|  |  |
| --- | --- |
| 成绩 |  |

操作系统课程设计报告

基于LRU置换算法的

请求分页模拟系统

学 院：

班 级：

姓 名：

学 号：

完成日期：

目录

[一、研发背景](#_Toc17925_WPSOffice_Level1) [1](#_Toc17925_WPSOffice_Level1)

[（一）课题内容](#_Toc2504_WPSOffice_Level2) [1](#_Toc2504_WPSOffice_Level2)

[（二）开发环境](#_Toc24001_WPSOffice_Level2) [1](#_Toc24001_WPSOffice_Level2)

[二、理论基础](#_Toc24001_WPSOffice_Level1) [1](#_Toc24001_WPSOffice_Level1)

[（一）传统存储方式的弊端](#_Toc8778_WPSOffice_Level2) [1](#_Toc8778_WPSOffice_Level2)

[（二）局部性原理](#_Toc26324_WPSOffice_Level2) [1](#_Toc26324_WPSOffice_Level2)

[（三）虚拟存储基本思想——“时间换空间”](#_Toc1977_WPSOffice_Level2) [2](#_Toc1977_WPSOffice_Level2)

[三、总体设计](#_Toc8778_WPSOffice_Level1) [2](#_Toc8778_WPSOffice_Level1)

[（一）提前说明](#_Toc10930_WPSOffice_Level2) [2](#_Toc10930_WPSOffice_Level2)

[（二）整体流程（用户）](#_Toc32062_WPSOffice_Level2) [3](#_Toc32062_WPSOffice_Level2)

[（三）整体流程（系统）](#_Toc11747_WPSOffice_Level2) [3](#_Toc11747_WPSOffice_Level2)

[（四）内存分配策略](#_Toc21251_WPSOffice_Level2) [4](#_Toc21251_WPSOffice_Level2)

[（五）页面置换算法](#_Toc28330_WPSOffice_Level2) [5](#_Toc28330_WPSOffice_Level2)

[四、详细设计](#_Toc26324_WPSOffice_Level1) [5](#_Toc26324_WPSOffice_Level1)

[（一）核心数据结构与流程](#_Toc10774_WPSOffice_Level2) [5](#_Toc10774_WPSOffice_Level2)

[1.请求页表](#_Toc26324_WPSOffice_Level3) [5](#_Toc26324_WPSOffice_Level3)

[2.内存使用情况说明](#_Toc1977_WPSOffice_Level3) [6](#_Toc1977_WPSOffice_Level3)

[3.地址变换过程&页面置换](#_Toc10930_WPSOffice_Level3) [6](#_Toc10930_WPSOffice_Level3)

[（二）其他数据结构和过程](#_Toc32352_WPSOffice_Level2) [8](#_Toc32352_WPSOffice_Level2)

[1.栈——LRU置换算法支持](#_Toc32062_WPSOffice_Level3) [8](#_Toc32062_WPSOffice_Level3)

[2.内存再分配过程](#_Toc11747_WPSOffice_Level3) [8](#_Toc11747_WPSOffice_Level3)

[3.硬件模拟过程](#_Toc21251_WPSOffice_Level3) [8](#_Toc21251_WPSOffice_Level3)

[五、测试与运行](#_Toc1977_WPSOffice_Level1) [9](#_Toc1977_WPSOffice_Level1)

[（一）用户使用说明](#_Toc6343_WPSOffice_Level2) [9](#_Toc6343_WPSOffice_Level2)

[（二）测试与运行](#_Toc17134_WPSOffice_Level2) [11](#_Toc17134_WPSOffice_Level2)

[六、总结与改进](#_Toc10930_WPSOffice_Level1) [12](#_Toc10930_WPSOffice_Level1)

[（一）总结](#_Toc23248_WPSOffice_Level2) [12](#_Toc23248_WPSOffice_Level2)

[（二）待改进](#_Toc2264_WPSOffice_Level2) [12](#_Toc2264_WPSOffice_Level2)

[八、参考文献](#_Toc32062_WPSOffice_Level1) [13](#_Toc32062_WPSOffice_Level1)

# 一、研发背景

**（一）课题内容**

（1）设计一个用请求分页技术管理内存的虚拟存储管理系统，模拟进程运行过程中的地址映射及缺页处理。

（2）设计合理的数据结构以表示进程的页表，设计表示内存信息分布的数据结构，以便进行页面置换。

（3）设计算法，产生进程对地址空间进行访问的地址流。

（4）用页面置换算法LRU模拟缺页中断处理。

（5）对于驻留集页面的访问，能给出逻辑地址到物理地址的映射，以模拟该指令被执行。

（6）运用简单的交互和图表等方式演示运行结果。

**（二）开发环境**

（1）开发语言：C语言

（2）集成开发环境：Visual Studio 2017

# 二、理论基础

**（一）传统存储方式的弊端**

虚拟存储源于解决传统存储器管理中内存容量不够的问题。

传统存储的通病在于它们都要求将一个作业全部装入内存后才能运行（一次性），且在装入内存后整个作业一直驻留在内存中（驻留性），这对内存的压力是非常大的，容易出现两种情况：

1.大作业要求的内存空间超过内存容量，作业不能全部被装入内存，致使无法执行。

2.大量作业要求运行，但受内存容量限制，只能将少量作业装入内存，剩余作业全部留在外存等待，致使运行效率低。

**（二）局部性原理**

1968年，P.Denning体系化的指出：程序在执行时将呈现出局部性规律，即在一较短的时间内，程序的执行仅局限于部分。相应地，它所访问的存储空间也局限于某个区域。他提出了下述几个论点:

1.程序执行时，除了少部分的转移和过程调用指令外，在大多数情况下是顺序执行的。

2.过程调用将会使程序的执行轨迹由一部分区域转至另一部分区域。但经研究看出过程调用的深度在大多数情况下都不超过5。这就是说，程序将会在一段时间内，都局限在这些过程的范围内运行。

3.程序中存在许多循环结构，这些结构虽然只由少数指令构成，但是它们将被多次执行

4.程序中还包括许多对数据结构的处理，如对数组进行操作，这些处理往往都局限于很小的范围内

局限性又表现在下述两个方面

1.时间局限性。如果程序中的某条指令被执行，则不久以后该指令可能再次执行。如果某数据被访问过，则不久以后该数据可能再次被访问。产生时间局限性的典型原因是在程序中存在着大量的循环操作。

2.空间局限性。一旦程序访问了某个存储单元，在不久之后，其附近的存储单元也将被访问，即程序在一段时间内所访问的地址可能集中在一定的范围之内，其典型情况便是程序的顺序执行。

