# DemandPaging设计方案

2152085孙亦菲

指导老师：张惠娟

—————————————

## 1 项目简介



### 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。

### 1.4 实现功能

* **置换算法选择**
* 可以选择使用FIFO算法或LRU算法进行页面调换。
* **内存页的展示**
* 项目绘制了内存中当前页面，展示已经执行的指令，从而能够比较清晰的了解内存状况以及当前所正在执行的指令。
* **多种执行方式**
* 项目提供了单步执行，以及一步执行到底的连续执行，以便观察指令执行效果。
* **相关信息的展示**
* 项目在界面左侧栏展示了当前的缺页数和缺页率，能够比较好的了解该种方式下的执行效率
* **重置清空当前内存**
* 可以选择重置清空当前内容，实现打开一次程序可以多次执行。

## 2 项目环境

### 2.1 开发环境

* Vue 3.0 + Node.js
* windows 11

### 2.2 项目结构

│ DemandPaging
  
│ DemandPaging.exe
  
└─ README.md

### 2.3 运行方法

* 直接运行：
* 直接点击 DemandPaging.exe
* 运行源码
* 本项目依赖于 npm 和 Vue CLI，运行前请确保环境中已安装[node.js](https://nodejs.org/en/)。
* 在DemandPaging文件夹中
* npm install
    
  npm run serve
* 可以在本地部署内存管理模拟的网页,默认运行在计算机<http://localhost:8080/>。

## 3 界面及功能介绍



* 单步执行
* 点击单步执行，执行下一条指令。
* 连续执行
* 点击连续执行，这时将连续执行直到320条指令执行完毕，此时页面置换选择、单步执行、重置的按钮都将禁用。
* 重置
* 点击后将回归到没有执行任何指令、没有页面被调入内存的状态。
* ！当320条指令执行完后，必须点击重置才能开始进行新一轮的演示。
* 分配给作业的内存展示
* 显示目前四个页框中页面的序号，及其中最近被执行的指令的相对地址用蓝色背景块标识出来。
* 如果页框中没有页，则显示NULL。
* 已执行指令展示
* 显示到目前为止已经执行的指令，具有信息标识如下：
  + 执行序号
  + 所在物理地址
  + 是否缺页
  + 换出页
  + 换入页
* 上下滑动右侧的滑动条，可以查看所有展示出来的指令信息。
* 页面置换算法选择
* 在可以选择的情况下，通过下拉条对置换算法进行选择。
* 
* 左侧信息展示
* 左侧部分展示信息如下：
  + 一共要执行的指令数
  + 所选择的置换算法
  + 已执行指令数
  + 目前访问页面总数
  + 缺页次数
  + 目前缺页率
* 执行完毕信息提示
* 

## 4 实现算法

### 4.1 页面置换算法

#### 4.1.1 FIFO算法

* 先进先出算法是最简单的页面换算法，是指每次有新的页面需要调入时，会选择调入内存时间最久的分页换出。它简单，容易实现，但这种绝对的公平方式容易导致效率的降低。
* 维护两个变量：
    
  FIFOPointer:-1,//最先进页面的指针
    
  FIFOList:[],//调进内存页面的内存块记录队列
* **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
* **当前页面不在内存中**
  + 内存块有空闲页框=>选择空闲页框为SelectFrame
  + 使用FIFO算法选择调出页面所在页框
  + SelectedFrame=this.FIFO();
      
    FIFO()
      
    {
      
     ++this.FIFOPointer;
      
     return this.FIFOList[this.FIFOPointer];
      
    }
* 将此时调入内存中的页所在的内存中页框序号加入调进内存页面的内存块记录队列。
* this.FIFOList.push(SelectedFrame);

#### 4.1.2 LRU算法

* 最近最少使用算法，是一种常用的页面置换算法，选择最近最久未使用的页面予以淘汰。该算法赋予每个页一个访问字段，用来记录一个页面自上次被访问以来所经历的时间 t，当须淘汰一个页面时，选择现有页面中其 t 值最大的，即最近最少使用的页面予以淘汰。
* 维护一个变量：
    
   LRUList:[],//每个页框上次使用时间间隔
* **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
* **当前页面不在内存中** =>
  + 内存块有空闲页框=>选择空闲页框为SelectFrame
  + 使用LRU算法选择调出页框
  + LRU()
      
    {
      
     let index=0;
      
     let maxTime=this.LRUList[index];
      
     for(let i=1;i<this.FrameNum;i++)
      
     {
      
     if(this.LRUList[i]>maxTime)//距现在时间最长的
      
     {
      
     maxTime=this.LRUList[i];
      
     index=i;
      
     }
      
     }
      
     return index;
      
    }
* 每个页框没有被使用的时间都要+1，而本次指令下被使用到的页框的时间值清零。
* for(let i=0;i<this.FrameNum;i++)
    
  {
    
   if(this.LRUList[i]!=-1)//到目前为止还没使用过则不用+1
    
   {
    
   this.LRUList[i]++;
    
   }
    
  }
    
  this.LRUList[SelectedFrame]=0;

### 4.2 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
  2. 顺序执行下一条指令，即序号为的指令
  3. 通过随机数，跳转到前地址部分0~中的某个指令处，其序号为
     + 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+n的指令
  5. 通过随机数，跳转到后地址部分~319中的某条指令处，其序号为
     + 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，即序号为的指令
  7. 重复3~7的步骤直到执行完320条指令

## 5 项目总结

### 5.1 项目亮点

* 实现了两种置换算法的模拟。
* 界面美观，能够比较好的展示内存模拟过程。
* 部署在网页，方便学习交流；也生成了相应的exe文件便于执行。
* 指令执行过程中，会亮起当前正在执行的指令，便于观察。
* 提示信息清晰。

### 5.2 项目改进方向

当执行指令时，目前指令的展示的表格没有实现能够自动滚动定位到当前指令位置的功能，可视化效果不佳，未来可以进一步学习前端部署功能改进。