教育部与高通公司产学合作协同育人项目

燕山大学课程设计说明书

课程设计名称：操作系统

题目：分页管理系统页面置换算法设计与实现

年级：2018 级

开发小组名称：OPT 小组

小组自评成绩：A+

小组负责人：乔翱

课题组成员：

姓名	学号	班级	自评成绩	分工	签字
乔翱	201811040809	软件 6 班	A+	独立完成	

课题开发日期：2020 年 12 月 21 日 - 2020 年 12 月 31 日

指导教师：申利民、李贤善、王林、穆运峰、陈真、尤殿龙 等

教育部与高通公司产学合作协同育人项目

基于操作系统课程的实践能力和创新能力培养模式建立与实践

目录

1 概述

1.1 目的

1、通过模拟实现几种基本的页面置换算法，更深一步了解页面置换算法，了解虚拟存储的特点以及虚拟存储的好处。

2、掌握虚拟存储页式管理中主要的页面置换算法的基本思想，包括 OPT、FIFO、LRU 等，巩固有关虚拟内存管理的知识，并且通过编程实现这些算法。

3、通过利用多进程程序，对不同的页面算法建立多个进程，并且并行执行，对不同的页面置换算法进行比较，比较各类页面置换算法的优点和缺点，思考各类算法在哪种情况下更加适用。

1.2 意义

操作系统是计算机中最重要的软件，而对于内存的管理更是操作系统中一个特别重要的部分。在地址映射过程中，若在页面中发现所要访问的页面不在内存中，则产生缺页中断。当发生缺页中断时，操作系统必须在内存中选择一个页面将其移出内存，用来选择淘汰哪一页的规则就是置换算法。

通过本次项目，加深对虚拟存储中页式存储管理的理解，进一步理解各类页面置换算法，比较对比各类页面置换算法的优缺点，体会页式存储管理的创新。

1.3 主要完成的任务

1、完成各类页面置换算法，包括：LRU、OPT、FIFO、LFU，采用多种不同的算法分别完成各类页面置换算法，并且最终选择出最高效的页面置换算法。

2、建立多个线程，每个线程执行一个页面置换算法，并且在屏幕上可以同时显示不同的算法的执行结果，设计显示方式使得对于不同算法的执行结果有一个直观的感受。

3、可以输入也可以随机产生逻辑地址访问序列，自动转换为逻辑页号，产生内存页号，分别由页面置换算法完成页面置换。

4、考虑到程序的通用性，本项目能够设定驻留内存页面的个数、内存的存取时间、缺页中断的时间、快表的时间，并提供合理省缺值，可以暂停和继续系统的执行。

5、本项目能够设定逻辑地址访问序列中地址的个数和地址的范围，输入非法数据会提示输入错误，只有输入正确的符合要求的数据才能进行页面置换。

6、本项目能够设定有快表和没有快表的两种运行模式。

7、提供良好图形界面，同时能够展示每个个算法当前运行的情况和运行的结果。

8、给出每种页面置换算法每次每个页面的存取时间、每个逻辑地址对应的物理页号和内存地址。

9、为了方便对实验数据进行分析，本项目能够将每次的实验输入和实验结果存储起来，并且随时可查询，方便用户对实验数据分析。

10、完成多次不同设置的实验，总结实验数据，分析实验数据，得出结论

1.4 使用的开发工具

本项目最终的结果需要用一个界面来很好的展示，所以本项目采用 C#中的 winform 进行编写，C#中的 winform 对于一些常用的数据结构有很好的封装，方便开发，并且 C#也有相应的线程库，方便多线程程序的建立，本项目使用的开发工具为 VS。

1.5 解决的主要问题

1、置换算法的实现

本项目首先需要解决的问题就是实现置换算法，置换算法的实现较为简单，通常是通过维护一个链表来实现不同的置换算法，但是考虑到要在图形界面上显示，对算法的速度有一定的要求，所以本项目力求采用最优的算法来实现各个页面置换算法。

2、多线程执行并行执行多个页面置换算法

在实现各个页面置换算法后，要实现多进程并行执行各个页面置换算法，直观的感受不同的页面置换算法的时间差距，并且可以随时暂停程序的执行，也就是把进程挂起。

3、对实验结果的存储和查询

本项目可以对实验结果进行存储，将每次的实验结果存储到文件中，方便用户查询，对结果进行进一步的分析。

4、良好的图形界面

对于本项目的结果要求使用界面来展示，这就需要精心设计一个界面，使得用户有一个很好的界面观感，提高用户友好性，设计美观简洁的界面来直观地展示结果。

1.6 人员分工

本项目小组只有一个人，所以所有任务由一个人单独完成，一个人独立完成上述所有的任务。一人完成上面所有的任务可能略有困难，但是是对自己编程能力的一个很好的锻炼。

1.7 开发日计划

为了能够按时顺利完成本项目，按照规定的时间制定了详细的开发日计划，每日严格按照此计划执行，以保证最后按时完成课程设计，如遇意外情况可微调每日计划。日计划如下表所示：

表 7.1 开发日计划表

时间	计划
2020.12..21-2020.12.22	设计项目具有的功能，编写课程设计计划书
2020.12.23-202012.24	编写主要的页面置换算法（OPT,LRU,FIFO）
2020.12.25-2020.12.26	设计并编写图形界面实现各部分的功能
2020.12.27	完善项目功能，美化界面
2020.12.28	反复测试项目，修改其中存在的 bug
2020.12.29-2020.12.30	编写课程设计说明书并制作答辩 PPT
2020.12.31	答辩验收课程设计

2 使用的基本概念和原理

2.1 多道程序

多道程序设计是在计算机内存中同时存放几道相互独立的程序，使它们在管理程序控制之下，相互穿插的运行。两个或两个以上程序在计算机系统中同处于开始到结束之间的状态。这就称为多道程序设计。

多道程序技术运行的特征：多道、宏观上并行、微观上串行。多道程序设计的出现，加快了 OS 的诞生。多道程序设计的基本特征：间断性、共享性、制约性。

计算机内存中可以同时存放多道（两个以上相互独立的）程序，它们都处于开始和结束之间。从宏观上看是并行的，多道程序都处于运行中，并且都没有运行结束；从微观上看是串行的，各道程序轮流使用 CPU，交替执行。引入多道程序设计技术的根本目的是为了提高 CPU 的利用率，充分发挥计算机系统部件的并行性，现代计算机系统都采用了多道程序设计技术。

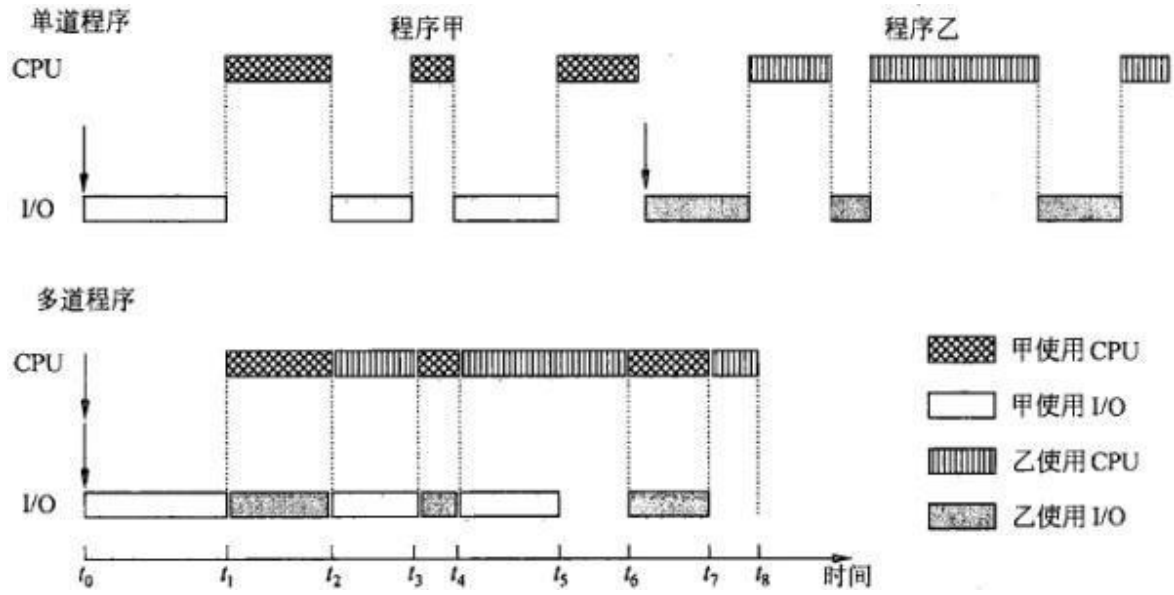

图 2.1 单道程序与多道程序比较

2.2 进程

在早期无操作系统和单道批处理系统计算机中，程序在系统中以顺序的方式执行，后进入内存的程序需要等待先进入内存的程序执行完毕才会分配 CPU 资源。在这种情况下，当某个程序因其他资源等待，一直占用 CPU 资源，其他程序将会一直处于等待状态，会导致系统的运行效率、资源的利用率以及 CPU 的使用效率比较低。