**（三）虚拟存储基本思想——“时间换空间”**

基于局部性原理，应用程序在运行之前没有必要将之全部装入内存。

操作系统给进程分配一定数量的内存空间，仅须将那些当前要运行的少数页面或段先装入内存便可运行，其余部分暂留在盘上。程序在运行时如果它所要访问的页(段)已调入内存，便可继续执行下去；但如果程序所要访问的页(段)尚未调入内存(称为缺页或缺段)，便发出缺页(段)中断请求，此时OS将利用请求调页(段)功能将它们调入内存，以使进程能继续执行下去。如果此时内存已满，无法再装入新的页(段)，OS还须再利用页(段)的置换功能，将内存中暂时不用的页(段)调至盘上，腾出足够的内存空间后，再将要访问的页(段)调入内存，使程序继续执行下去。这样，便可使一个大的用户程序在较小的内存空间中运行，也可在内存中同时装入更多的进程，使它们并发执行。

不难发现，虚拟存储的核心在于部分装入和置换。部分装入是为了减轻进程对内存空间的要求，置换是配合部分装入的使用需要。模拟系统的全程也正是围绕着如何给定“部分”、如何装入、什么情况需要置换、如何置换等展开。其余部分与传统存储方法类似。

本次课程设计的题目请求分页模拟系统也正是在分页存储的基础上增加了虚拟内存的部分。

# 三、总体设计

**（一）提前说明**

①请求分页存储器管理方式由分页存储器管理方式增加虚拟内存的功能演化而来，在此不再赘述传统分页存储的相关部分。

②用软件模拟操作系统必然带来硬件接口表达上的欠缺，所以对涉及硬件部分的模拟进行了有选择的简化。本次请求分页存储管理重点在于模拟页面调入与置换，故而主观上忽略了多道程序并发、缺页中断机构、磁盘读写、物理块分配算法、移位寄存器的部分，或者用其他简便的方法代替了它们的实际动作。

**（二）整体流程（用户）**

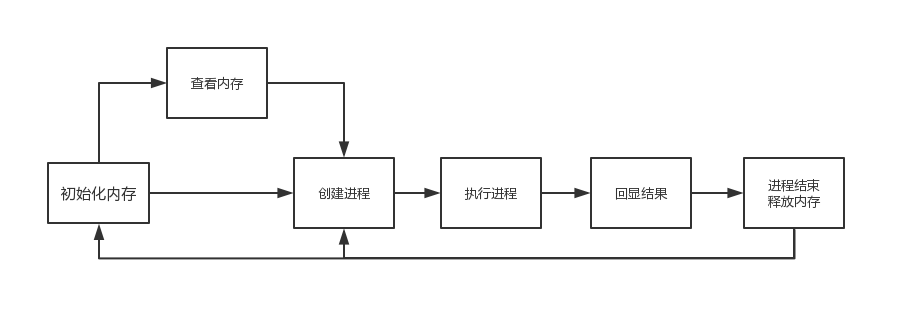
对于人机交互的过程而言，整体流程如下图所示：

图 1 整体流程（用户）

演示程序需要用户控制菜单操作和输入进程的逻辑地址长度，主要执行部分都以内置形式完成，无需用户操作。在实际情况中也确实如此，用户提交完作业后就不再对作业拥有控制权，执行进程和回显结果部分对用户是透明的，所有操作都由操作系统完成。

**（三）整体流程（系统）**

对于系统而言，它的流程图就需要增加对用户来说透明的部分，主要是执行进程中地址变换过程和增加内存分配的行为。如图2所示。

创建进程时会要求用户输入进程的逻辑地址长度，并检测逻辑地址长度的合法性，合法的逻辑地址长度应该是其对应的最小物理块数可以放入内存空间。

创建进程还需安排地址访问流传递给进程执行函数。在模拟系统中，安排地址访问流这一步是直接由系统通过一个随机函数实现，而实际情况会由程序本身决定它的地址流。在此还主观刻意避免了局部性原理，防止演示过程总停留在相近页面的访问上。

执行进程的过程就是对进程地址流逐一进行地址变换的过程，其运行结果会逐条回显。并在适当的情况改变物理块分配情况。执行进程部分是模拟系统的核心功能，将会在详细设计中阐述。

