterRequestTime(Integer.parseInt(printerReqTimeTextField.getText()));
14. }
15. **if**(!notInNewPcb(MainFrame.newPcb,pcb.getPid()))
16. showMsg(0);//进程标识号已存在
17. **else** **if**(pcb.getMem()==0)
18. showMsg(1);//进程占内存大小不能为0
19. **else** **if**(pcb.getMem()>PCB.comMemory)
20. showMsg(2);//进程大小超过系统最大内存空间
21. **else** **if**(pcb.getLastTime()==0)
22. showMsg(3);//进程持续时间不能为0
23. **else**    //满足创建条件，加入就绪进程队列和就绪Table
24. {
25. pcb.setInputTime(CpuColockTimer.time);
26. pcb.setStatus("就绪态");
27. //PCB.comMemory=PCB.comMemory-pcb.getMem();
28. MainFrame.newPcb.add(pcb);  //加入新进程newPcb
29. **if**(MainFrame.staticPriorityRadioB.isSelected())//如果是按优先级调度算法，则根据新建的进程有序的插入到就绪队列中
30. {
31. MainFrame.addPriorityPcb(MainFrame.readyPcb1,pcb);
32. MainFrame.updatePriorityTable();
33. }
34. **else**  //其他调度算法，则统一加入就绪队列1中
35. {
36. MainFrame.readyPcb1.add(pcb);
37. MainFrame.addReadyTable(pcb,1);
38. }
40. **if**(MainFrame.isFirstCreatePcb==1)//第一次创建进程，则自动开始运行第一次创建的进程，以后创建的进程由CPU调度
41. {
42. MainFrame.startRunPcb();
43. MainFrame.isFirstCreatePcb=0;
44. }
46. pidCount++;//进程标识号自动增1
47. pidTextField.setText(**new** Integer(pidCount).toString());
48. }
49. }
50. });

### 主窗体MainFrame中阻塞菜单触发事件

1. runBreakItem.addActionListener(**new** ActionListener() {
2. **public** **void** actionPerformed(**final** ActionEvent e) {
3. //监听点击阻塞进程菜单事件
4. **int** i=runTable.getSelectedRow();
5. **if**(i>=0)
6. {
7. PCB p=runPcb.get(i);
8. **if**(p.getStatus()=="执行态")
9. {
10. p.setStatus("等待态");
11. waitPcb.add(p);
12. addWaitTable(p);//加入等待进程队列中
14. runPcb.remove(i);
15. runMode.removeRow(i);//从执行进程中移除被阻塞的进程
17. startRunPcb();  //开始运行下个就绪进程
18. }
19. }
20. }
21. });

### 主窗体MainFrame中点击删除菜单触发事件

1. runDelItem.addActionListener(**new** ActionListener() {
2. **public** **void** actionPerformed(**final** ActionEvent e) {
3. //监听点击删除菜单事件
4. **int** i=runTable.getSelectedRow();
5. **if**(i>=0)
6. {
7. runPcb.remove(i);
8. runMode.removeRow(i);
9. }
10. }
11. });

### 主窗体MainFrame中点击暂停计时按钮触发事件

1. pauseButton.addActionListener(**new** ActionListener() {
2. **public** **void** actionPerformed(**final** ActionEvent e) {
3. CpuColockTimer.pauseCountTime=**true**;//暂停计时器计时
5. }
6. });

### 主窗体MainFrame中点击继续计时按钮触发事件

1. continueButton.addActionListener(**new** ActionListener() {
2. **public** **void** actionPerformed(**final** ActionEvent e) {
3. CpuColockTimer.pauseCountTime=**false**;//计时器继续计时
4. }
5. });

### 主窗体MainFrame中点击重置按钮触发事件

1. resetButton.addActionListener(**new** ActionListener() {
2. **public** **void** actionPerformed(**final** ActionEvent e) {
4. resetOs();//重置系统
5. }
6. });
7. **protected** **void** resetOs() {//重置系统
8. runMode.setRowCount(0);
9. readyMode1.setRowCount(0);
10. readyMode2.setRowCount(0);
11. waitMode.setRowCount(0);//清空Table
13. newPcb.clear();
14. runPcb.clear();
15. readyPcb1.clear();
16. readyPcb2.clear();
17. waitPcb.clear();//清空所以进程队列
19. CpuColockTimer.time=0;
20. timeLabel.setText("0   s");
21. CpuColockTimer.pauseCountTime=**false**;//计时器归零，重新计时
23. isFirstCreatePcb=1;//第一次创建进程标记置为1
25. PCBFrame.pidCount=1;//进程标识号置为1
26. PCBFrame.pidTextField.setText("1");
27. }

### 主窗体类MainFrame中调度下一个进程开始运行方法startRunPcb

1. **public** **static**  **void** startRunPcb() {
2. //开始运行下个进程，有顺序的判断各个进程队列是否为空
3. **int** isHavePcb=0;//判断是否还有进程要执行
4. PCB p=**new** PCB();
5. **if**(readyPcb1.size()>0)//就绪进程队列1不为空
6. {
7. isHavePcb=1;
8. p=readyPcb1.get(0);
10. readyMode1.removeRow(0);
11. readyPcb1.remove(0);
12. }
13. **else** **if**(readyPcb2.size()>0)就绪进程队列2不为空
14. {
15. isHavePcb=1;
16. p=readyPcb2.get(0);
18. readyMode2.removeRow(0);
19. readyPcb2.remove(0);
20. }
21. **else** **if**(waitPcb.size()>0)等待进程队列不为空
22. {
23. isHavePcb=1;
24. p=waitPcb.get(0);
26. waitMode.removeRow(0);
27. waitPcb.remove(0);
28. }
29. **if**(isHavePcb==1)//仍有进程要执行
30. {
31. p.setStatus("执行态");
32. runPcb.add(p);
33. addRunTable(p);
34. }

### 主窗体类MainFrame中addPriorityPcb和updatePriorityTable方法

1. **public** **static** **void** addPriorityPcb(Vector<PCB> readyPcb1, PCB pcb) {
2. //根据进程优先级有序地插入到就绪队列1中，末尾的进程优先级最小
3. **if**(readyPcb1.size()==0)
4. readyPcb1.add(pcb);
5. **else**
6. {
7. readyPcb1.setSize(readyPcb1.size()+1);
8. **int** i;
9. **for**(i=readyPcb1.size()-2;i>=0;i--)
10. {
11. PCB rPcb=readyPcb1.get(i);
12. **if**(rPcb.getPriority()<pcb.getPriority())
13. readyPcb1.set(i+1, rPcb);
14. **else**
15. **break**;
16. }
17. readyPcb1.set(i+1, pcb);
18. }
19. }
21. **public** **static** **void** updatePriorityTable() {
22. //更新就绪进程Table，先清空Table，再将有序的就绪进程队列加入Table中
23. readyMode1.setRowCount(0);
24. **for**(**int** i=0;i<readyPcb1.size();i++)
25. {
26. Vector<String> v=createVector(readyPcb1.get(i));
27. readyMode1.addRow(v);
28. }
29. }

