

内存管理 - 动态分区分配方式模拟

操作系统第二次课程作业 - 动态分区分配方式模拟

内存管理 - 动态分区分配方式模拟

- 项目需求
 - 基本任务
 - 功能描述
 - 项目目的
- 开发环境
- 项目结构
- 操作说明
- 系统分析
 - 置换算法
 - LRU算法
 - FIFO算法
 - 执行模式
 - 320条指令产生方式
- 系统设计
 - 类设计
 - 内存
 - 实体设计
 - 状态设计
- 系统实现
 - 请求调页存储管理方式模拟
 - 执行前320条指令
 - 执行一条指令
 - 请求调页
 - 打印信息
 - 产生随机指令
- 功能实现截屏展示
 - 选择配置
 - 模拟过程
 - 模拟结果
- 实验小结
 - 实验结果对比
 - LRU算法, 执行完所有指令
 - FIFO算法, 执行完所有指令
 - LRU算法, 执行前320条指令
 - FIFO算法, 执行前320条指令
 - 分析
- 作者

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

项目目的

- 页面、页表、地址转换
- 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

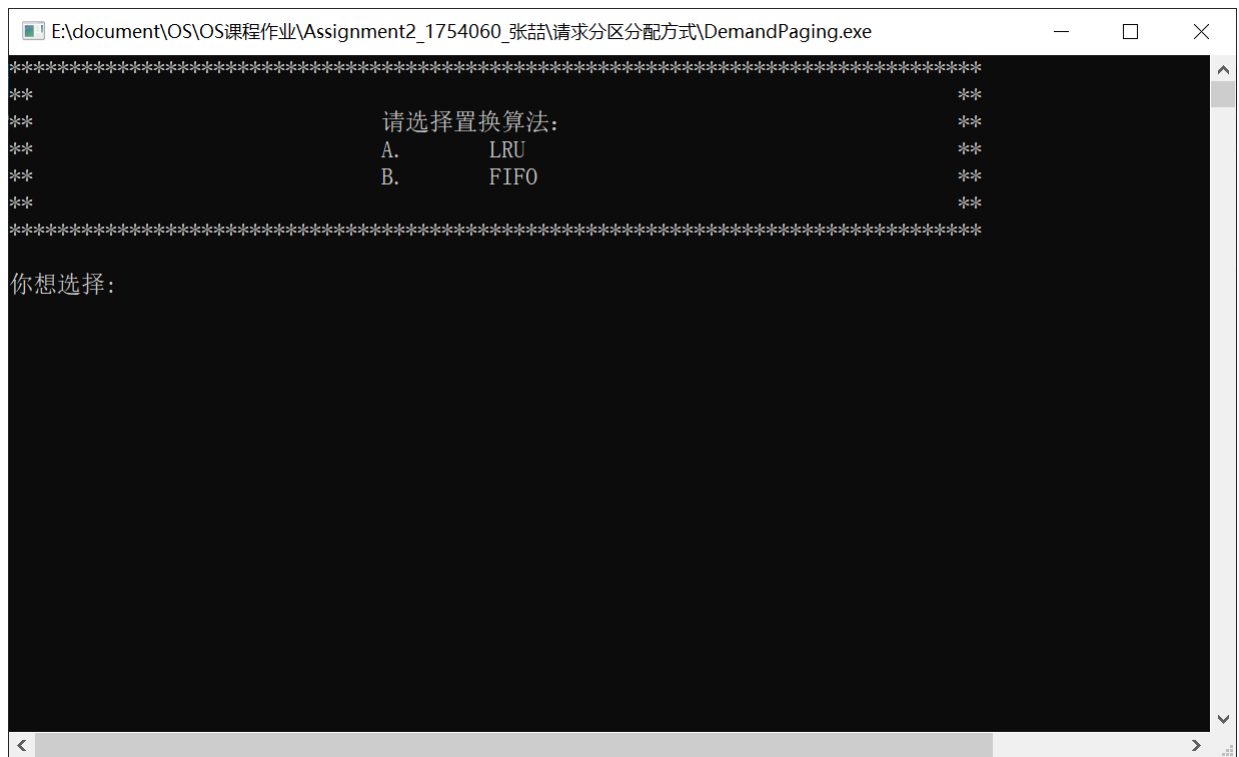
- **开发环境:** Windows 10
- **开发软件:**
Visual Studio 2017 15.9.28307.665
- **开发语言:** C++

