**同济大学软件工程**

**操作系统内存管理作业报告**

**——请求调页存储管理方式模拟**

****

**学 号 2151636**

**姓 名 李骁**

**专 业 软件工程**

**班 级 42036903**

**授课老师 王冬青**

**完成日期 2023/5/26**

**目录**

[请求调页存储管理方式模拟 2](#_Toc136469996)

[一 设计内容 2](#_Toc136469997)

[1. 主体要求 2](#_Toc136469998)

[2. 界面设计 3](#_Toc136469999)

[3. 操作说明 3](#_Toc136470000)

[二 项目组成： 3](#_Toc136470001)

[1. 代码文件 3](#_Toc136470002)

[三 设计思路 3](#_Toc136470003)

[1. 主要逻辑部分设计思路 4](#_Toc136470004)

[**1.1.** **类的设计** 4](#_Toc136470005)

[**1.2.** **Job类的实现** 6](#_Toc136470006)

[**1)** **GenerateInstructionQueue函数：** 6](#_Toc136470007)

[**2)** **execute函数** 7](#_Toc136470008)

[**3)** **HandleFault函数** 7](#_Toc136470009)

# 请求调页存储管理方式模拟

1. 设计内容
   1. 主体要求

**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。**

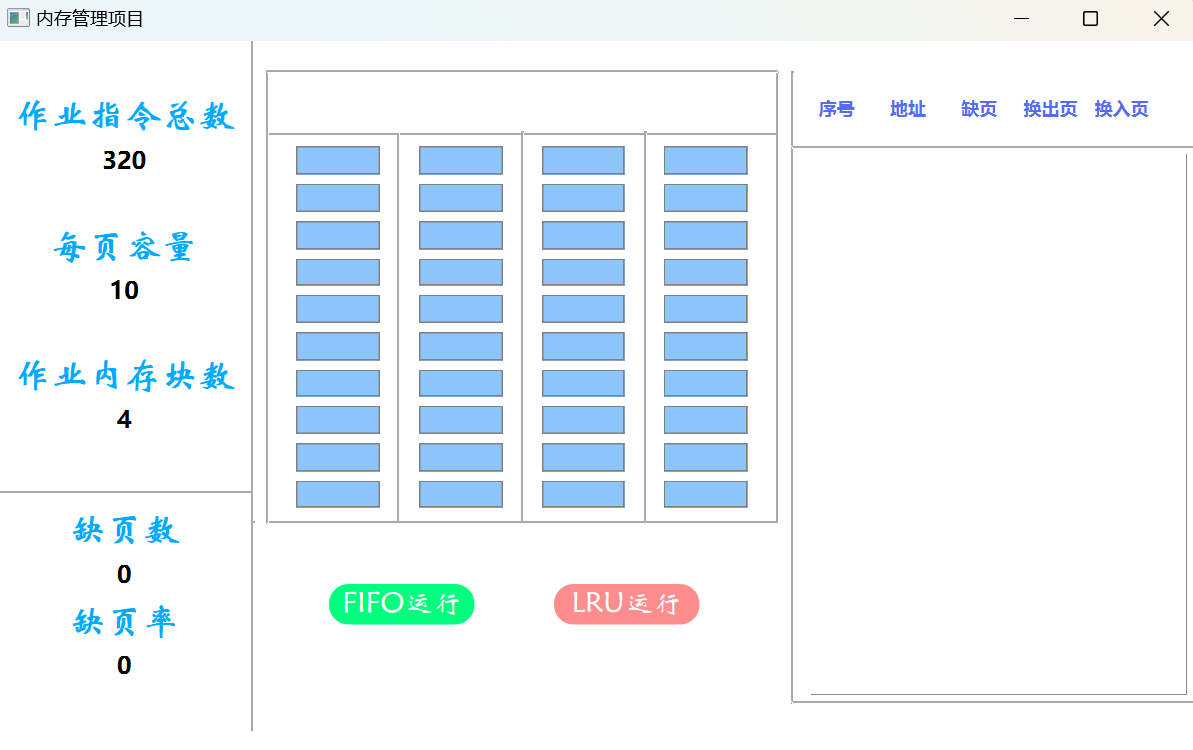
**所有320条指令执行完成后，计算并显示作业执行过程中发生的缺页率。**

**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. 界面设计



* 1. 操作说明

**若有qt相关库，打开debug文件夹下的exe即可运行。**

1. 项目组成：
   1. 代码文件

**SOURCES += \**

**MemoryPage.cpp \**

**main.cpp \**

**mainwindow.cpp**

**HEADERS += \**

**MemoryPage.h \**

**mainwindow.h**

**FORMS += \**

**mainwindow.ui**

1. 设计思路
   1. 主要逻辑部分设计思路
      1. **类的设计**

#ifndef MEMORYPAGE\_H

#define MEMORYPAGE\_H

#endif // MEMORYPAGE\_H

#include <algorithm>

#include <cstdlib>

#include <cstring>

#include <ctime>

#include <iostream>

#include <queue>

#include <random>

#include <vector>

//Qt有关头文件

#include <QMetaObject>

#include <QObject>

#include <QString>

using namespace std;

#define INF 2147483647

#define MEMORY\_SIZE 4 // 内存块数

#define PAGE\_SIZE 10  // 页面大小

#define JOB\_SIZE 320  // 作业大小（指令数）

// 定义页面类

class Page {

public:

