内存管理项目——请求调页存储管理

项目需求

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。

功能描述

- 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
- **额外功能：可统计多组实验数据结果，获取更准确的算法缺页率**

项目目的

- 熟悉页面、页表、地址转换
- 熟悉页面置换过程
- 加深对请求调页系统的原理和实现过程的理解

开发环境

- **操作系统平台：**Windows 10
- **开发语言：**Java SE
  - **JDK版本：**jdk1.8.0_152
- **开发软件：**Eclipse IDE *2020-03 (4.15.0)*

系统分析

置换算法

FIFO算法（先进先出算法）

思想：当需要进行页面置换时，置换最早进入内存中的页。

数据结构：维护一个`FIFO`**队列**，每次需要置换页面时，将队首的页面调出，并将调入页加到队尾

- 判断当前所需页是否在内存中

    - 若在内存，则无需置换
- 若**内存中存在空页**，则直接将当前页调入
- 若**内存已满**，调出FIFO队列中队首的页面，将当前页调入该页所在的内存，并把当前页加入到队尾

LRU算法（最近最少使用算法）

思想：当需要进行页面置换时，置换最长时间没有使用的页。

数据结构：维护一个`duration`数组，记录内存中每页最近使用到当前的时间

- 判断当前所需页是否在内存中
    - 若在内存，则无需置换
- 若**内存中存在空页**，则直接将当前页调入
- 若**内存已满**，遍历`duration`数组，将其中数值最大的元素所对应的页调出，并将当前页调入该页所在的内存，同时将该页对应的`duration`置0
- 每执行一条指令，将所有页面的`duraion`加1

指令执行模式

指令的执行顺序原则为： 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

- 设当前指令序号为`m`，随机生成0~3内的一个数字
- 根据随机数生成下一条指令
    - 若随机数为0或2，且第`m + 1`条指令未执行，则顺序执行下一条指令
    - 若随机数为1，且第`0`到`m - 1`条指令中存在未执行的指令，则在其中随机选择一条执行
    - 若随机数为3，且第`m - 1`到`319`条指令中存在未执行的指令，则在其中随机选择一条执行
- 如果在上一步中未能生成下一条指令，则重复执行前两个步骤，直至成功生成下一条指令

*注：若严格按照推荐做法，可能会造成**越界**或**无法按照指定顺序执行**，因此采用上述做法*

系统设计

类设计

1. **PagingUI类**

   用于实现程序的界面显示

```
public class PagingUI extends JFrame {

    private JPanel contentPane;

    PagingUI self;

    /* 请求调页逻辑部分 */
    Paging paging;

    /* 页表 */
    JTable[] pageTable;
    DefaultTableModel[] pageTableModel;

    /* 物理内存块 */
    JLabel[] frameJl, numJl;

    /* 统计部分 */
    JLabel curOrderJl, nextOrderJl, executeOrderJl, faultJl,
rateJl;
    JLabel algoJl;

    /* 按钮 */
    JButton nextJb, next10Jb, allJb;

    /* 统计实验结果 */
    JTable FIFO, LRU;
    DefaultTableModel FIFOModel, LRUModel;
    JScrollPane FIFOjsp, LRUjsp;
    JLabel FIFOall, LRUall;

    public PagingUI(String title);

    /* 初始化表格 */
    private void initPageTable(int index);

    /* 执行下一条指令，界面更新 */
    private void nextOrder();

    /* 更新表格 */
    private void refreshTable();
```

```
    /* 重置 */
    void reset(int algo);
}
```