### 主窗体类MainFrame中加入各个进程Table的方法AddTable

1. **public** **static** **void** addReadyTable(PCB p,**int** type) {
2. //将进程加入就绪进程队列中
3. Vector<String> v=createVector(p);
4. **if**(type==1)
5. readyMode1.addRow(v);
6. **else** **if**(type==2)
7. readyMode2.addRow(v);
8. }
10. **public** **static**  **void** addRunTable(PCB p) {
11. //将进程加入执行进程队列中
12. Vector<String> v=createVector(p);
13. runMode.addRow(v);
14. }
16. **public** **static** **void** addWaitTable(PCB p) {
17. //将进程加入等待进程队列中
18. Vector<String> v=createVector(p);
19. waitMode.addRow(v);
20. }

### 主窗体类MainFrame中唤醒进程的方法wakeWaitPcb()

1. **public** **static** **void** wakeWaitPcb() {
2. //唤醒等待进程队列到就绪进程队列
3. PCB p=waitPcb.get(0);
4. waitMode.removeRow(0);
5. waitPcb.remove(0);
6. p.setStatus("就绪态");
7. readyPcb1.add(p);
8. addReadyTable(p,1);
9. }

### 主窗体类MainFrame中将进程属性加入Vector中方法CreateVector

1. **public** **static** Vector<String> createVector(PCB p){
2. //将进程p 所有属性加入Vector向量v中，返回v
3. Vector<String> v=**new** Vector<String>();
4. v.addElement(**new** Integer(p.getPid()).toString());
5. v.addElement(**new** Integer(p.getPriority()).toString());
6. v.addElement(**new** Integer(p.getMem()).toString());
7. v.addElement(**new** Integer(p.getPrinter()).toString());
8. v.addElement(p.getStatus());
9. v.addElement(**new** Integer(p.getInputTime()).toString());
10. v.addElement(**new** Integer(p.getLastTime()).toString());
11. v.addElement(**new** Integer(p.getExecuteTime()).toString());
12. v.addElement(**new** Integer(p.getEndTime()).toString());
13. **return** v;
14. }

### 时钟计时器类CpuColockTimer的count方法

1. **private** **void** count() {
2. **if**(pauseCountTime==**false**)
3. {
4. **if**(MainFrame.fcfsRadioB.isSelected())
5. {
6. DispatchPcb(1);//先来先服务调度算法
7. }
8. **else** **if**(MainFrame.rrRadioB.isSelected())
9. {
10. DispatchPcb(2);//时间片轮转调度算法
11. }
12. **else** **if**(MainFrame.mulrrRadioB.isSelected())
13. {
14. DispatchPcb(3);//多级反馈轮转调度算法
15. }
16. **else** **if**(MainFrame.staticPriorityRadioB.isSelected())
17. {
18. DispatchPcb(4);//静态优先级调度算法
19. }
20. }
21. }

### 时钟计时器类CpuColockTimer的DispatchPcb调度方法

1. **private** **void** DispatchPcb(**int** type) {
2. time++;
3. MainFrame.timeLabel.setText(**new** String(**new** Integer(time).toString()+"  s"));
5. **int** size=MainFrame.runPcb.size();
7. **if**(size>0 && MainFrame.runPcb.get(size-1).getStatus()=="执行态" )
8. //若执行队列不为空且最后进程处在执行态则将其已执行时间加1s
9. {
10. PCB pcb=MainFrame.runPcb.get(size-1);
11. **int** exeTime=pcb.getExecuteTime();
12. exeTime+=1;//将进程已执行时间增1s
13. pcb.setExecuteTime(exeTime);
15. MainFrame.runMode.setValueAt(exeTime, MainFrame.runMode.getRowCount()-1, 7);
16. //更新执行Table中，处在执行态进程的已执行时间
18. **if**(type==2 || type==3)//当为时间片轮转法或多级反馈轮转法时
19. {
20. **if**(exeTime % timeSlice==0 && exeTime<pcb.getLastTime())
21. //时间片用完且进程还没有运行完
22. {
23. pcb.setStatus("就绪态");
24. MainFrame.runMode.removeRow(size-1);
25. MainFrame.runPcb.remove(pcb);//从执行队列中移除时间片用完的进程
27. **if**(type==2)//为时间片轮转法时，加入就绪队列1中
28. {
29. MainFrame.readyPcb1.add(pcb);
30. MainFrame.addReadyTable(pcb,1);
31. }
32. **else** //为多级反馈轮转法时，加入就绪队列2中
33. {
34. MainFrame.readyPcb2.add(pcb);
35. MainFrame.addReadyTable(pcb,2);
36. }
38. MainFrame.startRunPcb();
39. //开始运行下一个就绪进程，若没有就绪进程，则运行等待进程
40. }
41. }
42. **if**(exeTime==pcb.getLastTime())//进程运行完成
43. {
44. pcb.setEndTime(time);
45. pcb.setStatus("完成态");
46. MainFrame.runMode.removeRow(size-1);
47. MainFrame.addRunTable(pcb);
48. //更新进程状态为完成态。先从执行Table中移除，再重新加入执行Table中
49. **if**(MainFrame.waitPcb.size()>0)//若等待队列不为空，则唤醒等待队列到就绪队列中
50. MainFrame.wakeWaitPcb();
52. MainFrame.startRunPcb();
53. //开始运行下一个就绪进程，若没有就绪进程，则运行等待进程
54. }
55. }
56. **else** **if**(size>0)//执行队列中进程都已运行完成，则唤醒等待进程且调入下个就绪进程运行
57. {
58. **if**(MainFrame.waitPcb.size()>0)//若等待队列不为空，则唤醒等待队列到就绪队列中
59. MainFrame.wakeWaitPcb();
60. MainFrame.startRunPcb();//开始运行下一个就绪进程，若没有就绪进程，则运行等待进程
61. }
62. }

## 哲学家进餐模块

### 主要方法函数

1. **public** chopsticks(**int** i,JLabel cl, ImageIcon cp);  //定义哲学家进餐筷子类
2. **public** **synchronized** **void** P();                      //对象级别锁   取筷子，信号量限制互斥
3. **public** **synchronized** **void** V();                      //释放筷子，唤醒其他线程
4. **public** **void** actionPerformed(ActionEvent e);        // 启动5个哲学家线程
5. **public** **void** think();                               //思考
6. **public** **void** eat();                                 //吃饭
7. **public** **void** run();                                 //运行线程

