内存管理-请求分页分配方式-设计方案报告

学号	姓名	课号
1852137	张艺腾	42036901

目录

1. 项目需求

1.1 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。

1.2 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

1.3 项目目的

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

2. 开发环境

- **开发环境**：Windows 10
- **开发软件**：Eclipse
- **开发语言**：JavaSE (jdk1.8.0_241)
- **开发工具包**：Swing

3. 项目结构

```
|  memory.exe
|  memory.jar
|  README.md
|  内存管理 - 请求分页分配方式模拟.md
|  内存管理 - 请求分页分配方式模拟.pdf
|
└─src
    ├─Component
    |      AlgSelectBar.java
    |      Block.java
    |      EventListener.java
    |      Memory.java
    |      Moniter.java
    |      Page.java
    |      SpeedSlider.java
```

```
    |      WaitingList.java
    |
    └─UI
            MainFrame.java|  memory.jar
|  memory.exe
|  README.md
|  请求分区分配方式模拟_设计方案报告.md
|  请求分区分配方式模拟_设计方案报告.pdf
|
└─src
    ├─component
    |   AlgSelectBar.java
    |   Block.java
    |   EventListener.java
    |   Memory.java
    |   Moniter.java
    |   Page.java
    |   SpeedSlider.java
    |   WaitingList.java
    └─UI
        MainFrame.java
```

4. 操作说明

- 双击目录下的`memory.jar`(或`memory.exe`)文件进入模拟界面
  - 点击exe文件可能出现如下警告 -> 点击确定即可

- 在右上角的选项条中选择置换算法
  - FIFO-先进先出算法（默认值）
  - LRU-最近最少使用页面淘汰算法

  
- 点击开始模拟

  
- 滑动调节速度的滑块可以调整模拟速度

  
  - 慢速条件下可以看清**模拟调页的过程**

  
  - 快速条件下可以快速得到最终的结果——**缺页率**

  
- 点击数据清零可以清除本轮模拟产生的数据以进行下一轮模拟

5. 系统分析

5.1 置换算法

5.1.1 FIFO算法

- **当前页面已经在内存中** => 不需要进行调度，直接显示指令所在地址
- 当**内存中页面数小于分配给一个进程的内存容量（4页）**时 => 直接将页面顺序加入到内存的空闲块中，然后显示指令所在地址
- 当**内存满**时 => 每次替换掉最早进入内存块中的逻辑页面
  - 维护一个变量`turn`，每次执行一条指令，turn就自增1、模4，用来指定调出哪一块物理页中的逻辑页
    - `turn==0` 调出0号页面中的内容，将需要用的页调入物理0页
    - `turn==1` 调出1号页面中的内容，将需要用的页调入物理1页
    - `turn==2` 调出2号页面中的内容，将需要用的页调入物理2页
    - `turn==3` 调出3号页面中的内容，将需要用的页调入物理3页

5.1.2 LRU算法

- **当前页面已经在内存中** => 不需要进行调度，直接显示指令所在地址
- 当**内存中页面数小于分配给一个进程的内存容量（4页）**时 => 直接将页面顺序加入到内存的空闲块中，然后显示指令所在地址
- 当**内存满**时 => 每次替换掉最近最少使用的内存块中的页面
  - 用一个数组记录每个物理页的空闲次数
    - 每执行一条指令，未用到的物理页对应的空闲次数加一
    - 被执行到的指令所在的物理页空闲次数清零
  - 当遇到需要调页的情况时，选取空闲次数最多的页将其调出，将需要用到的页调入，并置该页的空闲次数为0，其余加一