为提高资源的利用率提出了“并发执行”的概念，但是并发执行会使程序失去顺序性、封闭性及可再现性，所以通常情况下程序不能并发执行。 之后引入了进程的概念，使用 PCB 来控制、管理进程，使各个程序并发独立的运行。在后来的多道批处理系统和分时处理系统中，内存中可以同时驻足多个程序，程序在系统中并发执行。并发执行表示多个程序可以在同一个时间间隔中执行，多个程序在系统中同时驻足，提高了 CPU 的使用率、资源利用率及系统的运行效率。

进程是程序在系统中执行的一个实例，是程序在一个数据集合上运行的动态过程。进程可以提高 CPU 的利用率、使程序在系统中并发执行并可以对并发执行的程序进行描述和控制，由程序、数据、程序控制块三部分组成。

进程的特性：动态性、并发性、独立性、异步性。

动态性：进程的实质是进程实体的执行过程，是动态的且有一定生命周期。由创建产生，由调度执行，由撤销消亡。

并发性：多个进程存在内存中且能在一段时间内同时执行。

独立性：进程是能独立运行、独立获得资源并独立接受调度的基本单位。

异步性：进程以不可预知的速度向前推进，按异步方式运行，进程的运行不是一气呵成的，是走走停停的。

2.3 线程

线程（thread）是操作系统能够进行运算调度的最小单位。它被包含在进程之中，是进程中的实际运作单位。一条线程指的是进程中一个单一顺序的控制流，一个进程中可以并发多个线程，每条线程并行执行不同的任务。

线程是独立调度和分派的基本单位。同一进程中的多条线程将共享该进程中的全部系统资源，如虚拟地址空间，文件描述符和信号处理等等。但同一进程中的多个线程有各自的调用栈（call stack），自己的寄存器环境（register context），自己的线程本地存储（thread-local storage）。

一个进程可以有很多线程，每条线程并行执行不同的任务。

2.4 同步和互斥

同步：异步环境下的一组并发进程因直接制约而互相发送消息、进行互相合作、互相等待，使得各进程按一定的速度执行的过程称为进程间的同步。具有同步关系的一组并发进程称为合作进程，合作进程间互相发送的信号称为消息或事件。

互斥：两个或两个以上的进程，不能同时进入关于同一组共享变量的临界区域，否则可能发生与时间有关的错误，这种现象被称作进程互斥。也就是说，一个进程正在访问临界资源，另一个要访问该资源的进程必须等待。

2.5 页面置换算法

地址映射过程中，若在页面中发现所要访问的页面不在内存中，则产生缺页中断。当发生缺页中断时，如果操作系统内存中没有空闲页面，则操作系统必须在内存选择一个页面将其移出内存，以便为即将调入的页面让出空间。而用来选择淘汰哪一页的规则叫做页面置换算法。

最佳置换算法（OPT）（理想置换算法）：从主存中移出永远不再需要的页面；如无这样的页面存在，则选择最长时间不需要访问的页面。于所选择的被淘汰页面将是以后永不使用的，或者是在最长时间内不再被访问的页面，这样可以保证获得最低的缺页率。

先进先出置换算法（FIFO）：是最简单的页面置换算法。这种算法的基本思想是：当需要淘汰一个页面时，总是选择驻留主存时间最长的页面进行淘汰，即先进入主存的页面先淘汰。其理由是：最早调入主存的页面不再被使用的可能性最大。FIFO 算法还会产生当所分配的物理块数增大而页故障数不减反增的异常现象，这是由 Belady 于 1969 年发现，故称为 Belady 异常。只有 FIFO 算法可能出现 Belady 异常，而 LRU 和 OPT 算法永远不会出现 Belady 异常。

最近最久未使用（LRU）算法：这种算法的基本思想是：利用局部性原理，根据一个作业在执行过程中过去的页面访问历史来推测未来的行为。它认为过去一段时间里不曾被访问过的页面，在最近的将来可

能也不会再被访问。所以，这种算法的实质是：当需要淘汰一个页面时，总是选择在最近一段时间内最久不用的页面予以淘汰。

时钟(CLOCK)置换算法：LRU 算法的性能接近于 OPT,但是实现起来比较困难，且开销大；FIFO 算法实现简单，但性能差。所以操作系统的设计者尝试了很多算法，试图用比较小的开销接近 LRU 的性能，这类算法都是 CLOCK 算法的变体。简单的 CLOCK 算法是给每一帧关联一个附加位，称为使用位。当某一页首次装入主存时，该帧的使用位设置为 1;当该页随后再被访问到时，它的使用位也被置为 1。对于页替换算法，用于替换的候选帧集合看做一个循环缓冲区，并且有一个指针与之相关联。当某一页被替换时，该指针被设置成指向缓冲区中的下一帧。当需要替换一页时，操作系统扫描缓冲区，以查找使用位被置为 0 的一帧。每当遇到一个使用位为 1 的帧时，操作系统就将该位重新置为 0；如果在这个过程开始时，缓冲区中所有帧的使用位均为 0，则选择遇到的第一个帧替换；如果所有帧的使用位均为 1,则指针在缓冲区中完整地循环一周，把所有使用位都置为 0，并且停留在最初的位置上，替换该帧中的页。由于该算法循环地检查各页面的情况，故称为 CLOCK 算法，又称为最近未用(Not Recently Used, NRU)算法。

3 总体设计

3.1 技术路线

本项目采取面向过程（POP）的思想，面向过程是一种以事件为中心的编程思想，编程的时候把解决问题的步骤分析出来，然后用函数把这些步骤实现，在一步一步的具体步骤中再按顺序调用函数。

面向过程的方法的流程化使得编程任务明确，在开发之前基本考虑了实现方式和最终结果，具体步骤清楚，便于节点分析。效率高，面向过程强调代码的短小精悍，善于结合数据结构来开发高效率的程序。

