请求调页储存方式模拟

2152814 周成杰

1. 引言  
   1.1 背景和目的  
    在计算机系统中，内存管理是一个重要的问题，其中请求调页存储管理方式是一种常见的策略。它将物理内存分成固定大小的物理块和逻辑上相同大小的逻辑块，每个逻辑块被称为一页。当程序需要访问某一页时，该页会被调入物理内存中，如果物理内存已满，则需要使用页面置换算法将某些页替换出去，以便为新的页腾出空间。本实验旨在通过模拟作业执行过程，实现请求调页存储管理方式，并比较FIFO算法和LRU算法的效果，计算缺页率和页面置换次数，以验证这两种算法的优劣。
   1. 实验环境和工具  
      本实验使用Java语言编写，开发环境为Eclipse IDE，实验在Windows 10操作系统的计算机上进行。
   2. 简述问题

* 基本任务

假设每个页面可存放10条指令，分配给一个作业的内存块为4。模拟一个作业的执行过程，该作业有320条指令，即它的地址空间为32页，目前所有页还没有调入内存。

* 模拟过程
  + 在模拟过程中，如果所访问指令在内存中，则显示其物理地址，并转到下一条指令；如果没有在内存中，则发生缺页，此时需要记录缺页次数，并将其调入内存。如果4个内存块中已装入作业，则需进行页面置换。
  + 所有320条指令执行完成后，计算并显示作业执行过程中发生的缺页率。
  + 置换算法可以选用FIFO或者LRU算法
  + 作业中指令访问次序可以按照下面原则形成：
  + 50%的指令是顺序执行的，25%是均匀分布在前地址部分，25％是均匀分布在后地址部分。

1. 请求调页存储管理方式模拟

2.1 基本任务

本实验的基本任务是通过模拟作业执行过程来实现请求调页存储管理方式。每个页面可存放10条指令，分配给一个作业的内存块为4。模拟一个作业的执行过程，该作业有320条指令，即它的地址空间为32页，目前所有页还没有调入内存。在模拟过程中，如果所访问指令在内存中，则显示其物理地址，并转到下一条指令；如果没有在内存中，则发生缺页，此时需要记录缺页次数，并将其调入内存。如果4个内存块中已装入作业，则需进行页面置换。所有320条指令执行完成后，计算并显示作业执行过程中发生的缺页率。

2.2 模拟过程

本实验的模拟过程主要包括作业执行过程、缺页处理和页面置换。

2.2.1 作业执行过程

本实验中，作业执行过程是模拟作业中每条指令的执行过程。指令访问次序可以按照下面原则形成：50%的指令是顺序执行的，25%是均匀分布在前地址部分，25％是均匀分布在后地址部分。

具体实现时，可以先生成320条指令的序列，并将其分为三个部分，分别是前地址部分、中间部分和后地址部分。前地址部分和后地址部分各包含80条指令，中间部分包含160条指令。然后，按照上述原则分别对这三个部分进行随机排序，最终得到一个具有一定规律性的指令序列。

在模拟执行过程中，可以使用一个指令计数器PC来记录当前执行的指令的地址。每执行一条指令，PC的值就会增加1。同时，还需要维护一个页表，用于记录每一页是否在内存中、在内存中的位置等信息。通过Java的图形库将按钮和执行函数相关联，通过Timer设置页面数据更新动画。

2.2.2 缺页处理

当执行一条指令时，需要判断其所在的页是否在内存中。如果在内存中，直接读取该页的物理地址；如果不在内存中，则发生缺页，需要将该页调入内存并记录缺页次数。具体实现时，可以使用一个缺页计数器来记录缺页次数，同时还需要实现页面调度算法，以确定哪个页将被调出内存以腾出空间给新页。

常用的页面调度算法有FIFO算法和LRU算法。FIFO算法选择最先进入内存的页进行置换，而LRU算法则选择最近最少使用的页进行置换。在本实验中，我们需要实现这两种算法，并比较它们在不同情况下的效果。

FIFO算法：

设置一个访问内存块顺序的队列，利用队列实现FIFO要求的页面先入先出，当出现缺页中断时，队列首的内存块将现有页面移除，将需要的页面装入。

LRU算法：

设置一个储存内存块访问优先级的Vector，当使用某个内存块时，将该内存块的优先级设为最高，将其他的内存块的优先级降一级。当发生缺页中断时，选择优先级最低的内存块移除现有的页表，装入新的页表。

2.3 作业执行过程的实现

本实验中，作业执行过程的实现涉及到指令序列的生成、作业执行过程的模拟以及缺页率的计算等方面。主要是使用Java图形库的实现整个页面调换过程中UI界面中的更新动画。

实现过程中使用的数据结构：

MemManage类：

// 调度算法类型

**public** AlgoType algo;

// UIWidget, 各种输出信息的组件，如果是静态的组件直接在ui图中画好

**public** Vector<JLabel> vLcd; // 显示当前内存块所对应页的页号

**public** Vector<DefaultListModel<String>> vBlockList;// 显示当前内存块访问指令的历史

**public** Vector<JLabel> vLabelFrame; // 显示当前内存块存放页号的历史

**public** DefaultListModel<String> instListAll; // 所有的待访问指令序列

**public** JList<String> instAllPage;// 装指令数据的容器

**public** DefaultListModel<String> instListFault; // 发生缺页的指令序列

**public** JList<String> instFaultData;// 装缺页指令数据的容器

**public** DefaultListModel<String> logList; // 所有访问信息

**public** JList<String> logData;// 装访问数据的容器

**public** JTable qPageTable; // 一级页表

**public** Vector<Integer> instArray; // 指令数组

**public** Vector<Integer> block; // 记录各内存块当前所存页

**public** **int** sizeFrame; // 内存块大小

**public** **int** sizeInst; // 指令数量

**public** **int** capacityPage; // 每个页能存放的指令数量

**public** **int** currentInst; // 当前指令对应序号，我们假设指令顺序存放，那么就是以10个为单位顺序存在一个页中

**public** **int** pageFaultCount; // 缺页数量

**private** Vector<Integer> pageTable; // 页表中记录各页(1-32逻辑页)对应的物理内存页号

**private** Queue<Integer> queueFIFO; // 记录FIFO算法中各内存块优先级，对应于相应页的存入时间

**private** Vector<Integer> priorityLRU; // 记录LRU 算法中各内存块优先级，对应于未被使用时间

Frame图形窗口类：

// 私有成员

private JLabel fedinstnum;

private JLabel queinstnum;

private JLabel querate;

private JLabel ways\_label;

private JLabel choose\_label;

private JButton startButton;

private JButton exitButton;

private JButton fifoButton;

private JButton lruButton;

private JButton f1\_Button;

private JButton f10\_Button;

private JButton fAll\_Button;