2. **Paging类**

   封装请求调页存储的逻辑部分

```
public class Paging {

    /* 请求分页算法 */
    int algo;

    static final int FIFO = 0;

    static final int LRU = 1;

    /* 共分配4页内存 */
    final int pagesForProcess = 4;

    /* 作业共32页，每页有10条指令 */
    final int pagesOfProcess = 32;

    /* 内部类，页表结构 */
    class Page {
        /* 页码 */
        int pageNumber;

        /* 内存页号 */
        int memoryPageNumber;

        /* 有效-无效位 */
        boolean isValid;

        Page(int num, int memoryNum) {
            pageNumber = num;
            memoryPageNumber = memoryNum;
            isValid = false;
        }

        Page(int num) {
            pageNumber = num;
            isValid = false;
        }
```

```
        void setMemoryPage(int page) {
            memoryPageNumber = page;
            isValid = true;
        }

        void outMemory() {
            isValid = false;
        }
    }

    /* 页表 */
    Page[] pageTable;

    /* 内存页 */
    int[] memoryPage;

    /* 当前执行的指令 */
    int curOrder = -1;

    /* 下一条指令 */
    int nextOrder = -1;

    /* 统计实验次数 */
    int count = 1;

    /*
     * FIFO算法用 先进先出，用队列实现
     */
    Queue<Integer> orderFIFO;

    /* LRU算法用，为每个物理内存中的页面设置访问位，记录最近一次访问到当前的时间 */
    int[] duration;

    /* 访问位 */
    boolean[] vis;

    /* 缺页次数 */
    int pageFault = 0;

    /* 已执行指令数 */
    int executedOrder = 0;

    /* 统计总次数 */
    int faultFIFO = 0, faultLRU = 0;
```

```
    int exeFIFO = 0, exeLRU = 0;

    public Paging(int algo);

    /* 执行下一条指令 */
    public void nextOrder();

    /* 获取下一条指令 */
    private void getNextOrder();

    /* 执行指令 */
    private void execute(int index);

    /* 获取指令在物理内存中的位置，若不在物理内存返回-1 */
    private int getMemoryPage(int index);

    /* 调页，返回调入的物理内存页码 */
    private int paging(int page);

    /* 重置 */
    public void reset(int algo);

    /* 返回当前实验缺页率 */
    public double getPageFaultRate();

    /* 返回FIFO算法平均缺页率 */
    public double getFIFOFaultRate();

    /* 返回LRU算法平均缺页率 */
    public double getLRUFaultRate();

    /* 获取页表 */
    Object[][] getPageList(int index);
}
```