5.2 指令产生方式

为了保证320条指令能够随机产生、均匀分布，模拟过程采用了下面这种循环产生指令的方式：

1. 在0~319条指令之间，随机选取一个起始执行指令，如序号为m
2. 顺序执行下一条指令，即序号为$m+1$的指令
3. 通过随机数，跳转到前地址部分0~$(m-1)$中的某个指令处，其序号为m_1
   - 若前面已经没有尚未执行的指令，则在全局范围内产生随机数，直到找到一个还未执行的指令
4. 顺序执行后面第一条未执行的指令，即序号为m_1+n的指令
5. 通过随机数，跳转到后地址部分(m_1+n+1)~319中的某条指令处，其序号为m_2
   - 若后面已经没有尚未执行的指令，则在全局范围内产生随机数，直到找到一个还未执行的指令
6. 顺序执行后面第一条未执行的指令，即序号为m_2+n的指令
7. 重复3~7的步骤直到执行完320条指令

6. 系统设计

6.1 类设计

6.1.1 算法选择条（AlgSelectBar）

6.1.2 页内代码块（Block）

```
public class Block extends JLabel {
    public Block() {
        setLayout(null);//设置布局方式
        setText("NULL");//设置初始文字
        setHorizontalAlignment(SwingConstants.CENTER);
        this.setBackground(new Color(88,201,185));
        setOpaque(true);
        setForeground(Color.white);
        setFont(new Font("楷体", Font.BOLD, 20));
    }
}
```

6.1.3 事件监听（EventListener）

```
public class EventListener implements ActionListener {
    public void actionPerformed(ActionEvent e) {
        System.out.println("I'm listening!");
    }
}
```

6.1.4 内存（Memory）

6.1.4.1 调度算法（dispatchPage）

```
public void dispatchPage(int physicPage, int logicalPage) {
    this.pages[physicPage].change(logicalPage);
}
```

6.1.4.2 检查内存中是否目标代码（check）

```
public int check(int num) {
    //检查是否有num对应的页在内存中
    //有则返回该页
    //否则返回-1
    for (int i = 0; i < 4; i++) {
        if (this.pages[i].lPage == num / 10) {
            return i;
        }
    }
    return -1;
}
```

6.1.4.3 清除内存（clear）

```
public void clear() {
    // 清空内存
    for (int i = 0; i < 4; i++) {
        pages[i].clear();
    }
}
```

6.1.5 显示器（Moniter）

6.1.6 页（Page）

6.1.7 速度选择滑块（SpeedSlider）

6.2 组件设计

- **窗体模型：** `Java. Swing. JFrame`
- **内存模型：** `Memory` 继承面板父类 `Java. Swing. JPanel`
  - 页 `component. Page`
  - 逻辑页号及物理页号 `Java. Swing. JLabel`
  - 内存文字 `Java. Swing. JLabel`
- **开始、清零按钮：** `Java.Swing.JButton`
- **速度选择条：** `Java.Swing.JSlider`
- **算法选择框：** `Java.Swing.JPanel`
  - 标题 `Java.Swing.JPanel`
  - 选择条 `Java.Swing.JList`
- **等待执行队列：** `Java.Swing.JPanel`
  - 指令 `Java.Swing.JLabel`
  - 标题 `Java.Swing.JLabel`
- **数码显示器：** `Java.Swing.JPanel`

7. 系统实现

7.1 320条指令的产生方式（MainFrame.next（））

- 先在320条指令中随机产生一条指令
- 第偶数条指令随机产生，其中
  - 第偶数条随机产生的指令从上一条指令之前产生
  - 第奇数条随机产生的指令从上一条指令之后产生
  - 模拟过程接近尾声时，大部分指令已经被执行过，容易出现在上一条指令执行之前或之后已经没有未执行的指令的情况，此时可以在整个指令范围内生成新的随机指令，直到找到一条可以执行的指令为止
- 第奇数条指令顺序执行