2.3.1 生成指令序列

为了满足题目要求，我们需要生成一个具有一定规律性的指令序列。具体实现时，可以先生成320条指令的序列，然后将其分为三个部分，分别是前地址部分、中间部分和后地址部分。前地址部分和后地址部分各包含80条指令，中间部分包含160条指令。然后，按照上述原则分别对这三个部分进行随机排序，最终得到一个具有一定规律性的指令序列。使用Java.util.Random包实现产生0-320中的随机数指令，具体生成指令的函数为：

// 产生指令访问顺序

**public** **void** instOrderGenerate() {

Random rand = **new** Random();// 伪随机数生成器

**int** count = sizeInst / 4; // 一共320条指令，一次循环生成4条指令，一共循环80次，该处4与内存块数量无关，取决于指令生成算法

**for** (**int** i = 0; i < count; i++) {

**int** m = rand.nextInt(sizeInst); // [0, sizeInst)；由于指令可重复指令并不考虑320条指令全不同

**while** (m == 0) {

m = rand.nextInt(sizeInst); // 如果m是0，后续取余不方便

}

**int** m1 = rand.nextInt(m);// 50%顺序执行,25%前地址部份，25%后地址部份

instArray.set(i \* 4, m1);

instArray.set(i \* 4 + 1, m1 + 1);

**int** m2 = rand.nextInt(sizeInst - 1 - m1) + m1 + 2;// 后半部分

m2 = (m2 >= sizeInst - 1 ? sizeInst - 2 : m2);// 防止指令越界

instArray.set(i \* 4 + 2, m2);

instArray.set(i \* 4 + 3, m2 + 1);

}

}

2.3.2 模拟作业执行过程

为了模拟作业执行过程，我们需要使用一个指令计数器PC来记录当前执行的指令的地址。每执行一条指令，PC的值就会增加1。同时，还需要维护一个页表，用于记录每一页是否在内存中、在内存中的位置等信息。具体实现过程中，需要考虑数据储存和图形容器，需要在更新储存的内部数据的同时更新图形容器显示的数据，例如java中类似的JList视图类不可一般搭配DefaultListModel数据模型使用。

每次执行一条指令时，需要将当前访问的指令所在的页的页号计算出来。然后，检查该页是否在4个内存块的其中一个中。如果在内存中，则直接读取该页的物理地址；如果不在内存中，则发生缺页，需要将该页调入内存并记录缺页次数。具体实现时，可以使用一个缺页计数器来记录缺页次数。

在进行缺页处理时，需要先检查内存中是否还有空闲的物理块。如果有空闲的物理块，则直接将该页调入空闲的物理块中，并更新页表中对应页的valid和frame属性。如果没有空闲的物理块，则需要使用页面置换算法来确定哪个页将被调出内存以腾出空间给新页。调出的页需要写回磁盘，以保证数据的一致性。

2.4 页面置换算法的实现

常用的页面置换算法有FIFO算法和LRU算法。FIFO算法选择最先进入内存的页进行置换，而LRU算法则选择最近最少使用的页进行置换。在本实验中，我们需要实现这两种算法，并比较它们在不同情况下的效果。

2.4.1 FIFO算法

FIFO算法选择最先进入内存的页进行置换。具体实现时，可以使用一个队列来维护当前在内存中的页。每当有新页调入内存时，就将其加入队列尾部；每当需要进行页面置换时，就将队列头部的页调出内存并写回磁盘。

FIFO算法的优点是实现简单，适用于长时间作业。但是，它可能导致最近使用的页被频繁调出内存，从而增加缺页率。

代码：

// FIFO调度算法

**public** **void** AlgoFIFO(**int** inst) {

**for** (**int** i = 0; i < sizeFrame; i++) {

**if** (block.get(i) == inst / capacityPage) {

setBlock(i, inst); // 当前指令所在的指令页面已在内存，设置后返回

**return**;

}

}

// 下面缺页中断；分两种情况，需要换出与不需要

updateFault(inst);

// 第一种：刚开始有空闲内存

**for** (**int** i = 0; i < sizeFrame; i++) {

**if** (block.get(i) == -1) {

setBlock(i, inst);

**if** (queueFIFO.offer(i) == **false**)

System.***out***.println("内存块队列已满，添加失败！！！");

**return**;

}

}

// 第二种：需要替换某一个

**if** (queueFIFO.size() == sizeFrame) {

**int** freeBlockID = queueFIFO.remove();// 获取头部的内存块号并将其删除

setBlock(freeBlockID, inst);

**if** (queueFIFO.offer(freeBlockID) == **false**)// 将当前使用的内存块放到队列最后

System.***out***.println("内存块队列已满，添加失败！！！");

}

}

2.4.2 LRU算法

LRU算法选择最近最少使用的页进行置换。具体实现时，可以使用一个链表来维护当前在内存中的页，链表头部表示最近使用的页，链表尾部表示最久未使用的页。每当有新页调入内存时，就将其加入链表头部；每当需要进行页面置换时，就将链表尾部的页调出内存并写回磁盘。

LRU算法的优点是可以有效地利用内存空间，降低缺页率。但是，实现相对复杂，需要记录每个页最近的使用时间，而且在更新链表时需要进行大量的指针操作。这里通过优先级数组简化了操作，但每次需要查找最久未使用的内存块也需要消耗，本项目中的内存块一共只涉及4块影响不明显。

代码：

**public** **void** AlgoLRU(**int** inst) {

**for** (**int** i = 0; i < sizeFrame; i++) {

**if** (block.get(i) == inst / capacityPage) {

priorityLRU.set(i, 0);// i号内存块使用，优先级设为0

**for** (**int** j = 0; j < sizeFrame; j++) {

**if** (j != i && block.get(j) != -1) {

priorityLRU.set(j, priorityLRU.get(j) + 1);// 其他几个内存块的优先级全部降一级

}

}

setBlock(i, inst); // 当前指令所在的页面已在内存内存块，设置

**return**;

}

}

// 显示缺页中断；下面是缺页中断的两种情况，需要换出与不需要

updateFault(inst);

// 第一种：刚开始有空闲内存

**for** (**int** i = 0; i < sizeFrame; i++) {

**if** (block.get(i) == -1) {

priorityLRU.set(i, 0);

**for** (**int** j = 0; j < sizeFrame; j++) {

**if** (j != i && block.get(j) != -1) {

priorityLRU.set(j, priorityLRU.get(j) + 1);

}

}

setBlock(i, inst);