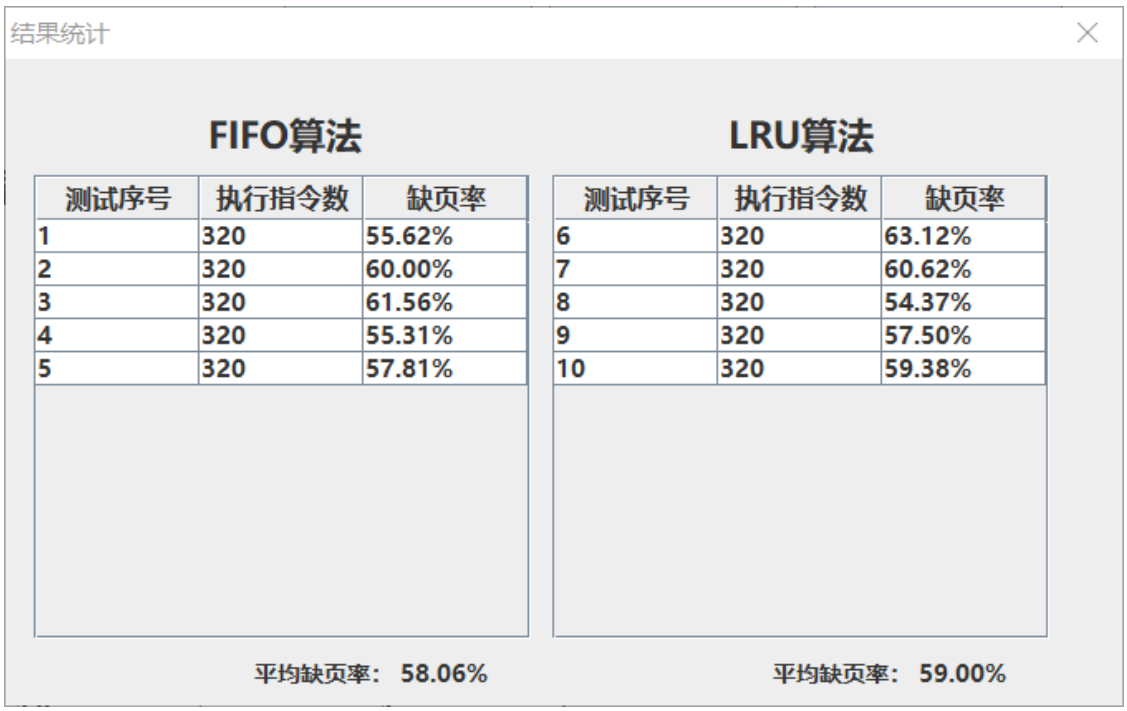
界面设计

1. 主界面
    - 显示当前执行指令序号、下一条指令序号
    - 显示已执行指令数
    - 显示缺页次数及当前**缺页率**
    - 显示当前**置换算法**
    - 显示**页表**状态
    - 显示**物理内存块**状态

- 执行指令按钮、执行10条指令按钮、执行所有指令按钮
- 重置按钮

2. 算法设置页面
   - 可选择**置换算法**

3. 结果统计页面
   - 分别显示各算法的每次实验结果
     - 测试序号、执行指令数、缺页率
   - 显示各算法的平均缺页率



常量设计

- 算法标识`algo`
  - `Paging.FIFO = 0`表示FIFO算法
  - `Paging.LRU = 1`表示LRU算法
- 内存相关
  - `pagesForProcess = 4` 分配4页物理内存
  - `pagesOfProcess  = 32` 作业共有32页，每页10条指令

系统实现

随机生成下一条指令

- 如果是第一条指令，则从0-319号指令随机生成一条指令
- 设当前指令序号为`m`，计算其前地址及后地址部分未执行的指令数，存于变量`left`和`right`
- 随机生成0-3中任意一个数
  - 若随机数为0或2，且第`m + 1`条指令未执行，则顺序执行下一条指令
  - 若随机数为1，且第`0`到`m - 1`条指令中存在未执行的指令
    - 随机生成`0`到`left - 1`中任意一个数`random`，执行从小到大第`random`条未执行的指令
  - 若随机数为3，且第`m - 1`到`319`条指令中存在未执行的指令，则在其中随机选择一条执行

    - 随机生成0到`right - 1`中任意一个数`random`，执行从小到大第`random`条未执行的指令
- 如果在上一步中未能生成下一条指令，则重复上一步，直至成功生成下一条指令
- 将下一条指令序号存于`nextOrder`中

```
private void getNextOrder() {
    if (executedOrder == 320) {
        nextOrder = -1;
        return;
    }

    Random rand = new Random();
    /* 随机选取开始指令 */
    if (curOrder == -1) {
        nextOrder = rand.nextInt(pagesOfProcess * 10);
        return;
    }

    int left = 0, right = 0;
    for (int i = 0; i < curOrder; i++) {
        if (!vis[i])
            left++;
    }
    for (int i = curOrder; i < pagesOfProcess * 10; i++) {
        if (!vis[i])
            right++;
    }

    int random = rand.nextInt(4), temp;

    while (true) {
        random = rand.nextInt(4);
        if (random == 0 || random == 2) {
            /* 顺序执行下一条 */
            if (curOrder + 1 != pagesOfProcess * 10 &&
!vis[curOrder + 1]) {
                nextOrder = curOrder + 1;
                return;
            }
        }
        if (random == 1) {
            /* 随机执行前地址中一条指令 */
            if (left == 0)
```

```
                continue;
            temp = rand.nextInt(left);
            for (int i = 0; i < curOrder; i++) {
                if (!vis[i]) {
                    if (temp == 0) {
                        nextOrder = i;
                        return;
                    } else
                        temp--;
                }
            }
        }
        if (random == 3) {
            /* 随机执行后地址中一条指令 */
            if (right == 0)
                continue;
            temp = rand.nextInt(right);
            for (int i = curOrder; i < pagesOfProcess * 10; i++)
{
                if (!vis[i]) {
                    if (temp == 0) {
                        nextOrder = i;
                        return;
                    } else
                        temp--;
                }
            }
        }
    }
}
```