### 记录型信号量

1. /////////////////////////////////////////////////////////////////////////////////////
2. // 名称:   philosopher
3. // 功能:   利用记录型信号量解决哲学家就餐问题（允许死锁）
4. // 算法:   当哲学家饥饿时，总是先去拿他左边的筷子，成功后再去拿他右边的筷子，进餐完毕后先放下他左边的筷子再放下他右边的筷子.(当五个哲学家同时拿起左边的筷子时会产生死锁)
5. // 伪码:
6. //      void philosopher(int i)           /\*i：哲学家编号，从0 到4\*/
7. //      {
8. //          while (TRUE)
9. //          {
10. //             think( );                /\*哲学家正在思考\*/
11. //             take\_fork(i);            /\*取左侧的叉子\*/
12. //             take\_fork((i+1) % N);    /\*取右侧叉子；％为取模运算\*/
13. //             eat( );                  /\*吃饭\*/
14. //             put\_fork(i);             /\*把左侧叉子放回桌子\*/
15. //             put\_fork((i+1) % N);     /\*把右侧叉子放回桌子\*/
16. //           }
17. //       }
18. // 返回:
19. // 注意事项:
20. ///////////////////////////////////////////////////////////////////////////////////////

### 最多只有四个哲学家同时就餐

1. /////////////////////////////////////////////////////////////////////////////////////
2. // 名称:   philosopher
3. // 功能:   最多只有四个哲学家同时就餐
4. // 算法:
5. // 伪码:
6. //          semaphore fork[5]={1，1，1，1，1};
7. //          semaphore count=4;
8. //          void philosopher(int i)
9. //          {
10. //              while(true)
11. //              {
12. //                  think();
13. //                  wait(count);                             //请求进入房间进餐
14. //                  wait(fork[i]);                           //请求左手边的叉子
15. //                  wait(fork[(i+1)%5]);                     //请求右手边的叉子
16. //                  eat();
17. //                  signal(fork[(i+1)%5]);                   //释放右手边的叉子
18. //                  signal(fork[i]);                         //释放左手边的叉子
19. //                  signal(count);                           //退出房间释放信号量room
20. //              }
21. //          }
22. // 返回:
23. // 注意事项:
24. ///////////////////////////////////////////////////////////////////////////////////////

### 哲学家就餐分奇偶

奇数号哲学家先取左边筷子再取右边筷子，偶数号哲学家先取右边筷子再取左边筷子

1. /////////////////////////////////////////////////////////////////////////////////////
2. // 名称:   philosopher
3. // 功能:   利用记录型信号量解决哲学家就餐问题（允许死锁）
4. // 算法:   奇数号哲学家先取左边筷子再取右边筷子，偶数号哲学家先取右边筷子再取左边筷子
5. // 伪码:
6. //       semaphore fork[5]={1，1，1，1，1};
7. //       void philosopher(int i)
8. //       {
9. //           while(true)
10. //           {
11. //               think();
12. //               if(i%2 == 0)                             //偶数哲学家，先右后左。
13. //               {
14. //                   wait (fork[ i + 1 ] mod 5) ;
15. //                   wait (fork[ i]) ;
16. //                   eat();
17. //                   signal (fork[ i + 1 ] mod 5) ;
18. //                   signal (fork[ i]) ;
19. //               }
20. //           else                                       //奇数哲学家，先左后右。
21. //           {
22. //               wait (fork[ i]) ;
23. //               wait (fork[ i + 1 ] mod 5) ;
24. //               eat();
25. //               signal (fork[ i]) ;
26. //               signal (fork[ i + 1 ] mod 5) ;
27. //           }
28. //       }
29. // 返回:
30. // 注意事项:
31. ///////////////////////////////////////////////////////////////////////////////////////

### 只有取到两把筷子才能吃饭

每位哲学家只有取到两把筷子才能吃饭，否则一把也不取。

1. /////////////////////////////////////////////////////////////////////////////////////
2. // 名称:   philosopher
3. // 功能:   利用记录型信号量解决哲学家就餐问题（允许死锁）
4. // 算法:   每位哲学家只有取到两把筷子才能吃饭，否则一把也不取，为保证哲学家不出现由于长时间等待得不到筷子而导致的饥饿,对哲学家进行优先级控制，哲学家每次吃完饭优先级降低，随饥饿次数的增加优先级增加
5. // 伪码:
6. //       semaphore fork[5]={1，1，1，1，1};
7. //       void philosopher(int i)
8. //       {
9. //           while(true)
10. //           {
11. //              think();
12. //              Swait(fork[(I+1)]%5,fork[I]);
13. //              eat();
14. //              Ssignal(fork[(I+1)]%5,fork[I]);
15. //            }
16. //       }
17. // 返回:
18. // 注意事项:
19. ///////////////////////////////////////////////////////////////////////////////////////