项目结构

```
| list.txt
| 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
DemandPaging.cpp
```

操作说明

- 双击目录下 `DemandPaing.exe` 可执行文件进入模拟界面

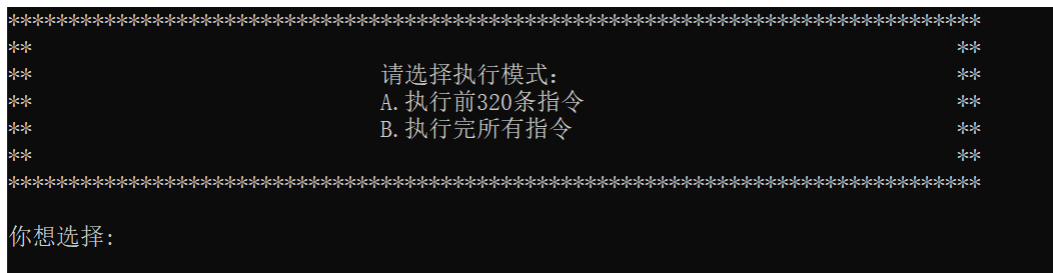


- 选择置换算法:

- 键入 **a** 或 **A** 代表选择LRU算法
- 键入 **b** 或 **B** 代表选择FIFO算法
- 输入无效算法将受到提示, 并允许重新选择

```
你想选择: d
您输入的置换算法有误, 请重新输入: P
您输入的置换算法有误, 请重新输入: 
```

- 选择执行模式:



- 键入 **a** 或 **A** 代表只执行前320条指令(指令可能重复)
- 键入 **b** 或 **B** 代表执行完所有指令(知道所有指令都被执行为止)
- 输入无效执行模式将受到提示, 并允许重新选择

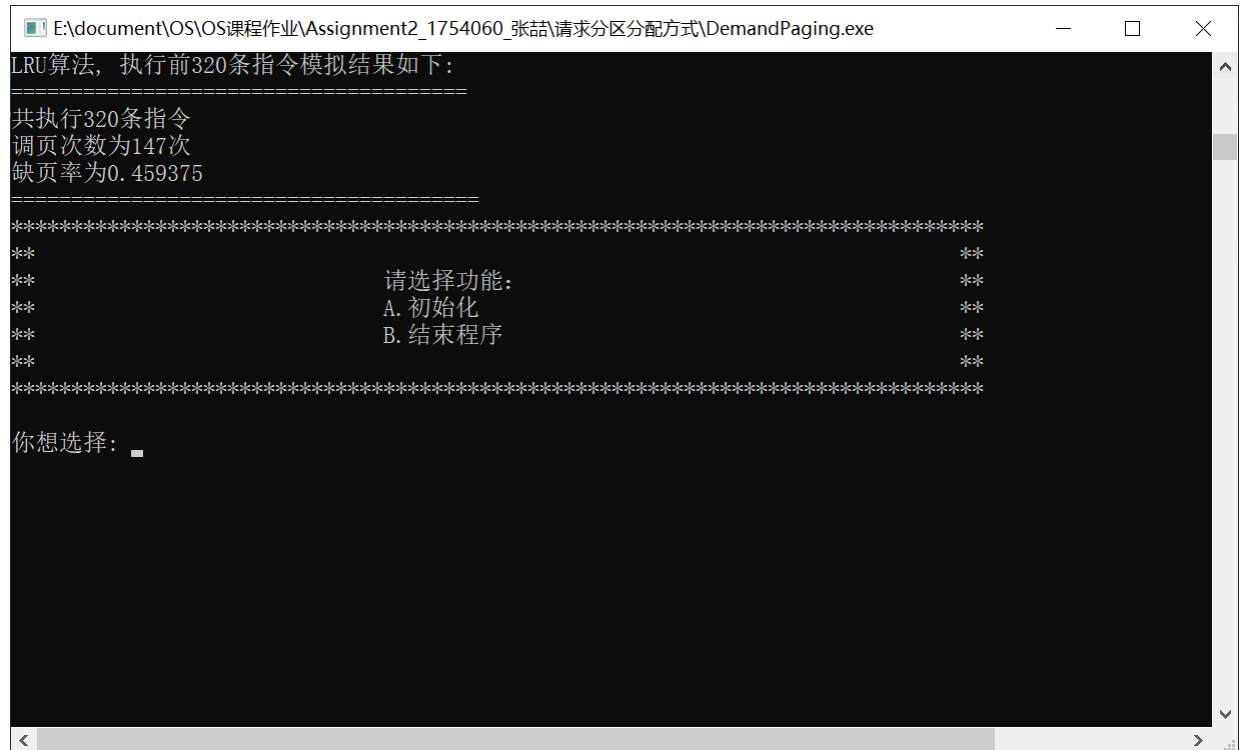
```
你想选择: t
您输入的执行模式有误, 请重新输入: U
您输入的执行模式有误, 请重新输入: 
```