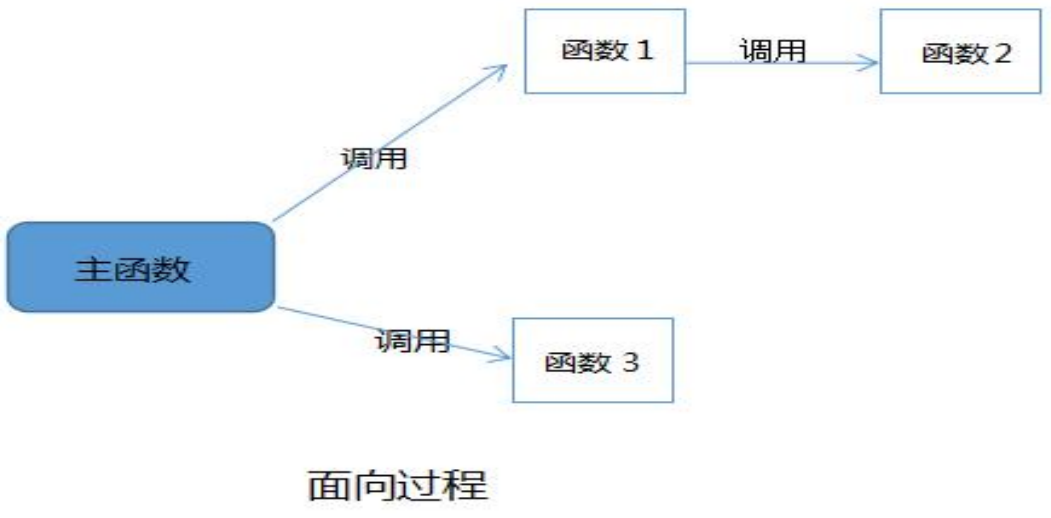

图 3.1 面向过程示意图

3.2 总体结构

本程序主要分成四个模块，分别是输入、运行、输出、保存模块。

本项目的总体结构图如下：

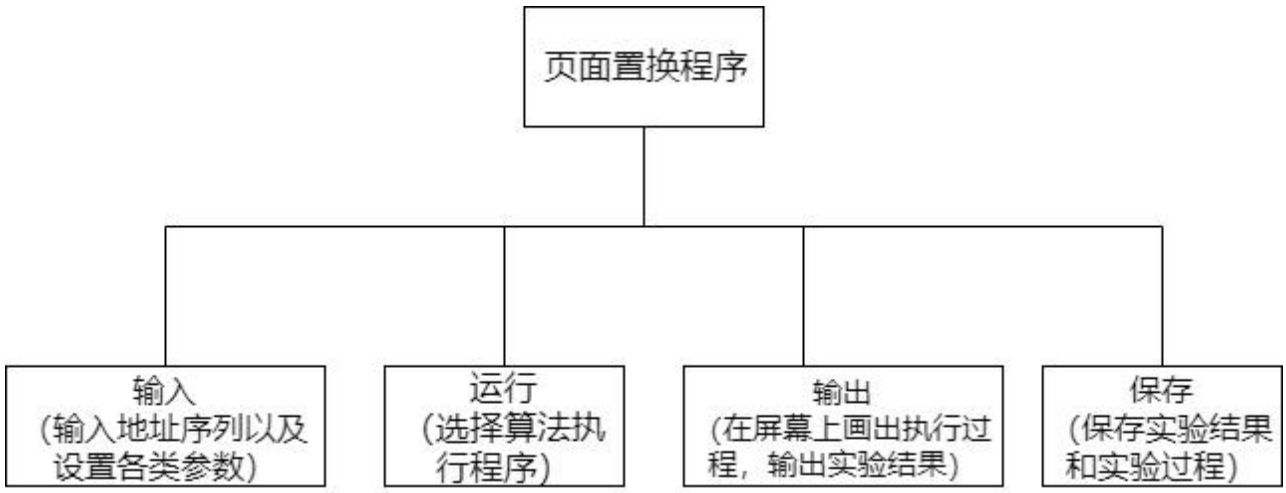

图 3.1 项目总体结构图

3.3 层次结构

根据项目的总体结构图进行细化，编写出项目的层次结构图。

本项目的层次结构图如下：

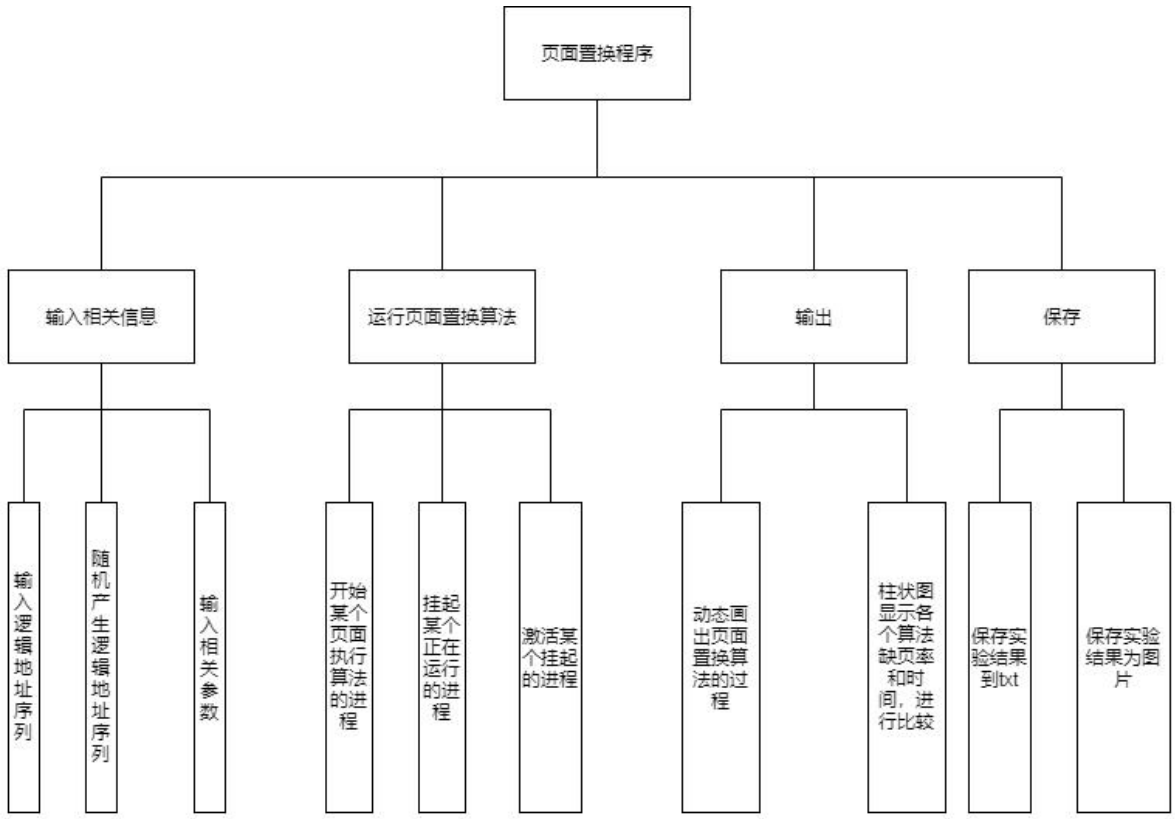

图 3.2 项目层次结构图

3.4 模块功能

本项目分为大的模块主要有四个模块：输入模块、进行页面置换算法模块、输出模块、保存模块。每个模块的功能如下所示：

输入模块：输入访问地址序列或者随机产生地址序列，还可以输入各种参数，若没有输入参数则采用默认缺省值。

进行页面置换算法模块：此模块可以开始每个页面算法线程，也可以暂停某个正在运行进程，还可以激活某个正在挂起的进程。

输出模块：此模块的功能是用来输出信息，包括动态输出实验的表格，输出实验数据的柱状图，方便用户比较各种算法。

保存模块：此模块的功能是用于保存实验数据，保存的实验数据有实验时间，实验中的逻辑地址与物理地址以及缺页率和访问时间等信息，此模块还用于保存实验过程，实验过程以图片的形式保存下来，图片名字为实验时间，方便用户查询数据。

3.5 总体流程

本项目的总体流程是：用户在进入主界面后，用户可以选择输入访问的地址序列，也可以选择随机生成地址序列，然后用户可以设置参数，若用户没有设置参数，则采用默认的缺省值。接着用户可以在控制区对各个算法线程进行控制，包括线程的开始、挂起和激活。在算法执行完毕后，会保存这次的实验结果，存储到文件中，并把实验过程以图片的形式存储起来。用户还可以对实验数据进行分析，本项目采取柱状图的形式展示不同算法的缺页率以及时间，方便用户对其进行比较。

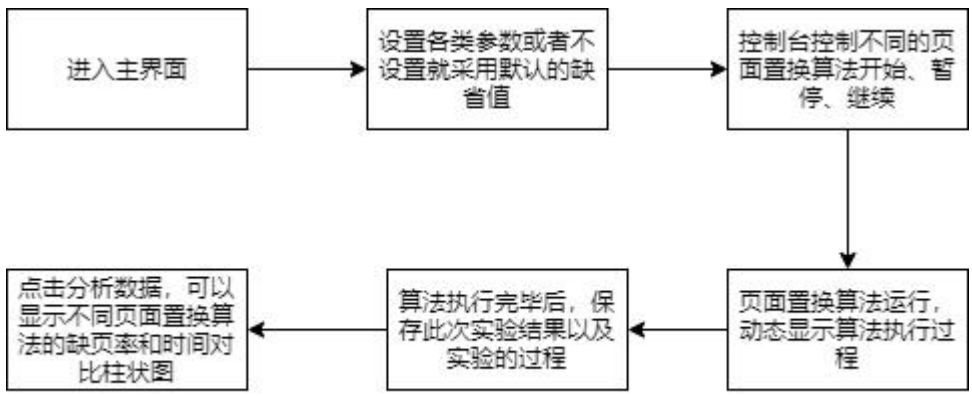

图 3.3 项目总体流程图

3.6 创立的进程、线程及其任务

本项目创立的线程主要有：主线程，FIFO 算法线程，LRU 算法线程，OPT 算法线程，画图线程，cmd 线程。各个线程的主要任务如下所示：

主线程：控制各个线程的创建、执行、挂起、激活，控制各个线程的运行。

FIFO 算法线程：执行 FIFO 算法的过程。

LRU 算法线程：执行 LRU 算法的过程。

OPT 算法线程：执行 OPT 算法的过程。

画图线程：画出屏幕上页面置换算法的动态过程，方便用户直观的看到整个页面置换算法的过程，并且对于不同的页面置换算法进行比较。

cmd 线程：执行嵌入其中的 cmd 程序，执行 cmd 程序中的页面置换算法。

4 详细设计

4.1 使用的核心函数

由于 C#对 windows api、线程都有一个很好的封装，所以本项目并没有直接调用 windows api，而是直接使用 C#封装好的库，这些库使用起来非常方便编程人员的使用。本项目主要使用的和线程有关的核心函数如下所示：

4.1.1 创建线程以及开始

创建进程并开始的函数如下所示：

```
LRUThread = new Thread(new ThreadStart(LRU));
LRUThread.IsBackground = true;
LRUThread.Start();
```

