**《操作系统》结课报告**

莫伟彬

2014010341

本报告主要围绕如何编程显示页面置换算法中的页面变化和缺页次数来展开。我们都知道，操作系统在处理缺页中断时有可能要用到页面置换算法。常见的页面置换算法主要有：OPT（最优页面置换）算法、LRU（最近最久未使用）算法、LFU（最不常用）算法、FIFO（先进先出）算法、Clock（时钟页面置换）算法。由于最终目的并非真正地访问页面，而是显示页面变化和缺页次数，所以不考虑Clock算法。

时隔一个学期，再拿起C语言这把利器的时候有些力不从心，在编程过程中不免存在很多考虑不周的地方，由于时间有限，来不及加以改进，还望见谅。（详细代码请见同时上交的TXT文件或源代码）

在算法的实现过程中

1. 为方便编程，基本所有数组均从第二个元素开始使用（即从a[1]开始使用，舍弃a[0]）；
2. 都采用了先接收逻辑页面序列（以下称逻辑列）后再填入物理页面的方法（方便打印结果）；
3. 能处理的逻辑页面列长度为99（当然这是可以通过改代码修改的）；
4. 每种算法均使用整型数据dis记录缺页次数，使用整型数据wlpagenum记录物理页面个数。

**（一）、OPT算法**

首先来看OPT算法（置换需要等待时间最长的页面）。接收逻辑页面列的为含有num（页面号）、amount（逻辑页面列里该页面号出现的次数）、xy[]（该页面号在逻辑页面列中的位置）三个变量的结构体数组ljpage[]。Ljpage[]中的每个元素记录着一个逻辑页面号，这个逻辑页面号的个数和每个相同逻辑页面所在的位置。

struct ljpagerange

{

int num;

int amount;

int xy[100];

};ljpagerange ljpage[100];

但这显然是不够的，因为我们难以直接根据xy[]（下文将它称为坐标）的值来判断逻辑页面的输入顺序，所以我们还需要一个数组来记录逻辑页面的输入顺序，因此创建了一个整型数组fuzhu[]。这个整型数组储存的并不是逻辑页面号，而是这个逻辑页面号对应的ljpage[i]的i。这将有利于fuzhu[]和ljpage[]建立联系。同时，我还设置了记录不同逻辑页面的总数的整型变量ljamount（其实其作用可用dis替代，但使用ljamount会比使用dis更方便与代码理解，故保留使用ljamount的方法）。

在将逻辑页面填入物理页面时，还要考虑不同的逻辑页面数ljamount和wlpagenum的大小关系。故分两种情况：

第一种：

if(wlpagenum > ljamount)

{

dis = 1;

page[1] = 1;

for(int j = 1 ; j < i ; j ++)

{

int temp2 = dis;

for(int k = 1; k <= temp2 ; k ++)

{

if(fuzhu[j] == page[k])break;

else if (k == temp2)

{

page[k + 1] = fuzhu[j];

dis = dis +1;

}

}

for(int k = 1; k <= dis; k ++)printf("%d ",ljpage[page[k]].num);

printf("\n\n");

}

if(ljamount == 0) dis = 0;

printf("缺页次数为：\n");

printf("%d\n",dis);

printf("是否继续使用OPT算法？\n\n");

printf("1.是 2.否\n");

fflush(stdin);

if(getch() == '1') goto there;

else return 0;

}

思路为：

1. 给入一个逻辑页面号
2. 判断该页号是否在物理页面中
3. 在：OK，不做任何变化

不在：在下一个物理页面中加入这个逻辑页号，且dis值+1。

值得注意的是：

1. 因为物理页面比逻辑页面（不同的）还多，所以每一个逻辑页面都有物理页面可存。中断次数 = 逻辑页面数。
2. 给dis赋值为1是在逻辑页面数 > 0的前提下得出的，所以在最后输出结果时，应单独讨论ljamount值为0时的情况。

第二种：

dis = 1;

page[1] = 1;

for(int j = 1 ; j <= sign ; j ++)

{

int temp2 = dis;

for(int k = 1; k <= temp2; k ++)

{

if(fuzhu[j] == page[k])break;

else if (k == temp2)

{

page[k + 1] = fuzhu[j];

dis = dis +1;

}

}

for(int k = 1; k <= dis; k ++)printf("%d ",ljpage[page[k]].num);

printf("\n");

}

int daxiao[100];

for(int j = sign + 1 ; j < i ; j ++)