进程结束后即释放内存，返回菜单。

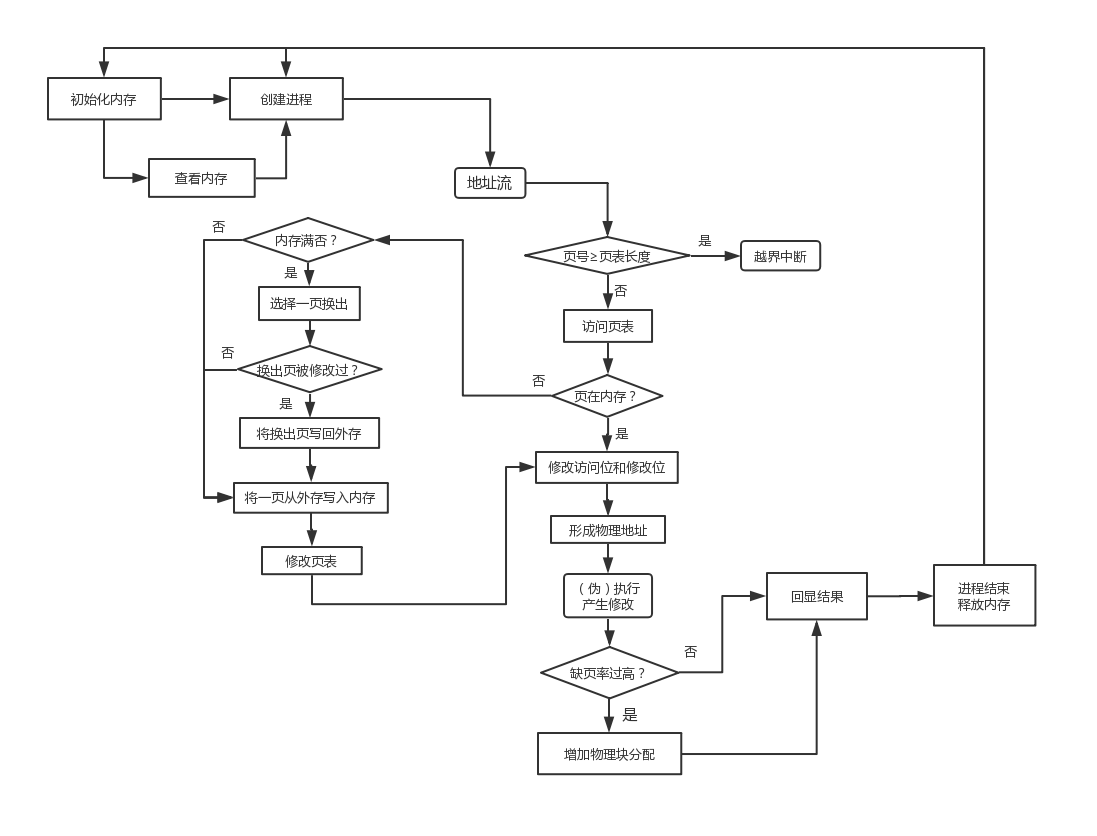


图 2 整体流程（系统）

**（四）内存分配策略**

本模拟系统选择采用可变分配局部置换的内存分配策略。

该策略是固定分配局部置换的改进。系统为每个进程分配一定数量的物理块，但当某进程发现缺页时，只允许从该进程在内存的页面汇总选择一页换出。如果进程在运行过程中频繁地发生缺页中断，则系统须再为该进程分配若干附加的物理块，直至该进程的缺页率f减少到适当的程度为止。如果缺页率特别低，则可以适当减少分配给该进程的物理块。

为了模拟系统能演示地址变换过程的各种情况，刻意将初始分配物理块的数量设置为（逻辑）页数的1/4。按照常规情况，这个分数设为1/2较为合适。

由于地址流用随机算法得出，且每次的物理块增加量都是一个较小值，无需减少物理块这一操作。

**（五）页面置换算法**

本模拟系统选择采用最近最久未使用（LRU）置换算法。

最近最久未使用(LRU)的页面置换算法是根据页面调入内存后的使用情况做出决策的。由于无法预测各页面将来的使用情况只能利用“最近的过去”作为“最近的将来”的近似，因此LRU置换算法是选择最近最久未使用的页面予以淘汰。该算法赋予每个页面一个访问字段，用来记录一个页面自上次被访问以来所经历的时间t。当需淘汰一个页面时，选择现有页面中其t值最大的，即最近最久未使用的页面予以淘汰。

实际操作时并没有使用访问字段来记录页面自上次被访问以来所经历的时间t，而是使用一个栈保存在内存中的那些页面的页号，以记录访问顺序。

# 四、详细设计

**（一）核心数据结构与流程**

模拟系统的核心数据结构是请求页表与内存使用情况说明，核心流程是地址变换过程。

**1.请求页表**

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| 页号 | 物理块号 | 状态位P | 访问字段A | 修改位M | 外存地址 |

简要说明各字段意义：

（1）物理块号：页面在内存中的实际位置

（2）状态位P：由于在请求分页系统中，只将应用程序的一部分调入内存，剩余部分仍在外存磁盘上，故须在页表中增加一个存在位字段。它用于指示该页是否已调入内存，供程序访时参考。

（3）访问字段A：用于记录本页在一段时间内被访问的次数，或记录本页最近已有多长时间未被访间，提供给置换算法程序在选择换出页面时参考。

（4）修改位M：标识该页在调入内存后是否被修改过。由于内存中的每一页都在外存上保留一份副本，因此在置换该页时，若未被修改，就不需再将该页写回到外存上，以减少系统的开销和启动磁盘的次数；若已被修改，则必须将该页重写到外存上，以保证外存中所保留的副本始终是最新的。

（5）外存地址:用于指出该页在外存上的地址，通常是物理块号，供调入该页时参考。

在实际操作过程中，某进程的页表是一个定长数组，存放页表项结构体。在模拟系统中，因为不涉及硬件，所以不使用外存地址这一项。

表1 请求页表

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 页号 | 物理块号 | 状态位P | 访问字段A | 修改位M |
| 0 |  |  |  |  |
| 1 |  |  |  |  |
| ... |  |  |  |  |
| n |  |  |  |  |