- 查看对应算法和对应执行模式下的模拟结果

LRU算法，执行前320条指令模拟结果如下：

```
=====
共执行320条指令
调页次数为147次
缺页率为0.459375
=====
```

- 选择功能:



```
E:\document\OS\OS课程作业\Assignment2_1754060_张喆\请求分区分配方式\DemandPaging.exe
LRU算法，执行前320条指令模拟结果如下：
=====
共执行320条指令
调页次数为147次
缺页率为0.459375
=====
*****
**                                     **
**               请选择功能：               **
**               A. 初始化                   **
**               B. 结束程序                 **
**                                     **
*****
你想选择: _
```

- 键入 **a** 或 **A** 代表初始化内存(可再次进行模拟)
- 键入 **b** 或 **B** 代表结束程序

- 初始化

```
E:\document\OS\OS课程作业\Assignment2_1754060_张喆\请求分区分配方式\DemandPaging.exe

*****
**                                     **
**               请选择功能:           **
**               A. 初始化             **
**               B. 结束程序           **
**                                     **
*****

你想选择: A

*****
**                                     **
**               请选择置换算法:       **
**               A.      LRU           **
**               B.      FIFO          **
**                                     **
*****

你想选择: B

*****
**                                     **
**               请选择执行模式:       **
**               A. 执行前320条指令    **
**               B. 执行完所有指令    **
**                                     **
*****

你想选择: A
```

- 结束程序

```
*****
**                                     **
**               请选择功能:           **
**               A. 初始化             **
**               B. 结束程序           **
**                                     **
*****

你想选择: B

*****
* 请求调页存储管理方式模拟结束 *
*****
```

系统分析

置换算法

LRU算法

- 当前页面已经在内存中 => 不需要进行调度
- 当内存中页面数小于内存容量时 => 直接将页面顺序加入到内存的空闲块中
- 当内存满时 => 每次替换掉最近最少使用的内存块中的页面
 - 维护一个 LRU 队列: 每当发生替换时取出队列头元素 => 将该内存块中的页面作为被替换掉的页面 => 将新页面加入到该页面中 => 将该内存块号重新压队

- **为每个内存中的页面维护一个变量 `sinceTime`**: 当需要进行页面替换的时候 => 选择 `sinceTime` 最大的页面替换掉 => 将新页面加入到该页面所在内存块中 => 将新页面的 `sinceTime` 置为0 => 内存中其他页面的 `sinceTime` 递增1
- 本模拟程序选择的是第一种方法, 即维护LRU队列