- 从上一条指令之后开始逐个向后找，找到第一个未执行的指令就返回该条指令的编号
- 可能出现编号大于319的情况，故需要让指令编号对319做模运算，也即最后找不到合适的指令就从头循环找

```
public int next(int cnt, int last) {
    //cnt--产生的第几个随机数[0,319]
    //last--上一条执行的指令序号
    //返回下一条执行的指令序号next
    int next = -1;

    //上一条是-1表示当前是第一条指令
    //则从0~319中随机产生一条
    if (last == -1) {
        next = (int) (Math.random() * 320);
        return next;
    }

    //第偶数条指令随机产生
    if (cnt % 2 == 0) {
        //第偶数条随机产生的指令从last前面产生
        if ((cnt / 2) % 2 == 0) {
            next = (int) (Math.random() * last);
            int times = 0;
            while (true == this.ins[next]) {
                times++;
                next = (int) (Math.random() * last);
                //如果last前面全部执行过了
                //就从所有指令中随机产生下一条
                if (times > last - 1) {
                    while (true == this.ins[next]) {
                        next = (int) (Math.random() * 320);
                    }
                }
            }
            //修改记录的标签位表示此指令已执行过
            this.ins[next] = true;
        }
        //第奇数条随机产生的指令从last后面产生
        else {
            next = (int) (last + Math.random() * (320 - last));
            int times = 0;
            while (true == this.ins[next]) {
                times++;
                next = (int) (last + Math.random() * (320 - last));
                //如果last后面全部执行过了
                //就从所有指令中随机产生下一条
                if (times > (319 - last)) {
                    while (true == this.ins[next]) {
                        next = (int) (Math.random() * 320);
                    }
                }
            }
            //修改记录的标签位表示此指令已执行过
            this.ins[next] = true;
        }
    }
    //第奇数条指令顺序产生--last之后第一条未执行的指令
```

```
    else {
        next = last + 1;
        //如果超过319则需要对320进行模运算
        //循环到最前面
        if (next > 319) {
            next = next % 320;
        }
        while (true == this.ins[next]) {
            next++;
            if (next > 319) {
                next = next % 320;
            }
        }
        //修改记录的标签位表示此指令已执行过
        this.ins[next] = true;
    }
    return next;
}
```

7.2 执行一条指令

- 初始情况先随机产生4条指令，放入等待队列（因为可视化的等待队列中可以放4条指令）
- 然后进行如下循环直到执行完全部的320条指令
  - 执行等待队列中最前面的一条指令
    - 在内存中 -> 显示该指令的物理地址（左侧内存块中高亮显示+右侧显示器显示该指令在内存i页中，无需调页）
    - 不在内存中 -> 按照用户所选算法（FIFO/LRU）将需要的页调入内存，然后在左侧内存中高亮显示，显示器显示i页调入内存，j页调出内存
  - 产生一条新指令，并加入等待队列
  - 休眠一段时间（由速度滑块决定）以方便用户观察调页过程

7.2.1 执行过程

```
new Thread(new Runnable() {
    // 要实时更新JLabel，所以需要单独开一个线程来刷新线程
    public void run() {
        MainFrame.clearButton.setEnabled(false);// 模拟过程中禁止点击清空按钮
        // 先生成前四个，加到等待队列
        for (int i = 0; i < 4; i++) {
            num = next(i, num);
            waitingList.turn(num);
        }

        // 逐个执行320条指令
        for (int i = 0; i < 320; i++) {
            /*
            * 此处是 FIFO 或 LRU
            */
            // 产生下一个待执行指令
            if (i < 316) {
                num = next(i, waitingList.insNum[3]);
```

```
                // 修改等待队列
                waitingList.turn(num);
            } else {
                waitingList.turn(-1);
            }

            // 休眠一段时间以方便观察
            try {
                Thread.sleep(speed);
            } catch (InterruptedException e) {
            // TODO Auto-generated catch block
                e.printStackTrace();
            }
        }
        // 显示结果——缺页率
        Moniter.setResult(((double) cnt_miss / (double) 320));
        // 恢复清零按钮的功能
        MainFrame.clearButton.setEnabled(true);
    }
}).start();
```

