内存管理 - 动态分区分配方式模拟

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 11
- **开发软件:**

  Visual Studio Code *1.81.1*
- **开发语言:** C++

项目结构

```
|  请求分区分配方式模拟_设计方案报告.md
|  DemandPaging.exe
└─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
- 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
- 查看对应算法和对应执行模式下的模拟结果

- 选择功能:
    - 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
- 结束程序

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

2. **分配给作业的总内存块数:** `#define MaxSize 4`
3. **内存块为空标识:** `#define EMPTY -1`
4. **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
- 检测内存中有无空闲块:
    - 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
*/
```

```cpp
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;
        }
    }
    /*检测内存中有无空闲块*/
    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);
}
```