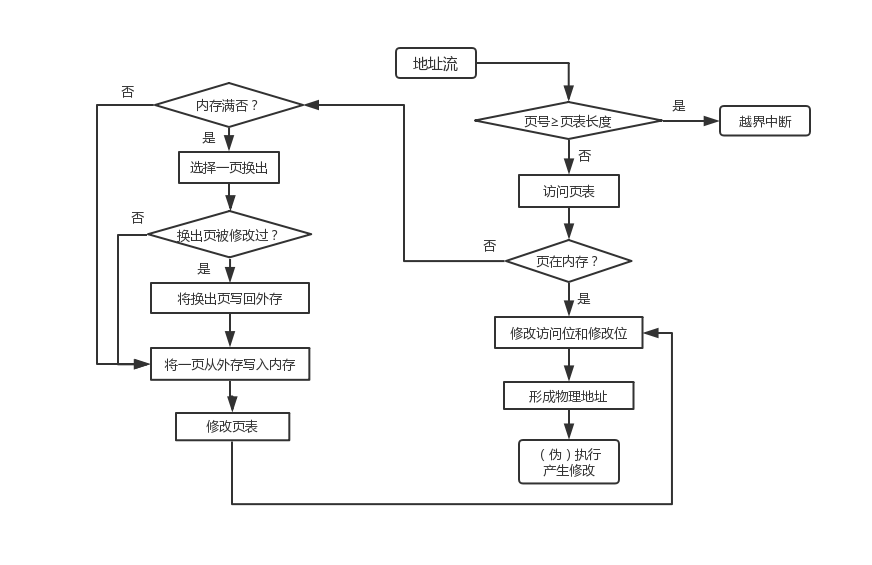
**2.内存使用情况说明**

内存使用情况均以物理块（页框）为单位，说明主要分为：内存剩余容量和各块使用情况。

内存剩余容量供判断初分配物理块和再分配物理块合法性时使用。

各块使用情况供页面载入时使用。使用简单的一维数组标记所有物理块的使用情况。

**3.地址变换过程&页面置换**

图 3 地址变换

地址变换过程正是用户视角中程序执行的那一部分。地址变换过程带来了整个系统最核心的部分，也就是LRU页面置换过程中的三种情况：

**（1）页面不在内存中，且内存有物理块待分配**

将空闲物理块分配给该页面，并修改页表项即可。（见图4）



图 4 页面置换情况（1）

**（2）页面不在内存中，且内存物理块已被分配完**

在栈中找到LRU页面，将其从内存踢出，并修改页表项。

如果LRU页面在执行过程中被修改过，那么需要将其写回内存。如果未修改，则无动作。将LRU页面的物理块传给新页面使用，修改新页面的页表项。（见图5）

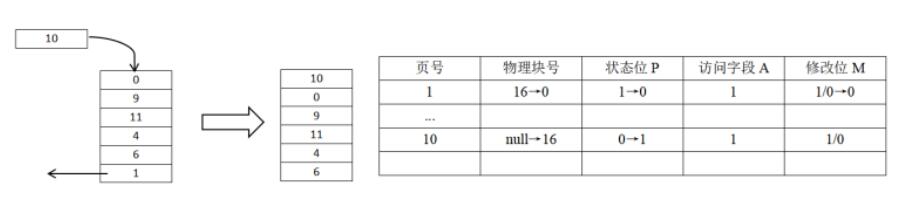


图 5 页面置换情况（2）

**（3）页面已在内存中**

只需要将页面提到栈顶，并修改页表项必要的项即可。（见图6）

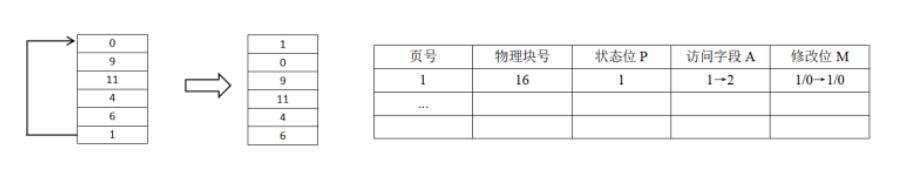


图 6 页面置换情况（3）

因为模拟系统的限制，故而删去了外存取读的动作，只保留了软件系统可以模拟的部分，主要是对页表的操作。

为模拟过程不至于被罕见的逻辑地址请求越界打断，所以设计的地址流规定是合法的，人为地屏蔽了越界中断的情况。

在设计模拟系统的初期忘记考虑可以模拟快表，是一大缺憾。

**（二）其他数据结构和过程**

**1.栈——LRU置换算法支持**

这个特殊的栈可能是一个硬件，如果是软件的话，使用栈这个数据结构可能并不很好，需要大量的移动数据。但在模拟过程中，还是尊重事实，使用栈。

每当进程访问某页面时，便将该页面的页号放在栈顶。因此栈顶始终是最近被访问页面的页号，栈底是最久未使用页面的页号。

栈的变化情况详见之前的地址变化过程的图标演示（三种情况）。

此外，用一个变量记录空闲物理块号的数量，方便对栈进行操作。

**2.内存再分配过程**

每次地址变换过程后都需要计算缺页数及缺页率，如果一段时间内f过高则需要再分配一些物理块给进程，以缓解抖动的情况。

具体实现过程是在发生缺页并需要置换的情况下，计算缺页率f。在达到增加分配的要求后分配，注意增加分配是在此次地址变换过程末尾执行的，并不影响此次置换情况。该置换的页面仍需置换，下一次页面装入才可以享受这一次增加分配带来的空闲内存空间。

