# 调页存储管理方式模拟器开发文档

## 1. 项目背景

调页存储管理是计算机内存管理的重要方式之一，在进行页面调度的过程中，不可避免的会出现内存块已满，需要进行页面调换的情况。此时，一个合理的页面调度算法往往能极大程度上降低程序后续执行过程中的缺页率，从而降低调页开销、提高程序运行的速度。常见的页面置换算法有 FIFO、LRU、OPT 和 LFU 等。这里我们选取 FIFO 和 LRU 两种页面调度算法进行调度模拟，来对两种页面调度算法进行比较和分析。

## 2. 功能需求

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。

※ 在模拟过程中，如果所访问指令在内存中，则显示其物理址，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。

※ 指令访问次序具体实施方法如下：

★ 在 0-319 条指令之间，随机选取一个其实执行指令，如序号为 m

★ 顺序执行下一条指令，即序号为 m+1 的指令

★ 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 $m_1$

★ 顺序执行下一条指令，即序号为 $m_1$+1 的指令

★ 通过随机数，跳转到后地址部分 m1+2~319 中的某条指令处，其序号为 $m_2$

★ 顺序执行下一条指令，即 m2+1 处的指令

★ 重复跳转到前地址部分、顺序执行、跳转到后地址部分、顺序执行、顺序执行的过程，直到执行完 320 条指令。

## 3. 开发环境

为了方便 UI 设计以及对页面调度模拟器随时随地地访问，这里选择 Web 开发模式。

1. 客户端的开发建立在 HTML5 + CSS + jQuery 环境下，开发测试工具为 WebStorm 和 Chrome。

2. 服务端的服务器选用阿里云 ECS 云服务器，服务器的系统为 Windows Server 2012，网络服务器搭建的环境为:Apache + MySQL + PHP5，php 开发工具为 Sublime Text 3，需要结合前端进行测试。

## 4. 设计思路

### 4.1. 客户端设计思路

客户端负责与用户的实时交互，需要将页面调度的结果以图形化界面的方式反馈给用户。客户端的设计分为控制部件设计和结果反馈组件的设计。

这里对项目需求进行了一定程度上的扩展——内存块的大小和每个页面的指令条数并不固定，用户可以根据需求对这两个参量进行

相应的调整，从而可以比较某一具体算法下页面调度情况随内存块的大小和页面指令条数的变化。

对于结果反馈部分，通过扇形图来直观地向用户呈现出缺页率，而详细的缺页情况则以表格的方式呈现出来。

### 4.2. 服务端设计思路

服务端的业务逻辑较为简单，这里选择项目需求中给定的指令执行算法来确定指令的执行顺序。其次，我们需要定义数据结构来存储每种算法的缺页情况，最后返回给前端。

## 5. 项目开发

### 5.1. Web 端开发

#### 5.1.1. jQuery 全局变量设置

```
//FIFO 算法缺页记录数组
var FIFOPageFault = [];
//LRU 算法缺页记录数组
var LRUPageFault = [];
//FIFO 算法缺页率
var FIFOPageFaultRatio = 0.0;
//LRU 算法缺页率
var LRUPageFaultRatio = 0.0;
//记录模拟器状态
var running = false;
```