**return**;

}

}

// 第二种：缺页中断，换出最久未被访问内存

**int** maxIndex = 0;

**int** maxTime = -1;

**for** (**int** i = 0; i < sizeFrame; i++) {

**if** (priorityLRU.get(i) > maxTime) {

maxTime = priorityLRU.get(i);

maxIndex = i;

}

}

// 使用最久未被访问的内存块

priorityLRU.set(maxIndex, 0);

**for** (**int** i = 0; i < sizeFrame; i++) {

priorityLRU.set(i, priorityLRU.get(i) + 1);

}

setBlock(maxIndex, inst);

}

2.5 实验结果和分析

在模拟作业执行过程中，我们可以记录缺页次数和页面置换次数，并计算缺页率和页面置换效率。缺页率和页面置换效率是评价请求调页存储管理方式效果的重要指标，它们可以反映出系统对内存的利用程度。缺页次数是指在作业执行过程中发生的缺页次数，它可以用来衡量内存的利用程度。缺页

1. 总结
   1. 结论

本实验通过模拟作业执行过程，实现了请求调页存储管理方式，并比较了FIFO算法和LRU算法的效果。实验结果表明，LRU算法的缺页率低于FIFO算法，页面置换次数也更少。这是因为LRU算法能够更好地利用内存空间，避免了最近使用的页被频繁调出内存的问题。

同时，本实验还发现，作业执行过程中的指令访问次序对系统性能的影响比较大。当指令访问次序随机分布时，缺页率和页面置换次数都明显高于按照一定规律排序的情况。这提示我们在设计作业时应尽量避免随机的指令访问次序，以提高系统性能。

* 1. 建议

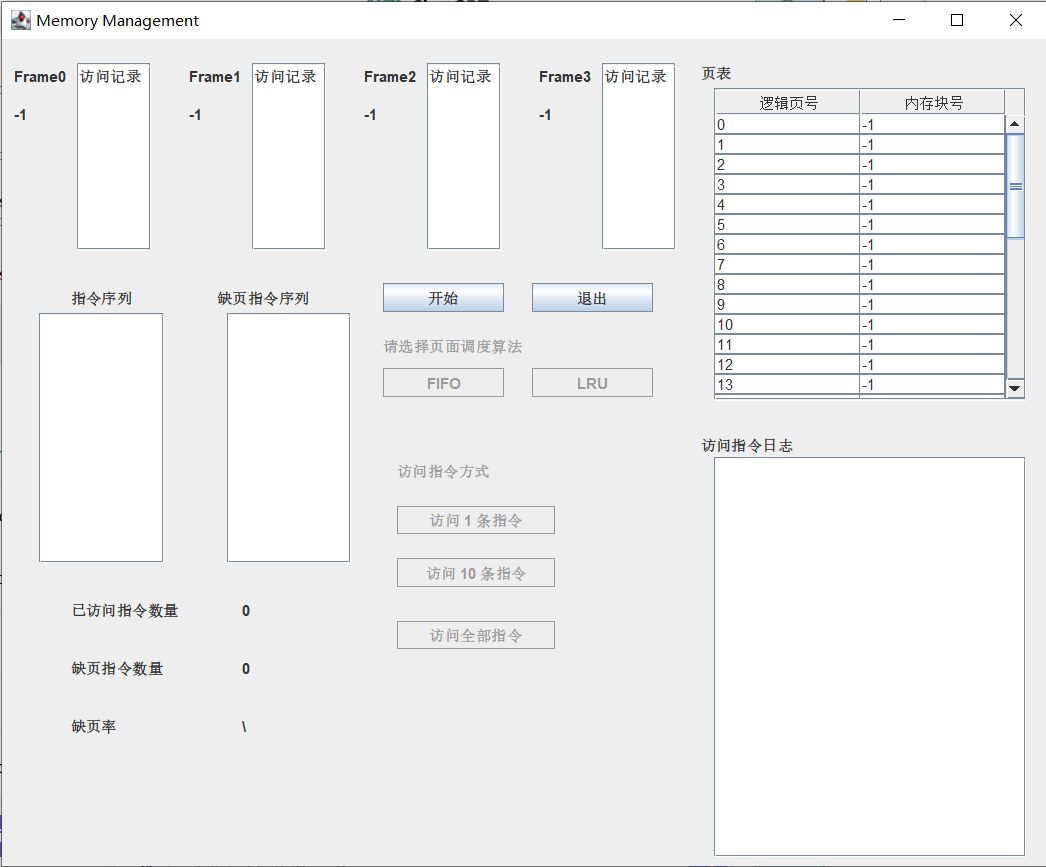
在本实验中，我们比较了FIFO算法和LRU算法在请求调页存储管理中的效果。实验结果表明，LRU算法的效果更好，建议在实际应用中优先考虑使用LRU算法。同时，需要注意作业的指令访问次序对系统性能的影响，应尽量按照一定规律排序以提高系统性能。

此外，在实际应用中，还需要考虑内存大小、作业大小、作业数量等因素对系统性能的影响，以选择合适的页面置换算法和调整合适的参数，以达到更好的性能表现。

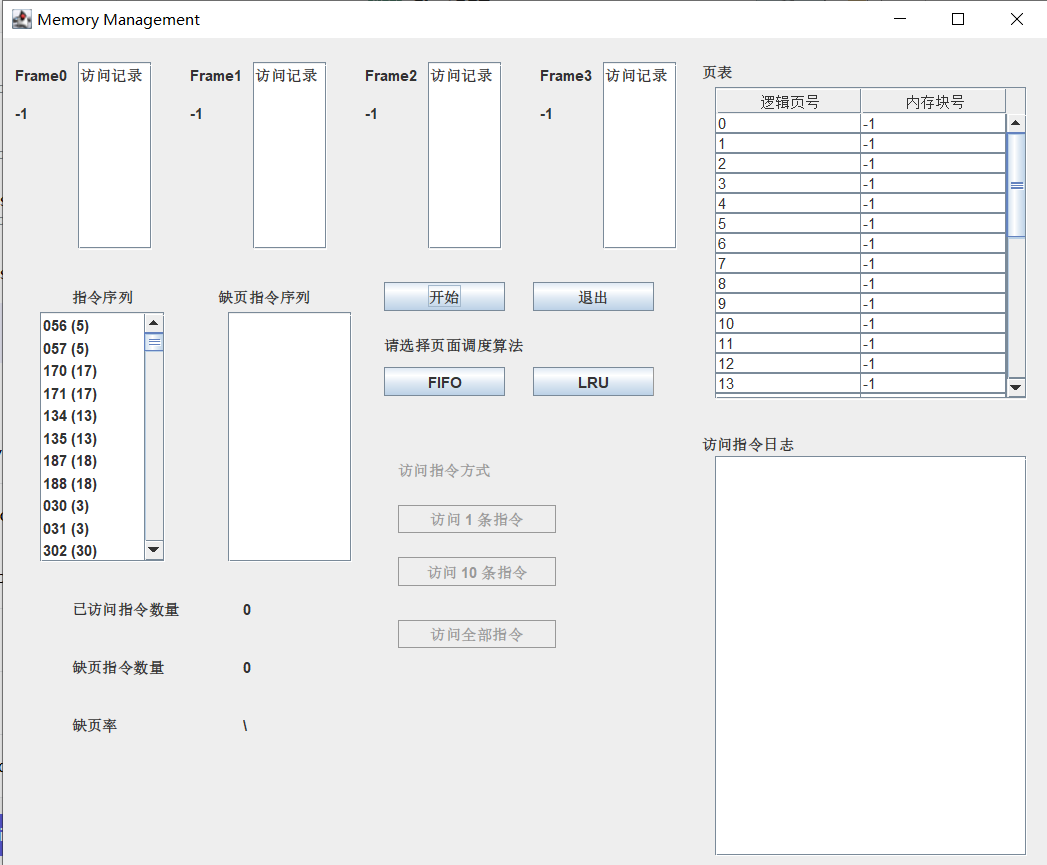
最后，本实验实现的请求调页存储管理方式仅为简单模拟，并且由于实验数据为320条并不是很多设计指令访问次序时也许有不合理的情况，最终的缺页率差异不够明显，应该改进。而在实际应用中可能会涉及更复杂的情况和更多的因素。因此，在实际应用中，需要根据具体情况进行选择和调整，以达到更好的系统性能。