## 读者写者问题模块

### 主要方法函数

1. **void** rreadlock(String a);      // 读优先
2. **void** wreadlock(String a);      // 写优先
3. **void** readunlock(String a);     //读解锁
4. **void** writelock(String a);      //写上锁
5. **void** writeunlock(String a);    //写解锁
6. **void** initComponents();         //初始化构件
7. **void** MainJFrame();             //读写模块主要启动函数
8. ///////////////////////////////////////////////////////////////////////////////////////
9. // 名称:      myThread
10. // 功能:      读写主要操作
11. // 算法:      ①允许多个读者可以同时对文件执行读操作；
12. ②只允许一个写者往文件中写信息；
13. ③任一写者在完成写操作之前不允许其他读者或写者工作；
14. ④写者执行写操作前，应让已有的读者和写者全部退出。
15. // 返回:      如果成功，UI上面返回结果显示
16. // 注意事项:
17. ///////////////////////////////////////////////////////////////////////////////////////
18. **class** myThread **extends** Thread {
19. **static** ReadWriteLock Lock = **new** ReadWriteLock();
20. Random r = **new** Random();
21. **static** **int** readcount = 0;
22. **static** **int** writecount = 0;
23. **boolean** flag;
24. String a, b;
25. **double** c;
27. myThread(String q, String s, **boolean** choose) {
28. c = r.nextDouble() \* 1;
29. a = q;
30. b = s;
31. flag = choose;
32. }
34. **public** **void** run() {
35. **if** (b == "R") {
36. Lock.aa.fill("读线程" + a + "被创建\n");
37. /\*
38. \* try{ Thread.sleep((int)(1000\*c)); } catch(InterruptedException e){}
39. \*/
40. Lock.aa.fill("读线程" + a + "申请读操作\n");
41. **try** {
42. **if** (!flag) {
43. Lock.rreadlock(a);
44. } **else** {
45. Lock.wreadlock(a);
46. }
47. } **catch** (InterruptedException e1) {
48. }
49. Lock.aa.fill("读线程" + a + "开始读操作\n");
50. **try** {
51. Thread.sleep((**int**) (1000 \* c));
52. } **catch** (InterruptedException e) {
53. }
54. Lock.aa.fill("读线程" + a + "结束读操作\n");
55. Lock.readunlock(a);
56. } **else** **if** (b == "W") {
57. Lock.aa.fill("写线程" + a + "被创建\n");
58. /\*
59. \* try{ Thread.sleep((int)(1000\*c)); } catch(InterruptedException e){}
60. \*/
61. Lock.aa.fill("写线程" + a + "申请写操作\n");
62. **try** {
63. Lock.writelock(a);
64. } **catch** (InterruptedException e1) {
65. }
66. Lock.aa.fill("写线程" + a + "开始写操作\n");
67. **try** {
68. Thread.sleep((**int**) (1000 \* c));
69. } **catch** (InterruptedException e) {
70. }
71. Lock.aa.fill("写线程" + a + "结束写操作\n");
72. Lock.writeunlock(a);
73. }
74. }
75. }
77. **class** ReadWriteLock {
78. **public** ResultJFrame aa = **new** ResultJFrame();
79. **private** **int** readingThreads = 0;
80. **private** **int** writingThreads = 0;
81. **private** **int** waitingThreads = 0;
83. **public** **synchronized** **void** rreadlock(String a) **throws** InterruptedException {
84. // 读优先
85. **while** ((writingThreads > 0)) {
86. aa.fill("读线程" + a + "等待\n");
87. **this**.wait();
88. }
89. readingThreads++;
90. }
92. **public** **synchronized** **void** wreadlock(String a) **throws** InterruptedException {
93. // 写优先
94. **while** ((writingThreads > 0 || (waitingThreads > 0))) {
95. aa.fill("读线程" + a + "等待\n");
96. **this**.wait();
97. }
98. readingThreads++;
99. }
101. **public** **synchronized** **void** readunlock(String a) {
102. readingThreads--;
103. aa.fill("读线程" + a + "释放\n");
104. notifyAll();
105. // notify();
106. }
108. **public** **synchronized** **void** writelock(String a) **throws** InterruptedException {
109. waitingThreads++;
110. **try** {
111. **while** (readingThreads > 0 || writingThreads > 0) {
112. aa.fill("写线程" + a + "等待\n");
113. **this**.wait();
114. }
115. } **finally** {
116. waitingThreads--;
117. }
118. writingThreads++;
119. }
121. **public** **synchronized** **void** writeunlock(String a) {
122. writingThreads--;
123. aa.fill("写线程" + a + "释放\n");
124. notifyAll();
125. // notify();
126. }
127. }

# 测试计划

## 进程调度管理

### 先来先服务调度

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **进程标识号** | **提交时间** | **持续时间** | **测试:完成时间** | **实际完成时间** |
| 1 | 3 | 4 | 7 | 7 |
| 2 | 8 | 2 | 11 | 11 |
| 3 | 12 | 6 | 19 | 19 |
| 4 | 17 | 8 | 27 | 27 |

### 短作业优先调度

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **进程标识号** | **提交时间** | **持续时间** | **完成时间** | **实际完成时间** |
| 1 | 2 | 10 | 12 | 12 |
| 4 | 10 | 4 | 16 | 16 |
| 2 | 8 | 5 | 21 | 21 |
| 3 | 9 | 16 | 37 | 37 |

### 优先级调度算法

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **进程标识号** | **优先级** | **提交时间** | **持续时间** | **完成时间** | **实际完成时间** |
| 1 | 3 | 3 | 10 | 13 | 13 |
| 3 | 4 | 9 | 10 | 23 | 23 |
| 2 | 2 | 5 | 12 | 35 | 35 |
| 4 | 1 | 13 | 10 | 45 | 45 |

### 时间片轮转算法

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **进程标识号** | **提交时间** | **持续时间** | **完成时间** | **实际完成时间** |
| 1 | 2 | 10 | 27 | 27 |
| 2 | 5 | 7 | 31 | 31 |
| 4 | 10 | 7 | 35 | 35 |
| 3 | 7 | 11 | 37 | 37 |

### 多级反馈队列算法

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **进程标识号** | **优先级** | **提交时间** | **持续时间** | **完成时间** | **实际完成时间** |
| 1 | 1 | 2 | 10 | 33 | 33 |
| 2 | 1 | 4 | 14 | 50 | 50 |
| 3 | 1 | 8 | 13 | 51 | 51 |
| 4 | 1 | 13 | 17 | 56 | 56 |

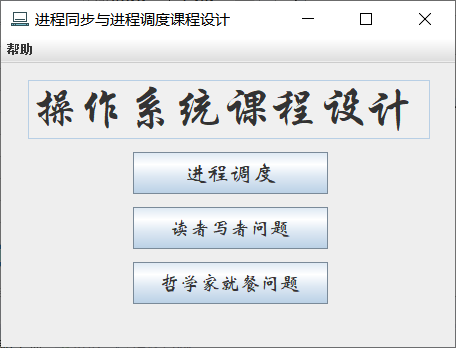
## 哲学家进餐

## 读者写者问题

|  |  |  |  |
| --- | --- | --- | --- |
| **优先** | **添加读者或写者** | **测试:运行结果** | **实际运行结果** |
| **读者优先** | 读者  写者  读者  写者 | 读线程1被创建  读线程1申请读操作  读线程1开始读操作  写线程2被创建  写线程2申请写操作  写线程2等待  读线程3被创建  读线程3申请读操作  读线程3开始读操作  写线程4被创建  写线程4申请写操作  写线程4等待  读线程3结束读操作  读线程3释放  写线程4等待  写线程2等待  读线程1结束读操作  读线程1释放  写线程2开始写操作  写线程4等待  写线程2结束写操作  写线程2释放  写线程4开始写操作  写线程4结束写操作  写线程4释放 | 读线程1被创建  读线程1申请读操作  读线程1开始读操作  写线程2被创建  写线程2申请写操作  写线程2等待  读线程3被创建  读线程3申请读操作  读线程3开始读操作  写线程4被创建  写线程4申请写操作  写线程4等待  读线程3结束读操作  读线程3释放  写线程4等待  写线程2等待  读线程1结束读操作  读线程1释放  写线程2开始写操作  写线程4等待  写线程2结束写操作  写线程2释放  写线程4开始写操作  写线程4结束写操作  写线程4释放 |
| **写者优先** | 读者  写者  读者  写者 | 读线程1被创建  读线程1申请读操作  读线程1开始读操作  写线程2被创建  写线程2申请写操作  写线程2等待  读线程3被创建  读线程3申请读操作  读线程3等待  写线程4被创建  写线程4申请写操作  写线程4等待  读线程1结束读操作  读线程1释放  写线程4开始写操作  写线程2等待  写线程4结束写操作  写线程4释放  写线程2开始写操作  读线程3等待  写线程2结束写操作  写线程2释放  读线程3开始读操作  读线程3结束读操作  读线程3释放 | 读线程1被创建  读线程1申请读操作  读线程1开始读操作  写线程2被创建  写线程2申请写操作  写线程2等待  读线程3被创建  读线程3申请读操作  读线程3等待  写线程4被创建  写线程4申请写操作  写线程4等待  读线程1结束读操作  读线程1释放  写线程4开始写操作  写线程2等待  写线程4结束写操作  写线程4释放  写线程2开始写操作  读线程3等待  写线程2结束写操作  写线程2释放  读线程3开始读操作  读线程3结束读操作  读线程3释放 |