全局变量 FIFOPageFault 负责接收并存储服务端传来的利用 FIFO 算法进行页面调度的缺页记录；

全局变量 LRUPageFault 负责接收并存储服务端传来的利用 LRU 算法进行页面调度的缺页记录；

全局变量 FIFOPageFaultRatio 负责接收并存储服务端传来的利用 FIFO 算法进行页面调度的缺页率；

全局变量 LRUPageFaultRatio 负责接收并存储服务端传来的利用 LRU 算法进行页面调度的缺页率；

全局变量 running 负责标识模拟器当前的状态：当其值为 false 表示模拟器当前处于空闲等待状态，若值为 true 表示模拟器当前处于运行状态。

### 5.1.2. 前后端数据交互

客户端的核心部分为向后端 POST 数据以及接受服务端传来的数据的过程，具体实现细节如下所示：

```
6.      $.post("MemoryManagement.php",
7.          {
8.              MemoryBlockNum: MemoeryBlockNum,
9.              PageInstructionNum: PageInstructionNum
10.         },
11.         function(data,status){
12.             var json_data = JSON.parse(data);
13.             FIFOPageFault = json_data.FIFOPageFault;
14.             FIFOPageFaultRatio = parseFloat(json_data.FIFOPageFaultRatio);
15.             LRUPageFault = json_data.LRUPageFault;
16.             LRUPageFaultRatio = parseFloat(json_data.LRUPageFaultRatio);
17.             barAnimation(1, FIFOPageFaultRatio, document.getElementsByClassName("bar-
        container-1")[0]);
18.             tableAnimation(0, FIFOPageFault, "f");
19.             barAnimation(1, LRUPageFaultRatio, document.getElementsByClassName("bar-
        container-2")[0]);
20.             tableAnimation(0, LRUPageFault, "l");
21.         }
22.     );
```

这里客户端需要向后台的 php 文件传送 MemoryBlockNum（内存块大小）和 PageInstructionNum（页面指令条数），这两者是用户自行调节的（调参方式请参考使用说明）。

在接收到服务端传来的数据时，首先要将数据解析成 json 格式，然后使用 5.1.1.节中所给出的全局变量来存储相应的数据。赋值完成后，我们下一步就需要调用相应的函数来将此次调页结果呈现给用户——函数 barAnimation( )负责实现显示缺页率的扇形图部分的动画，函数 tableAnimation( )负责实现显示缺页情况的表格部分的动画。

### 5.1.3. 动画实现

① barAnimation 函数

```
function barAnimation(i, percent, element){
    if(i > percent * 100){
        percent *= 100;
        percent = percent.toFixed(4);
        element.parentNode.querySelector("span").innerHTML = percent + "%";
        running = false;
        return;
    }
    updateDonut(i, element);
    setTimeout(function () {
        i++;
        barAnimation(i, percent, element);
    }, 60);
}

function updateDonut(percent, element){
    if (percent < 50){
        offset = (360 / 100) * percent;
        element.parentNode.querySelector("#section3").style.webkitTransform = "rotate(" + offset + "deg)";
        element.parentNode.querySelector("#section3 .item").style.webkitTransform = "rotate(" + (180 - offset) + "deg)";
        element.parentNode.querySelector("#section3").style.msTransform = "rotate(" + offset + "deg)";
        element.parentNode.querySelector("#section3 .item").style.msTransform = "rotate(" + (180 - offset) + "deg)";
        element.parentNode.querySelector("#section3").style.MozTransform = "rotate(" + offset + "deg)";
        element.parentNode.querySelector("#section3 .item").style.MozTransform = "rotate(" + (180 - offset) + "deg)";
        element.parentNode.querySelector("#section3 .item").style.backgroundColor = "#41A8AB";
    } else {
        offset = ((360 / 100) * percent) - 180;
        element.parentNode.querySelector("#section3").style.webkitTransform = "rotate(180deg)";
        element.parentNode.querySelector("#section3 .item").style.webkitTransform = "rotate(" +  offset + "deg)";
        element.parentNode.querySelector("#section3").style.msTransform = "rotate(180deg)";
        element.parentNode.querySelector("#section3 .item").style.msTransform = "rotate(" +  offset + "deg)";
        element.parentNode.querySelector("#section3").style.MozTransform = "rotate(180deg)";
        element.parentNode.querySelector("#section3 .item").style.MozTransform = "rotate(" +  offset + "deg)";
        element.parentNode.querySelector("#section3 .item").style.backgroundColor = "#E64C65";
    }
}
```

上述 updateDonut( )函数实现对于扇形图状态的更新：根据其 percent 参量来设置扇形图中红色扇形块所占比例。为了实现扇形图的动画效果，barAnimation( )封装了 updateDonut 的调用细节，并

通过 setTimeOut( )函数进行延迟，延迟时间设为 60ms，这样就形成了一个动画帧，然后通过递归调用 barAnimation 来将这样一组动画帧连接起来，形成动画。