1. UI界面的实现效果

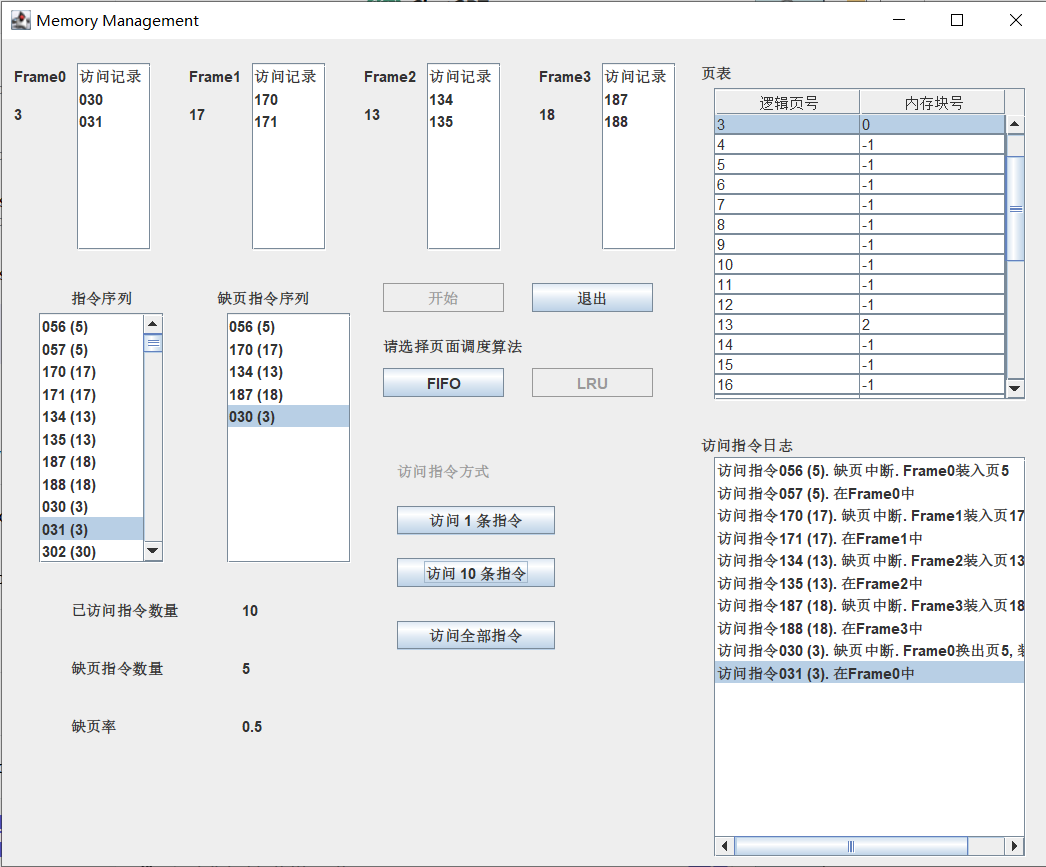
* “退出“按钮一直可用，任何时候点击即可退出程序。
* 内存块中未装入页面时显示-1
* 点击“开始“按钮——使”FIFO“和”LRU“按钮可用选择调度算法——访问方式按钮可用。
* 多条指令访问时，可看见每次页面更新数据的过程，而不是直接显示最终的结果
  1. 初始页面