**3.硬件模拟过程**

# 模拟系统自然不可能使用硬件接口，但不妨碍我们模拟使用硬件。用若干函数打印硬件需求或硬件过程，以示完成了硬件动作。如：写入内存，写回磁盘、执行命令等。

# 五、测试与运行

**（一）用户使用说明**

1.主菜单



图 7 主菜单

主菜单共有4个功能，但在初次进入系统时强烈建议先初始化内存。如果未初始化进程时误入了查看内存和执行程序，系统会强制用户初始化内存。如图8所示。

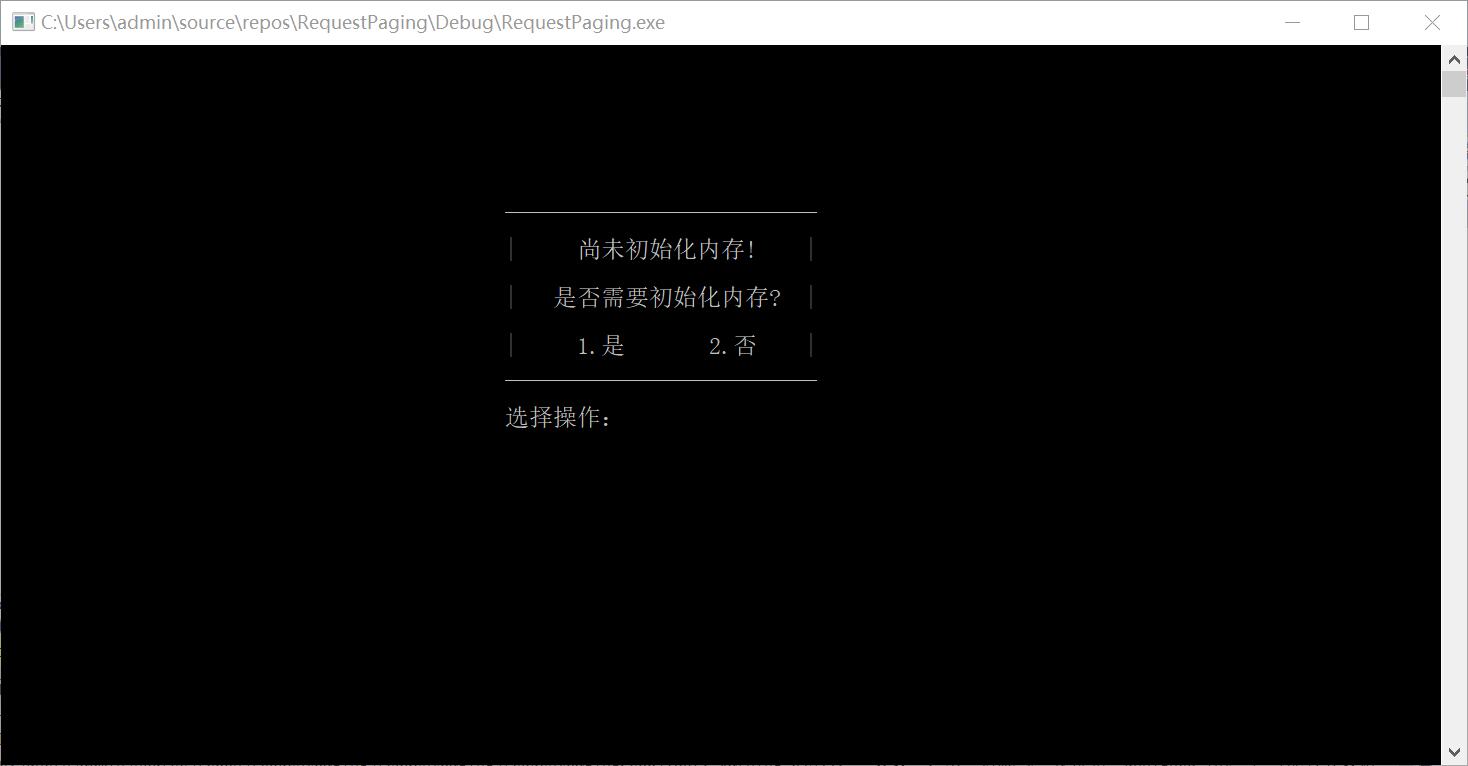


图 8 检测是否初始化内存

2.初始化内存



图 9 初始化内存

初始化内存的功能是模拟一部分内存假设已被使用，剩余空间供进程使用。可以重复初始化内存，但没有必要。

3.创建进程



图 10 创建进程

系统给出了逻辑地址长度的安全范围，非法输入将被退回。输入合法长度后系统将给出页数和分配给进程的块数。

4.执行进程

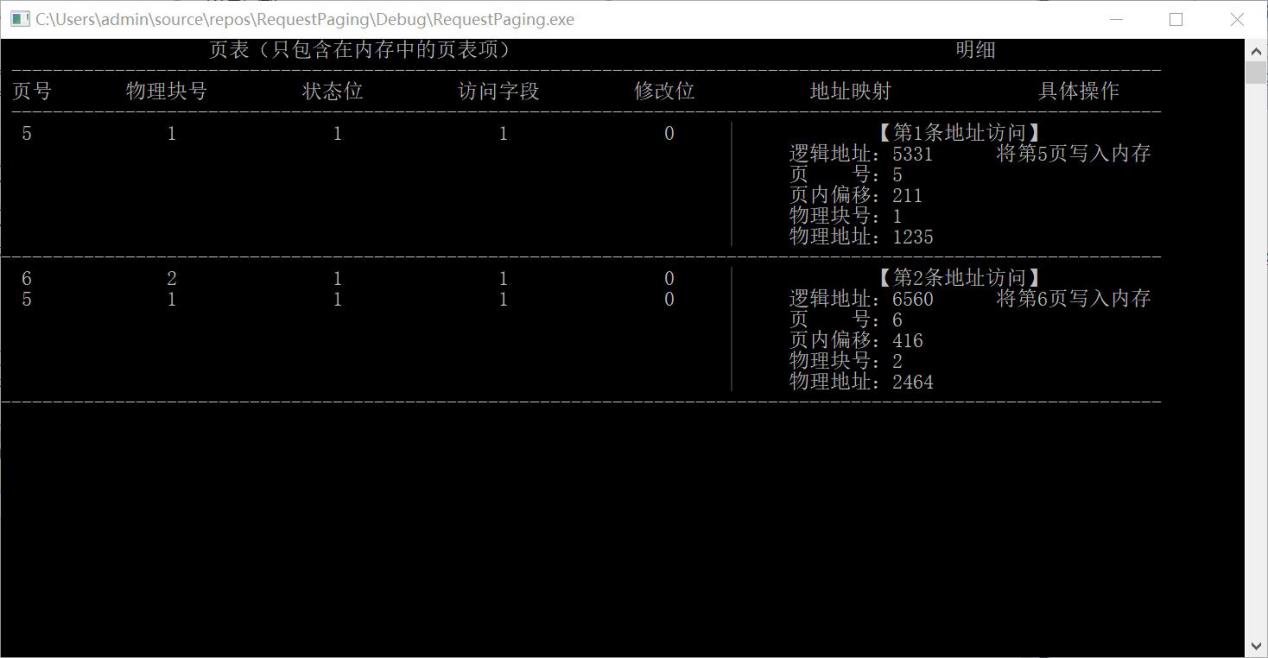


图 11 执行进程

用户只需回车即可逐条阅览页面访问时的页表变化情况和操作明细。

**（二）测试与运行**

严格得说，模拟系统只有一处需要手工输入数据，即逻辑地址总长度。但只对逻辑地址总长度选择不同的测试数据是不可行的，因为地址访问流的随机性不能确保所有页面置换情况都被覆盖到。所以在简单测试不同逻辑地址总长度的基础上，反复多次测试，以覆盖所有页面置换情况。