该函数可以创建一个线程，并且可以设置此线程是否为后台线程，控制线程的开始。参数为函数名，返回值为一个线程对象。

4.1.2 挂起线程

挂起进程并开始的函数如下所示：

```
if(FIFOThread.IsAlive)
    FIFOThread.Suspend();
```

首先利用 isalive（）判断线程是否在运行状态，接着利用 suspend（）来挂起线程，该函数没有参数，是线程类的成员函数，使用该函数可以挂起线程，即暂停线程的执行。

4.1.3 激活线程

对于挂起的进程，可以使用 Resume（）函数进行唤醒，该函数可以对挂起的线程进行挂起，该函数的使用如下所示：

```
FIFOThread.Resume();
```

4.1.4 让线程休眠

在多线程应用程序中，有时候并不希望某一个线程继续执行，而是希望该线程暂停一段时间，等待其他线程执行之后再继续执行。这时可以调用 Thread 类的 Sleep 方法，即让线程休眠。例如：

```
    Thread.Sleep(1000);
```

这条语句的功能是让当前线程休眠 1000 毫秒。调用 Sleep 方法的是类本身，而不是类的实例。休眠的是该语句所在的线程，而不是其他线程。

4.1.5 互斥锁

Mutex 是一个互斥的对象，同一时间只有一个线程可以拥有它，该类还可用于进程间同步的同步基元。如果当前有一个线程拥有它，在没有释放之前，其它线程是没有权利拥有它的。可以把 Mutex 看作洗手间，

上厕所的人看作线程；上厕所的人先进洗手间，拥有使用权，上完厕所之后出来，把洗手间释放，其他人才可以使用。

线程使用 Mutex.WaitOne()方法等待 Mutex 对象被释放，如果它等待的 Mutex 对象被释放了，它就自动拥有这个对象，直到它调用 Mutex.ReleaseMutex()方法释放这个对象，而在此期间，其他想要获取这个 Mutex 对象的线程都只有等待。

```
Mutex mutex= new Mutex();
 mutex.WaitOne();
 mutex.ReleaseMutex();
```

4.2 核心算法设计

本程序主要的算法就是页面置换算法，每个算法都采取适合的数据结构来实现。

4.2.1 FIFO 算法设计

用一个队列记录内存块中的页面，由于队列的特性是先进先出，所以队头的页面是最先进入的，呆的时间最久，每次在置换的时候只需要置换队头的页面，让队头的页面出队，新来的页面入队，插入到队尾。

在内存块初始化后，取出页面访问序列队列的队头。首先判断内存块中是否已经存在该队头页面，如果存在则直接显示内存块当前情况；否则，判断此时内存是否已满。如果内存未满，循环遍历找出空闲内存块，进行页面置换；若内存已满，置换内存块队列的队头页面，缺页数加 1，如此循环迭代，直到页面访问序列队列为空时，整个算法执行完毕。最后计算并显示缺页率和总的访问时间。其流程图如下：

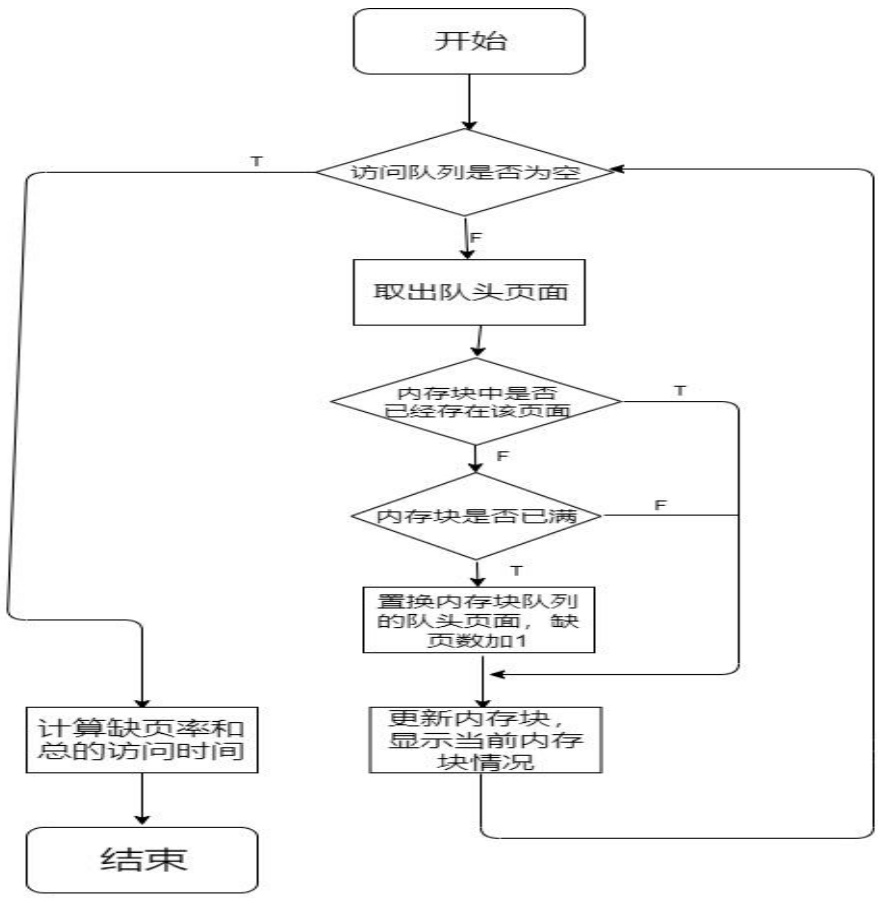

图 4.1 FIFO 算法流程图

4.2.2 LRU 算法设计

对于 LRU 算法，由于 C#中没有 map 这种数据结构，但是 C#中有 Directory 这种数据结构，其用法和 map 类似，声明一个 Directory 用来记录每个页面出现的时间，完后每次淘汰最近最久未使用，即淘汰记录的时间值（出现的序号）最小的页面。

内存块初始化后，取出页面访问序列队列的队头。首先判断内存块中是否已经存在该队头页面，如果存在则直接显示内存块当前情况，并且更新页面出现的时间值；否则，判断此时内存是否已满。如果内存未满，循环遍历找出空闲内存块，进行页面置换，并且更新页面出现的时间值；若内存已满，则比较当前内存中的页面出现的时间值，选出时间值最小的，即最近最久未使用的页面，置换该页面，缺页数加 1，如此循环迭代，直到页面访问序列队列为空时，整个算法执行完毕。最后计算并显示缺页率和总的访问时间。其流程图如下：