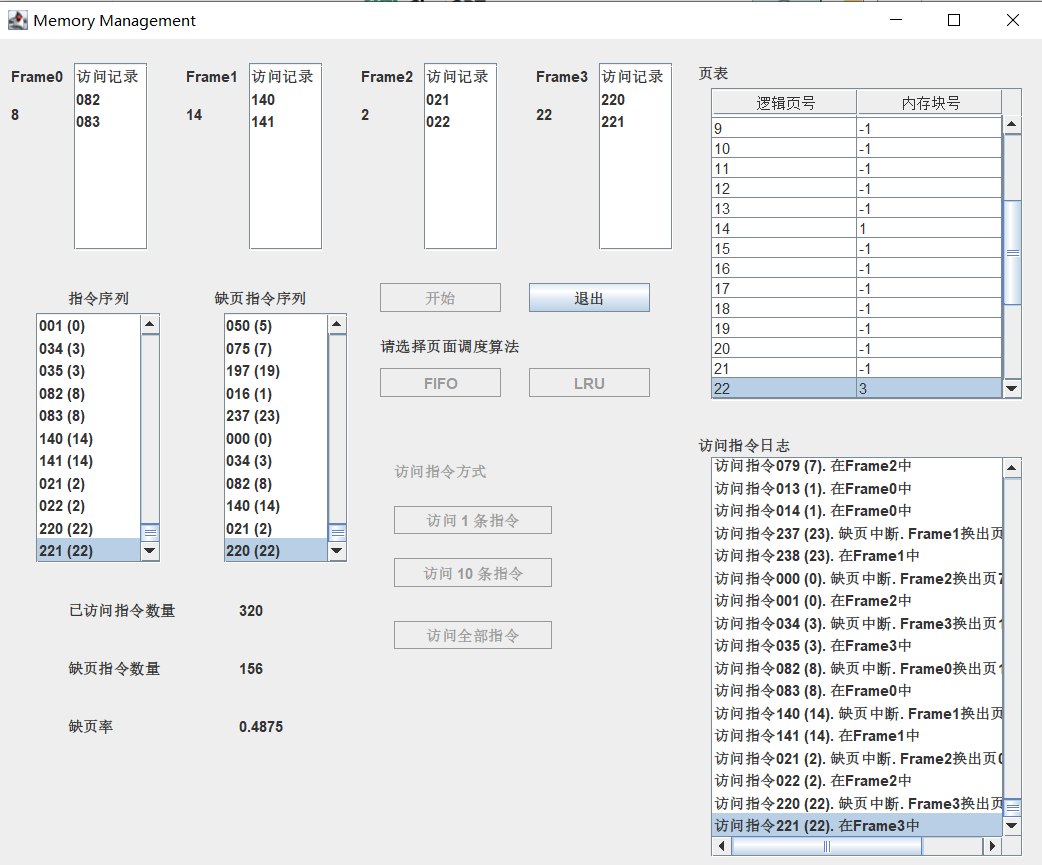
* 1. 点击“开始”后



* 1. 点击“访问10条指令”后



* 1. 访问完所有指令后



1. 附录代码

AlgoType.java\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

**package** Component;

**public** **enum** AlgoType {

***LRU***,***FIFO***

}

MemManage.java\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

package Component;

import javax.swing.JLabel;

import javax.swing.JTable;

import javax.swing.JList;

import javax.swing.DefaultListModel;

import javax.swing.SwingUtilities;

import java.awt.Rectangle;

import java.util.Vector;

import java.util.Queue;

import java.util.LinkedList;

import java.util.Random;

public class MemManage {

private static final int gSizeFrame = 4;

private static final int gSizeInst = 320;

private static final int gCapacityPage = 10; // 每个页面的容量为10，可以容纳10个指令

// 构造函数

public MemManage() {

initData();

}

// 析构函数

public void finalize() {

/\*

\* 是Java中的一个方法，用于在对象被垃圾回收之前执行一些清理操作。 这个方法通常被子类重写以实现自定义的清理逻辑。当一个对象即将

\* 被垃圾回收时，Java虚拟机会自动调用该对象的finalize()方法。但 是，由于finalize()方法的调用时机是不确定的，因此不建议在该方

\* 法中执行重要的清理操作。相反，应该使用try-finally块或者Java 7

\* 中引入的try-with-resources语句来确保资源的正确释放。希望这可 以帮助你更好地理解finalize()方法的作用和使用方式

\*/

System.out.println("程序已结束！");

}

// 数据空间初始化

public void initData() {

sizeFrame = gSizeFrame;

sizeInst = gSizeInst;

capacityPage = gCapacityPage;

// 初始化空间

instArray = new Vector<Integer>();

block = new Vector<Integer>();

priorityLRU = new Vector<Integer>();

pageTable = new Vector<Integer>();

queueFIFO = new LinkedList<Integer>();

vLcd = new Vector<JLabel>();

vBlockList = new Vector<DefaultListModel<String>>();

vLabelFrame = new Vector<JLabel>();

// 已访问指令数量和缺页指令数量的初始化

currentInst = 0;

pageFaultCount = 0;

instArray.setSize(sizeInst); // 指令大小初始化

block.setSize(sizeFrame); // 内存块数量初始化

priorityLRU.setSize(sizeFrame); // 各内存块优先级

pageTable.setSize(sizeInst / capacityPage); // 页表数目，为一级页表

// 指令地址初始化，也就是逻辑地址，我们假设指令顺序存放，那么就是以10个为单位顺序存放在一个页中

for (int i = 0; i < sizeInst; i++)

instArray.set(i, i);

// 内存块尚未存放任何逻辑页面

for (int i = 0; i < block.size(); i++)

block.set(i, -1);

// 页表内容初始化

for (int i = 0; i < pageTable.size(); i++)

pageTable.set(i, -1);

// LRU内存块优先级初始化，全部设为0

for (int i = 0; i < priorityLRU.size(); i++)

priorityLRU.set(i, 0);

}

// 产生指令访问顺序

public void instOrderGenerate() {

Random rand = new Random();// 伪随机数生成器

int count = sizeInst / 4; // 一共320条指令，一次循环生成4条指令，一共循环80次，该处4与内存块数量无关，取决于指令生成算法

for (int i = 0; i < count; i++) {

int m = rand.nextInt(sizeInst); // [0, sizeInst)；由于指令可重复指令并不考虑320条指令全不同

while (m == 0) {

m = rand.nextInt(sizeInst); // 如果m是0，后续取余不方便

}

int m1 = rand.nextInt(m);// 50%顺序执行,25%前地址部份，25%后地址部份

instArray.set(i \* 4, m1);

instArray.set(i \* 4 + 1, m1 + 1);

int m2 = rand.nextInt(sizeInst - 1 - m1) + m1 + 2;// 后半部分

m2 = (m2 >= sizeInst - 1 ? sizeInst - 2 : m2);// 防止指令越界

instArray.set(i \* 4 + 2, m2);

instArray.set(i \* 4 + 3, m2 + 1);

}

}

// 显示指令序列

public void instOrderShow() {

for (int i = 0; i < sizeInst; i++) {

String inst = String.format("%03d", instArray.get(i));

String instPage = String.valueOf(instArray.get(i) / capacityPage);

instListAll.addElement(inst + " (" + instPage + ")");

}

}

// 延时函数

public static void delay(int millis) {

SwingUtilities.invokeLater(() -> {

try {

Thread.sleep(millis);

} catch (InterruptedException e) {

// 处理中断异常

e.printStackTrace();

}

});

}

// 两种页面调度算法

// LRU调度算法

public void AlgoLRU(int inst) {

for (int i = 0; i < sizeFrame; i++) {

if (block.get(i) == inst / capacityPage) {

priorityLRU.set(i, 0);// i号内存块使用，优先级设为0

for (int j = 0; j < sizeFrame; j++) {

if (j != i && block.get(j) != -1) {

priorityLRU.set(j, priorityLRU.get(j) + 1);// 其他几个内存块的优先级全部降一级

}

}

setBlock(i, inst); // 当前指令所在的页面已在内存内存块，设置

return;

}

}

// 显示缺页中断；下面是缺页中断的两种情况，需要换出与不需要

updateFault(inst);

// 第一种：刚开始有空闲内存

for (int i = 0; i < sizeFrame; i++) {

if (block.get(i) == -1) {

priorityLRU.set(i, 0);

for (int j = 0; j < sizeFrame; j++) {

if (j != i && block.get(j) != -1) {

priorityLRU.set(j, priorityLRU.get(j) + 1);