需要出现的情况有：

1.页面不在内存中，且内存有物理块待分配（图12）；

2.页面已在内存中（图13）；

3.页面不在内存中，且内存物理块已被分配完，无需增加物理块（图14）；

4.页面不在内存中，且内存物理块已被分配完，需增加物理块（图15）。

在测试的同时给出运行结果。

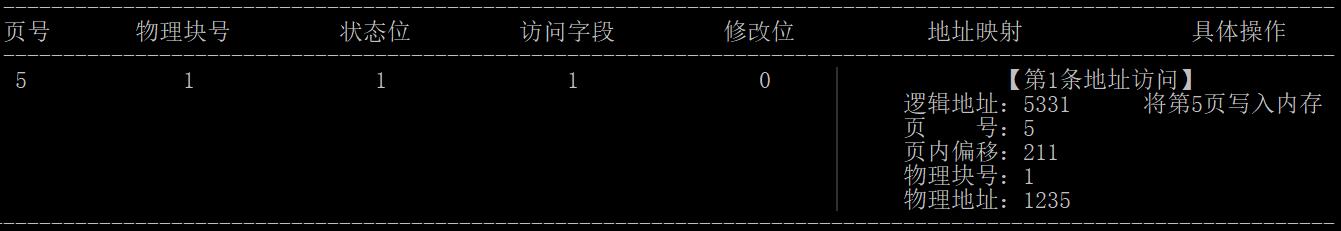


图 12 情况1

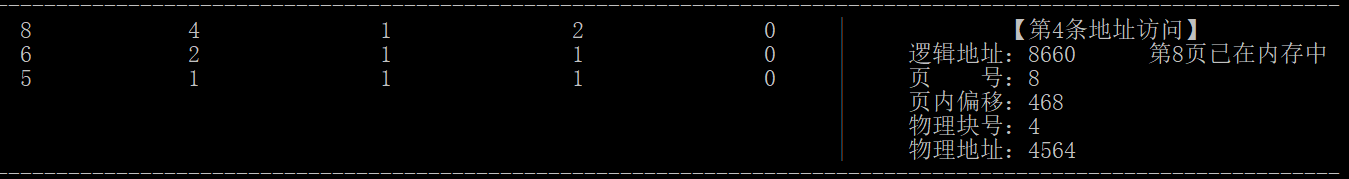


图 13 情况2

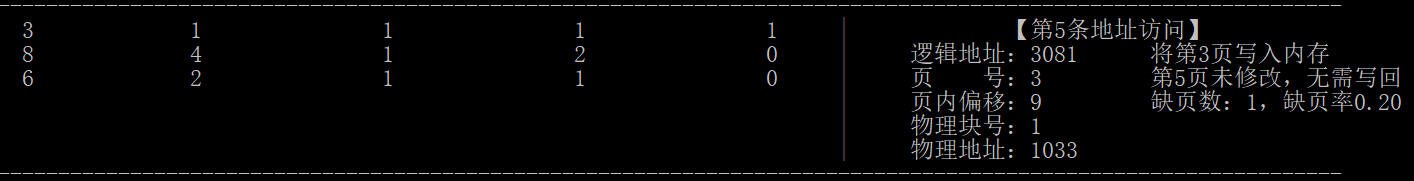


图 14 情况3

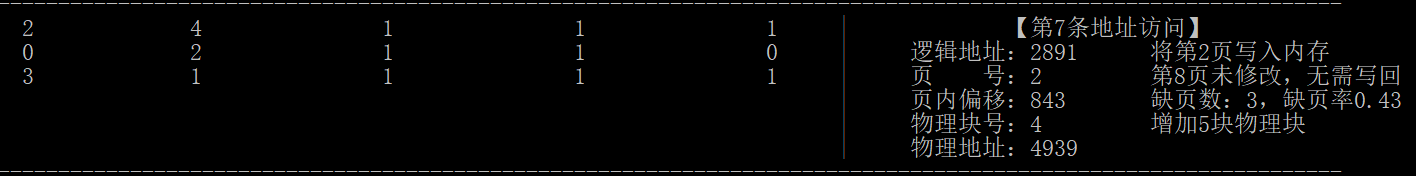


图 15 情况4

# 六、总结与改进

**（一）总结**

此次课设目的在于完成一个基于LRU置换算法的请求分页模拟系统。用较为合理的数据结构和算法基本实现了其核心功能：页面置换和地址映射，并用较为清晰的界面演示了整个过程。

此模拟系统的难点在于使用LRU置换算法对下一逻辑地址访问的处理，主要在于页面是否需要置换、置换哪个页面、如何置换页面、页表如何更新、页面号与物理块号的映射及逻辑地址与物理地址的映射。

私以为在更进一步的模拟系统中可以考虑与其他部分的衔接，如与进程并发、磁盘调度的配套使用等。

未使用可视化开发是因为认为控制台页面足以将流程表述清晰。

**（二）待改进**

1.快表模拟

在地址映射过程中应该增加快表模拟，单一个慢表显得过于简单了，也不符合实际情况。如果需要模拟快表，需要注意快表的置换方法和快慢表的配合。

2.进程并发

实现多进程并发对请求页表的过程本质上并无影响，对每个进程而言地址访问流都是固定的，并不随着并发而产生变化。如果能实现进程并发，对内存的控制会显得更加重要。

3.多种页面置换算法

虽然在实际情况中，只可能采用一种页面置换算法，但作为模拟系统实现多种页面置换算法显然更符合“展示”的要求

4.避免页号请求大于页表长度的情况

为了使每一次程序都能完整走完，避免在演示过程中出现断档的情况，在产生逻地址访问流时刻意只产生合法的地址。如果从演示的角度而言，可以“显示中断”而不真正中断，这样既可以演示越界错误，也不会打断程序的正常运行。

# 八、参考文献

1.汤子瀛.计算机操作系统[M].西安电子科技大学出版社:西安,2014:164-180.

# 九、附录

#include "stdafx.h"

#include "stdlib.h"

#include "windows.h"

#include "string.h"

#include "time.h"

//页表项

struct pageTableEntry {

int frame = -1; //物理块号

int state = 0; //状态位（是否调入内存）

int visitFrequency = 0; //访问字段（访问次数）

int change = 0; //修改位（是否发生变化，需要重新写入磁盘）

};

int RAM\_Length = 1048576; //内存地址总行数,假定1MB

int pageLength = 1024; //页长、块长,假定1KB

int frameNum = 1024; //物理块数,假定1K块

int frame\_used[1024]; //标记物理块是否被使用

int remainingFrame = 1024; //剩余块数

double f\_standard = 0.35; //标准高位缺页率

int frame\_increase = 5; //标准增加块数（再分配到某进程的物理块数）

int frame\_proportion = 4; //初分配时物理块数量占页数的frame\_proportion分之一

int initRAM\_flag = 0; //是否初始化内存

//函数声明

void menu();

void loca(int x, int y);

void writeToRAM(int instruction\_page);

void writeToDisk(int last\_recently\_used);

void initRAM();

void createProcess();

void run(int physical\_address);

void LRU(struct pageTableEntry \*, int, int, int);

void print\_pageTableEntry(struct pageTableEntry \*pagetable, int instruction\_page);

void over();

void checkRAM();

void uninitialized\_RAM();

int get\_free\_frame();

//菜单

void menu() {

while (1)

{

system("cls"); //清屏

loca(46, 6);printf("请求分页存储管理模拟系统");

loca(46, 10);printf("| 1.初始化内存 |");

loca(46, 12);printf("| 2.查看内存 |");

loca(46, 14);printf("| 3.执行程序 |");

loca(46, 16);printf("| 4.退出系统 |");

loca(80, 26);printf("Designed by Zhou");

loca(46, 20);printf("------------------------");

loca(52, 22);

printf("请选择操作：");

int t; //选择功能

scanf\_s("%d", &t);

switch (t)

{

case 1: initRAM();; break;

case 2: checkRAM(); break;

case 3: createProcess();break;

case 4: over(); break;

default:break;

}

}

}

void createProcess() {

uninitialized\_RAM();

void uninitialized\_RAM();

int logic\_address\_length; //逻辑地址总长度

while (1) {

system("cls");

loca(40, 8);

int logic\_address\_length\_max = 256 \* pageLength < remainingFrame \* frame\_proportion \* pageLength ?