{

for(int k = 1; k <= wlpagenum; k ++)

{

if(fuzhu[j] == page[k])break;

else if (k == wlpagenum)

{

for(int t = 1; t <= wlpagenum; t ++)

{

for(int p = 1; p <= ljpage[page[t]].amount; p ++)

if( ljpage[page[t]].xy[p] > j)

{

daxiao[t] = ljpage[page[t]].xy[p];

break;

}

else if (p == ljpage[page[t]].amount)

{

daxiao[t] = 200 - ljpage[page[t]].xy[p];

}

}

int temp2 = 0;

for(int t = 1; t <= wlpagenum; t ++)

{

if (daxiao[t] > temp2) temp2 = daxiao[t];

}

int sign2;

for(int t = 1; t <= wlpagenum; t ++)

{

if(daxiao[t] == temp2)

{

sign2 = t;

break;

}

}

page[sign2] = fuzhu[j];

dis = dis + 1;

}

}

for(int j = 1; j <= wlpagenum;j ++)printf("%d ",ljpage[page[j]].num);

printf("\n");

}

在这部分中，在物理页面没有完全被占用前的填充情况和第一种是完全一样的，之所以没有合并是因为这样更方便于输出结果，在接下来的三种方法里我使用的都是合并的方法。

在物理页面完全填满后的填充思路为：

1. 给入一个逻辑页面号
2. 判断该页号是否在物理页面中
3. 在：OK，不做任何变化

不在：进行页面置换

进行页面置换是算法的重头戏，也是最头疼的部分，在确定最终方案前我曾考虑过很多种方法，最终都因太过繁琐而放弃了。现在的算法是目前我能想到的最为简单的算法（不一定是事实上的最简便的方法）。算法的思路如下：

1. 找出目前占有内存的逻辑页面下一次出现时在逻辑列中的位置。具体实现过程为：
2. 通过物理页page[]找到对应的逻辑列ljpage[] （即ljpage[ page[] ]） \*所以说page[]记录i比记录num好
3. 在ljpage[] 的坐标xy[]找到大于当前位置的最小坐标值，赋给数组daxiao[]备用（有的可能没有，这时应采用先进先出的原则，给daxiao[]赋值为

200 - ljpage[page[t]].xy[ljpage[page[t]].amount]

这样赋值是有特殊考虑的。一方面要遵循换掉坐标值大的逻辑页的原则，所有已填完最后页的逻辑页由先进先出原则，先进的页的最后的坐标值小，故应写为某个数减去这个坐标；另一方面因为要保证它的假的“xy[]值”大于所有真的xy[]值（理论上假的坐标值应理解为无穷大），所以使用200来减去它最后出现的位置，使假的“xy[]”大于

100，达到大于所有真的xy[]的目的。

1. 在所有占有内存的逻辑页的xy[]值中找出最大值，并通过这个最大值找出对应的逻辑页面，将其替换掉，并将缺页次数dis加1。

至此OPT算法已经实现。

**（二）、LRU算法**

除了OPT算法比较特殊含有3个成员的结构体（ljpagerange）、特殊的输入方式（如使用sign来标记正好填满物理页时的逻辑页对应的ljpage[]的位置；page[]不储存页号而储存位置）外，LRU、FIFO、LFU算法都采用同一种结构体（dafuzhu）、同一种输入方式，即以一个数组temp[]先接收逻辑列的页号，再将逻辑列填入物理页的方式。

在将逻辑页填入物理页时，LRU、FIFO、LFU算法都将填入过程分为dis < wlpagenum 的时候和dis >= wlpagenum（这个过程可能不经历）的时候。

其它填入方式比较平凡，不再赘述，现讨论置换算法。由于用来模拟物理页的结构体数组lru[]中成员range存有逻辑页的新旧程度（离上一次使用间隔了多少次访问），所以只需要找出拥有最大range的逻辑页并把它替换掉即可。

LRU算法的特别之处在于对物理页的排序输出。由于在逻辑列的填入过程中我没有对物理页中的逻辑页进行排序。所以，在输出结果时，采用了冒泡排序法对每个占据内存的逻辑页的range进行了排序。当然，并没有直接排，而是创建一个新的数组paixv[]，将range[]赋值给paixv[]，在paixv[]上进行排列，再将结果反馈给lru[]，使lru[]的成员num按顺序输出。