}

}

setBlock(i, inst);

return;

}

}

// 第二种：缺页中断，换出最久未被访问内存

int maxIndex = 0;

int maxTime = -1;

for (int i = 0; i < sizeFrame; i++) {

if (priorityLRU.get(i) > maxTime) {

maxTime = priorityLRU.get(i);

maxIndex = i;

}

}

// 使用最久未被访问的内存块

priorityLRU.set(maxIndex, 0);

for (int i = 0; i < sizeFrame; i++) {

priorityLRU.set(i, priorityLRU.get(i) + 1);

}

setBlock(maxIndex, inst);

}

// FIFO调度算法

public void AlgoFIFO(int inst) {

for (int i = 0; i < sizeFrame; i++) {

if (block.get(i) == inst / capacityPage) {

setBlock(i, inst); // 当前指令所在的指令页面已在内存，设置后返回

return;

}

}

// 下面缺页中断；分两种情况，需要换出与不需要

updateFault(inst);

// 第一种：刚开始有空闲内存

for (int i = 0; i < sizeFrame; i++) {

if (block.get(i) == -1) {

setBlock(i, inst);

if (queueFIFO.offer(i) == false)

System.out.println("内存块队列已满，添加失败！！！");

return;

}

}

// 第二种：需要替换某一个

if (queueFIFO.size() == sizeFrame) {

int freeBlockID = queueFIFO.remove();// 获取头部的内存块号并将其删除

setBlock(freeBlockID, inst);

if (queueFIFO.offer(freeBlockID) == false)// 将当前使用的内存块放到队列最后

System.out.println("内存块队列已满，添加失败！！！");

}

}

// 指令inst将要进入id为blockid的内存块中

public void setBlock(int blockID, int inst) {

updateLog(blockID, inst);

// 不是目标页，需要替换出去

int instPage = inst / capacityPage;

if (instPage != block.get(blockID)) {

block.set(blockID, instPage);

vBlockList.get(blockID).clear();

// 清除原有界面中的指令信息，实际中需要把这块复制到磁盘里面

vBlockList.get(blockID).addElement("访问记录");

vLcd.get(blockID).setText(Integer.toString(instPage));

}

String accessInfo = String.format("%03d", inst);

vBlockList.get(blockID).addElement(accessInfo);

}

// 更新日志信息

public void updateLog(int blockID, int inst) {

String newPage = Integer.toString(inst / capacityPage);

String LogString = "访问指令" + String.format("%03d", inst);

LogString += " (";

LogString += newPage;

LogString += "). ";

int instPage = inst / capacityPage;

if (instPage == block.get(blockID)) {

LogString += "在Frame" + Integer.toString(blockID) + "中";

} else {

LogString += "缺页中断. ";

if (block.get(blockID) != -1) {

LogString += "Frame" + Integer.toString(blockID) + "换出页" + Integer.toString(block.get(blockID))

+ ", 装入页" + newPage;

updatePageTable(block.get(blockID), Integer.toString(-1));// 换出页面的内存块变回空—— -1

} else {

LogString += "Frame" + Integer.toString(blockID) + "装入页" + newPage;

}

updatePageTable(inst / capacityPage, Integer.toString(blockID));// 装入页的内存块

}

logList.addElement(LogString);

logData.setSelectedIndex(currentInst-1);// 设置选中的项

logData.ensureIndexIsVisible(currentInst-1);

}

// 更新页表

public void updatePageTable(int row, String newItem) {

qPageTable.setValueAt(newItem, row, 1);

qPageTable.setRowSelectionInterval(row, row);//该方法可以选中多行，设为相同值选中一行

Rectangle rect = qPageTable.getCellRect(row, 0, true);

qPageTable.scrollRectToVisible(rect);

}

// 更新缺页指令与缺页率

public void updateFault(int inst) {

String faultInst = String.format("%03d", inst);

String faultPage = Integer.toString(inst / capacityPage);// 页面号

instListFault.addElement(faultInst + " (" + faultPage + ")");

pageFaultCount++;// 缺页数增加

instFaultData.setSelectedIndex(pageFaultCount - 1);

instFaultData.ensureIndexIsVisible(pageFaultCount - 1);// 设置跟随显示

}

// 调度算法类型

public AlgoType algo;

// UIWidget, 各种输出信息的组件，如果是静态的组件直接在ui图中画好

public Vector<JLabel> vLcd; // 显示当前内存块所对应页的页号

public Vector<DefaultListModel<String>> vBlockList;// 显示当前内存块访问指令的历史

public Vector<JLabel> vLabelFrame; // 显示当前内存块存放页号的历史

public DefaultListModel<String> instListAll; // 所有的待访问指令序列

public JList<String> instAllPage;// 装指令数据的容器

public DefaultListModel<String> instListFault; // 发生缺页的指令序列

public JList<String> instFaultData;// 装缺页指令数据的容器

public DefaultListModel<String> logList; // 所有访问信息

public JList<String> logData;// 装访问数据的容器

public JTable qPageTable; // 一级页表

public Vector<Integer> instArray; // 指令数组

public Vector<Integer> block; // 记录各内存块当前所存页

public int sizeFrame; // 内存块大小

public int sizeInst; // 指令数量

public int capacityPage; // 每个页能存放的指令数量

public int currentInst; // 当前指令对应序号，我们假设指令顺序存放，那么就是以10个为单位顺序存在一个页中

public int pageFaultCount; // 缺页数量

private Vector<Integer> pageTable; // 页表中记录各页(1-32逻辑页)对应的物理内存页号

private Queue<Integer> queueFIFO; // 记录FIFO算法中各内存块优先级，对应于相应页的存入时间

private Vector<Integer> priorityLRU; // 记录LRU 算法中各内存块优先级，对应于未被使用时间

// data and structure

}

Frame.java:

package UI;

import Component.MemManage;

import java.awt.EventQueue;

import javax.swing.JFrame;

import javax.swing.JPanel;

import javax.swing.border.EmptyBorder;

import javax.swing.JLabel;

import javax.swing.JButton;

import javax.swing.JScrollPane;

import javax.swing.JTable;

import javax.swing.JList;

import javax.swing.DefaultListModel;

import javax.swing.Timer;

import java.awt.event.ActionEvent;

import java.awt.event.\*;

public class Frame extends JFrame {

// 在java中serialVersionUID是唯一控制着能否反序列化成功的标志， 只要这个值不一样，就无法反序列化成功。

private static final long serialVersionUID = 7794841109938881742l;// 强制修改Uid删除警告

// 主视图容器

private JPanel contentPane;

// 创建一个内存管理

private MemManage MM;

/\*\*

\* Launch the application.

