《操作系统原理》实验报告

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| 姓名 | 邬雪菲 | 学号 | U202112131 | 专业班级 | 网安2104 | 时间 | 2023.12.5 |

**一、实验目的**

1）理解页面淘汰算法原理，编写程序演示页面淘汰算法。

2）验证Linux虚拟地址转化为物理地址的机制

3）理解和验证程序运行局部性的原理。

4）理解和验证缺页处理的流程。

**二、实验内容**

1）Win/Linux编写二维数组遍历程序，理解局部性的原理。

2）Windows/Linux模拟实现OPT或FIFO或LRU淘汰算法。

3）研读并修改Linux内核的缺页处理函数do\_no\_page 或页框分配函数get\_free\_page，并用printk打印调试信息。注意：需要编译内核。建议优麒麟或麒麟系统。

4）Linux下利用/proc/pid/pagemap技术计算某个变量或函数虚拟地址对应的物理地址等信息。建议优麒麟或麒麟系统。

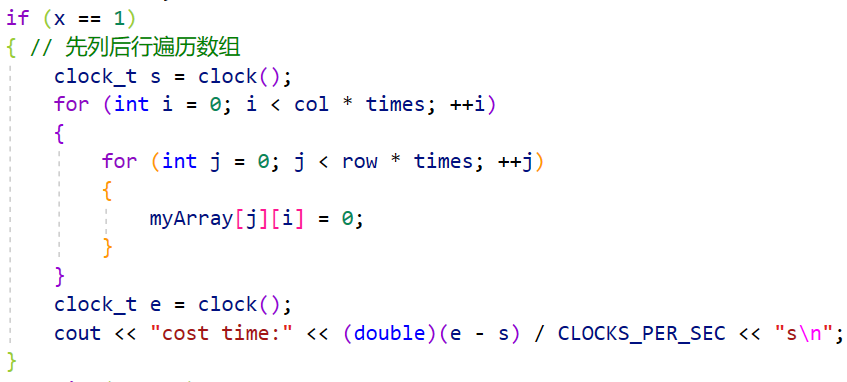
**三、实验过程**

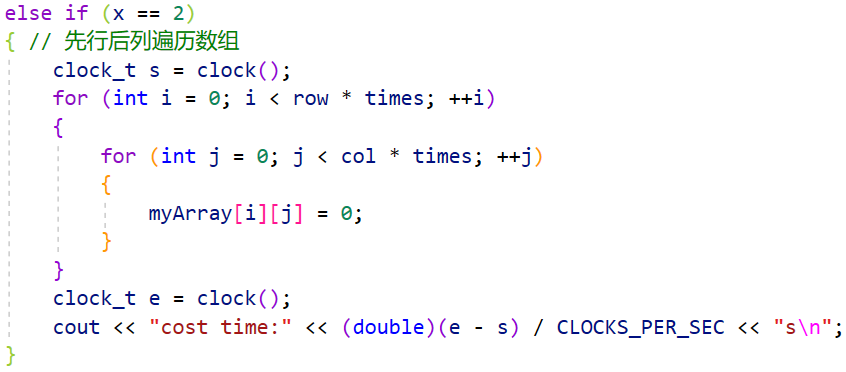
**3.1 windows二维数组遍历**

1）定义遍历函数

分别编写四种遍历情况，先行后列、先列后行、大数组和小数组，并在遍历的开始和结束读取时间并计算遍历耗时，由此可初步观察遍历效率和局部性好坏。

代码在Visual Studio Code上编写，代码如下所示。

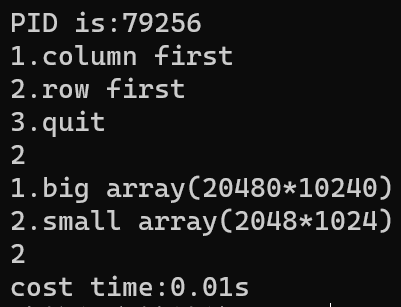
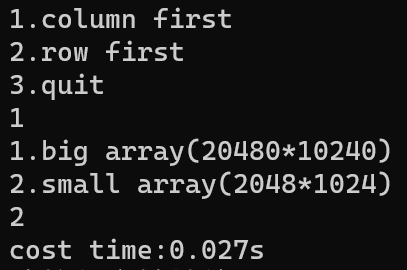