# 运行结果截图

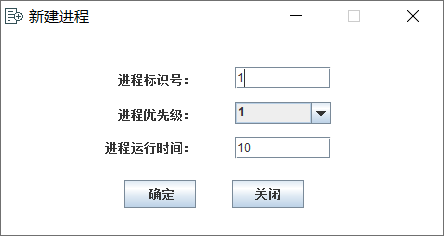
## 项目启动程序



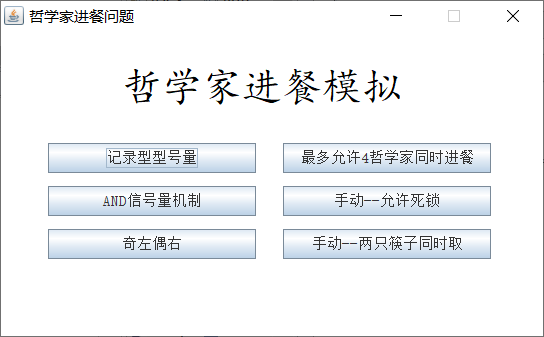


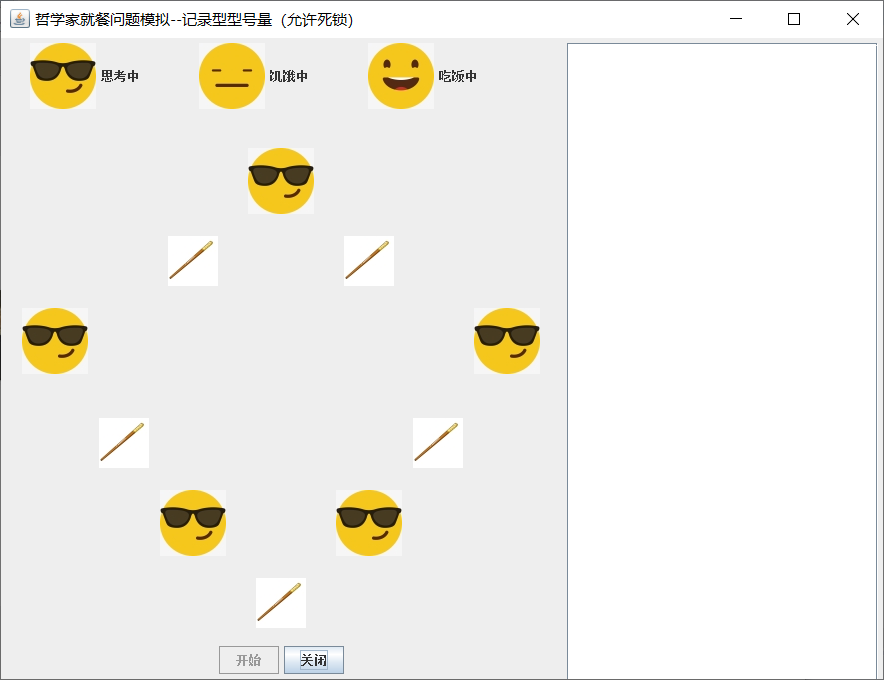
## 进程调度管理

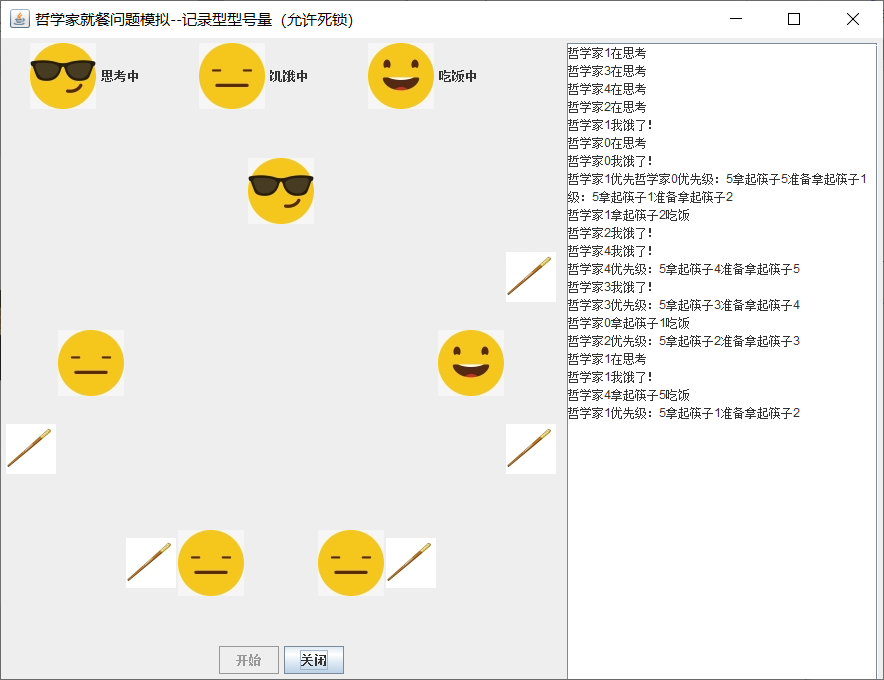




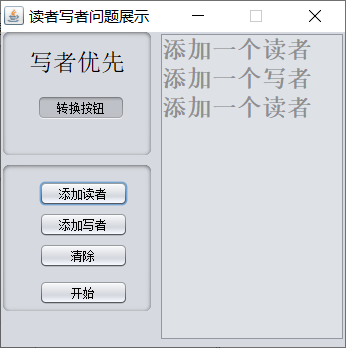
## 哲学家进餐

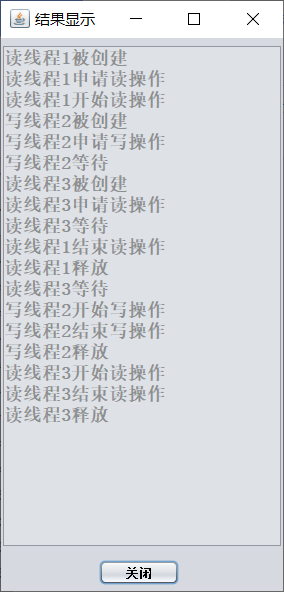






## 读者写者问题





# 团队分工

| **姓名** | **学号** | **分工细则** |
| --- | --- | --- |
| 许东明 | YA1714180 | 1. 组长：对小组进行任务分工，并且实时关注组内动态（主要） 2. 负责程序代码撰写 3. 负责文档撰写 4. 负责汇报PPT大纲制作 |
| 方俊翔 | YA1714036 | 1. 负责文档撰写（主要） 2. 绘制项目所有流程图、功能模块图 |
| 汪鹏 | YA1714037 | 1. 负责程序代码撰写（主要） 2. 负责UI界面设计 |
| 袁兴武 | YA1714052 | 1. 负责程序算法测试 2. 负责汇报PPT制作（主要） |

