**广州大学学生实验报告**

开课学院及实验室: 计算机科学与网络工程学院 2024年 5 月 17 日

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| 学院 | 计算机科学与网络工程学院 | 年级/专业/班 |  | 姓名 |  | 学号 |  |
| 实验课程名称 | 操作系统实验 | | | | | 成绩 |  |
| 实验项目名称 | 内存管理 | | | | | 指导老师 |  |

**实验三 内存管理**

1. **实验目的**

1、了解虚拟存储技术的特点，掌握虚拟存储请求页式存储管理中几种基本页面置换算法的基本思想和实现过程，并比较它们的效率。

2、了解程序设计技术和内存泄露的原因

**二、基本要求**

**1、画出每个页面置换算法流程图；**

**2、对算法所用的数据结构进行说明；**

**3、测试数据随机产生。不可手工输入；**

**4、编写程序并调试；**

**5、多次测试程序，截屏输出实验结果；**

**6、根据实验结果与理论课讲述的原理进行实验分析。**

**三、实验软件**

Visual studio C++2022

**四、实验内容：**

**随机生成随机指令序列并进行调试:**

const int N = 320;//指令集长度

int Instructions[N];//指令集

//生成指令序列

void Init\_instructions()

{

    srand(time(NULL));//设置随机数种子

    int i = 0;

    while (i<N)

    {

        int x = rand() % N;

        while (x > N - 2)//生成随机数，并防止越界

        {

            x = rand() % N;

        }

        if(i<N)Instructions[i++] = x + 1; //加入指令集

        int y = rand() % (x + 1);

        while (y > N - 3)//生成随机数，并防止越界

        {

            y = rand() % (x + 1);

        }

        if(i<N)Instructions[i++] = y;//加入指令集

        if(i<N)Instructions[i++] = y + 1;//加入指令集

        int z = rand() % (N - (y + 2)) + y + 2;

        if(i<N)Instructions[i++] = z;//加入指令集

    }

}

int main()

{

    Init\_instructions();

    for (int i = 0; i < N; ++i)

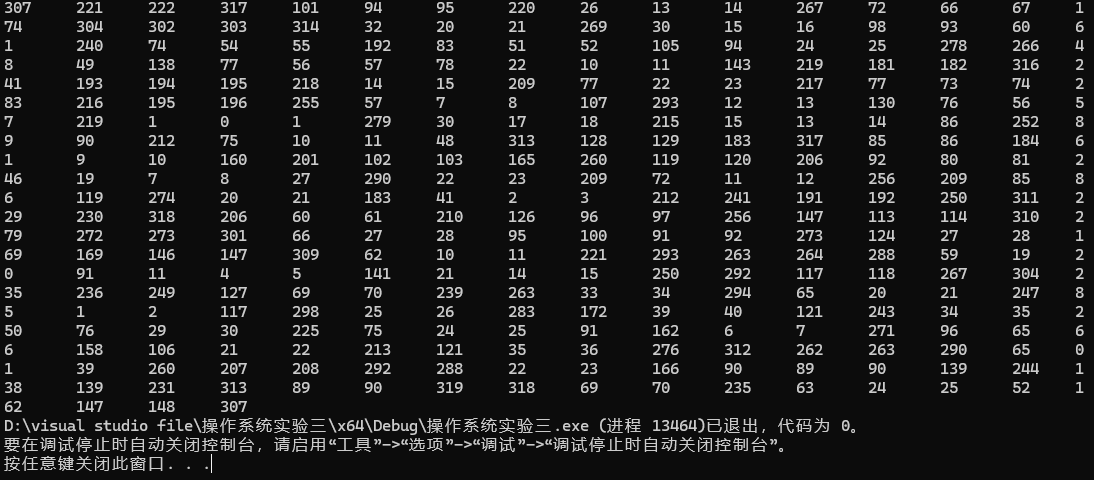
    {

        cout << Instructions[i] << "\t";