FIFO算法

- 当前页面已经在内存中** => 不需要进行调度
- 当**内存中页面数小于内存容量**时 => 直接将页面顺序加入到内存的空闲块中
- 当**内存满**时 => 每次一次替换掉内存块中的页面
 - 维护变量 `adjustTime`, 用来计算缺页次数
 - `adjustTime` 为1 => 将0号内存的页调出, 将当前指令调入0号内存中
 - `adjustTime` 为2 => 将1号内存的页调出, 将当前指令调入1号内存中
 - `adjustTime` 为3 => 将2号内存的页调出, 将当前指令调入2号内存中
 - `adjustTime` 为4 => 将3号内存的页调出, 将当前指令调入3号内存中

执行模式

320条指令产生方式

为了保证320 条指令能够随机产生, 并且能够均匀分布, 采用了下面这种循环产生指令的方式:

- 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
- 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m1
- 顺序执行下一条指令, 即序号为m1+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m2
- 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、顺序执行的过程, 直到执行完320条指令。

系统设计

类设计

内存

```
class Memory
{
private:
    vector<PageNum> block;           //内存块
    vector<bool> visited;           //是否执行过该指令
    queue<BlockNum> LRU_Queue;     //最近最少使用队列
```

```

int runTime = 0; //运行次数
int adjustTime = 0; //调页次数
int restInst = TOTALNUM; //剩余未执行指令

void execute(string algorithm, InstNum aim); //按照算法执行一条指令
PageNum adjust(string algorithm, BlockNum &pos); //页面置换

void displayPosMess(InstNum aim) { //打印指令地址信息
    cout << "物理地址为:" << setw(3)<<aim
        << ", 地址空间页号为:" <<setw(2)<< aim / 10
        << ", 页内第" << setw(2) << aim % 10 << "条指令.";
}
void displayLoadMess(PageNum fresh, BlockNum pos, bool flag) { //打印未发生调页的信息
    cout << endl;
    if (flag) { //已经在内存块中
        cout << fresh << "号页已经在内存中第" << pos << "号块中了, 未发生调页." << endl <<
endl;
    }
    else { //没在内存块中, 但是内存块没满
        cout << fresh << "号页放在内存中第" << pos << "号块中, 未发生调页." << endl << endl;
    }
}
void displayLoadMess(PageNum old, PageNum fresh, BlockNum pos) { //打印发生调页的信息
    cout << " || 调出内存中第" << setw(2)<<pos
        << "块中第" <<setw(2)<< old
        << "号页, 调入第" << setw(2) << fresh << "号页." << endl << endl;
}

public:
    Memory() = default;
    ~Memory() = default;

    void Init(); //初始化内存
    void Simulate(string algorithm, char type); //按照算法和执行模式执行指令

    int getRunTime() { return this->runTime; } //返回运行次数
    int getAdjustTime() { return this->adjustTime; } //返回调页次数
    double getAdjustRate(){ return (1.0*this->adjustTime / this->runTime); } //返回缺页率
};

```

实体设计

1. 指令号: `typedef int InstNum;`
2. 页号: `typedef int PageNum;`
3. 块号: `typedef int BlockNum;`

状态设计

1. 分配给作业的总内存块数: `#define MaxSize 4`

2. 内存块为空标识: `#define EMPTY -1`
3. 指令总条数: `#define TOTALNUM 320`

系统实现

请求调页存储管理方式模拟

执行前320条指令

- 设置变量 `cnt` 用于记录当前已执行的指令条数, 初始化为0
- 随机选取一个起始指令 => 递增 `cnt` => 顺序执行下一条指令 => 递增 `cnt`
- 进入循环:
 - 判断是否执行满320条指令(`cnt`是否为320) => 跳转到前地址部分 => 递增 `cnt`
 - 判断是否执行满320条指令 => 顺序执行下一条指令 => 递增 `cnt`
 - 判断是否执行满320条指令 => 跳转到后地址部分 => 递增 `cnt`
 - 判断是否执行满320条指令 => 顺序执行下一条指令 => 递增 `cnt`

