**设计方案报告**

**——内存管理**

**目录**

[**1.项目背景及需求** 2](#_Toc166869237)

[**1.1项目背景** 2](#_Toc166869238)

[**1.2项目需求** 2](#_Toc166869239)

[**2.模型假设** 3](#_Toc166869240)

[**3.功能介绍** 3](#_Toc166869241)

[**3.1界面展示** 3](#_Toc166869242)

[**3.2功能介绍** 4](#_Toc166869243)

[**4.页面置换算法** 4](#_Toc166869244)

[**4.1 FIFO算法** 4](#_Toc166869245)

[**4.2 LRU算法** 5](#_Toc166869246)

[**5.测试** 6](#_Toc166869247)

[**5.1 FIFO算法测试** 6](#_Toc166869248)

[**5.2 LRU算法测试** 6](#_Toc166869249)

**1.项目背景及需求**

**1.1项目背景**

假设每个页面可以存放10条指令，分配给一个作业的内存块为4页。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。初始状态下，所有逻辑页均没有调入内存。

**1.2项目需求**

1.2.1 物理内存显示

在模拟过程中，如果访问的指令在内存中，则显示该指令的物理地址，即指令在内存中的地址。如果该指令不在内存中，则显示该指令调入内存后的物理地址。

1.2.2 缺页数记录及缺页率的计算

如果访问指令不在内存中，则发生缺页，此时需要记录缺页次数。同时还要保存当前执行指令的条数，以计算当前的缺页率。缺页率计算公式如下：



1.2.3 页面置换算法

如果所有的内存块都已装入作业，且需要访问的指令不在内存中，则需要进行页面置换。本项目需要实现两种页面置换算法：FIFO（先进先出）和LRU（最近最久未使用）。

FIFO算法：按照指令进入内存的顺序进行替换，最早进入内存的指令最早被替换。

LRU算法：替换最近最久未使用的指令，即替换最近最长时间未被访问的指令。

1.2.4 指令访问原则和算法

指令访问要保证50%的指令是顺序执行，25%的指令是均匀分布在前地址部分，25%的指令均匀分布在后地址部分。

具体实现算法如下。第一步，现在0至319条指令中随机选取一个起始执行指令，记为m；第二步，顺序执行下一条指令，即m+1；第三步，随机选取前地址部分0至m-1中的某条指令，记为m1；第四步，顺序执行下一条指令，即m1+1；第五步，随机选取后地址部分m1+1至319中的某条指令，记为m2；第六步，顺序执行下一条指令，即m2+1。重复上述过程，直到执行完320条指令。

**2.模型假设**

首先，系统共有320条指令，其序号范围为0至319。其中，指令按照每10条指令划分为一个逻辑页。例如，0至9号指令属于逻辑页0，10至19号指令属于逻辑页1，以此类推。这样，整个指令范围共划分为32个逻辑页。

其次，系统共有4个内存块，编号为1至4，每个内存块可以存放一个逻辑页。四页内存空间是连续的，但起始地址在初始化或重置时是随机生成的。内存按字节编址，默认每个指令占用一个字节。

根据上述假设，当一个逻辑页被加载到某个内存块中时，该逻辑页的所有指令的物理地址可以通过内存块的起始地址加上指令在该页中的偏移量计算得到。例如，如果起始地址为mem\_start，逻辑页p被加载到内存块i中，且指令号为n，则该指令的物理地址计算公式为：



**3.功能介绍**

**3.1界面展示**

图 1 请求调页存储管理方式模拟UI图

**3.2功能介绍**

指令执行分为单步执行和自动执行。其中，每点击一次Single Step按钮，程序会随机选取一条指令执行；调整执行速度后，点击一次Auto Execution按钮，程序会按照指定速度执行指令。