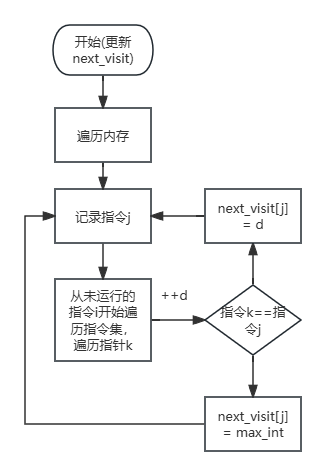
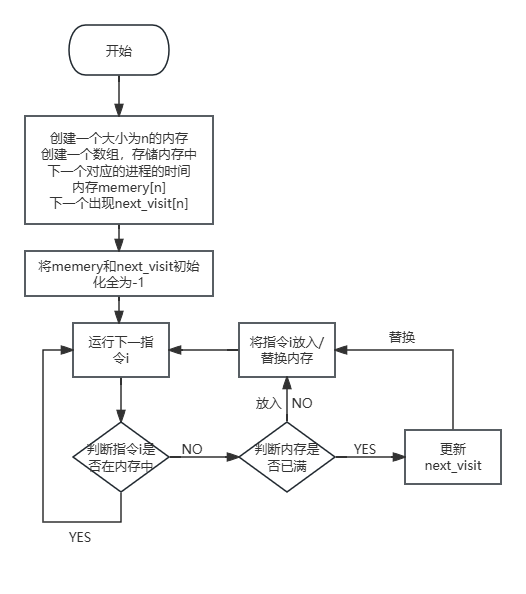
    }

}

**调试结果:**

****

**模拟实现请求页式存储管理的几种基本页面置换算法**

**（1）最佳淘汰算法（OPT）1.算法流程图:**

**2.数据结构说明:**

const int N = 320;//指令集长度

int Instructions[N];//指令集序列

int n;//内存大小

int memery[n];//内存

int next\_visit[n];//下一个相同进程出现的时间

const int M = 10;//每k存放10条指令

**Instructions是要运行的指令序列，由Init\_instructions()随机生成/初始化。**

**next\_visit是内存memery中位置i对应指令中下一条指令i在序列中出现的时间。**

**3.编写程序:**

**核心代码:**

float OPT(int n)

{

    //初始化

    double count = 0;

    int\* memery;

    memery = new int[n];

    int\* next\_visit;

    next\_visit = new int[n];

    for (int i = 0; i < n; ++i)

    {

        memery[i] = -1;

        next\_visit[i] = -1;

    }

    for(int i = 0;i<N;++i)//运行指令i

    {

        //若i所在内存页不在内存中

        if (!i\_in\_memery(i, n, memery))

        {

            //内存已满

            if (memery\_full(n, memery))

            {

                //更新next\_visit

                //遍历内存

                for (int j = 0; j < n; ++j)

                {

                    int k = i;

                    int d = 1;

                    while (k < N)

                    {

                        if (memery[j] == Instructions[k]/M)

                        {

                            next\_visit[j] = d;

                            break;

                        }

                        else next\_visit[j] = INT32\_MAX;

                        ++d;

                        ++k;

                    }

                }

                //找出next\_visit最大值的位置

                memery[find\_max\_next\_visit(n, next\_visit)] = Instructions[i]/M;

            }

            else//内存未满

            {

                //找出第一个未满的位置

                int j = 0;

                while (memery[j] != -1)++j;

                memery[j] = Instructions[i]/M;//将指令所在页放入内存

            }

        }

        else ++count;//i所在内存页在内存中

    }

    return count;

}

**函数返回类型为float，返回内存命中次数count**

**判断指令i在内存中:**

bool i\_in\_memery(int i,int n,int memery[])//判断指令i所在页是否在内存中

{

    for (int j = 0; j < n; ++j)

        if (Instructions[i]/M == memery[j])

            return true;//在内存中返回true

    return false;//不在内存中返回false

}

**判断内存是否已满:**

bool memery\_full(int n,int memery[])

{

    for (int i = 0; i < n; ++i)

        if (memery[i] == -1)

            return false;//内存未满

    return true;//内存已满

}

**查找next\_visit的最大值，并返回最大值在next\_visit中的最大值:**

int find\_max\_next\_visit(int n,int next\_visit[])

{

    int k = 0;

    int max = 0;

    for (int i = 0; i < n; ++i)

    {

        if (next\_visit[i] > max)

        {

            k = i;

            max = next\_visit[i];

        }

    }

    return k;

}

**主函数:**

int main()

{

    Init\_instructions();

    for(int n = 4;n<33;++n)

    {

        double p = OPT(n);