请求调页

- 若有空页，则直接调入内存
- 按照相应置换算法进行置换

```
/* 调页，返回调入的物理内存页码 */
private int paging(int page) {

    /* 有空页直接调页 */
    for (int i = 0; i < pagesForProcess; i++) {
        if (memoryPage[i] == -1) {
            memoryPage[i] = page;
            pageTable[page].setMemoryPage(i);
            orderFIFO.add(i);
```

```
                return i;
            }
        }

        int index = -1;
        /* FIFO算法调页 */
        if (algo == FIFO) {
            index = orderFIFO.poll();
            orderFIFO.add(index);
        }

        /* LRU算法调页 */
        if (algo == LRU) {
            int max = -1;
            for (int i = 0; i < pagesForProcess; i++) {
                if (duration[i] > max) {
                    index = i;
                    max = duration[i];
                }
            }
        }
        pageTable[memoryPage[index]].outMemory();
        pageTable[page].setMemoryPage(index);
        memoryPage[index] = page;
        return index;
    }
```

执行指令

- 按下`nextJb`按钮，调用函数`execute()`执行指令
- 调用函数`getMemoryPage(index)`获取当前指令所在内存位置
    - 若在内存中，则直接执行指令，无需置换
- 调用函数`paging(index)`，按照相应置换算法进行调页
- 更新页表数据及算法对应数据结构
    - 对于FIFO算法，更新`orderFIFO`队列
    - 对于LRU算法，更新`duration`数组

```
/* 执行指令 */
private void execute(int index) {
    int loc = getMemoryPage(index);

    if (loc == -1) {
        /* 缺页 */
```

```
            int page = paging(index / 10);
            duration[page] = -1;
            pageFault++;
        } else {
            duration[loc] = -1;
        }

        /* 服务于LRU算法 */
        for (int i = 0; i < pagesForProcess; i++)
            duration[i]++;
        vis[index] = true;
    }

    /* 获取指令在物理内存中的位置，若不在物理内存返回-1 */
    private int getMemoryPage(int index) {
        if (pageTable[index / 10].isValid)
            return pageTable[index / 10].memoryPageNumber;
        else
            return -1;
    }
```

显示信息

- 每执行一条指令后，将调用函数`refreshTable()`更新页表、物理内存块、数据信息

```
    private void refreshTable() {
        pageTableModel[0].setRowCount(0);
        pageTableModel[1].setRowCount(0);

        /* 更新页表 */
        int index = 0;
        for (int i = 0; i < paging.pagesOfProcess; i++) {
            boolean valid = paging.pageTable[i].isValid;

            Object[] temp = { paging.pageTable[i].pageNumber, valid ?
    paging.pageTable[i].memoryPageNumber : "",
                    valid ? "v" : "i" };

            pageTableModel[index].addRow(temp);
            if ((i + 1) % 16 == 0)
                index++;
        }

        /* 更新物理内存 */
```

```
    for (int i = 0; i < paging.pagesForProcess; i++) {
        if (paging.memoryPage[i] == -1)
            frameJl[i].setText("");
        else {
            int temp = paging.memoryPage[i];
            frameJl[i].setText("页" + temp + " 指令" + temp * 10 +
"-" + (temp * 10 + 9));
            frameJl[i].setBackground(Color.pink);
        }
    }
}
```

重置系统

- 按下`resetJb`按钮进行重置
- 调用函数`reset()`，将主界面的显示设置为初始状态

注：此部分代码较长，且与核心逻辑关联较小，在文档中将不展示

选择置换算法

- 选择菜单栏“设置”--“算法设置”
- 在复选框中选择算法，并按下确定键`okJb`
- 执行**重置系统**操作，并将`paging`中属性`algo`设置为对应的`FIFO`或`LRU`

```
okJb.addActionListener(new ActionListener() {
    @Override
    public void actionPerformed(ActionEvent e) {
        PagingUI p = (PagingUI) parent;
        int num = comboBox.getSelectedIndex();
        p.reset(num);
        self.setVisible(false);
    }
});
```