# 小结

我们设计了一个模拟进程调度、哲学家进餐和读写者管理系统，通过用户按照一定的形式进行选择相应功能模块，并且输入正确的参数可实现操作系统课程设计相关的模拟。主要设计的工作可概括如下：

通过这次课程设计，我们小组收获到了以下几点。

1.对操作系统的进程调度管理、进程同步以及哲学家进餐算法过程有了更清晰的了解和掌握，以前在课堂上都没有怎么弄明白的原理，通过和同学讨论问题，自己动手写程序，一下子就搞明白了，这说明平常学习过程中要和同学老师多讨论讨论，才能弄懂知识点，才能做到学以致用。

2.这次课程设计，我们是用java实现的，对java桌面应用程序的开发又有了进一步的熟悉和掌握，尤其是用java swing 插件来设计界面，方便又快捷，大大提高了系统开发效率。

3.团队合作很重要，每个成员都有自己不同的想法和解决相同问题的方法，要尽量汲取他人的优点。

# 参考文献

[1]王育勤,等，《计算机操作系统》[M].北京，北京交通大学出版社，2004.

[2]汤子瀛，等，《计算机操作系统》[M].北京，西安电子科技大学出版社，2001

[3]张顺香，等，《操作系统》[M].北京，武汉大学出版社，2009

[4]张尧学，等，《计算机操作系统教程》[M].北京，清华大学出版社，2006

# 附录：主要代码

CpuColockTimer.java

1. **import** java.util.Timer;
2. **import** java.util.TimerTask;
4. **public** **class** CpuColockTimer {
6. **public** **static** Timer timer = **new** Timer(); // 计时
7. **public** **static** **int** time = 0;
8. **public** **static** **int** timeSlice = 5;// cpu 时间片，默认为5s
9. **public** **static** **boolean** pauseCountTime = **false**;// 用于控制计时器是否计时，false则计时，true 则停止计时
11. **public** CpuColockTimer() {
12. }
14. **public** **void** start() {
15. timer.schedule(**new** TimerTask() {
16. **public** **void** run() {
17. count(); // 时间每次加1s
18. **if** (time > 600)
19. timer.cancel();
20. }
22. **private** **void** count() {
23. **if** (pauseCountTime == **false**) {
24. **if** (MainFrame.fcfsRadioB.isSelected()) {
25. DispatchPcb(1);// 先来先服务调度算法
26. } **else** **if** (MainFrame.rrRadioB.isSelected()) {
27. DispatchPcb(2);// 时间片轮转调度算法
28. } **else** **if** (MainFrame.mulrrRadioB.isSelected()) {
29. DispatchPcb(3);// 多级反馈队列调度算法
30. } **else** **if** (MainFrame.staticPriorityRadioB.isSelected()) {
31. DispatchPcb(4);// 静态优先级调度算法
32. } **else** **if** (MainFrame.sjrRadioB.isSelected()) {
33. DispatchPcb(5);// 最短作业优先调度算法
34. }
35. }
36. }
38. **private** **void** DispatchPcb(**int** type) {
39. time++;
40. MainFrame.timeLabel.setText(**new** String(**new** Integer(time).toString() + "  s"));
41. **int** size = MainFrame.runPcb.size();
42. **if** (size > 0 && MainFrame.runPcb.get(size - 1).getStatus() == "执行态")
43. // 若执行队列不为空且最后进程处在执行态则将其已执行时间加1s
44. {
45. PCB pcb = MainFrame.runPcb.get(size - 1);
46. **int** exeTime = pcb.getExecuteTime();
47. exeTime += 1;// 将进程已执行时间增1s
48. pcb.setExecuteTime(exeTime);
50. MainFrame.runMode.setValueAt(exeTime, MainFrame.runMode.getRowCount() - 1, 5);// 更新执行Table中，处在执行态进程的已执行时间
52. **if** (type == 2 || type == 3)// 当为时间片轮转法或多级反馈轮转法时
53. {
54. **if** (exeTime % timeSlice == 0 && exeTime < pcb.getLastTime())
55. // 时间片用完且进程还没有运行完
56. {
57. pcb.setStatus("就绪态");
58. MainFrame.runMode.removeRow(size - 1);
59. MainFrame.runPcb.remove(pcb);
60. // 从执行队列中移除时间片用完的进程
62. **if** (type == 2)// 为时间片轮转法时，加入就绪队列1中
63. {
64. MainFrame.readyPcb1.add(pcb);
65. MainFrame.addReadyTable(pcb, 1);
66. } **else** // 为多级反馈轮转法时，加入就绪队列2中
67. {
68. MainFrame.readyPcb2.add(pcb);
69. MainFrame.addReadyTable(pcb, 2);
70. }
72. MainFrame.startRunPcb();
73. // 开始运行下一个就绪进程，若没有就绪进程，则运行等待进程
74. }
75. }
76. **if** (exeTime == pcb.getLastTime())// 进程运行完成
77. {
78. pcb.setEndTime(time);
79. pcb.setStatus("完成态");
80. MainFrame.runMode.removeRow(size - 1);
81. MainFrame.addRunTable(pcb);
82. // 更新进程状态为完成态。先从执行Table中移除，再重新加入执行Table中
84. **if** (MainFrame.waitPcb.size() > 0)
85. // 若等待队列不为空，则唤醒等待队列到就绪队列中
86. MainFrame.wakeWaitPcb();
88. MainFrame.startRunPcb();
89. // 开始运行下一个就绪进程，若没有就绪进程，则运行等待进程
90. }
91. } **else** **if** (size > 0)
92. // 执行队列中进程都已运行完成，则唤醒等待进程且调入下个就绪进程运行
93. {
94. **if** (MainFrame.waitPcb.size() > 0)
95. // 若等待队列不为空，则唤醒等待队列到就绪队列中
96. MainFrame.wakeWaitPcb();
98. MainFrame.startRunPcb();
99. // 开始运行下一个就绪进程，若没有就绪进程，则运行等待进程
100. }
101. }
102. }, 1, 1000);// 1---表示计时从1ms开始，1000---表示每隔1s计一次时间
103. }
104. }