执行完所有指令

- 设置变量 `restInst` 用于记录剩余未执行指令数, 初始化为320
- 设置布尔向量 `visited` 用于记录是否执行过该指令, 初始化为false
- 随机选取一个起始指令 => 递减 `restInst` => `visited` 中标记为true => 顺序执行下一条指令 => 递减 `restInst` => `visited` 中标记为true
- 进入循环:
 - 判断是否执行完所有指令(`restInst`是否为0) => 跳转到前地址部分 => 递减 `restInst` => `visited` 中标记为true
 - 判断是否执行完所有指令 => 顺序执行下一条指令 => 递减 `restInst` => `visited` 中标记为true
 - 判断是否执行完所有指令 => 跳转到后地址部分 => 递减 `restInst` => `visited` 中标记为true
 - 判断是否执行完所有指令 => 顺序执行下一条指令 => 递减 `restInst` => `visited` 中标记为true

```
/* 请求调页存储管理方式模拟
 * @param {置换算法} algorithm
 * @param {用户选择的执行类型} type
 */
void Memory::Simulate(string algorithm, char type)
{
    InstNum aim;
    if (type == 'A' || type == 'a')
    {
        int cnt = 0;

        //随机选取一个起始指令
        aim = getRand(0, TOTALNUM - 1);
        execute(algorithm, aim); cnt++;
        //顺序执行下一条指令
        aim++;
    }
}
```



```

execute(algorithm, aim); cnt++;
while (true)
{
    if (cnt == TOTALNUM) { break; }
    //跳转到前地址部分
    aim = getRand(0, aim - 1);
    execute(algorithm, aim); cnt++;

    if (cnt == TOTALNUM) { break; }
    //顺序执行下一条指令
    aim++;
    execute(algorithm, aim); cnt++;

    if (cnt == TOTALNUM) { break; }
    //跳转到后地址部分
    aim = getRand(aim + 1, TOTALNUM - 1);
    execute(algorithm, aim); cnt++;

    if (cnt == TOTALNUM) { break; }
    //顺序执行下一条指令
    aim++;
    execute(algorithm, aim); cnt++;
}
}
else if (type == 'B' || type == 'b')
{
    //随机选取一个起始指令
    aim = getRand(0, TOTALNUM - 1);
    execute(algorithm, aim);
    restInst--; visited[aim] = true;
    //顺序执行下一条指令
    aim++;
    execute(algorithm, aim);
    restInst--; visited[aim] = true;

    while (true)
    {
        if (!restInst) { break; }
        //跳转到前地址部分
        aim = getRand(0, aim - 1);
        execute(algorithm, aim);
        if (aim!=TOTALNUM && !visited[aim]) { restInst--; visited[aim] = true; }

        if (!restInst) { break; }
        //顺序执行下一条指令
        aim++;
        execute(algorithm, aim);
        if (aim != TOTALNUM && !visited[aim]) { restInst--; visited[aim] = true; }

        if (!restInst) { break; }
        //跳转到后地址部分
        aim = getRand(aim + 1, TOTALNUM - 1);
        execute(algorithm, aim);
    }
}

```

```

        if (aim != TOTALNUM && !visited[aim]) { restInst--; visited[aim] = true; }

        if (!restInst) { break; }
        //顺序执行下一条指令
        aim++;
        execute(algorithm, aim);
        if (aim != TOTALNUM && !visited[aim]) { restInst--; visited[aim] = true; }
    }
}
}

```

执行一条指令

- 更新运行次数
- 计算页号并输出该指令的信息(物理地址, 页号, 页内地址)
- 检测该页是否已经在内存中:
 - 如果是 => 打印已经在内存块中相应信息
- 检测内存中是否有空闲块:
 - 如果有 => 打印没在内存块中, 但是内存块没满相应信息
- 按照相应的算法请求调页

```

/* 执行一条指令
 * @param {置换算法} algorithm
 * @param {待执行指令} aim
 */
void Memory::execute(string algorithm, InstNum aim)
{
    this->runTime++;          //更新运行次数

    PageNum page = aim / 10;    //计算页号
    BlockNum pos = 0;

    displayPosMess(aim);

    /*检测该页是否已经在内存中*/
    for (pos = 0; pos < MaxSize; ++pos)
    {
        if (block[pos] == page)
        {
            displayLoadMess(page, pos, true);

            return;
        }
    }
    /*检测内存中是否有空闲块*/
    for (pos = 0; pos < MaxSize; ++pos)
    {
        if (block[pos] == EMPTY)
        {