按钮Reset按钮，程序会回到初始状态，内存中原来的指令会清空。再次执行时，4个内存块的地址会重新分配，即内存页面中起始物理地址会随机分配。

接下来，当内存块已满且发生缺页时，需要进行页面置换。我们可以选择FIFO或LRU算法。但需要注意的是，每次改变置换算法前，请先按Reset按钮恢复初始化。

**4.页面置换算法**

**4.1** **FIFO算法**

FIFO页面置换算法是最简单的一种页面置换算法，它的基本思想是：最先进入内存的页面在需要置换时最先被淘汰。当有指令需要执行时，我们按内存块1至4的顺序存放。当所有内存块均有指令且发生缺页时，再按照1至4的顺序将页调出内存块。在项目中，我使用一个大小为4的数组来表示内存，并使用一个指针来记录下一个要被替换的页面位置。这个指针从0到3循环，当指针超过3时回到0。

初始化。创建一个大小为4的数组memory，用于存储内存中的页面。初始化一个指针memoryindex用于指向下一个要被替换的位置。初始值为0。当需要加载一个页面时，首先检查该页面是否已经在内存中。如果页面已经在内存中，则指针不需要任何操作。如果页面不在内存中，则发生缺页，需要进行页面置换。先将调入内存的逻辑页替换掉指针memoryindex指向的位置的内容，并将指针memoryindex向前移动一位，即pointer = (pointer + 1) % 4，这样当指针超过3时会回到0。

下面给出一个简单的例子。

表 1 FIFO算法内存变化表

|  |  |  |
| --- | --- | --- |
| 状态 | 内存 | 指针 |
| 初始状态 | [-, -, -, -] | 0 |
| 加载页面A | [A, -, -, -] | 1 |
| 状态 | 内存 | 指针 |
| 加载页面B | [A, B, -, -] | 2 |
| 加载页面C | [A, B, C, -] | 3 |
| 加载页面D | [A, B, C, D] | 0 |
| 加载页面E | [E, B, C, D] | 1 |

所以，当所有内存块都被占用且发生缺页时，新逻辑页将替换最早进入内存的逻辑页，从而实现FIFO算法。

**4.2 LRU算法**

LRU页面置换算法是一种基于访问历史的页面置换算法，它的基本思想是：在需要置换页面时，选择最近最少使用的页面进行置换。在项目中，我使用一个大小为4的数组来表示内存，并使用一个大小为320的数组来记录取出指令的次序。

初始化。创建一个大小为4的数组memory，用于存储内存中的页面。创建一个大小为320的数组instructions，用于记录取出指令的次序。每次需要页面置换的时候，依次检测内存中的指令在instructions中出现位置的下标，新的逻辑页替换进入下标最小的指令对应的内存块。

下面给出一个简单的例子。

表 2 LRU算法内存变化表

|  |  |
| --- | --- |
| 状态 | 内存 |
| 初始状态 | [-,-, -, -] |
| 加载页面A | [A, -, -, -] |
| 加载页面B | [A, B, -, -] |
| 加载页面C | [A, B, C, -] |
| 加载页面D | [A, B, C, D] |
| 加载页面A(访问更新) | [A, B, C, D] |
| 加载页面E | [A, E, C, D] |

所以，当所有内存块都被占用且发生缺页时，新逻辑页将替换内存中最近最少使用的逻辑页，从而实现LRU算法。

**5.测试**

**5.1 FIFO算法测试**

点击Single Step按钮，直至出现页面置换。



图 2 FIFO算法测试图

如上图所示。首先，对于指令序号为29的指令，因为逻辑页号2已经在内存块1中，所以直接执行该指令即可。对于指令5，因为逻辑页号0不在内存中且内存中已满，因此需要找出最早调入内存的逻辑页号，即逻辑页号2。因此，将内存块1中逻辑页号2调出，将逻辑页号0调入。结果符合预期。

**5.2 LRU算法测试**

点击Single Step按钮，直至出现页面置换。



图 3 LRU算法测试图

如上图所示。首先，对于指令序号为295和296的指令，因为逻辑页号29已经在内存块1中，所以直接执行该指令即可。对于指令42，因为逻辑页号4不在内存中且内存中已满，因此需要找出最近最少使用的逻辑页号进行置换。从上图中可以看出，指令序号42之前的逻辑页号依次是29、30、22、22、29、29、11、11，因此最近最少使用的逻辑页号是30。因此，将内存块2中逻辑页号30调出，将逻辑页号4调入。结果符合预期。