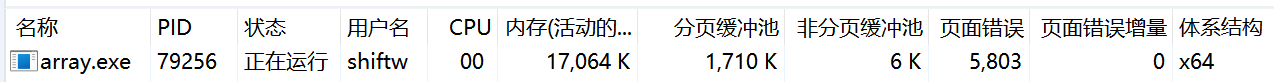


2）测试程序

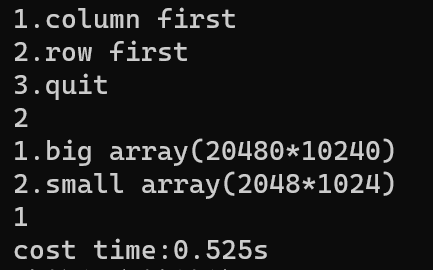
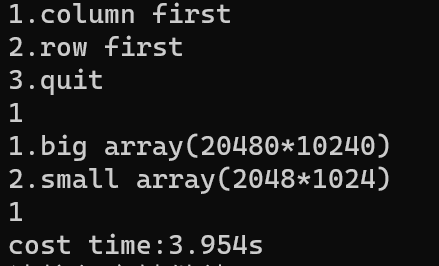
Windows11下编译运行cpp程序，分别测试四种情况下的遍历过程，同时在任务管理器中检测程序缺页情况，结果如下：

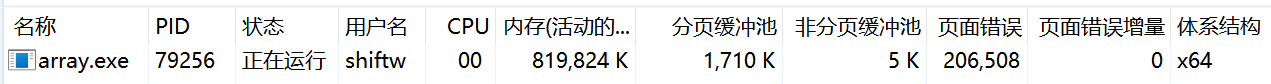
首先是小数组的运行情况，数组大小为2048\*1024，可以由输出结果知先行后列和先列后行的遍历效率差距较大，先行后列的局部性较好，遍历速度快了几倍。但是观察缺页率会发现两种方式一致。



然后是大数组的运行情况，数组大小为20480\*10240，可以观察到两种运行方式的效率差异更大了，先列后行的方式局部性较低，在数据量增大的情况下运行时间急剧增大，而先行后列的方式运行效率则比小数据量情况下更显著地优于先列后行，从几倍提高到了几十倍的差异。不过两者的缺页率一致。