```

```

        block[pos] = page;
        displayLoadMess(page, pos, false);

        if (algorithm == string("LRU"))
        {
            LRU_Queue.push(pos);          //将其压入最近最少使用队列
        }

        return;
    }
}

//执行到这说明：1.内存块是满的 2.要进行调页
PageNum old = adjust(algorithm, pos);
block[pos] = page;
displayLoadMess(old, page, pos);
}

```

请求调页

- 更新调页次数
- FIFO算法:
 - 根据缺页次数计算哪个页面要被替换掉, 位置记录在 pos 中
 - 内存块中 pos 位置页面记录为 old
- LRU算法:
 - 访问LRU队列头, 获取最近最少使用页面位置, 记录在 pos 中
 - 将该位置填入新的页面
 - 压入队列尾

```

/* 请求调页
 * @returnValue {要被替换掉的页号}
 * @param {置换算法} algorithm
 * @param {调入调出的位置} pos
 */
PageNum Memory::adjust(string algorithm, BlockNum &pos)
{
    this->adjustTime++;          //更新调页次数

    PageNum old;
    if (algorithm == "FIFO")
    {
        pos = (this->adjustTime-1) % 4; //缺页次数为1, 则将0号内存的页调出, 将当前指令调入0 号内存
        //中...以此类推
        old = block[pos];
    }
    else if (algorithm == "LRU")
    {
        pos = LRU_Queue.front();          //取队列头元素 => 最近最少使用的页面
        LRU_Queue.pop();
    }
}

```

```

        LRU_Queue.push(pos);                //将其压入队尾

        old = block[pos];
    }

    return old;
}

```

打印信息

- 打印指令的物理地址, 页面, 页内地址
- 打印未发生调页的信息
- 打印发生调页的信息

```

void displayPosMess(InstNum aim) {                //打印指令地址信息
    cout << "物理地址为:" << setw(3)<<aim
        << ", 地址空间页号为:" <<setw(2)<< aim / 10
        << ", 页内第" << setw(2) << aim % 10 << "条指令.";
}

void displayLoadMess(PageNum fresh, BlockNum pos, bool flag) {    //打印未发生调页的信息
    cout << endl;
    if (flag) {            //已经在内存块中
        cout << fresh << "号页已经在内存中第" << pos << "号块中了, 未发生调页." << endl <<
endl;
    }
    else {                //没在内存块中, 但是内存块没满
        cout << fresh << "号页放在内存中第" << pos << "号块中, 未发生调页." << endl << endl;
    }
}

void displayLoadMess(PageNum old, PageNum fresh, BlockNum pos) {    //打印发生调页的信息
    cout << "    || 调出内存中第" << setw(2)<<pos
        << "块中第" <<setw(2)<< old
        << "号页, 调入第" << setw(2) << fresh << "号页." << endl << endl;
}

```

产生随机指令

```

/*返回[low, high]间的随机指令*/
InstNum getRand(InstNum low, InstNum high)
{
    if (high - low == -1) { return high; }        //消除作业中指令访问次序产生high比low小1的问题
    return (rand() % (high - low + 1) + low);
}

```

功能实现截屏展示

选择配置

```

*****
**                                     **
**                               **
**      请选择功能：              **
**      A. 初始化                **
**      B. 结束程序              **
**                               **
*****

你想选择： a
*****
**                                     **
**                               **
**      请选择置换算法：          **
**      A.        LRU            **
**      B.        FIFO           **
**                               **
*****

你想选择： a
*****
**                                     **
**                               **
**      请选择执行模式：          **
**      A. 执行前320条指令       **
**      B. 执行完所有指令       **
**                               **
*****

你想选择： b_