7.2.2 调度算法

FIFO

- 先进先出相当于物理页轮流调出其存储的逻辑页，故申请一个`int`型数据`turn`用来记录轮转到哪一页
- 遇到需要调页的情况直接将第turn个物理页所存放的逻辑页调出，调入所需页即可

```
int cnt_miss = 0;// 记录缺页次数
int turn = 0;// 先进先出相当于物理页轮流调出其存储的逻辑页，turn用来记录轮转到哪一页
int num = -1;// 当前要执行的指令的编号
num = waitingList.insNum[0];
// 判断在不在里面
if (memory.check(num) != -1) {
    // 在内存中，显示信息
    show(memory.check(num), num, true, -1);
} else {
    // 不在内存中，调度
    show(memory.check(num), num, false, turn);
    memory.dispatchPage(turn, num / 10);

    // 高亮显示目标指令所在位置
    Memory.pages[turn].blocks[num % 10].setBackground(new Color(209, 182,
225));
    try {
        Thread.sleep(speed);
    } catch (InterruptedException e) {
        // TODO Auto-generated catch block
        e.printStackTrace();
    }
    Memory.pages[turn].blocks[num % 10].setBackground(new Color(88, 201,
185));

    // 修改相应计数器
    turn++;
    cnt_miss++;
```

```
    if (turn > 3) {
        turn = turn % 4;
    }
}
```

LRU

- 申请一个`int`型数组`free`用来记录每个物理页的空闲次数
- 每当遇到要调出页的时候选取空闲次数最多的页调出
- 每执行一条指令就将所有物理页的空闲次数加一，然后将使用到的页的空闲次数置为0

```
int cnt_miss = 0;// 记录缺页次数
int[] free = new int[4];// 记录每个物理页的空闲次数
int num = -1;// 当前要执行的指令的编号
num = waitingList.insNum[0];
// 判断在不在里面
if (memory.check(num) != -1) {
    // 在内存中
    // 现将各页闲置次数加一
    for (int j = 0; j < 4; j++) {
        free[j]++;
    }
    // 再将当前执行的页限制次数置为零
    free[memory.check(num)] = 0;
    // 显示信息
    show(memory.check(num), num, true, -1);
} else {
    // 不在内存中，调度
    int turn = 0;// 要调度的页的物理页号
    int longest = 0;// 最长闲置时间
    // 判断调度哪一页
    for (int j = 0; j < 4; j++) {
        if (free[j] > longest) {
            turn = j;
            longest = free[j];
        }
    }
    // 现将各页闲置次数加一
    for (int j = 0; j < 4; j++) {
        free[j]++;
    }
    // 再将当前执行的页限制次数置为零
    free[turn] = 0;
    // 显示信息
    show(memory.check(num), num, false, turn);
    // 调度
    memory.dispatchPage(turn, num / 10);
    // 高亮显示目标指令所在位置
    Memory.pages[turn].blocks[num % 10].setBackground(new Color(209, 182,
225));
    try {
        Thread.sleep(speed);
    } catch (InterruptedException e) {
        // TODO Auto-generated catch block
        e.printStackTrace();
    }
```

```
    Memory.pages[turn].blocks[num % 10].setBackground(new Color(88, 201,
185));

    // 修改相应计数器
    cnt_miss++;
}
```

7.3 更新等待队列（WaitingList.turn()）

- 将前面三个还未执行的指令依次向前移动
- 将新产生的指令加到队列尾部

```
public void turn(int newIns) {
    for (int i = 0; i < 3; i++) {
        // 还未执行的前三个逐个前移
        insNum[i] = insNum[i + 1];
        lists[i].setText("" + insNum[i]);
    }
    // 新来的指令加在最后
    insNum[3] = newIns;
    lists[3].setText("" + insNum[3]);
    return;
}
```