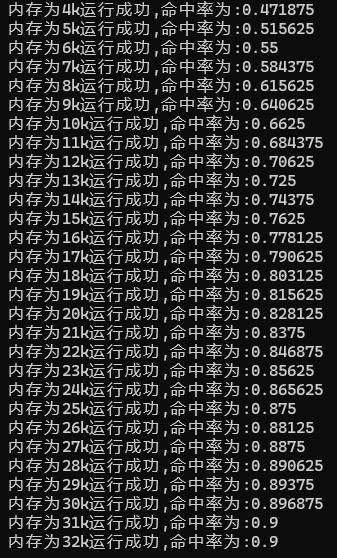
        double p0 = p / N;

        cout << "内存为" << n << "k运行成功," <<"命中率为:"<<p0<< endl;

    }

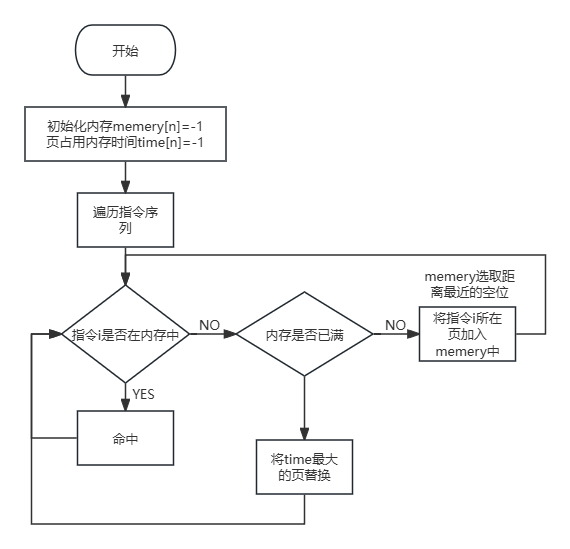
}

**4.调试结果:**

****

**（2）先进先出的算法（FIFO）**

**1.算法流程图:**

****

**2.数据结构说明:**

int memery[n];//内存

int time[n];//某页在内存中存在的时间

const int M;//每页的指令数

const int N;//指令集的大小

Instructions[N];//指令集

**指令集Instructions[N]由Init\_instructions()随机生成**

**3.编写程序:**

**核心代码:**

void FIFO(int n)

{

    double count = 0;//命中次数

    int\* memery = new int[n];//内存

    int\* time = new int[n];//页在内存中的时间

    for (int i = 0; i < n; ++i)

    {

        time[i] = -1;

        memery[i] = -1;

    }

    //运行指令

    for (int i = 0; i < N; ++i)

    {

        //指令所在页在内存中

        if (instruction\_in\_memery(n, i, memery))

        {

            ++count;//命中

            //刷新时间

            upgrade\_time(n,time);

        }

        //指令所在页不在内存中

        else

        {

            //内存已满

            if (memery\_is\_full(n, memery))

            {

                //寻找time最大的页，替换

                int j = 0;

                int max\_time = 0;

                for (int k = 0; k < n; ++k)

                {

                    if (max\_time < time[k])

                    {

                        max\_time = time[k];

                        j = k;

                    }

                }

                memery[j] = Instructions[i] / M;

                time[j] = 0;

                upgrade\_time(n, time);

            }

            //内存未满

            else

            {

                //寻找内存中顺序第一个空位

                int j = 0;

                while (memery[j] != -1)++j;

                memery[j] = Instructions[i] / M;

                time[j] = 0;//将指令放入内存

                upgrade\_time(n, time);

            }

        }

    }

    cout << "内存为" << n << "k运行成功," << "命中率为:" << count/N << endl;

}

**判断某指令对应的某页是否在内存中:**

bool instruction\_in\_memery(int n,int i,int memery[])

{

    for (int j = 0; j < n; ++j)

    {

        if (Instructions[i] / M == memery[j])

            //一个页可存储M条指令0....319,Instructions[i]/M->页号

            return true;

    }

    return false;

}

**判断内存是否已满,满则返回true，否则返回false:**

bool memery\_is\_full(int n,int memery[])

{

    for (int i = 0; i < n; ++i)

        if (memery[i] == -1)return false;

    return true;

}

**刷新时间:**

void upgrade\_time(int n,int time[])