```

模拟过程

```
物理地址为：31，地址空间页号为：3，页内第 1 条指令。  
3号页放在内存中第0号块中，未发生调页。  
  
物理地址为：32，地址空间页号为：3，页内第 2 条指令。  
3号页已经在内存中第0号块中了，未发生调页。  
  
物理地址为：12，地址空间页号为：1，页内第 2 条指令。  
1号页放在内存中第1号块中，未发生调页。  
  
物理地址为：13，地址空间页号为：1，页内第 3 条指令。  
1号页已经在内存中第1号块中了，未发生调页。  
  
物理地址为：106，地址空间页号为：10，页内第 6 条指令。  
10号页放在内存中第2号块中，未发生调页。  
  
物理地址为：107，地址空间页号为：10，页内第 7 条指令。  
10号页已经在内存中第2号块中了，未发生调页。  
  
物理地址为：52，地址空间页号为：5，页内第 2 条指令。  
5号页放在内存中第3号块中，未发生调页。  
  
物理地址为：53，地址空间页号为：5，页内第 3 条指令。  
5号页已经在内存中第3号块中了，未发生调页。  
  
物理地址为：258，地址空间页号为：25，页内第 8 条指令。 || 调出内存中第 0 块中第 3 号页，调入第25号页。  
  
物理地址为：259，地址空间页号为：25，页内第 9 条指令。  
25号页已经在内存中第0号块中了，未发生调页。  
  
物理地址为： 8，地址空间页号为：0，页内第 8 条指令。 || 调出内存中第 1 块中第 1 号页，调入第 0 号页。  
  
物理地址为： 9，地址空间页号为：0，页内第 9 条指令。  
0号页已经在内存中第1号块中了，未发生调页。  
  
物理地址为：59，地址空间页号为：5，页内第 9 条指令。  
5号页已经在内存中第3号块中了，未发生调页。  
  
物理地址为：60，地址空间页号为：6，页内第 0 条指令。 || 调出内存中第 2 块中第10号页，调入第 6 号页。
```

模拟结果

```
LRU算法，执行前320条指令模拟结果如下：  
=====  
共执行320条指令  
调页次数为144次  
缺页率为0.45  
=====
```

实验小结

通过本次实验，我充分理解了操作系统中内存管理的原理以及页面调度的具体实现过程，对FIFO 和LRU 两种调页算法有了更深刻的理解。

实验结果对比

LRU算法, 执行完所有指令

```
LRU算法, 执行完所有指令模拟结果如下:
```

```
=====
共执行1940条指令
调页次数为891次
缺页率为0.459278
=====
```

FIFO算法, 执行完所有指令

```
FIFO算法, 执行完所有指令模拟结果如下:
```

```
=====
共执行1834条指令
调页次数为870次
缺页率为0.474373
=====
```

LRU算法, 执行前320条指令

```
LRU算法, 执行前320条指令模拟结果如下:
```

```
=====
共执行320条指令
调页次数为136次
缺页率为0.425
=====
```

FIFO算法, 执行前320条指令

```
FIFO算法, 执行前320条指令模拟结果如下:
```

```
=====
共执行320条指令
调页次数为154次
缺页率为0.48125
=====
```

分析

在本次项目中, 我使用自己编写的程序, 进行了多次实验, 并且每次实验均使用了两种调页算法, 以比对两种调页算法的效率, 但是多次实验之后, 我的数据显示两种调页算法中的缺页率一致的情况出现了很多次, 我尝试着对所取的指令顺序在自己进行两种调页方式的部分模拟, 发现两种调页算法并没有出错。经过思考, 我认为是指令选取算法的问题, 虽然我的指令选取算法保证了指令的分布范围能够比较平均, 但是, 因为指令的总数目过多, 选取的指令很少能够抖动现象, 当抖动现象出现的次数极少或者根本不出现时, 两种调页算法的效率基本是无差别的。但是当选取的指令顺序使得抖动现象出现时, 根据实验结果, LRU 调页算法的效率优于FIFO 调页算法的效率。

作者

学号 1754060

姓名 张喆

指导老师 王冬青老师

上课时间 周三/周五 上午一二节

联系方式 email: doubleZ0108@gmail.com