PCB.java

1. **public** **class** PCB {
2. **int** pid=0; //进程标识号
3. **int** priority=0;//优先级
4. String status="";//当前状态
5. **int** lastTime=0;//持续时间
6. **int** inputTime=0;//提交时间
7. **int** endTime=0;//完成时间
8. **int** executeTime=0;//已执行时间
10. //获得进程标识号
11. **public** **int** getPid() {
12. **return** pid;
13. }
14. //设置进程标识号
15. **public** **void** setPid(**int** pid) {
16. **this**.pid = pid;
17. }
18. //获得优先级
19. **public** **int** getPriority() {
20. **return** priority;
21. }
22. //设置优先级
23. **public** **void** setPriority(**int** priority) {
24. **this**.priority = priority;
25. }
26. //获得持续时间
27. **public** **int** getLastTime() {
28. **return** lastTime;
29. }
30. //设置持续时间
31. **public** **void** setLastTime(**int** lastTime) {
32. **this**.lastTime = lastTime;
33. }
34. //获得提交时间
35. **public** **int** getInputTime() {
36. **return** inputTime;
37. }
38. //设置提交时间
39. **public** **void** setInputTime(**int** inputTime) {
40. **this**.inputTime = inputTime;
41. }
42. //获得完成时间
43. **public** **int** getEndTime() {
44. **return** endTime;
45. }
46. //设置完成时间
47. **public** **void** setEndTime(**int** endTime) {
48. **this**.endTime = endTime;
49. }
50. //获得已执行时间
51. **public** **int** getExecuteTime() {
52. **return** executeTime;
53. }
55. // 设置已执行时间
56. **public** **void** setExecuteTime(**int** executeTime) {
57. **this**.executeTime = executeTime;
58. }
59. //获得当前状态
60. **public** String getStatus() {
61. **return** status;
62. }
63. //设置当前状态
64. **public** **void** setStatus(String status) {
65. **this**.status = status;
66. }
67. }

myThread.java

1. **class** myThread **extends** Thread {
2. **static** ReadWriteLock Lock = **new** ReadWriteLock();
3. Random r = **new** Random();
4. **static** **int** readcount = 0;
5. **static** **int** writecount = 0;
6. **boolean** flag;
7. String a, b;
8. **double** c;
10. myThread(String q, String s, **boolean** choose) {
11. c = r.nextDouble() \* 1;
12. a = q;
13. b = s;
14. flag = choose;
15. }
17. **public** **void** run() {
18. **if** (b == "R") {
19. Lock.aa.fill("读线程" + a + "被创建\n");
20. /\*
21. \* try{ Thread.sleep((int)(1000\*c)); } catch(InterruptedException e){}
22. \*/
23. Lock.aa.fill("读线程" + a + "申请读操作\n");
24. **try** {
25. **if** (!flag) {
26. Lock.rreadlock(a);
27. } **else** {
28. Lock.wreadlock(a);
29. }
30. } **catch** (InterruptedException e1) {
31. }
32. Lock.aa.fill("读线程" + a + "开始读操作\n");
33. **try** {
34. Thread.sleep((**int**) (1000 \* c));
35. } **catch** (InterruptedException e) {
36. }
37. Lock.aa.fill("读线程" + a + "结束读操作\n");
38. Lock.readunlock(a);
39. } **else** **if** (b == "W") {
40. Lock.aa.fill("写线程" + a + "被创建\n");
41. /\*
42. \* try{ Thread.sleep((int)(1000\*c)); } catch(InterruptedException e){}
43. \*/
44. Lock.aa.fill("写线程" + a + "申请写操作\n");
45. **try** {
46. Lock.writelock(a);
47. } **catch** (InterruptedException e1) {
48. }
49. Lock.aa.fill("写线程" + a + "开始写操作\n");
50. **try** {
51. Thread.sleep((**int**) (1000 \* c));
52. } **catch** (InterruptedException e) {
53. }
54. Lock.aa.fill("写线程" + a + "结束写操作\n");
55. Lock.writeunlock(a);
56. }
57. }
58. }
60. **class** ReadWriteLock {
61. **public** ResultJFrame aa = **new** ResultJFrame();
62. **private** **int** readingThreads = 0;
63. **private** **int** writingThreads = 0;
64. **private** **int** waitingThreads = 0;
66. **public** **synchronized** **void** rreadlock(String a) **throws** InterruptedException {
67. // 读优先
68. **while** ((writingThreads > 0)) {
69. aa.fill("读线程" + a + "等待\n");
70. **this**.wait();
71. }
72. readingThreads++;
73. }
75. **public** **synchronized** **void** wreadlock(String a) **throws** InterruptedException {
76. // 写优先
77. **while** ((writingThreads > 0 || (waitingThreads > 0))) {
78. aa.fill("读线程" + a + "等待\n");
79. **this**.wait();
80. }
81. readingThreads++;
82. }
84. **public** **synchronized** **void** readunlock(String a) {
85. readingThreads--;
86. aa.fill("读线程" + a + "释放\n");
87. notifyAll();
88. // notify();
89. }
91. **public** **synchronized** **void** writelock(String a) **throws** InterruptedException {
92. waitingThreads++;
93. **try** {
94. **while** (readingThreads > 0 || writingThreads > 0) {
95. aa.fill("写线程" + a + "等待\n");
96. **this**.wait();
97. }
98. } **finally** {
99. waitingThreads--;
100. }
101. writingThreads++;
102. }
104. **public** **synchronized** **void** writeunlock(String a) {
105. writingThreads--;
106. aa.fill("写线程" + a + "释放\n");
107. notifyAll();
108. // notify();
109. }
110. }