256 \* pageLength : remainingFrame \* 2 \* pageLength;

printf("逻辑地址长度要求为 [ 1 , %d ]", logic\_address\_length\_max);

loca(40, 10);

printf("请输入逻辑地址总长度：");

loca(62, 10);

scanf\_s("%d", &logic\_address\_length);

//判定一下逻辑地址大小不合格就重来

//逻辑地址大小取决于页表长度和内存的剩余物理块数量

if (logic\_address\_length > 0 && logic\_address\_length < 256 \* pageLength

&& logic\_address\_length < remainingFrame \* 2 \* pageLength) {

break;

}

else{

loca(40, 15);printf("逻辑地址长度异常，请重新输入!");

Sleep(1000);

}

}

int page\_count = logic\_address\_length / pageLength + 1; //逻辑地址的页数

int frame\_count = page\_count / frame\_proportion + 1; //按照页数的若干分之一分配物理块数

if (frame\_count > remainingFrame) {

printf("内存不够了");

}

else //内存够则开始执行程序

{

//剩余物理块数减少

remainingFrame -= frame\_count;

//创建页表

/\*用固定长度的数组必然会有一些问题，用可变数组会比较好，但暂不解决\*/

struct pageTableEntry pageTable[256];

struct pageTableEntry \*pagetable= pageTable;

loca(40, 12);

printf("页数：%d，分配到物理块数：%d", page\_count, frame\_count);

getchar(); getchar();

LRU(pagetable, page\_count, frame\_count, logic\_address\_length);

free(pagetable);

}

}

void LRU(struct pageTableEntry \*pagetable, int page\_count, int frame\_count, int logic\_length) {

system("cls");

int loca\_control = 4; //控制光标的行数

int missing\_page = 0; //缺页数

double f = 1; //缺页率

int arr\_length = frame\_count; //空物理块数量

//需要一个数据结构记录访问页号的顺序

int \*arr; //arr数组记录页面使用顺序，下标越大越是近期使用

if ((arr = (int \*)malloc(frame\_count \* sizeof(int))) == NULL)

{

printf("（实际）分配内存空间失败，程序退出！");

exit(0);

}

srand(time(NULL)); //与时间有关的种子，确保每次随机的结果不同

loca(20, 0);

printf("页表（只包含在内存中的页表项） 明细");

loca(1, 1);

printf("--------------------------------------------------------------------------------------------------------------------");

loca(1, 2);

printf("页号 物理块号 状态位 访问字段 修改位 地址映射 具体操作");

loca(1, 3);

printf("-------------------------------------------------------------------------------------------------------------------- ");

//假设需要访问xxx条逻辑地址，每次逻辑地址通过随机函数得到

for (int i = 0; i < page\_count; i++) {

/\*虽然因为局部性原理，逻辑地址总是抱团出现，但对模拟系统而言，随机产生逻辑地址能更好的展示各种情况\*/

int instruction = (int)(rand() % logic\_length); //随机下一条指令的逻辑地址

int instruction\_page = instruction / pageLength; //逻辑地址对应的页号

int instruction\_shifting = instruction % pageLength; //逻辑地址的页内偏移

loca(84, loca\_control);

printf("【第%d条地址访问】", i + 1);

loca(76, loca\_control + 1);

printf("逻辑地址：%d", instruction);

loca(76, loca\_control + 2);

printf("页 号：%d", instruction\_page);

loca(76, loca\_control + 3);

printf("页内偏移：%d", instruction\_shifting);

//如果下一个使用的页号已经在内存中了

if ((pagetable + instruction\_page)->state == 1) {

int record = 0;

for (int i = 0; i < frame\_count - arr\_length; i++) {

if (arr[i] == instruction\_page) {

record = i; //记录instruction\_page原来所在的数组下标

break;

}

}

if (record != frame\_count - arr\_length - 1) {

for (int i = record + 1; i < frame\_count - arr\_length; i++) {

arr[i - 1] = arr[i];

}

arr[frame\_count - arr\_length - 1] = instruction\_page;

}

(pagetable + instruction\_page)->visitFrequency++;

loca(96, loca\_control + 1);

printf("第%d页已在内存中", instruction\_page);

loca(76, loca\_control + 4);

printf("物理块号：%d", (pagetable + instruction\_page)->frame);

}//if

else //下一个页号不在内存中,需要调入页面

{

loca(96, loca\_control + 1);

writeToRAM(instruction\_page);

//如果需要换出页面

if (arr\_length == 0) {

int last\_recently\_used = arr[0]; //取出最近最久未使用的页面

for (int i = 1; i < frame\_count; i++) {

arr[i - 1] = arr[i];

}

//换出页面的页表调整

(pagetable + last\_recently\_used)->state = 0;

(pagetable + instruction\_page)->frame = (pagetable + last\_recently\_used)->frame;

(pagetable + last\_recently\_used)->frame = -1;

(pagetable + last\_recently\_used)->visitFrequency = 0;

//打印物理块号

loca(76, loca\_control + 4);

printf("物理块号：%d", (pagetable + instruction\_page)->frame);

//处理撤出内存的页面的回写

loca(96, loca\_control + 2);

if ((pagetable + last\_recently\_used)->change == 1) {

writeToDisk(last\_recently\_used); //页面修改过，需要写回

(pagetable + last\_recently\_used)->change = 0;

}

else {

printf("第%d页未修改，无需写回", last\_recently\_used);

}

//换入页面的页表调整

(pagetable + instruction\_page)->state = 1; //调入的页面需改变状态

(pagetable + instruction\_page)->visitFrequency++;

arr[frame\_count - 1] = instruction\_page;

//实现可变分配局部置换，在缺页率过高时多分配一些物理块

missing\_page+=1;

f = ((double)missing\_page) / (i+1); //计算缺页率

loca(96, loca\_control + 3);

printf("缺页数：%d，缺页率%.2lf \n", missing\_page,f);

//如果缺页率>标准缺页率 且 内存有可以增加的位置，就增加物理块分配

if (f > f\_standard && remainingFrame>=f\_standard) {

loca(96, loca\_control + 4);

printf("增加%d块物理块",frame\_increase);

arr\_length += frame\_increase;

frame\_count += frame\_increase;

//更新arr数组，使其可以容纳增加的物理块

int \* p= (int \*) malloc(frame\_count \* sizeof(int));

if ( p == NULL )

{

printf("（实际）分配内存空间失败");

exit(0);

}

for (int j = 0; j < frame\_count - arr\_length ; j++)

{

p[j] = arr[j];

}

arr = p;

//free(p);

}//if

else{} //为了抵消if，防止影响下面的else

}

//如果还有物理块可分配

else if (arr\_length > 0) {

int frame = get\_free\_frame(); //找一个空闲的物理块

loca(76, loca\_control + 4);

printf("物理块号：%d", frame);

if (frame >= 0) {

arr[frame\_count - arr\_length] = instruction\_page;

arr\_length--;