统计实验结果

- 选择菜单栏“统计”--“缺页率”
- 跳出**结果统计**界面，显示统计结果

功能展示

选择置换算法

模拟过程

页号	物理内存位置	有效位
0		i
1		i
2		i
3		i
4	3	v
5	2	v
6		i
7		i
8		i
9		i
10		i
11		i
12		i
13		i
14		i
15		i

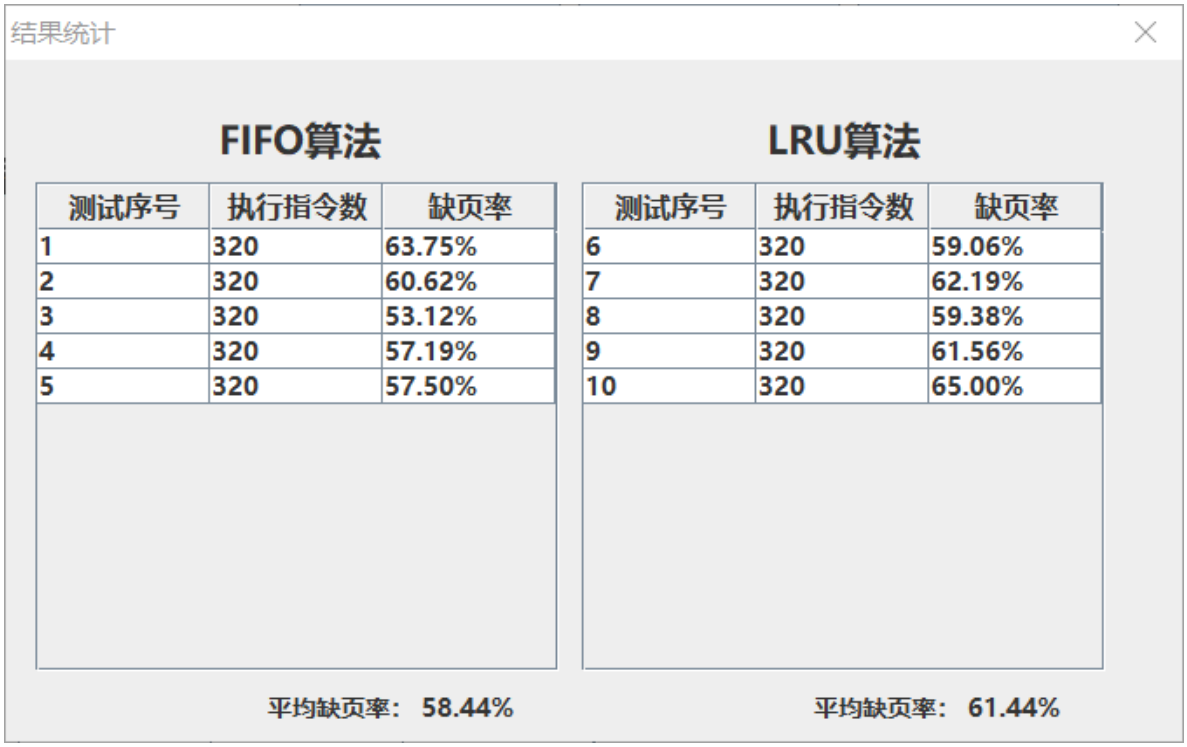
页号	物理内存位置	有效位
16		i
17		i
18		i
19		i
20		i
21		i
22		i
23		i
24		i
25	0	v
26		i
27		i
28	1	v
29		i
30		i
31		i

模拟结果

页号	物理内存位置	有效位
0		i
1		i
2		i
3		i
4		i
5		i
6		i
7		i
8		i
9		i
10		i
11		i
12		i
13	0	v
14		i
15		i

页号	物理内存位置	有效位
16	3	v
17		i
18		i
19		i
20	2	v
21	1	v
22		i
23		i
24		i
25		i
26		i
27		i
28		i
29		i
30		i
31		i