② tableAnimation 函数

```
function tableAnimation(i, PageFaultArray, tag){
    if(i == PageFaultArray.length){
        return;
    }
    var selector = "td." + tag + (i + 1).toString();
    $(selector).css("background-color", "#F2385A");
    $(selector).text(PageFaultArray[i].toString());
    setTimeout(function () {
        tableAnimation(++i, PageFaultArray, tag);
    }, 20);
}
```

该函数的逻辑较为简单，基本操作是：通过 jQuery 的选择器选择标签并更改标签的样式。动画效果的实现也是通过递归调用，这与 barAnimation 大同小异。

## 5.2. 服务端开发

服务端的逻辑较为简单，主要工作是模拟项目需求中所给的指令执行顺序算法，具体方法封装在了 MemoryManagement 类中。

### 5.2.1. 类属性

MemoryManagement 类中的属性如下所示，为了增加代码可读性，降低开发难度，这里的变量命名与前端保持一致。

```
private $InstructionSet;
private $FIFOMemoryBlock;
private $LRUMemoryBlock;
private $FIFOPageFault;
private $LRUPageFault;
private $MemoryBlockNum;
private $PageInstructionNum;
```

其中变量 InstructionSet 用于模拟 320 条指令在辅存中的相对位置关系；变量 MemoryBlockNum 和 PageInstructionNum 接收前端传来的数据。

### 5.2.2. 类方法

① MemoryManagement( )

```
1.  private function MemoryManagement(){
2.      $InstructionSet = $this->InstructionSet;
3.      $InstructionPointer = mt_rand(0, 319);
4.      //顺序执行两条指令
5.      $this->SequentialInstruction($InstructionSet, $InstructionPointer);
6.      while (!emptyempty($InstructionSet)) {
7.          if($InstructionPointer > 0){
8.              //跳转到前面顺序执行两条指令
9.              $InstructionPointer = mt_rand(0, $InstructionPointer - 1);
10.             $this->SequentialInstruction($InstructionSet, $InstructionPointer);
11.
12.         }
13.         if($InstructionPointer <= count($InstructionSet) - 1){
14.             //跳转到后面顺序执行两条指令
15.             $InstructionPointer = mt_rand($InstructionPointer, count($InstructionSe
    t) - 1);
16.             $this->SequentialInstruction($InstructionSet, $InstructionPointer);
17.         }
18.     }
19. }
```

我们用 mt_rand( )函数来初始化 InstructionPointer（整型变量，储存的是当前执行的指令在 InstructionSet 中的下标）。然后，按照项目需求中给出的算法：先顺序执行两条指令，然后跳转到前地址指令指令处顺序执行两条指令，最后跳转到后地址指令处，这样就完成了一个周期，重复此过程，直到所有指令执行完毕。

② SequentialInstruction( )

```
1.  private function SequentialInstruction(&$InstructionSet, &$InstructionPointer){
2.      for($i = $InstructionPointer; $i - $InstructionPointer <= 1 && $i < count($Inst
    ructionSet); $i++){
3.          $this->PageSearch($this->FIFOMemoryBlock, $this->FIFOPageFault, floor($Inst
    ructionSet[$i] / $this->PageInstructionNum) + 1, FALSE);
4.          $this->PageSearch($this->LRUMemoryBlock, $this->LRUPageFault, floor($Instru
    ctionSet[$i] / $this->PageInstructionNum) + 1, TRUE);
5.      }
6.      array_splice($InstructionSet, $InstructionPointer, $i - $InstructionPointer);
7.  }
```

此函数负责顺序执行两条指令（当然，顺序情况下的指令条数不足两条的时候除外）。针对每条指令，我们都会分别在 FIFO 和 LRU 两种情况下调用 PageSearch( )函数。