{

    for (int i = 0; i < n; ++i)

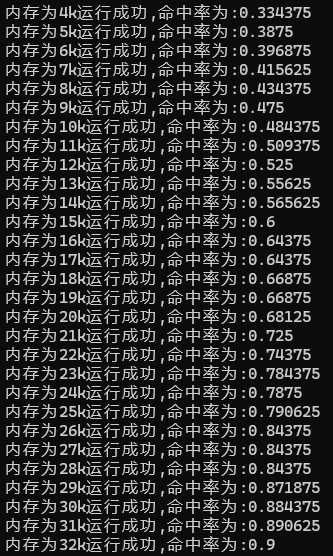
    {

        if (time[i] != -1)++time[i];

    }

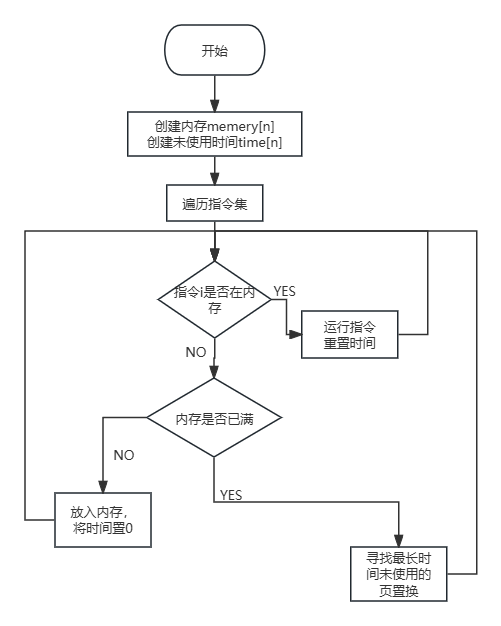
}

**4.调试结果:**

****

**（3）最近最久未使用算法（LRU）**

**1.算法流程图:**

****

**2.数据结构说明:**

int memery[n];//内存

int time[n];//存储某页未使用的时间

int count;//统计内存命中次数

Instructions[N];//指令集

**time[n],初始化全置为-1，当memery在i位置加入指令后，time[i] = 0,若指令命中memery[i]，则重新将time[i]置为0；**

**3.编写程序:**

**核心代码:**

void LRU(int n)

{

    //初始化

    float count = 0;

    int\* memery = new int[n];

    int\* time = new int[n];

    for (int i = 0; i < n; ++i)

    {

        memery[i] = -1;

        time[i] = -1;

    }

    //遍历指令集

    for (int i = 0; i < N; ++i)

    {

        //指令i在内存中

        if (i\_in\_memery(i, n, memery))

        {

            //指令使用

            //将时间置为0

            //查找指令在memery中的位置

            int j = 0;

            while (memery[j]!=Instructions[i]/10)++j;

            time[j] = 0;

            ++count;

            upgrade\_time(n,time);

        }

        //指令i不在内存中

        else

        {

            //内存已满

            if (memery\_is\_full(n, memery))

            {

                //找出time中的最大值

                int j = 0;

                int max = 0;

                int max\_time = 0;

                while (j<n)

                {

                    if (max\_time < time[j])

                    {

                        max\_time = time[j];

                        max = j;

                    }

                    ++j;

                }

                memery[max] = Instructions[i] / 10;//将页替换内存

                time[max] = 0;//重置时间

                upgrade\_time(n, time);

            }

            //内存未满

            else

            {

                //寻找内存中最前面的空位，将指令放入内存

                int j = 0;

                while (memery[j] != -1)++j;

                memery[j] = Instructions[i] / M;

                //将时间置为0

                time[j] = 0;

                //刷新时间

                upgrade\_time(n, time);//++time

            }

        }

    }

    cout << "内存为" << n << "k运行成功," << "命中率为:" << count / N << endl;

}

**判断指令i是否在内存中:**

bool instruction\_in\_memery(int n,int i,int memery[])

{

    for (int j = 0; j < n; ++j)

    {

        if (Instructions[i] / M == memery[j])

            //一个页可存储M条指令0....319,Instructions[i]/M->页号

            return true;

    }

    return false;

}

**判断内存是否已满:**

bool memery\_is\_full(int n,int memery[])