下面是在输出物理页未填满前的页面变化情况的代码（填满后的代码类似，不赘述）：

for(int t = 1; t <= wlpagenum; t ++)

paixv[t] = lru[t].range;

for(int t = 1; t <= wlpagenum; t ++)

for(int p = 1; p <= wlpagenum - t; p ++)

{

int omg;

if(paixv[p] < paixv[p + 1])

{

omg = paixv[p];

paixv[p] = paixv[p + 1];

paixv[p + 1] = omg;

}

}

for(int t = 1; t <= wlpagenum; t ++)

for(int p = 1; p <= wlpagenum ; p ++)

{

if(lru[p].range == paixv[t])

{

printf("%d",lru[p].num);

break;

}

}

**（三）、FIFO算法**

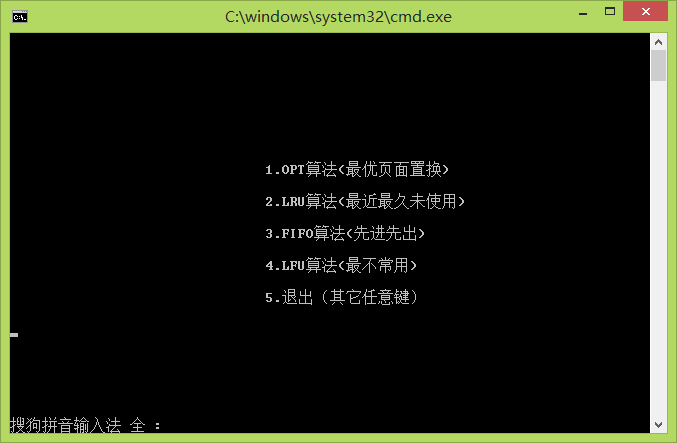
这个算法实现起来最为简单。只需要建立一个数组page[]，每填进一个逻辑页就将数组元素整体右移一个单位，在page[1]处填入新逻辑页即可。（再次重申，page[0]已经被弃用）

**（四）、LFU算法**

LFU算法的实现也很简单，在记录逻辑页的结构体数组中标记各个逻辑页的已被使用的次数即可。值得一提的是，由于LFU算法在模拟过程中难以模拟到所有情况（因为在页面置换过程中已占据内存的逻辑页的使用次数可能相同），所以只模拟出了其中一种算法。