统计实验结果



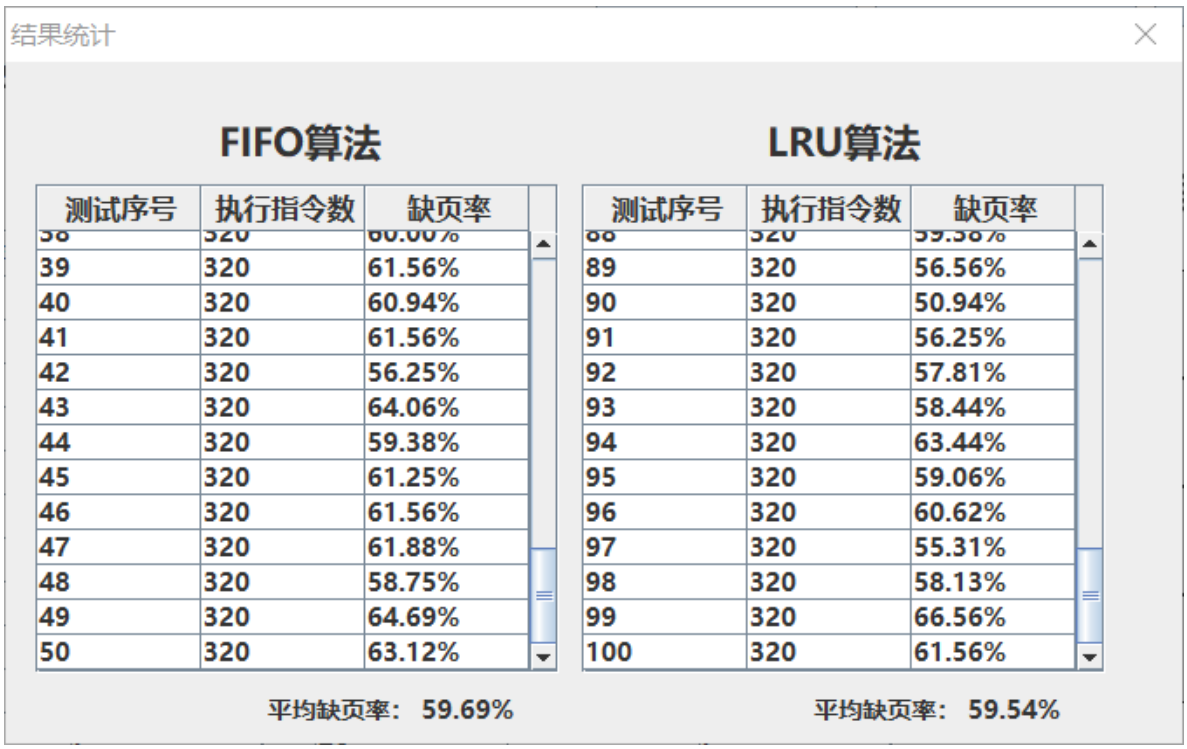
测试序号	执行指令数	缺页率
1	320	63.75%
2	320	60.62%
3	320	53.12%
4	320	57.19%
5	320	57.50%

测试序号	执行指令数	缺页率
6	320	59.06%
7	320	62.19%
8	320	59.38%
9	320	61.56%
10	320	65.00%

实验小结

实验结果对比

(FIFO算法和LRU算法各进行50次实验)



分析

- 理论来说，在实际情况下，LRU算法的缺页率比FIFO算法的缺页率要低，但实验结果显示两者的缺页率差不多，我认为原因如下：
  - 本系统采用的是随机产生下一条指令（50%顺序执行，50%从前地址或后地址任意执行）来进行模拟，而FIFO算法的缺点为**与进程访问内存的动态特征不符，最先进入的页面也很可能是被频繁访问的，从而导致缺页率较高**
  - 但本系统的随机模式很少出现这种情况，*如先访问第10页，再访问第8页，再跳回访问第10页*，因此FIFO算法和LRU算法在实验中的缺页率相近
- 本项目是对请求调页存储管理模式的模拟，对调页的具体过程有了更直观的了解，帮助我们更好理解请求调页的机制，但就各算法的缺页率而言，与实际情况还是略有偏差的