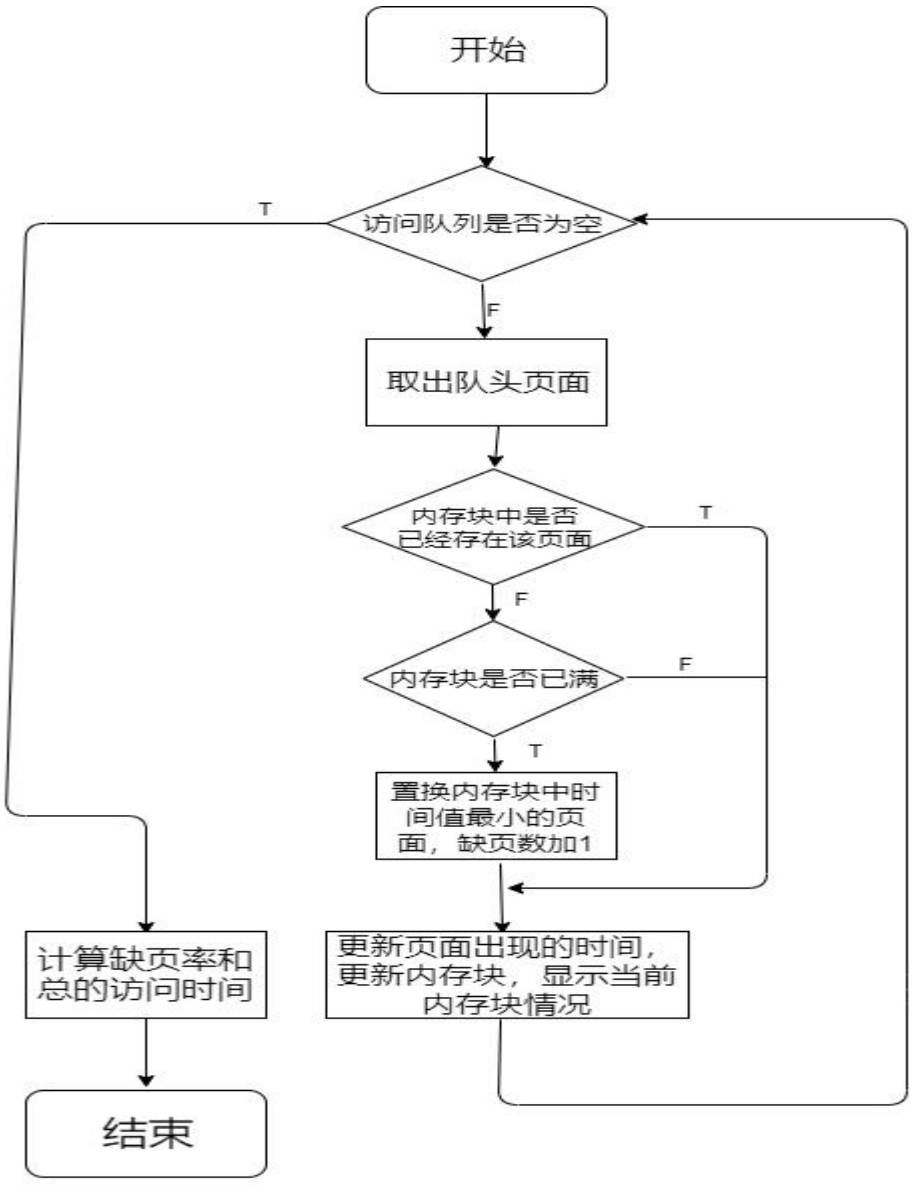

图 4.2 LRU 算法流程图

4.2.3 OPT 算法设计

OPT 算法的设计思想和 LRU 类似，LRU 是向前看，淘汰时间最小的页面，而 OPT 是向后看，淘汰最后出现或者将来没有出现的页面。在 OPT 算法中也声明一个

Directory 记录页面出现的时间，当需要发生页面置换的时候，向后遍历还未读入的页面，记录内存中存在的页面在将来出现的时间值，完后比较选出时间值最大的或者将来没出现的进行淘汰。

内存块初始化后，取出页面访问序列队列的队头。首先判断内存块中是否已经存在该队头页面，如果存在则直接显示内存块当前情况；否则，判断此时内存是否已满。如果内存未满，循环遍历找出空闲内存块，进行页面置换；若内存已满，则遍历之后要出现的界面，记录内存块中存在的页面将来出现的时间，置换出现时间最晚的页面，缺页数加 1，如此循环迭代，直到页面访问序列队列为空时，整个算法执行完毕。最后计算并显示缺页率和总的访问时间。其流程图如下：

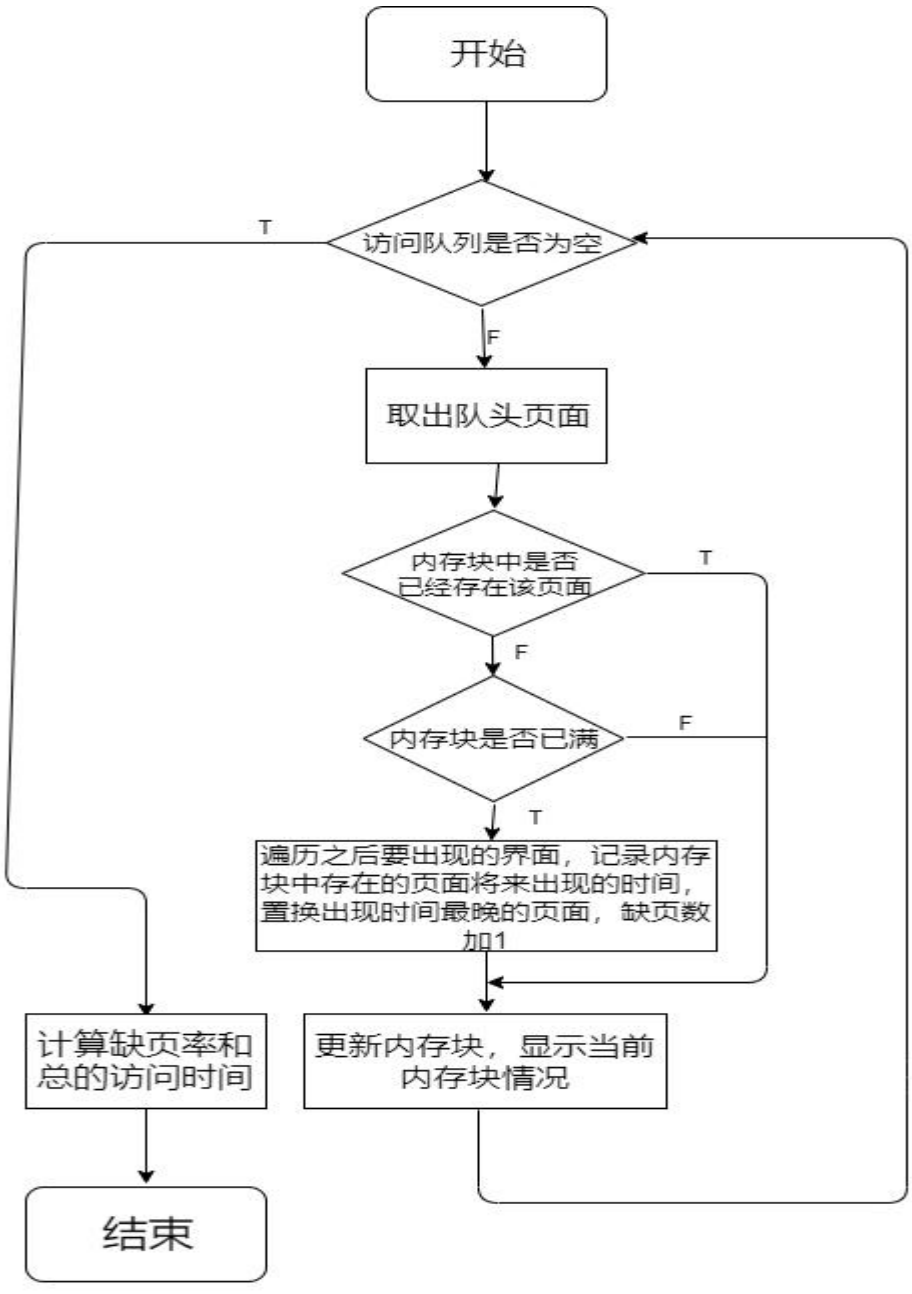

图 4.3 OPT 算法流程图

4.2.4 CLOCK 算法设计

简单 Clock 算法需要根据页面内存是否被访问来决定是否置换该页面。实际编程中，与最近最久未置换算法类似，用整型数组来表示当前每个内存页面是否被访问，其中 1 代表被访问过，0 代表未访问过。每次置换，指针循环遍历，找出第一个访问位不为 1 的那个内存页面。并且在找到被置换页面之前，将所经过的所有页面内存对应的访问位置 0.

在内存块初始化后，取出页面访问序列队列的队头。首先判断内存块中是否已经存在该队头页面，如果存在则直接显示内存块当前情况，相应访问位置 1，指针循环下移；若不存在，循环遍历内存块，找出第一个访问位不为 1 的那个内存页面。并且在找到被置换页面之前，将所经过的所有页面内存对应的访问位置置 0. 如果找到的内存页面不为空闲位，则将缺页数加 1. 如此循环迭代，直到页面访问序列队列为空时，整个算法执行完毕。最后计算并显示缺页率。其流程图如图所示：