**运行概况：**

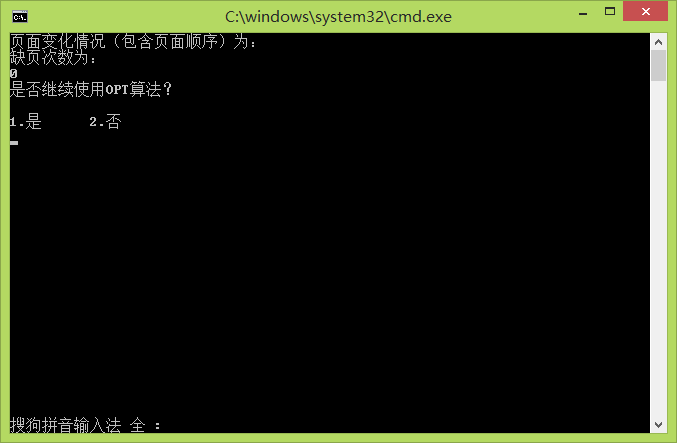
**主菜单：**



1. **OPT算法**

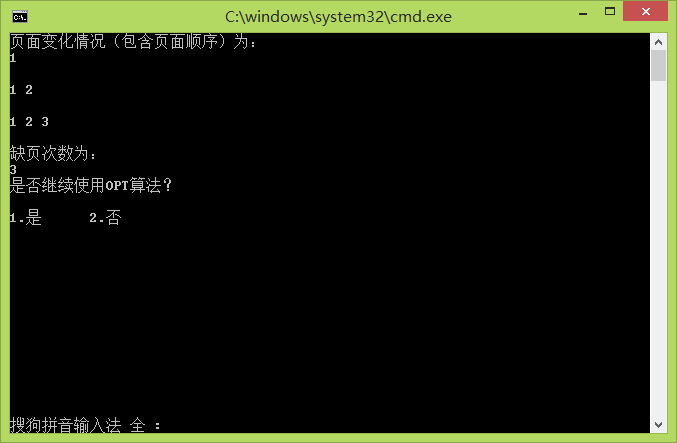
情形一：无逻辑页面





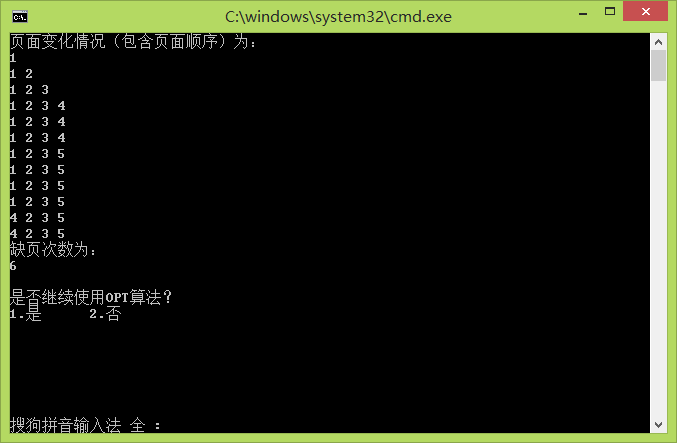
情形二：缺页次数少于物理页面数





情形三：缺页次数多于物理页面数（使用教材例子）

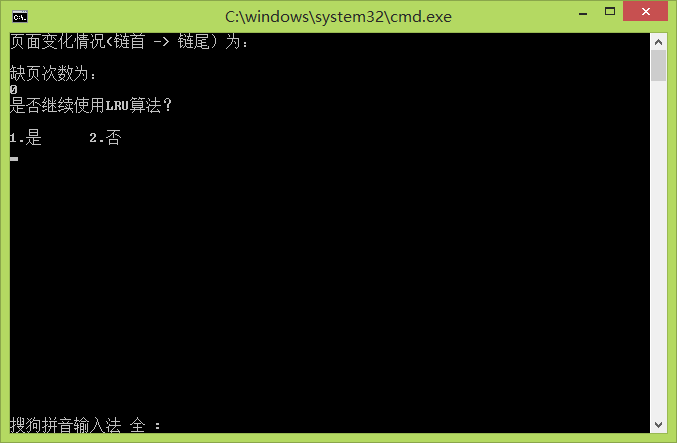




1. **LRU算法**

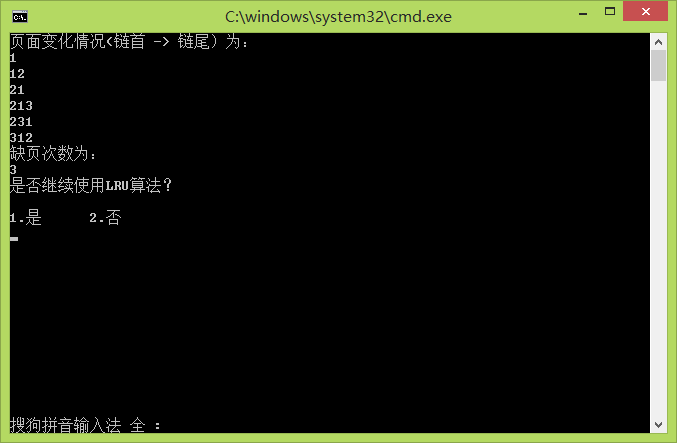
情形一：无逻辑页面





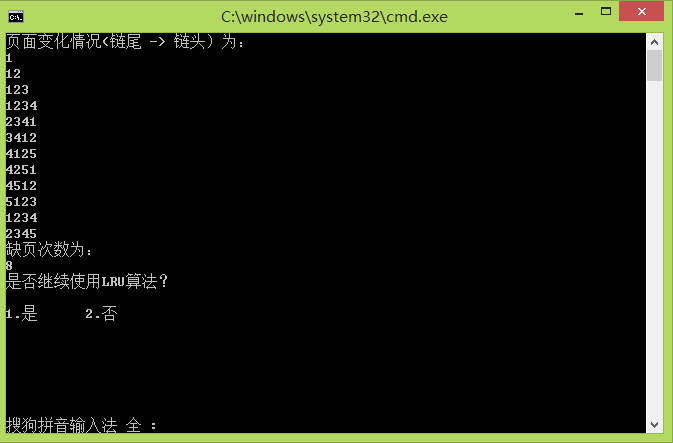
情形二：缺页次数少于物理页面数