(pagetable + instruction\_page)->frame = frame; //页表项中的物理块号改一下

(pagetable + instruction\_page)->state = 1; //确认该页已被调入内存

(pagetable + instruction\_page)->visitFrequency++;//访问字段自加，暂时没有用

frame\_used[frame] = 1; //标记该物理块号已被使用

}

else{

loca(76, loca\_control + 4);

printf("无空闲物理块可用");

}

}//else if

}//else

//计算物理地址，run代表其已被执行

int physical\_address = pageLength \* (pagetable + instruction\_page)->frame + instruction\_shifting;

loca(76, loca\_control + 5);

run(physical\_address);

//模拟页面修改。随机到1就代表内存中的页面发生了更改，需要写回。

int flag = (int)rand() % 2;

if (flag == 1) (pagetable + instruction\_page)->change = 1;

//打印页表信息

for (int k = loca\_control; k < loca\_control + frame\_count - arr\_length; k++) {

loca(2, k);

//print\_pageTableEntry(pagetable, instruction\_page);

print\_pageTableEntry(pagetable, arr[frame\_count - arr\_length - (k - loca\_control) - 1]);

}

int cc = frame\_count - arr\_length < 6 ? 6 : frame\_count - arr\_length;

loca(0, cc+loca\_control);

printf("---------------------------------------------------------------------------------------------------------------------");

for (int z = loca\_control; z < cc + loca\_control; z++) {

loca(70, z); printf("|");

}

loca\_control += cc+1;

//手动控制，一步一步显示变化

getchar();

}//for

loca(40, loca\_control + 5);

printf(" 进程结束");

loca(40, loca\_control + 7);

printf("任意键返回菜单");

char g = getchar();

free(arr);

remainingFrame+= frame\_count - arr\_length;

menu();

}

//打印页表项(页号、物理块号、状态位、访问字段、修改位）

void print\_pageTableEntry(struct pageTableEntry \*pagetable,int instruction\_page){

printf("%d\t\t%d\t\t%d\t\t%d\t\t%d", instruction\_page, (pagetable + instruction\_page)->frame,

(pagetable + instruction\_page)->state, (pagetable + instruction\_page)->visitFrequency, (pagetable + instruction\_page)->change);

}

//找空闲物理块

int get\_free\_frame() {

for (int i = 0; i < frameNum; i++) {

if (frame\_used[i] == 0) return i;

}

return -1;

}

//从磁盘写入内存，因为是模拟，所以只打印语句表示执行

void writeToRAM(int instruction\_page) {

printf("将第%d页写入内存", instruction\_page);

};

//写入磁盘，因为是模拟，所以只打印语句表示执行

void writeToDisk(int last\_recently\_used) {

printf("将第%d页的内容写回磁盘\n",last\_recently\_used);

};

//执行物理地址的命令

void run(int physical\_address) {

printf("物理地址：%d", physical\_address);

}

//处理未初始化内存的情况

void uninitialized\_RAM() {

if (initRAM\_flag == 0) {

system("cls");

loca(42, 6);

printf("\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_");

loca(42, 8);

printf("| 尚未初始化内存! |");

loca(42, 10);

printf("| 是否需要初始化内存? |");

loca(42, 12);

printf("| 1.是 2.否 |");

loca(42, 13);

printf("\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_");

loca(42, 15);

printf("选择操作：");

char s;

while (1) {

getchar();

s = getchar();

switch (s)

{

case'1':initRAM(); break;

case'2':menu(); break;

default:break;

}

}

}

}

//内存物理块的初始化

void initRAM()

{

system("cls");

initRAM\_flag = 1; //标志内存已初始化

remainingFrame = RAM\_Length / pageLength; //每次初始化内存都是全新的开始

//刻意制造一些内存块已被使用的情况

srand(time(NULL));

for (int i = 0; i < frameNum; i++) {

if ((frame\_used[i] = (int)rand() % 2) == 1)

remainingFrame--;

}

loca(42, 6);

printf("\* 内存物理块已经成功初始化 \*");

loca(42, 8);

printf("\* 共有%d个可用物理块 \*",remainingFrame);

loca(42, 10);

printf("\* 每块大小为%dB \*", pageLength);

loca(48, 13);

printf("------------------");

loca(48, 14);

printf("| 1.执行程序 |");

loca(48, 16);

printf("| 2.返回菜单 |");

loca(48, 17);

printf("------------------");

loca(48, 19);

printf("请选择操作：");

char t;

while (1) {

t = getchar();

switch (t)

{

case'1':createProcess();break;

case'2':menu(); break;

default:break;

}

}

}

//查看内存

void checkRAM() {

uninitialized\_RAM(); //未初始化内存处理

system("cls");

loca(42, 8); printf("\* 共有%d个可用物理块 \*", remainingFrame);

loca(42, 10); printf("\* 每块大小为%dB \*", pageLength);

loca(48, 13); printf("------------------");

loca(48, 14); printf("| 1.执行程序 |");

loca(48, 16); printf("| 2.返回菜单 |");

loca(48, 17); printf("------------------");

loca(48, 19); printf("请选择操作：");

char t;

while (1) {

t = getchar();

switch (t)

{

case'1':createProcess(); break;

case'2':menu(); break;

default:break;

}

}

}

//退出系统

void over() //退出系统

{

char t;

system("cls");

loca(48, 11);

printf("-----------------------");

loca(48, 12);

printf("| 您确定要退出吗? |");

loca(48, 14);

printf("| 1.确定 2.取消 |");

loca(48, 15);

printf("-----------------------");

loca(48, 16);

while (1)

{

t = getchar();

switch (t)

{

case '1':

system("cls");

loca(48, 10);

printf("正在退出....");

Sleep(1500);

system("cls");

loca(48, 10);

printf("已退出软件");

loca(48, 12);

printf("谢谢使用！");

Sleep(1000);

loca(48, 14);

exit(0); break; //终止程序

case '2':

menu(); break; //调用函数，进入菜单

default:break;

}

}

}

//将光标移动到x,y坐标处

void loca(int x, int y)

{

COORD pos = { x,y };

//COORD是Windows API中定义的一种结构，表示一个字符在控制台屏幕上的坐标

HANDLE Out = GetStdHandle(STD\_OUTPUT\_HANDLE);

//GetStdHandle（）返回标准的输入、输出或错误的设备的句柄，也就是获得输入、输出/错误的屏幕缓冲区的句柄

SetConsoleCursorPosition(Out, pos);

//API中定位光标位置的函数

}

int main()

{

menu();

return 0;

}