\*/

public static void main(String[] args) {

EventQueue.invokeLater(new Runnable() {

public void run() {

try {

Frame frame = new Frame();

frame.setVisible(true);

} catch (Exception e) {

e.printStackTrace();

}

}

});

}

// 初始化UI界面

@SuppressWarnings("deprecation")

public void InitUI() {

MM = new MemManage();

// UI界面设置

for (int i = 0; i < MM.sizeFrame; i++) {

MM.vLabelFrame.add(new JLabel("Frame" + String.valueOf(i)));

MM.vLabelFrame.get(i).move(10 + i \* 140, 20);

MM.vLabelFrame.get(i).setSize(70, 20);

contentPane.add(MM.vLabelFrame.get(i));

}

// 数值

for (int i = 0; i < MM.sizeFrame; i++) {

MM.vLcd.add(new JLabel(String.valueOf(-1)));

MM.vLcd.get(i).move(10 + i \* 140, 30);

MM.vLcd.get(i).resize(60, 60);

contentPane.add(MM.vLcd.get(i));

}

// 显示器

for (int i = 0; i < MM.sizeFrame; i++) {

MM.vBlockList.add(new DefaultListModel<String>());

MM.vBlockList.get(i).addElement("访问记录");

JList<String> list = new JList<String>(MM.vBlockList.get(i));

JScrollPane midScPane = new JScrollPane(list);

midScPane.setViewportView(list);

midScPane.move(60 + i \* 140, 20);

midScPane.resize(60, 150);

contentPane.add(midScPane);

}

{ // 输出详细指令信息初始化UI

// 输出详细指令信息初始化UI

MM.instListAll = new DefaultListModel<String>();

MM.instAllPage = new JList<String>(MM.instListAll);

JScrollPane midScPane = new JScrollPane(MM.instAllPage);

midScPane.setViewportView(MM.instAllPage);

midScPane.move(30, 220);

midScPane.resize(100, 200);

contentPane.add(midScPane);

// 缺页指令

MM.instListFault = new DefaultListModel<String>();

MM.instFaultData = new JList<String>(MM.instListFault);

JScrollPane midScPane1 = new JScrollPane(MM.instFaultData);

midScPane1.setViewportView(MM.instFaultData);

midScPane1.move(180, 220);

midScPane1.resize(100, 200);

contentPane.add(midScPane1);

// 访问日志

MM.logList = new DefaultListModel<String>();

MM.logData = new JList<String>(MM.logList);

JScrollPane midScPane2 = new JScrollPane(MM.logData);

midScPane2.setViewportView(MM.logData);

midScPane2.move(570, 335);

midScPane2.resize(250, 320);

contentPane.add(midScPane2);

// 页表,并进行初始化，由于这里未使用如Tablemodel的表数据模型，只能将所有的页行添加，JTable不能直接添加行数据

String[] columnNames = { "逻辑页号", "内存块号" };

String[][] data = { { "0", "-1" }, { "1", "-1" }, { "2", "-1" }, { "3", "-1" }, { "4", "-1" },

{ "5", "-1" }, { "6", "-1" }, { "7", "-1" }, { "8", "-1" }, { "9", "-1" }, { "10", "-1" },

{ "11", "-1" }, { "12", "-1" }, { "13", "-1" }, { "14", "-1" }, { "15", "-1" }, { "16", "-1" },

{ "17", "-1" }, { "18", "-1" }, { "19", "-1" }, { "20", "-1" }, { "21", "-1" }, { "22", "-1" },

{ "23", "-1" }, { "24", "-1" }, { "25", "-1" }, { "26", "-1" }, { "27", "-1" }, { "28", "-1" },

{ "29", "-1" }, { "30", "-1" }, { "31", "-1" }, { "32", "\_avoid\_out\_" } };// 添加32号行防止越界

MM.qPageTable = new JTable(data, columnNames);

MM.qPageTable.setEnabled(false);

JScrollPane midScPane3 = new JScrollPane(MM.qPageTable);

midScPane3.setViewportView(MM.qPageTable);

midScPane3.move(570, 40);

midScPane3.resize(250, 250);

contentPane.add(midScPane3);

}

}

/\*

\* 下面几个是事件监听函数，处理按钮点击事件，实现用户交互

\*/

// 点击开始按钮

public void start\_clicked(ActionEvent e) {

MM.instOrderGenerate();

MM.instOrderShow();

choose\_label.setEnabled(true);

fifoButton.setEnabled(true);

lruButton.setEnabled(true);

}

// 点击退出按钮，终止程序

public void exit\_clicked(ActionEvent e) {

System.exit(0);

}

// 点击FIFO选择调度方式FIFO

public void fifo\_clicked(ActionEvent e) {

startButton.setEnabled(false);

lruButton.setEnabled(false);

MM.algo = Component.AlgoType.FIFO;

f1\_Button.setEnabled(true);

f10\_Button.setEnabled(true);

fAll\_Button.setEnabled(true);

}

// 点击LRU选择调度方式为LRU

public void lru\_clicked(ActionEvent e) {

startButton.setEnabled(false);

fifoButton.setEnabled(false);

MM.algo = Component.AlgoType.LRU;

f1\_Button.setEnabled(true);

f10\_Button.setEnabled(true);

fAll\_Button.setEnabled(true);

}

// 单步访问一条指令

// 访问一条指令

public void OneStep() {

// 指令已经访问完

if (MM.currentInst == MM.sizeInst) {

fifoButton.setEnabled(false);

lruButton.setEnabled(false);

startButton.setEnabled(false);

f1\_Button.setEnabled(false);

f10\_Button.setEnabled(false);

fAll\_Button.setEnabled(false);

return;

}

fedinstnum.setText(String.valueOf(MM.currentInst));

MM.instAllPage.setSelectedIndex(MM.currentInst);

MM.instAllPage.ensureIndexIsVisible(MM.currentInst);// 设置跟随显示

if (MM.algo == Component.AlgoType.FIFO)

MM.AlgoFIFO(MM.instArray.get(MM.currentInst++));

else if (MM.algo == Component.AlgoType.LRU)

MM.AlgoLRU(MM.instArray.get(MM.currentInst++));

// 刷新显示

float faultRate = (float) MM.pageFaultCount / (float) MM.currentInst;

fedinstnum.setText(String.valueOf(MM.currentInst));// 更新已经访问指令数量

queinstnum.setText(String.valueOf(MM.pageFaultCount));// 更新缺页数量

querate.setText(String.valueOf(faultRate));// 更新缺页率

}

// 点击访问1条指令、10条指令、全部指令

public void f1\_clicked(ActionEvent e) {

OneStep();