情形三：缺页次数多于物理页面数（使用教材例子）

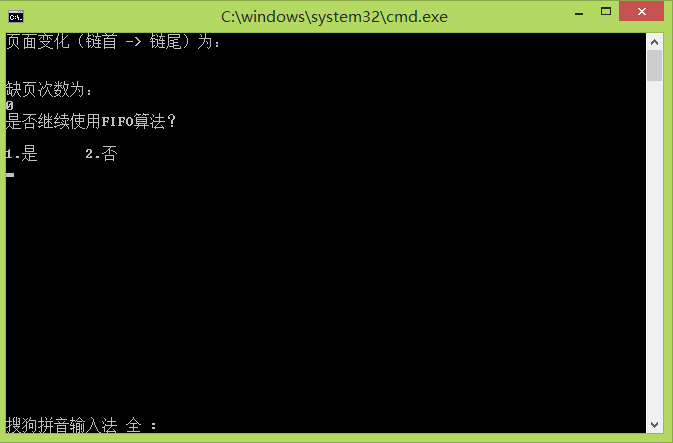




1. **FIFO算法**

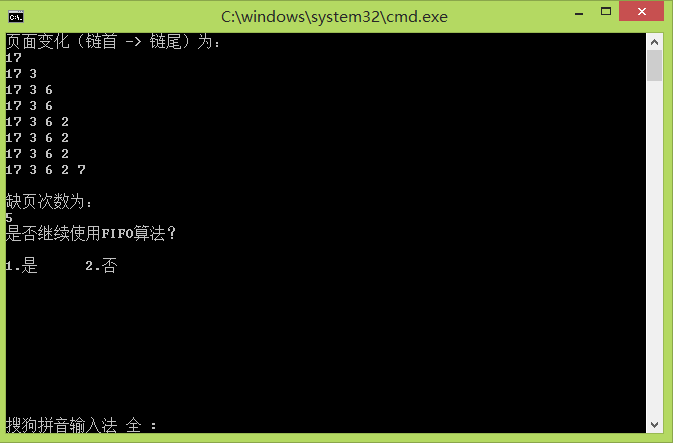
情形一：无逻辑页面





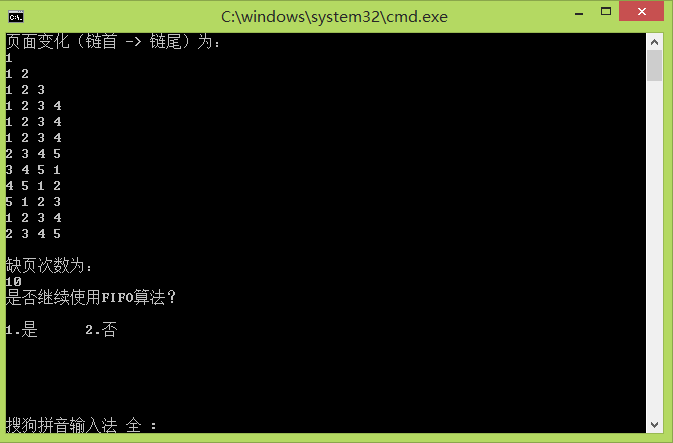
情形二：缺页次数少于物理页面数





情形三：缺页次数多于物理页面数（使用教材例子）

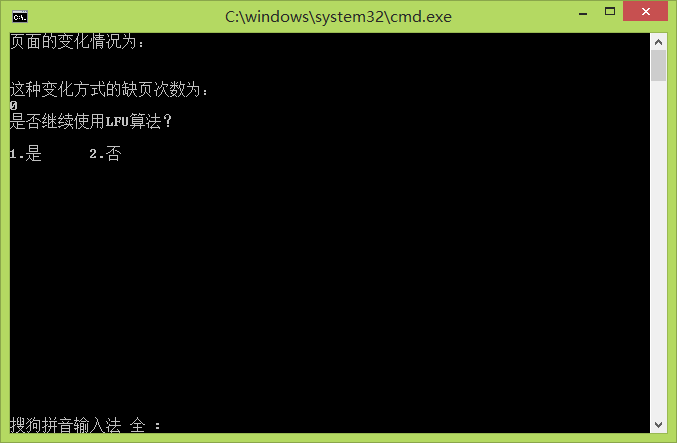




1. **LFU算法**

情形一：无逻辑页面





情形二：缺页次数少于物理页面数





情形三：缺页次数多于物理页面数（使用教材例子）





1. **退出**



**莫伟彬**

**水利水电工程系**

**邮箱：thumwb14@qq.com**