    // int list[PAGE\_SIZE];  //存储指令地址

    bool loaded;   //是否装载

    int last\_used; //上次使用时间

    Page() {

        loaded = false;

        last\_used = -1;

    }

};

// 定义作业类

class Job {

public:

    Job() {

        numLoadedPages = 0;

        numPageFaults = 0;

        FIFO\_cnt = 0;

        for (int i = 0; i < MEMORY\_SIZE; ++i) {

            MemoryIndex[i] = 0;

        }

        for (int i = 0; i < JOB\_SIZE; ++i) {

            InstructionQueue[i] = 0;

        }

        GenerateInstructionQueue();

    }

    // 随机生成指令访问次序

    void GenerateInstructionQueue();

    //执行指令

    void execute(const char \*mode);

    //解决缺页问题（置换）

    //返回换出页在任务内存块中序号

    int HandlePageFault(const int pageIndex, const char \*mode);

    //处理指令信号量(序号、地址、是否缺页、换出页、换入页)

    using SignalType\_Instruction = function<void(int, int, bool, int, int)>;

    //处理缺页信号量(缺页数、缺页率)

    using SignalType\_FaultRate = function<void(int ,int)>;

    //向主进程发送更改ui界面的信号(列表添加已执行指令信息)

    void SignaInstruction(SignalType\_Instruction signal) {

        signal\_instruction = signal;

    }

    //向主进程发送更改ui界面的信号(缺页率)

    void SignalFaultRate(SignalType\_FaultRate signal) {

        signal\_FaultRate = signal;

    }

private:

    int numLoadedPages;             //已装载页面数

    int numPageFaults;              //缺页数

    Page pages[JOB\_SIZE];           //页面容器

    int MemoryIndex[MEMORY\_SIZE];   //记录

    int FIFO\_cnt;                   //先进先出受害者指针

    int InstructionQueue[JOB\_SIZE]; //任务序列

    double faultsRate;

    SignalType\_Instruction signal\_instruction;

    SignalType\_FaultRate signal\_FaultRate;

};

* + 1. **Job类的实现**

**首先通过GenerateInstructionQueue函数随根据要求，顺序与随机混合初始化指令序列，然后根据选择的模式决定FIFO或LRU式替换页面。**

* + - 1. **GenerateInstructionQueue函数：**

#include "MemoryPage.h"

// 随机生成指令访问次序

void Job::GenerateInstructionQueue() {

    random\_device rd;

    mt19937 gen(rd());

    int cnt = 0;

    int vis[JOB\_SIZE];

    int randomqueue[JOB\_SIZE];

    for (int i = 0; i < JOB\_SIZE; ++i) {

        randomqueue[i] = i;

        vis[i] = false;

    }

    shuffle(randomqueue, randomqueue + JOB\_SIZE, gen);

    for (int i = 0; i < JOB\_SIZE; ++i) {

        if (!vis[randomqueue[i]]) {

            InstructionQueue[cnt++] = randomqueue[i];

            vis[randomqueue[i]] = true;

            if (rd() % 2 && !vis[(randomqueue[i] + 1) % JOB\_SIZE]) { //一半的顺序

                InstructionQueue[cnt++] = (randomqueue[i] + 1) % JOB\_SIZE;

                vis[(randomqueue[i] + 1) % JOB\_SIZE] = true;

            }

        }

    }

}

* + - 1. **execute函数**

//执行指令

void Job::execute(const char \*mode) {

    for (int i = 0; i < JOB\_SIZE; ++i) {

        int instruction = InstructionQueue[i];

        int pageIndex = instruction / PAGE\_SIZE;

        bool Fault=!pages[pageIndex].loaded;

        int victim = -1;

        cout << "第" << i + 1 << "条指令的物理地址为 " << InstructionQueue[i]

             << endl;

        pages[pageIndex].last\_used = i; //更新“上一次使用”时间为第i个指令

        if (Fault) {

            victim = HandlePageFault(pageIndex, mode);

            ++numPageFaults; //缺页数+1

            faultsRate = numPageFaults / (i + 1);

            signal\_FaultRate(numPageFaults,faultsRate);

        } else {

            for (int i = 0; i < MEMORY\_SIZE; ++i) {

                if (MemoryIndex[i] == pageIndex) {

                    cout << i << "号有" << pageIndex << endl;

                }

            }

        }

        signal\_instruction(i,instruction,Fault,victim,pageIndex);

    }

}

* + - 1. **HandleFault函数**

    //解决缺页问题（置换）

    //返回换出页在任务内存块中序号

int Job::HandlePageFault(const int pageIndex, const char \*mode) {

    int victimIndex; //被替换页面的序号

    if (!strcmp(mode, "FIFO")) {

        victimIndex = FIFO\_cnt;

        FIFO\_cnt = (FIFO\_cnt + 1) % 4;

    } else {

        int last\_used = INF; //要找出last\_uesd最小（使用最久远）的页

        for (int i = 0; i < MEMORY\_SIZE; ++i) {

            if (last\_used > pages[MemoryIndex[i]].last\_used) {

                last\_used = pages[MemoryIndex[i]].last\_used;

                victimIndex = i;

            }

        }

    }

    pages[MemoryIndex[victimIndex]].loaded = false;

    pages[pageIndex].loaded = true; //更改装载状态

    MemoryIndex[victimIndex] = pageIndex;

    return victimIndex;

}