}

// 点击访问10条指令

public void f10\_clicked(ActionEvent event) {

// 访问过程中不能点击

f1\_Button.setEnabled(false);

fAll\_Button.setEnabled(false);

// for (int i = 0; i < 10 && MM.currentInst < MM.sizeInst; i++) {

// OneStep();

// }

// 设定可以看到每次的更新，避免直接看到最后结果

Timer timer = new Timer(30, null);

int nowInst=MM.currentInst;//记录当前指令位置，便于判断后面定时器执行了几次

ActionListener taskPerformer = e -> {

OneStep();

if (MM.currentInst == nowInst + 10 ||MM.currentInst >= MM.sizeInst)

timer.stop();

};

timer.addActionListener(taskPerformer);

timer.start();

f1\_Button.setEnabled(true);

fAll\_Button.setEnabled(true);

// 指令已经访问完

if (MM.currentInst == MM.sizeInst) {

fifoButton.setEnabled(false);

lruButton.setEnabled(false);

startButton.setEnabled(false);

f1\_Button.setEnabled(false);

f10\_Button.setEnabled(false);

fAll\_Button.setEnabled(false);

return;

}

}

// 点击访问所有剩余指令

public void fAll\_clicked(ActionEvent event) {

// 访问过程中不能点击

f1\_Button.setEnabled(false);

f10\_Button.setEnabled(false);

// while (MM.currentInst < MM.sizeInst) {

// OneStep();

// }

// 设定可以看到每次的更新，避免直接看到最后结果

Timer timer = new Timer(30, null);

ActionListener taskPerformer = e -> {

OneStep();

if (MM.currentInst == MM.sizeInst) {// 指令已经访问完

timer.stop();

fifoButton.setEnabled(false);

lruButton.setEnabled(false);

startButton.setEnabled(false);

f1\_Button.setEnabled(false);

f10\_Button.setEnabled(false);

fAll\_Button.setEnabled(false);

}

};

timer.addActionListener(taskPerformer);

timer.start();

}

// 构造函数，创建整体的UI

public Frame() {

setDefaultCloseOperation(JFrame.EXIT\_ON\_CLOSE);

setBounds(100, 100, 850, 700);

contentPane = new JPanel();

contentPane.setBorder(new EmptyBorder(5, 5, 5, 5));

// 设置窗口标题

setTitle("Memory Management");

InitUI();

setContentPane(contentPane);

contentPane.setLayout(null);

JLabel lblNewLabel = new JLabel("指令序列");

lblNewLabel.setBounds(55, 200, 58, 15);

contentPane.add(lblNewLabel);

JLabel lblNewLabel\_1 = new JLabel("缺页指令序列");

lblNewLabel\_1.setBounds(172, 200, 95, 15);

contentPane.add(lblNewLabel\_1);

JLabel lblNewLabel\_2 = new JLabel("已访问指令数量");

lblNewLabel\_2.setBounds(55, 450, 95, 15);

contentPane.add(lblNewLabel\_2);

JLabel lblNewLabel\_3 = new JLabel("缺页指令数量");

lblNewLabel\_3.setBounds(55, 496, 95, 15);

contentPane.add(lblNewLabel\_3);

JLabel lblNewLabel\_4 = new JLabel("缺页率");

lblNewLabel\_4.setBounds(55, 543, 71, 15);

contentPane.add(lblNewLabel\_4);

fedinstnum = new JLabel("0");

fedinstnum.setBounds(192, 450, 58, 15);

contentPane.add(fedinstnum);

queinstnum = new JLabel("0");

queinstnum.setBounds(192, 496, 58, 15);

contentPane.add(queinstnum);

querate = new JLabel("\\");

querate.setBounds(192, 543, 58, 15);

contentPane.add(querate);

choose\_label = new JLabel("请选择页面调度算法");

choose\_label.setBounds(305, 239, 127, 15);

choose\_label.setEnabled(false);

contentPane.add(choose\_label);

JLabel lblNewLabel\_9 = new JLabel("访问指令日志");

lblNewLabel\_9.setBounds(559, 318, 95, 15);

contentPane.add(lblNewLabel\_9);

JLabel lblNewLabel\_10 = new JLabel("页表");

lblNewLabel\_10.setBounds(559, 20, 58, 15);

contentPane.add(lblNewLabel\_10);

startButton = new JButton("开始");

startButton.addActionListener(this::start\_clicked);// 绑定事件监听

startButton.setBounds(305, 196, 97, 23);

contentPane.add(startButton);

exitButton = new JButton("退出");

exitButton.setBounds(424, 196, 97, 23);

exitButton.addActionListener(this::exit\_clicked);// 绑定事件监听

contentPane.add(exitButton);

fifoButton = new JButton("FIFO");

fifoButton.setBounds(305, 264, 97, 23);

fifoButton.addActionListener(this::fifo\_clicked);// 绑定事件监听

fifoButton.setEnabled(false);

contentPane.add(fifoButton);

lruButton = new JButton("LRU");

lruButton.setBounds(424, 264, 97, 23);

lruButton.addActionListener(this::lru\_clicked);// 绑定事件监听

lruButton.setEnabled(false);

contentPane.add(lruButton);

f1\_Button = new JButton("访问 1 条指令");

f1\_Button.setBounds(316, 374, 127, 23);

f1\_Button.addActionListener(this::f1\_clicked);// 绑定事件监听

f1\_Button.setEnabled(false);

contentPane.add(f1\_Button);

f10\_Button = new JButton("访问 10 条指令");

f10\_Button.setBounds(316, 416, 127, 23);

f10\_Button.addActionListener(this::f10\_clicked);// 绑定事件监听

f10\_Button.setEnabled(false);

contentPane.add(f10\_Button);

fAll\_Button = new JButton("访问全部指令");

fAll\_Button.setBounds(316, 466, 127, 23);

fAll\_Button.addActionListener(this::fAll\_clicked);// 绑定事件监听

fAll\_Button.setEnabled(false);

contentPane.add(fAll\_Button);

ways\_label = new JLabel("访问指令方式");

ways\_label.setBounds(316, 339, 127, 15);

ways\_label.setEnabled(false);

contentPane.add(ways\_label);

}

// 原来是在design页面创建的，现调整为私有成员方便设置值

// 私有成员

private JLabel fedinstnum;

private JLabel queinstnum;

private JLabel querate;

private JLabel ways\_label;

private JLabel choose\_label;

private JButton startButton;

private JButton exitButton;

private JButton fifoButton;

private JButton lruButton;

private JButton f1\_Button;

private JButton f10\_Button;

private JButton fAll\_Button;

}