7.5 打印信息（MainFrame.show()）

```
// 展示在内存中的信息
public void show(int pageNum, int num, boolean tag, int remove) {
    /*
     * pageNum--物理页号
     * num--指令编码
     * tag--是否命中
     * remove--调出页的页号
     */
    if (tag) {
        // 如果命中
        this.moniter.showInf(num, true, -1);// 展示信息
        // 高亮显示指令所在位置
        this.memory.pages[pageNum].blocks[num % 10].setBackground(new
Color(209, 182, 225));
        try {
            Thread.sleep(speed);
        } catch (InterruptedException e) {
            // TODO Auto-generated catch block
            e.printStackTrace();
        }
        this.memory.pages[pageNum].blocks[num % 10].setBackground(new
Color(88, 201, 185));
    } else {
        // 缺页
        int rem = this.memory.pages[remove].lPage;// 计算所调的逻辑页郝
        this.moniter.showInf(num / 10, false, rem);// 显示调页信息
```

```
    }
}
```

```
public void showInf(int ins,boolean IsContained,int remove) {
    /*
     * ins--当前执行代码所在的逻辑页页号
     * IsContained--该页是否在内存中
     * remove--将要移出的逻辑页页号
     */
    if(IsContained) {
        // 在内存中，显示信息
        this.result.setText("第"+ins+"页在内存中，不需要调度");
        this.inf.setText("");
    }else {
        // 不在内存中，显示信息
        this.result.setText("调出第"+remove+"页，调入第"+ins+"页");
        this.inf.setText("");
    }
}
```

```
static public void setResult(double rateOfMiss) {
    result.setText("缺页率："+rateOfMiss*100+"%");
}
```

7.6 数据清零（MainFrame.clear()）

主类中的`clear`函数分别调用内存、显示器和等待队列的`clear`函数，从而将所有组件恢复到初始状态。

```
// clear
public void clear() {
    waitingList.lists[0].setBackground(new Color(88, 201, 185));// 清除等待队列的高亮
    MainFrame.startButton.setEnabled(true);// 恢复开始按钮
    this.memory.clear();// 清除内存
    this.moniter.clear();// 清除显示器
    this.waitingList.clear();// 清空等待队列
    // 清除标识位
    for (int i = 0; i < 320; i++) {
        this.ins[i] = false;
    }
}
```

```
// Memory.clear()
public void clear() {
    // 清空内存
    for (int i = 0; i < 4; i++) {
        pages[i].clear();
    }
}

// Page.clear()
public void clear() {
```

```
    // 清空页
    for (int i = 0; i < 10; i++) {
        blocks[i].setText("NULL");
    }
    lPage = -1;
    this.logicalPage.setText("逻辑"+lPage+"页");
}
```

```
// Moniter.clear()
public void clear() {
    // 清空显示器
    this.result.setText("");
    this.inf.setText("");
}
```

```
// WaitingList.clear()
public void clear() {
    // 清空等待列表
    for (int i = 0; i < 4; i++) {
        this.insNum[i] = -1;
        this.lists[i].setText("" + insNum[i]);
    }
}
```

7.7 模拟速度调节

- `MainFrame`中有一个静态属性`static public int speed = 509;`用来表示模拟速度，`speed`默认值509，由滑动条动态改变
- 指令执行过程每执行一条指令就有一个休眠时间，通过改变`speed`来修改休眠时间长短，从而达到调节模拟速度的目的

```
// 休眠一段时间以方便观察
try {
    Thread.sleep(speed);
} catch (InterruptedException e) {
    // TODO Auto-generated catch block
    e.printStackTrace();
}
```

- 在`MainFrame`中添加速度滑块，并添加事件监听

```
// 添加速度滑块
SpeedSlider speedSlider = new SpeedSlider();
this.getContentPane().add(speedSlider);
speedSlider.setLocation(597, 420);
speedSlider.speed_.addChangeListener(new ChangeListener() {
    public void stateChanged(ChangeEvent e) {
        MainFrame.speed = 1009 - speedSlider.speed_.getValue();
    }
});
```