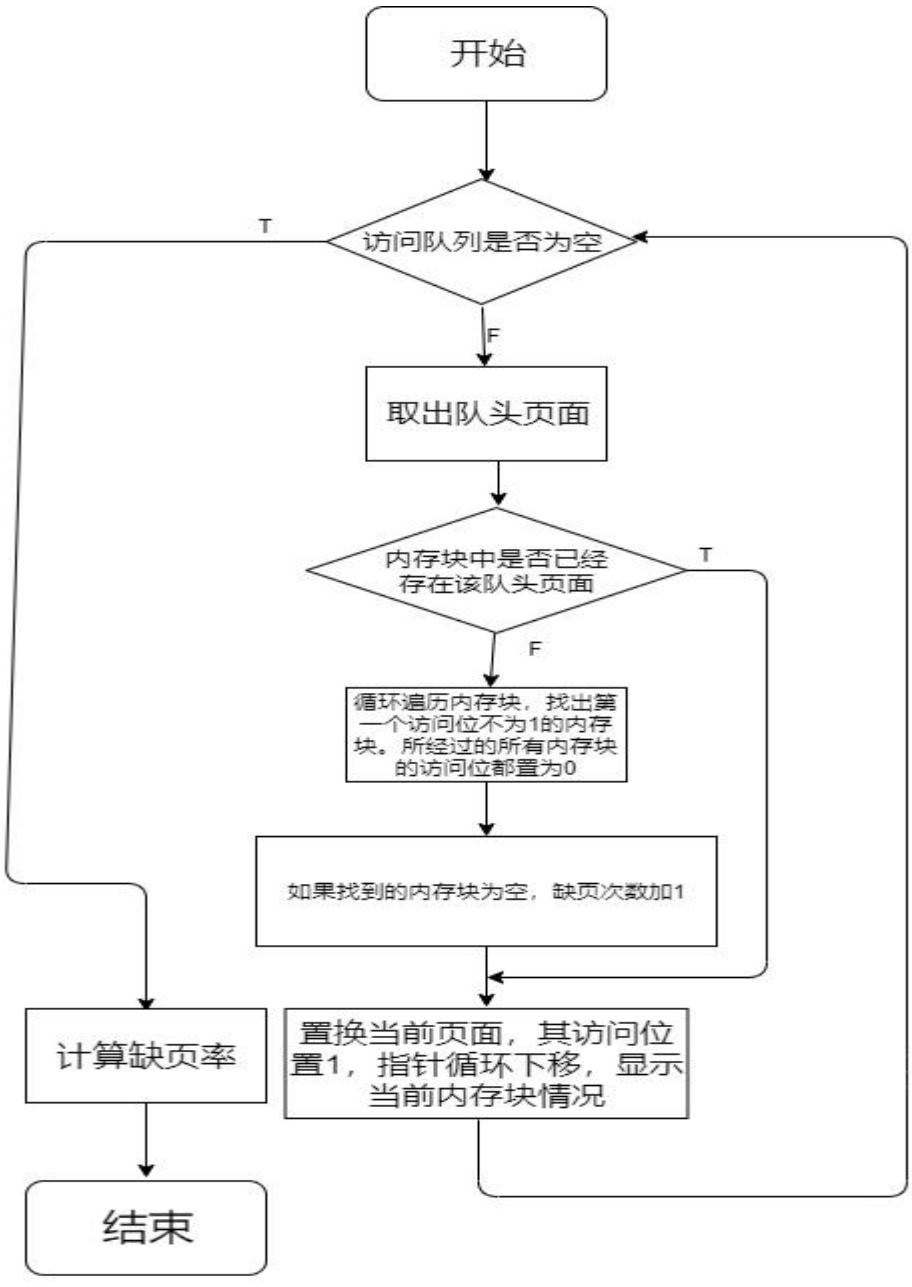

图 4.4 CLOCK 算法流程图

4.3 界面设计

对于本项目的界面设计采取 C#中的 winform 进行界面设计，C#中封装好了很多的控件，可以直接使用，在设计界面时充分考虑用户的感受，设计的界面对于用户很友好，方便用户的使用，用户可以在很短时间内学会使用该项目。但是由于在美工方面没有专业的学习，界面的美化程度还有待提高，这也是在将来的学习过程中需要改进的地方。

图 4.5 界面设计

4.4 数据结构设计以及变量说明

由于本项目采用 C#进行编写，C#中并不像 C++对很多数据结构进行一个很好的封装，很多数据结构需要手动自己进行设计来实现。本项目中主要使用的数据结构就是数组，利用数组来保存各类信息，例如：逻辑地址、物理地址等。

另外主要使用的数据结构就是 C#中的 Dictionary,Dictionary 里面的每一个元素都是一个键值对(由二个元素组成：键和值)，键必须是唯一的,而值不需要唯一的，键和值都可以是任何类型(比如：string, int, 自定义类型，等等)， 通过一个键读取一个值的时间是接近 O(1)。

本项目主要使用 Directory 来记录页面出现的时间，在 LRU 页面置换算法和 OPT 页面置换算法中有使用，用来记录页面出现的时间，来淘汰最近最久未使用的页面或者将来最远使用的页面。

本项目使用的主要变量及其说明如下表所示：

表 4.1 项目主要变量说明

变量名	变量类型	说明
logicalAddress	String 类型的数组	输入的逻辑地址
visitPages	Int 类型的数组	访问的页面序列
visitNum	Int	需要访问的页面数
memBlockNum	Int	物理块数量
memoryAccessTime	Int	访问一次内存的时间
FasttableAccessTime	Int	访问快表时间
MissingpagesTime	Int	缺页中断时间
OPTphysicalAddress	Int 类型的数组	OPT 得到的物理块号
FIFOphysicalAddress	Int 类型的数组	FIFO 得到的物理块号
LRUphysicalAddress	Int 类型的数组	LRU 得到的物理块号
pageTable	Int 类型的数组	页表
withinPageaddress	String 类型的数组	页内地址
FIFOThread	Thread	FIFO 线程
LRUThread	Thread	LRU 线程
OPTThread	Thread	OPT 线程

5 编码设计

5.1 开发环境

1、操作系统：windows10

2、编程语言：C#

3、开发语言版本：.NET Framework 4.6.1

5.2 程序设计注意的事项

在编程时考虑程序的健壮性、鲁棒性、可维护性等，需要注意以下几点：

1、代码的执行效率，需要运行多长时间。

2、要求的最大等待响应时间能否满足。

3、并发吞吐能力如何。

4、运行的稳定性和各种边界、异常处理是否考虑到了。

5、上线后，出现 Bug，相关的监控、日志能否帮助快速定位。

6、编码是否规范，注意编码的规范，例如：变量命名规范等。

5.3 主要的程序代码设计及注释

本程序的核心代码部分是三个算法的设计，针对这三个算法进行了精心的设计，力求设计出最优的算法，实现高效的算法，部分算法核心代码如下：

FIFO 核心代码：

```
        public void FIFO()
        {
            for(int i = 0; i < memBlockNum; i++)
            {
            int totalSleep = 0;
            Queue<int> q=new Queue<int>();
            FIFOphysicalAddress = new int[visitNum];
            int[] memBlock = new int[memBlockNum];
            for (int i = 0; i < memBlockNum; i++)
                memBlock[i] = -1;
            int missingPageNum = 0;
            int sleepTime = 0;
            int num = 0;
            for(int i = 0; i < visitNum; i++)
            {
                sleepTime = 0;
                bool flag = false;
                    for(int j= 0; j < memBlockNum; j++)
                    {
                        if(visitPages[i]==memBlock[j])
                        {
                            flag = true;
                            break;
                        }
                    }
                    if (!flag)
                    {
                        if (missingPageNum < memBlockNum)
                    {
                        memBlock[num] = visitPages[i];
                        FIFOphysicalAddress[i] = num;
                        num++;
                        q.Enqueue(visitPages[i]);
                            missingPageNum++;
                            if (fast)
                            sleepTime = FasttableAccessTime + memoryAccessTime +
MissingpagesTime + FasttableAccessTime + memoryAccessTime;
                            else
                                sleepTime = memoryAccessTime * 3 + MissingpagesTime;
                        }
```

```
                    else
                    {
                        int page = q.Dequeue();
                        q.Enqueue(visitPages[i]);
                        for (int j = 0; j < memBlockNum; j++)
                        {
                            if (page == memBlock[j])
                            {
                                memBlock[j] = visitPages[i];
                            FIFOphysicalAddress[i] = j;
                            break;
                            }
                        }
                        missingPageNum++;
                    if (fast)
                        sleepTime = FasttableAccessTime + memoryAccessTime +
MissingpagesTime + FasttableAccessTime
                                + memoryAccessTime;
                    else
                        sleepTime = memoryAccessTime * 3 + MissingpagesTime;
                }
                }
                else
                {
                for (int j = 0; j < memBlockNum; j++)
                {
                    if (visitPages[i] == memBlock[j])
                    {
                        FIFOphysicalAddress[i] = j;
                        break;
                    }
                }
                    if (fast)
                        sleepTime = FasttableAccessTime + memoryAccessTime;
                    else
                        sleepTime = memoryAccessTime * 2;
                }
            totalSleep += sleepTime;
            this.BeginInvoke(new Action(() =>
            {
```

```
                    FIFOLacklabel.Text = "FIFO 当前缺页率：" + missingPageNum + "/"
+ (i + 1) + "=" + ((missingPageNum * 1.0 / (i + 1)) * 100).ToString("F2") + "%";
                }));
                this.BeginInvoke(new Action(() =>
                {
                     lableFIFOsleep.Text = "FIFO 当前累计用时：" + totalSleep + "nm";
                }));
                g.Save();
                pictureBoxFIFO.Image = bp;
                Thread.Sleep(sleepTime);
            }
        }
```