需要特别注意的是：每次执行完执行，我们都会把该条指令从 InstructionSet 中删除，表示该条指令不会再次被执行到。

③ PageSearch( )

```
1.  private function PageSearch(&$MemoryBlock, &$PageFault, $PageNum, $Tag){
2.      for($i = 0; $i < count($MemoryBlock); $i++){
3.          if($MemoryBlock[$i] == $PageNum){
4.              if($Tag){
5.                  $Temp = $MemoryBlock[$i];
6.                  array_splice($MemoryBlock, $i, 1);
7.                  $MemoryBlock[count($MemoryBlock)] = $Temp;
8.              }
9.              return;
10.         }
11.     }
12.     $PageFault[count($PageFault)] = $PageNum;
13.     if(count($MemoryBlock) == $this->MemoryBlockNum){
14.         array_splice($MemoryBlock, 0, 1);
15.         $MemoryBlock[$this->MemoryBlockNum - 1] = $PageNum;
16.     }else{
17.         $MemoryBlock[count($MemoryBlock)] = $PageNum;
18.     }
19. }
```

该函数接收的参数 Tag 为一个 bool 型变量，当其为 false 时表示当前执行的是 FIFO 页面调度算法，其值为 true 时表示当前执行的是 LRU 页面调度算法。

★ FIFO 页面调度算法：

(1) 当内存块中存在目标页面时，直接返回，不做进一步处理。

(2) 当在内存块中未找到目标页面时，如果内存块未满，则直接将页面追加到数组末尾。如果内存块已满，则将内存块数组中的第一个元素删除，并在数组末尾插入目标页面。每次调入新的页面都会将其插入数组末尾，这样一来数组的首元素就是最早被调入内存的，所以在页面调换的时候仅需将首元素删除。

★ LRU 页面调度算法：

(1) 当内存块中存在目标页面时，将内存中的目标页面元素位置压到数组末尾，其余元素（这里指原来在目标页面元素之后的）按次序向前移动一个位置。

(2) 当在内存块中未找到目标页面时，如果内存块未满，则直接将目标页面追加到数组尾部。如果内存块已满，则将内存块数组中的第一个元素删除，并在数组末尾插入目标页面。每次使用页面都会将其位置压到数组末尾，这样一来数组的首元素就是最不经常使用的，所以在页面调换的时候仅需将首元素删除。

④ GetInstructionExecutionResult( )

```
1.  public function GetInstructionExecutionResult(){
2.      $this->MemoryManagement();
3.      $RetArray = array('FIFOPageFault' => array(), 'FIFOPageFaultRatio' => 0.0, 'LRU
    PageFault' => array(), 'LRUPageFaultRatio' => 0.0);
4.      $RetArray['FIFOPageFault'] = $this->FIFOPageFault;
5.      $RetArray['FIFOPageFaultRatio'] = (float)count($this->FIFOPageFault) / 320;
6.      $RetArray['LRUPageFault'] = $this->LRUPageFault;
7.      $RetArray['LRUPageFaultRatio'] = (float)count($this->LRUPageFault) / 320;
8.      echo json_encode($RetArray);
9.  }
```

该函数负责将结果数据打包成关联数组，echo 给前端。

## 6. 项目测试

在实际测试过程中，两种算法的优劣性并无明显差别，在相当一部分情况下，两者的缺页情况相同。

在内存块的大小为 4，页面指令条数为 10 的情况下，100 次运行结果中——41 次两种算法的缺页情况完全相同，27 次 FIFO 算法的缺页率稍低于 LRU 算法的缺页率，32 次 LRU 算法的缺页率低于 FIFO 算法的缺页率。

在内存块的大小为 8，页面指令条数为 10 的情况下，100 次运行结果中——34 次两种算法的缺页情况完全相同，30 次 FIFO 算法的缺页率稍低于 LRU 算法的缺页率，36 次 LRU 算法的缺页率低于 FIFO 算法的缺页率。

## 7. 项目总结

本调页存储管理模拟器基本满足的项目的需求，与此同时在项目需求之外对模拟器的功能进行了一定程度的拓展。但是就实验结果来看，该模拟器并不能明显突出 LRU 算法的优势，通过分析，总结出以下两点主要原因：

① 该指令执行顺序算法不满足程序实际执行过程中的局部性原理：该算法完全依赖随机算法，完全违背了程序执行的局部性原理，这对 LRU 算法有致命影响；

② 内存块大小偏小，发生换页的可能性过大，并不能突出算法优劣性；指令数量偏少也会导致两种算法的差异不能凸显出来。