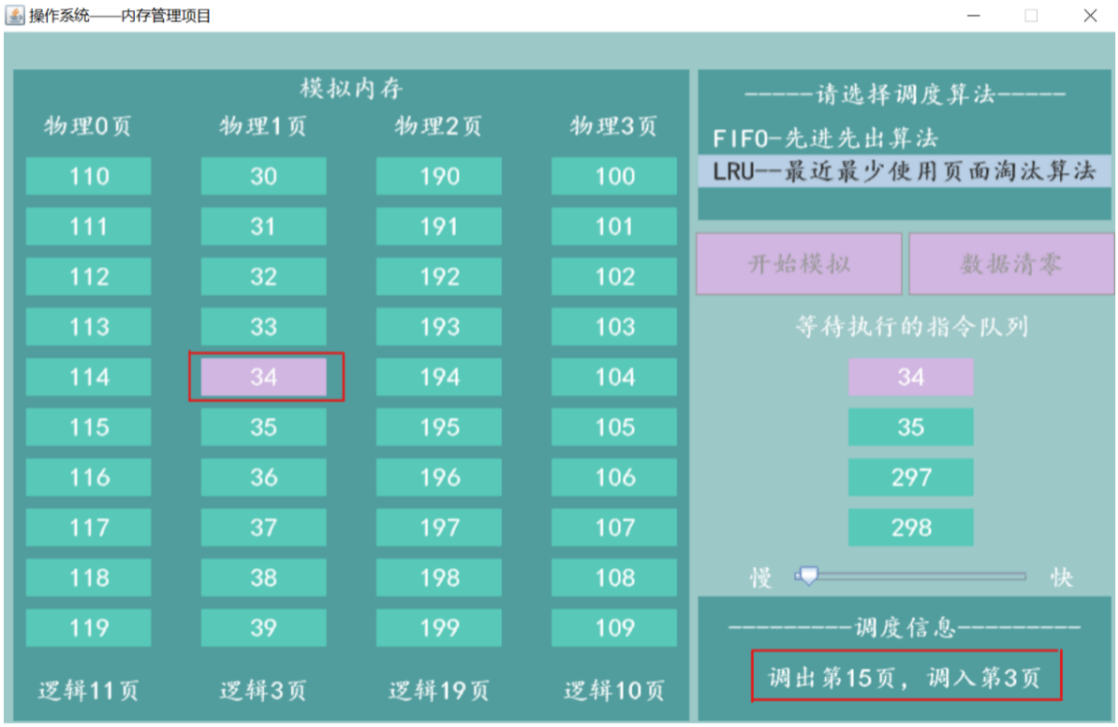
8. 功能实现截屏演示

- **初始界面**

- **选择模拟方式及速度**

- **模拟过程**
  - 命中

- 缺页



- 加快模拟速度得出结果

9. 实验结果分析

9.1 使用FIFO算法的十次模拟结果

1	2	3	4	5	6	7	8	9	10	平均缺页率
54.1%	51.9%	53.8%	55.9%	53.1%	52.2%	53.4%	55.9%	52.8%	54.4%	53.75%

9.2 使用LRU算法的十次模拟结果

1	2	3	4	5	6	7	8	9	10	平均缺页率
52.8%	53.8%	49.69%	50.9%	53.4%	51.6%	54.4%	53.1%	54.1%	55.0%	52.88%

9.3 分析

- 开始做这个项目之前，我大致预判了一下按照 随机-顺序-随机-顺序.....这样的执行方式，缺页率应该在50%左右，做好之后发现两种算法都在50%以上，仔细分析了一下发现
  - 随机产生新指令大概率不会命中
  - 顺序执行大部分情况下可以命中，但执行次数多了之后，顺序找下一条未执行的指令时往往需要往后找很多条，就可能出现跨两个页的情况(如125和132)，此时一般会出现不命中
  - 因此综合考虑这两点影响因素平均缺页率在50%以上是合理的
- 理论上讲LRU算法是优于FIFO算法的，缺页率相对较低，实验结果也证实了这一点，虽然LRU算法偶尔会出现57%~59%，但总体平均下来还是优于FIFO的。

作者

姓名： Kerr

联系方式： email:kerr99801@gmail.com