{

    for (int i = 0; i < n; ++i)

        if (memery[i] == -1)return false;

    return true;

}

**刷新时间:**

void upgrade\_time(int n,int time[])

{

    for (int i = 0; i < n; ++i)

    {

        if (time[i] != -1)++time[i];

    }

}

**主函数:**

int main()

{

    Init\_instructions();

    for (int i = 4; i < 33; ++i)

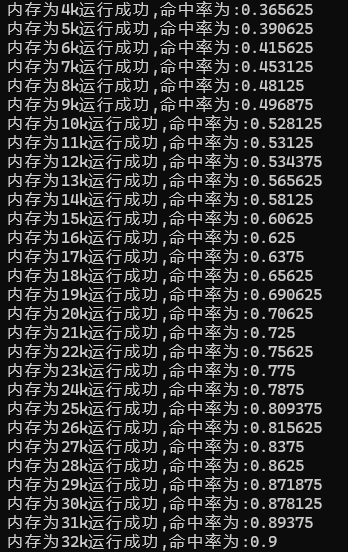
    {

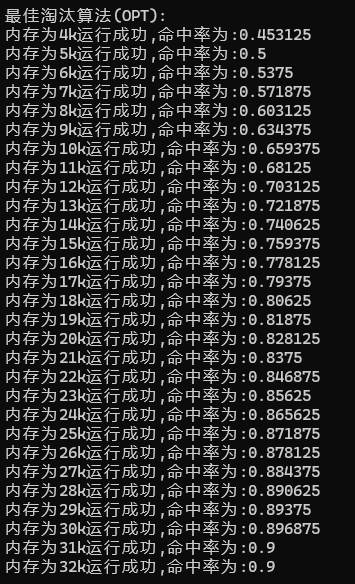
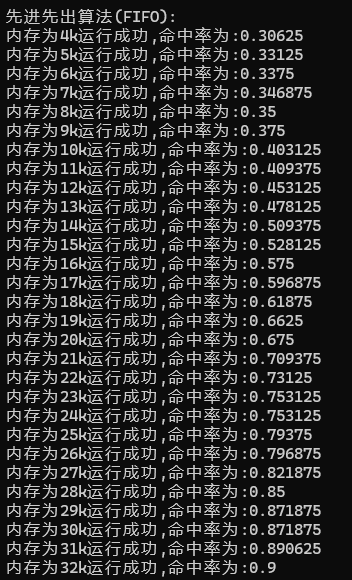
        LRU(i);

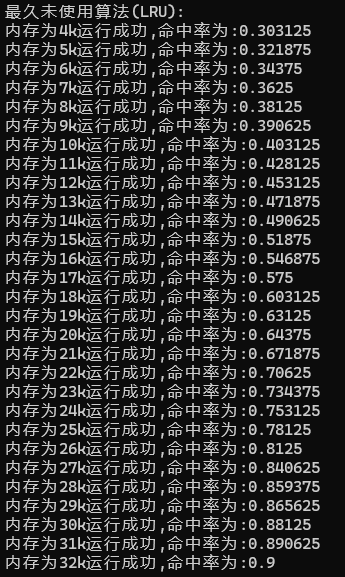
    }

}

**4.调试结果:**

****

**多次调试:**

****

**五、实验思考**

**1、从几种算法的命中率看，哪个算法最高？哪个算法最低？对每个页面的执行结果进行分析。**

**答:**

**从实验结果来看，三种算法在内存足够大时命中率相差不大，接近90%。但在内存较小时，例如4K、5K时，OPT算法命中率高达50%，FIFO算法命中率约为35%，LRU算法命中率约为36%。可见OPT算法命中率最高，其次是LRU算法，最后是FIFO算法。**

**从理论上来看，OPT算法对后续需要执行的指令进行了预测，是一种理想状态，因此命中率较高。FIFO使用了先进先出的替换策略，而LRU使用了最少使用的替换策略，使用了一定的统计规律进行预测，因而LRU算法要优于FIFO算法。**

**2、OPT算法在执行过程中可能会发生错误，为什么？**

**答:**

**OPT算法是对未来的进行预测，但这个是一个理想状态，通常现实中后面的页面可能会发生改变，所以导致算法预测的结果可能不一样，可变系数大，除此之外进程也会不一样。因此在执行过程中就容易发生错误。**