interface1.java

1. /\*\*
2. \* 最多允许四个哲学家同时进餐
3. \*/
4. **package** code;
6. **import** java.awt.\*;
7. **import** java.awt.event.\*;
8. **import** javax.swing.\*;
10. **public** **class** interface1 **extends** JFrame **implements** ActionListener {
11. JButton startBt, endBt;
12. **static** JTextArea area = **new** JTextArea(50, 28);
13. JLabel cl[];
14. JLabel h[];
15. JLabel pl[];
16. chopsticks c[];
17. philoDemo1 p[];
18. ImageIcon cp[], empty, tp, sp, ep;//五支筷子的图片组，空的图，思考中的图，饥饿中的图，吃饭中的图
20. interface1() {
21. **super**("哲学家就餐问题模拟--最多允许4哲学家同时进餐");
22. **this**.setDefaultCloseOperation(JFrame.EXIT\_ON\_CLOSE);
23. Container ct = **this**.getContentPane();
24. ct.setBackground(java.awt.Color.YELLOW);
26. cp = **new** ImageIcon[6];//五支筷子的图片组
27. cp[1] = **new** ImageIcon("c1.jpg");
28. cp[2] = **new** ImageIcon("c2.jpg");
29. cp[3] = **new** ImageIcon("c3.jpg");
30. cp[4] = **new** ImageIcon("c4.jpg");
31. cp[5] = **new** ImageIcon("c5.jpg");
32. tp = **new** ImageIcon("thinking1.jpg");//空的图
33. sp = **new** ImageIcon("starving1.jpg");//思考中的图
34. ep = **new** ImageIcon("eating1.jpg");//饥饿中的图
35. empty = **new** ImageIcon("empty.jpg");//吃饭中的图
37. JPanel panel = **new** JPanel();
39. pl = **new** JLabel[6];
40. **for** (**int** i = 1; i <= 5; i++) {
41. pl[i] = **new** JLabel();
42. pl[i].setIcon(tp);
43. }
45. cl = **new** JLabel[6];
46. cl[1] = **new** JLabel();
47. cl[1].setIcon(cp[1]);
48. cl[2] = **new** JLabel();
49. cl[2].setIcon(cp[2]);
50. cl[3] = **new** JLabel();
51. cl[3].setIcon(cp[3]);
52. cl[4] = **new** JLabel();
53. cl[4].setIcon(cp[4]);
54. cl[5] = **new** JLabel();
55. cl[5].setIcon(cp[5]);
57. c = **new** chopsticks[6];
58. **for** (**int** i = 1; i <= 5; i++)
59. c[i] = **new** chopsticks(i, cl[i], cp[i]);
61. h = **new** JLabel[13];
62. **for** (**int** i = 1; i <= 12; i++) {
63. h[i] = **new** JLabel();
64. h[i].setIcon(empty);
65. }
67. p = **new** philoDemo1[6];
69. panel.setLayout(**new** GridBagLayout());// 设置布局
70. GridBagConstraints gbc = **new** GridBagConstraints();
71. gbc.weightx = 1;
72. gbc.weighty = 1;
73. gbc.gridwidth = 1;
74. gbc.gridheight = 1;
75. gbc.gridy = 0;
76. gbc.gridx = 4;
77. panel.add(h[2], gbc);
78. gbc.gridx = 6;
79. panel.add(h[1], gbc);
80. gbc.gridy = 1;
81. gbc.gridx = 5;
82. panel.add(pl[1], gbc);
83. gbc.gridy = 2;
84. gbc.gridx = 0;
85. panel.add(h[3], gbc);
86. gbc.gridx = 3;
87. panel.add(cl[1], gbc);
88. gbc.gridx = 7;
89. panel.add(cl[5], gbc);
90. gbc.gridx = 10;
91. panel.add(h[10], gbc);
92. gbc.gridy = 3;
93. gbc.gridx = 1;
94. panel.add(pl[2], gbc);
95. gbc.gridx = 9;
96. panel.add(pl[5], gbc);
97. gbc.gridy = 4;
98. gbc.gridx = 0;
99. panel.add(h[4], gbc);
100. gbc.gridx = 10;
101. panel.add(h[9], gbc);
102. gbc.gridy = 5;
103. gbc.gridx = 2;
104. panel.add(cl[2], gbc);
105. gbc.gridx = 8;
106. panel.add(cl[4], gbc);
107. gbc.gridy = 7;
108. gbc.gridx = 2;
109. panel.add(h[5], gbc);
110. gbc.gridx = 3;
111. panel.add(pl[3], gbc);
112. gbc.gridx = 7;
113. panel.add(pl[4], gbc);
114. gbc.gridx = 8;
115. panel.add(h[8], gbc);
116. gbc.gridy = 8;
117. gbc.gridx = 3;
118. panel.add(h[6], gbc);
119. gbc.gridx = 5;
120. panel.add(cl[3], gbc);
121. gbc.gridx = 7;
122. panel.add(h[7], gbc);
123. ct.add(panel, BorderLayout.CENTER);
124. // 建立哲学家对象
125. p[0] = **new** philoDemo1(0, pl[1], c[5], cl[5], c[1], cl[1], cp[5], cp[1], h[1], h[2], area);
126. p[1] = **new** philoDemo1(1, pl[2], c[1], cl[2], c[2], cl[1], cp[2], cp[1], h[4], h[3], area);
127. p[2] = **new** philoDemo1(2, pl[3], c[2], cl[2], c[3], cl[3], cp[2], cp[3], h[5], h[6], area);
128. p[3] = **new** philoDemo1(3, pl[4], c[3], cl[4], c[4], cl[3], cp[4], cp[3], h[8], h[7], area);
129. p[4] = **new** philoDemo1(4, pl[5], c[4], cl[4], c[5], cl[5], cp[4], cp[5], h[9], h[10], area);
131. startBt = **new** JButton("开始");
132. startBt.setEnabled(**true**);
133. startBt.addActionListener(**this**);
134. endBt = **new** JButton("关闭");
135. endBt.addActionListener(**this**);
137. JPanel panel1 = **new** JPanel();
138. panel1.setLayout(**new** FlowLayout());
139. panel1.add(**new** JLabel(tp));
140. panel1.add(**new** JLabel("思考中                  "));
141. panel1.add(**new** JLabel(sp));
142. panel1.add(**new** JLabel("饥饿中                  "));
143. panel1.add(**new** JLabel(ep));
144. panel1.add(**new** JLabel("吃饭中                  "));
146. JPanel panel2 = **new** JPanel();
147. panel2 = **new** JPanel(**new** FlowLayout());
148. panel2.add(startBt);
149. panel2.add(endBt);
151. JPanel panel3 = **new** JPanel();
152. panel3.setLayout(**new** BorderLayout());
153. panel3.add(panel1, "North");
154. panel3.add(panel, "Center");
155. panel3.add(panel2, "South");
156. ct.add(panel3, "Center");
158. JPanel panel4 = **new** JPanel();
159. panel4.add(**new** JScrollPane(area));
160. ct.add(panel4, "East");
161. }
163. **public** **void** actionPerformed(ActionEvent e) {
164. **if** (e.getSource() == startBt) {
165. **for** (**int** i = 0; i <= 4; i++) // 启动5个哲学家线程
166. {
167. p[i].start();
169. }
170. startBt.setEnabled(**false**);
171. } **else** **if** (e.getSource() == endBt) {
172. // System.exit(0);
173. dispose();
174. }
175. }
177. **public** **static** **void** main(String[] args) {
178. interface1 f = **new** interface1();
179. f.setSize(900, 700);
180. f.setVisible(**true**);
181. }
183. }

**课程设计成绩评定表**

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| 综合评定 | YA1714180 | YA1714036 | YA1714037 | YA1714052 |  |  |
| 许东明 | 方俊翔 | 汪鹏 | 袁兴武 |  |  |
| 演示系统效果25% |  |  |  |  |  |  |
| 程序结构合理性、算法、  编程风格25% |  |  |  |  |  |  |
| 设计报告规范  （条理清晰，重点突出）30% |  |  |  |  |  |  |
| 创新性及难易程度/工作量10% |  |  |  |  |  |  |
| 设计表现（工作态度、动手能力）10% |  |  |  |  |  |  |
| 总评成绩 |  |  |  |  |  |  |