LRU 页面置换算法核心代码：

```
  public void LRU()
        {
             int totalSleep = 0;
            int sleepTime = 0;
            int num = 0;
            Dictionary<int, int> map = new Dictionary<int, int>();
            LRUphysicalAddress = new int[visitNum];
            int[] memBlock = new int[memBlockNum];
            for (int i = 0; i < memBlockNum; i++)
                memBlock[i] = -1;
            int missingPageNum = 0;
            for (int i = 0; i < visitNum; i++)
            {
                LRUtableCurrentY = LRUtableY;
                bool flag = false;
                    for (int j = 0; j < memBlockNum; j++)
                    {
                        if (visitPages[i] == memBlock[j])
                        {
                            flag = true;
                            break;
                        }
                    }
                    if (!flag)
                    {
                        if (missingPageNum < memBlockNum)
                        {
```

```
                    memBlock[num] = visitPages[i];
                    LRUphysicalAddress[i] = num;
                    num++;
                        missingPageNum++;
                        map.Add(visitPages[i], i);
                    if (fast)
                        sleepTime = FasttableAccessTime + memoryAccessTime +
MissingpagesTime + FasttableAccessTime
                            + memoryAccessTime;
                    else
                        sleepTime = memoryAccessTime * 3 + MissingpagesTime;
                }
                    else
                    {
                        int mn = 999;
                        int page = -1;
                        foreach (var item in map)
                        {
                            for (int j = 0; j < memBlockNum; j++)
                                if (mn > item.Value && item.Key == memBlock[j])
                                {
                                    mn = item.Value;
                                    page = item.Key;
                                }
                        }
                        for (int j = 0; j < memBlockNum; j++)
                        {
                            if (page == memBlock[j])
                            {
                                memBlock[j] = visitPages[i];
                                LRUphysicalAddress[i] = j;
                            break;
                            }
                        }
                        missingPageNum++;
                        if (map.ContainsKey(visitPages[i]))
                        {
                            map[visitPages[i]] = i;
                        }
                        else
```

```
                        {
                            map.Add(visitPages[i], i);
                        }
                    if (fast)
                        sleepTime = FasttableAccessTime + memoryAccessTime +
MissingpagesTime + FasttableAccessTime
                            + memoryAccessTime;
                    else
                        sleepTime = memoryAccessTime * 3 + MissingpagesTime;
                }
                }
                else
                {
                    if (map.ContainsKey(visitPages[i]))
                        map[visitPages[i]] = i;
                for (int j = 0; j < memBlockNum; j++)
                {
                    if (visitPages[i] == memBlock[j])
                    {
                        LRUphysicalAddress[i] = j;
                        break;
                    }
                }
                    if (fast)
                        sleepTime = FasttableAccessTime + memoryAccessTime;
                    else
                    sleepTime = memoryAccessTime * 2;
                }
            totalSleep += sleepTime;
            this.BeginInvoke(new Action(() =>
            {
                LRULacklabel.Text = "LRU 当前缺页率：" + missingPageNum + "/" +
(i + 1) + "=" + ((missingPageNum * 1.0 / (i + 1)) * 100).ToString("F2") + "%";
            }))
            this.BeginInvoke(new Action(() =>
            {
                lableLRUsleep.Text = "LRU 当前累计用时：" + totalSleep + "nm";
            }));
            Thread.Sleep(sleepTime);
        }
```

```
        }
```

OPT 算法核心代码：

```
 public void OPT() {
           int totalSleep = 0;
           int sleepTime = 0;
           int num = 0;
           Dictionary<int, int> map = new Dictionary<int, int>();
           int[] memBlock = new int[memBlockNum];
           for (int i = 0; i < memBlockNum; i++)
               memBlock[i] = -1;
           int missingPageNum = 0;
           OPTphysicalAddress = new int[visitNum];
           for (int i = 0; i < visitNum; i++)
           {
               OPTtableCurrentY = OPTtableY;

               bool flag = false;
                   for (int j = 0; j < memBlockNum; j++)
                   {
                       if (visitPages[i] == memBlock[j])
                       {
                           flag = true;
                           break;
                       }
                   }
                   if (!flag)
                   {
                       if (missingPageNum < memBlockNum)
                       {
                           memBlock[num] = visitPages[i];
                           OPTphysicalAddress[i] = num;
                           num++;
                           missingPageNum++;
                                if (fast)
                           sleepTime = FasttableAccessTime + memoryAccessTime +
MissingpagesTime + FasttableAccessTime
                               + memoryAccessTime;
                       else
                           sleepTime = memoryAccessTime * 3 + MissingpagesTime;
                   }
```

```
                else
                {
                    map.Clear();
                    int mm = -1;
                    for (int z = 0; z < memBlockNum; z++)
                        for (int j = i; j < visitNum; j++)
                        {
                            if (memBlock[z] == visitPages[j])
                            {
                                map.Add(visitPages[j], j);
                                break;
                            }
                        }
                    int page = -1;
                    foreach (var item in map)
                    {
                        for (int j = 0; j < memBlockNum; j++)
                            if (mm < item.Value && item.Key == memBlock[j])
                            {
                                mm = item.Value;
                                page = item.Key;
                            }
                    }
                    for (int j = 0; j < memBlockNum; j++)
                    {
                        if (!map.ContainsKey(memBlock[j]))
                        {
                            page = memBlock[j];
                            break;
                        }
                    }
                    for (int j = 0; j < memBlockNum; j++)
                    {
                        if (page == memBlock[j])
                        {
                            memBlock[j] = visitPages[i];
                            OPTphysicalAddress[i] = j;
                            break;
                        }
                    }
```

```
                        missingPageNum++;
                    if (fast)
                        sleepTime = FasttableAccessTime + memoryAccessTime +
MissingpagesTime + FasttableAccessTime
                            + memoryAccessTime;
                    else
                        sleepTime = memoryAccessTime * 3 + MissingpagesTime;
                }
                }
                else
                {
                for (int j = 0; j < memBlockNum; j++)
                {
                    if (visitPages[i] == memBlock[j])
                    {
                        OPTphysicalAddress[i] = j;
                        break;
                    }
                }
                    if (fast)
                        sleepTime = FasttableAccessTime + memoryAccessTime;
                    else
                        sleepTime = memoryAccessTime * 2;
                }
            totalSleep += sleepTime;
            this.BeginInvoke(new Action(() =>
            {
                OPTLacklabel.Text = "OPT 当前缺页率：" + missingPageNum + "/" +
(i + 1) + "=" + ((missingPageNum * 1.0 / (i + 1)) * 100).ToString("F2") + "%";
            }));
            this.BeginInvoke(new Action(() =>
            {
                lableOPTsleep.Text = "OPT 当前累计用时：" + totalSleep + "nm";
            }));
            Thread.Sleep(sleepTime);
        }
```

CLOCK 页面置换算法核心代码（C++实现）：

```
int nru[N];
int page_in_block[N];
int CLOCK(){
```

```
    cout<<"*********CLOCK*********"<<endl;
    int index = 1;
    int page_lack = 0;
    memset(block, -1, sizeof(block));
    for(int i=1; i<=n; ++i){
        if(page_in_block[page[i]]){
            nru[page_in_block[page[i]]] = 1;
        }
        else {
            while(true){
                if(index > m) index = 1;
                if(block[index] == -1) {
                    nru[index] = 1;
                    page_in_block[page[i]] = index;
                    block[index++] = page[i];
                    ++page_lack;
                    break;
                }
                if(block[index] == page[i]){
                    nru[index++] = 1;
                    break;
                } else {
                    if(nru[index] == 0){
                        nru[index] = 1;
                        page_in_block[block[index]] = 0;
                        page_in_block[page[i]] = index;
                        block[index++] = page[i];
                        ++page_lack;
                        break;
                    } else
                        nru[index++] = 0;
                }
            }
        }
        for(int k=1; k<=m; ++k)
            cout<<block[k]<<" ";
        cout<<endl;
    }
    return page_lack;
}
```

5.4 解决的技术难点

本项目整体设计思路比较清晰，容易编程实现，对于本项目存在的技术难点，主要是算法的设计以及图形的动态显示。

5.4.1 算法的设计

本项目是用 C#语言进行开发的，对于页面置换算法的设计不是特别方便（C#没有像 C++封装好的数据结构），所以在设计的时候遇到了很多的困难，

5.4.2 图形的动态显示

本项目的一个难点是对于图形的动态显示，也就是动态显示一个页面置换算法的过程，这个过程要是想动态显示，就需要设计好怎么显示，显示在哪里，什么时候显示，这些都是编码阶段的一个难点，需要着重考虑。

5.5 经常犯的错误

1、算法设计考虑不当，考虑情况不全，出现一系列的问题。

2、界面设计不好，画出的东西不能很好在界面上画出。

3、代码注释较少，当代码量较多的时候修改不易。

6 测试时出现过的问题及其解决方法

6.1 算法的设计

本项目的难点在于对于算法的设计，在算法设计中遇到了很多的问题，开发初期设计的算法往往对于某些特殊情况没有能考虑到，没有考虑临界情况，这是自己的编码能力不足所导致的，这也是在之后急需提高的方面。

在编码算法前一定不要急于上手编码，要先画好算法流程图，理清算法的流程才能更容易进行编码，在之后的学习过程也要牢记这点，不要急于编码，有了大概的思路，要先把思路理清，这样才能快速编写出代码，并且对于编写的代码要进行不断的调试改进，才能最终得到最优的代码。

6.2 动态过程的显示

本项目在开发初期，采用 graphics 类在屏幕上画出页面置换的动态过程，但是此种方法有很多的问题，例如：当画的内容过大，显示不全；画面上的内容在移动的时候会消失等问题，总之，此种方法虽然能满足程序的基本要求，但是对于某些特殊情况的页面置换过程常常会发生显示不全的错误，所以开发过程中弃用了这种方法。

在开发后期采取了新的方法，通过 picturebox 来显示图片，还是用 graphics 画，但是是把图片显示在 picturebox 上面，这样图片在移动过程中就不会消失，并且针对可能出现的特殊情况，采用 C#中的控件，采用滑动条，这样就可以显示完整所有的动态过程。

6.3 文件创建和保存

对于文件的创建，由于平时对这方面的使用较少，在使用的时候对于一些路径的使用出现了很多的错误，使用的时候经常会报错，例如：地址不存在的问题，或者文件名不符合要求的错误，从这也可以看出

自身基本功的欠缺，不扎实，在学文件这块没有多加练习。

最后在查阅了相关的资料进行改进，能够正确进行文件的创建以及存储和读取。

7 软件使用说明

7.1 基本功能

1、各类页面置换算法，包括：LRU、OPT、FIFO、LFU，采用多种不同的算法分别完成各类页面置换算法，并且最终选择出最高效的页面置换算法。

2、建立多个线程，每个线程执行一个页面置换算法，并且在屏幕上可以同时显示不同的算法的执行结果，设计显示方式使得对于不同算法的执行结果有一个直观的感受。

3、可以输入也可以随机产生逻辑地址访问序列，自动转换为逻辑页号，产生内存页号，分别由页面置换算法完成页面置换。

4、本项目能够设定驻留内存页面的个数、内存的存取时间、缺页中断的时间、快表的时间，并提供合理省缺值，可以暂停和继续系统的执行。

5、本项目能够设定逻辑地址访问序列中地址的个数和地址的范围，输入非法数据会提示输入错误，只有输入正确的符合要求的数据才能进行页面置换。本项目能够设定有快表和没有快表的两种运行模式。

6、提供良好图形界面，同时能够展示每个个算法当前运行的情况和运行的结果。

7、给出每种页面置换算法每次每个页面的存取时间、每个逻辑地址对应的物理页号和内存地址。

8、为了方便对实验数据进行分析，本项目能够将每次的实验输入和实验结果存储起来，并且以图片的形式把实验过程存储起来，随时可查询，方便用户对实验数据分析。

9、完成多次不同设置的实验，总结实验数据，分析实验数据，采用柱状图显示实验数据。

7.2 需要运行的环境

本项目采用 C#进行开发，采用 C#中的 winform 开发，需要运行的环境为配置了.net framework 的计算机。经过测试在 windows 系统和 linux 系统都可以正常运行。

7.3 使用步骤

本项目使用起来非常简单，容易上手，只需要点击 exe 文件即可进入主界面进行运行，在主界面可以进行参数的设置，来改变程序中的一系列参数，在控制台可以控制各个页面置换算法的线程的开始、暂停、继续。在程序运行区动态显示程序运行过程，在输入区可以输入访问地址序列或者随机生成地址序列。

图 7.1 软件使用说明图

8 总结

8.1 完成的部分

对于本项目，要求的所有功能都已经完全完成，其中包括：各类页面置换算法，包括：LRU、OPT、FIFO、LFU，采用多种不同的算法分别完成各类页面置换算法。建立多个线程，每个线程执行一个页面置换算法，并且在屏幕上可以同时显示不同的算法的执行结果，动态显示方式使得对于不同算法的执行结果有一个直观的感受。可以输入也可以随机产生逻辑地址访问序列，自动转换为逻辑页号，产生内存页号，分别由页面置换算法完成页面置换。本项目能够设定驻留内存页面的个数、内存的存取时间、缺页中断的时间、快表的时间，并提供合理省缺值，可以暂停和继续系统的执行。本项目能够设定逻辑地址访问序列中地址的个数和地址的范围，输入非法数据会提示输入错误，只有输入正确的符合要求的数据才能进行页面置换。本项目能够设定有快表和没有快表的两种运行模式。本项目提供良好图形界面，同时能够展示每个个算法当前运行的情况和运行的结果。给出了每种页面置换算法每次每个页面的存取时间、每个逻辑地址对应的物理页号和内存地址。为了方便对实验数据进行分析，本项目能够将每次的实验输入和实验结果存储起来，并且以图片的形式把实验过程存储起来，随时可查询，方便用户对实验数据分析。完成多次不同设置的实验，总结实验数据，分析实验数据，采用柱状图显示实验数据。

8.2 未完成的部分

由于本项目的开发时间有限，本项目的很多功能还不够完善，例如对于某些特殊的情况，程序可能会出 bug，在测试程序的时候时间有限，没能完全测试到大多数情况，另外还可能会在执行时出错。在之后的学习过程中，将在以下几个方面改进项目：

1、界面的美化，设计改进界面，考虑用户感受，增强用户对于界面的友好性。

2、原有功能的完善，目前本项目的有些功能还会出现一些小的问题，需要在之后进行进一步的完善。

3、新功能的开发，开发新的功能，例如研究更多的页面置换算法。

8.3 创新功能

8.3.1 动态显示页面置换过程

动态显示了每个页面置换过程中内存页块的变化，每个页面还可以动态的实时显示缺页率和访问时间的变化。动态显示过程的变化，对于用户比较每个算法的不同有一个直观的体验，并且方便用户对于不同的算法进行比较。

8.3.2 实现了多种页面置换算法

课设基本要求是实现两个页面置换算法，但是本项目实现了四种页面置换算法，其中包括 FIFO、LRU、OPT、CLOCK。

8.3.3 分目录保存了各个算法的执行结果，图片形式保存过程

本项目对于实验数据的保存进行了一个保存，不同的算法的实验结果放在不同的目录下的文件中，在保存的 txt 文件中记录了实验数据，其中包括实验时间，实验中的逻辑地址和物理地址，还记录了实验的缺页率和总的访问时间。

另外，本项目对每次实验的实验过程以图片的形式进行了一个存储，每个图片的名字为实验时间，方便用户进行查找，图片也以一种生动的方式展示了实验的过程和结果。

8.3.4 柱状图分析实验数据

对于实验数据，本项目可以读取数据进行分析，并采用柱状图的方式更生动的显示实验数据，方便用户对实验数据进行一个很好的对比。

8.3.5 嵌入控制台程序执行多种算法

本项目还嵌入了一个 C++语言编写的 cpp 程序，在控制台模拟了多种页面置换算法，其中包括 FIFO、LRU、OPT、CLOCK，其中 CLOCK 算法是一个创新。

8.4 团队合作情况

本项目团队只有一人，所以全部工作一人完成，一人完成整个项目工作量还是很大的，但是一人成组也是很好地锻炼了自己各方面的能力，尤其是对于自己编程能力的提高很有帮助。

8.5 收获和经验教训

通过本次实验了解到了很多相关的知识，对于页面置换算法有了更加深刻的理解，通过本次课设，对于自己编程能力有了一个很大的提高，本项目代码大概有一千多行，通过本次项目，自己编写的代码量也有了很大的提升，对于操作系统课程也有了更加深入的研究和理解，加深了对操作系统认识，了解了操作系统各种资源分配算法的实现，特别是对于虚拟存储和页面置换有了一定的了解。

本次课设是一人成组，整个代码量以及工作量都是很大的，对自己是一个很大的挑战，最终还是经过自己的努力完成了这次课设，完成后有很大的成就感。

通过本次课设发现自己有很大的不足，自己平时动手少，导致编码能力低，在之后的学习过程中需要多动手，提升自己的编程能力。

9 参考文献

[1] 谢旭东,朱明华.操作系统教程[M].北京:机械工业出版社,2012:145-153.
[2] 孙钟秀.操作系统教程(第 4 版)[M].北京:高等教育出版社,2008:258-275.
[3] 阳慧.LRU 算法的研究及实现[J].计算机时代, 2004 (2) :28.
[4] 屠立德, 屠祁.操作系统基础[M].北京:清华大学出版社, 2000.
[5] 汤子瀛.计算机操作系统[M].西安:西安电子科技大学出版社, 2003.