综上所述，可以看到数组更大的情况下缺页次数更多，遍历耗时也更长。但是同规模数组情况下，遍历过程的缺页次数一致，执行效率却差别很大，先行后列的遍历方式耗时更短。

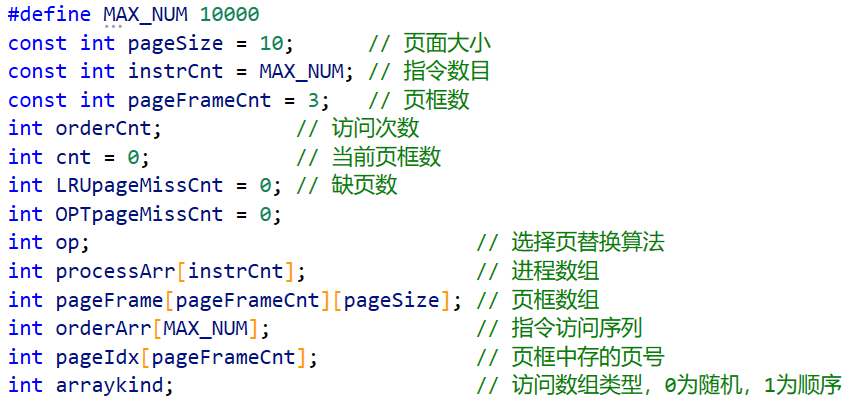
**3.2 Windows模拟OPT和LRU算法**

本文在Windows11上使用Visual Studio Code编写cpp程序模拟OPT和LRU的淘汰页面算法，FIFO算法的实现较为简单，在此不作模拟。

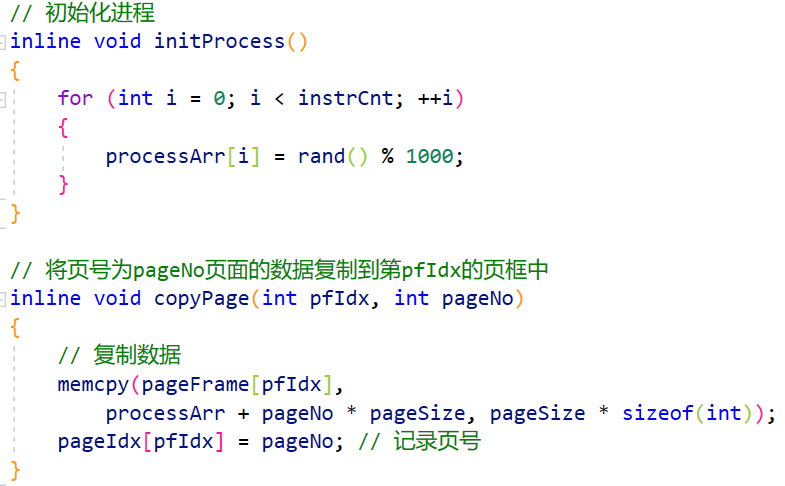
1）代码编写

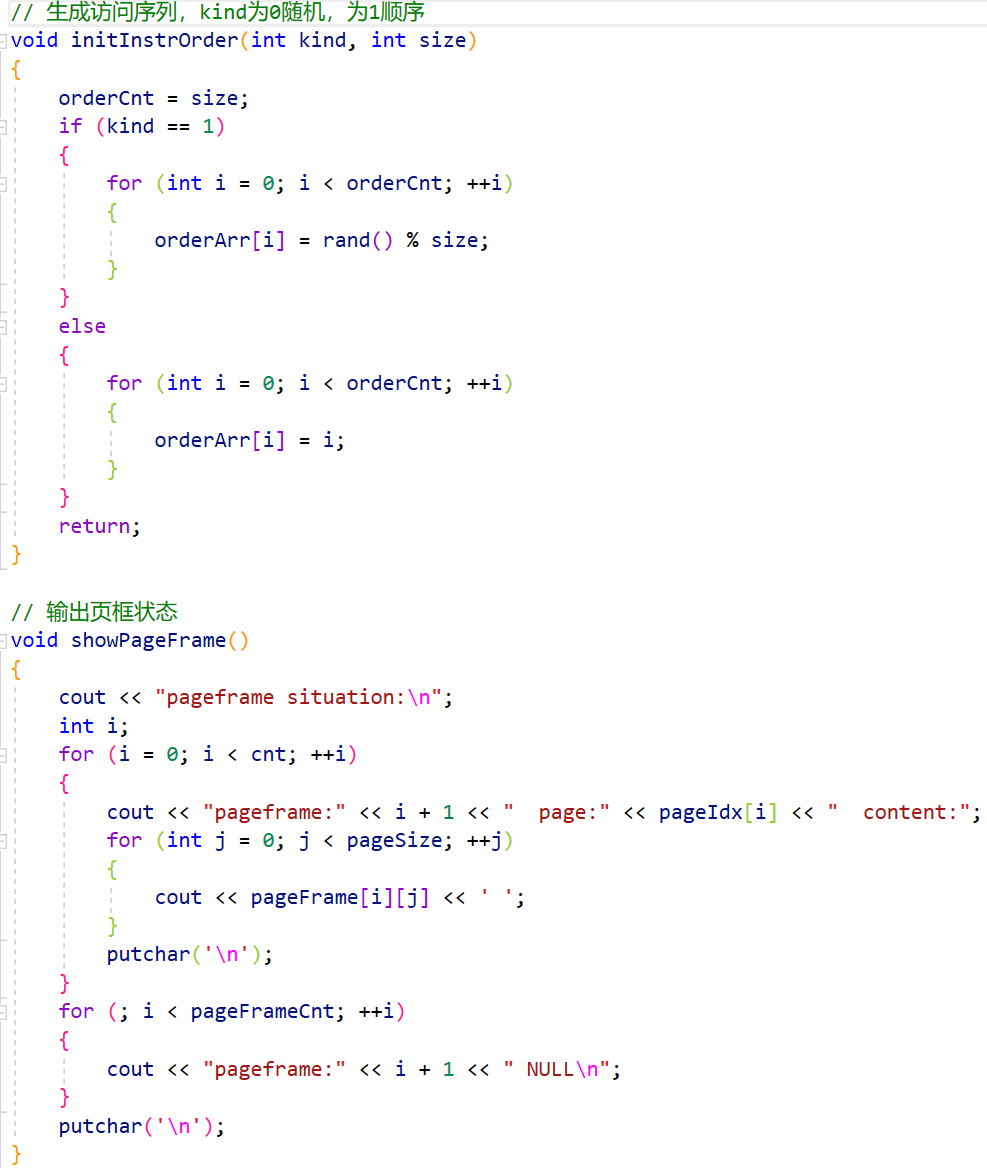
首先进行一些全局常量的定义，包括页面大小、页框数、等等，方便根据需要进行调整。

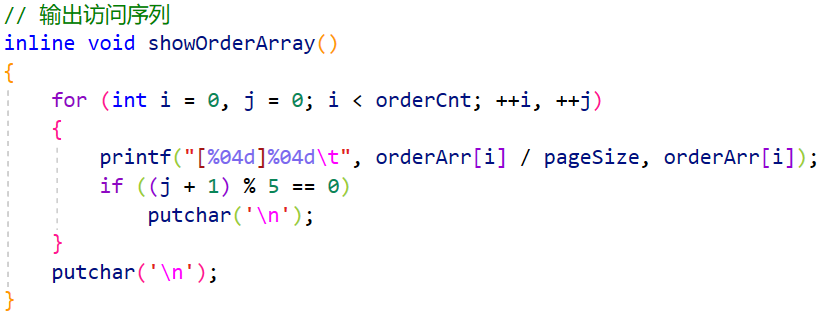
然后定义全局变量用于记录模拟过程中的关键信息，包括用于记录当前物理内存中已经装有数据的页框数目、缺页次数、进程数组、指令序列等等。

****

接着定义一些模拟页面淘汰过程中的辅助函数，比如进程初始化函数，以及访问序列输出函数，访问序列生成函数，可以生成随机访问序列和顺序访问序列，如需扩展更多情况只需要修改该函数即可。再者就是页框情况的查看和打印，可按需调用。



****

****

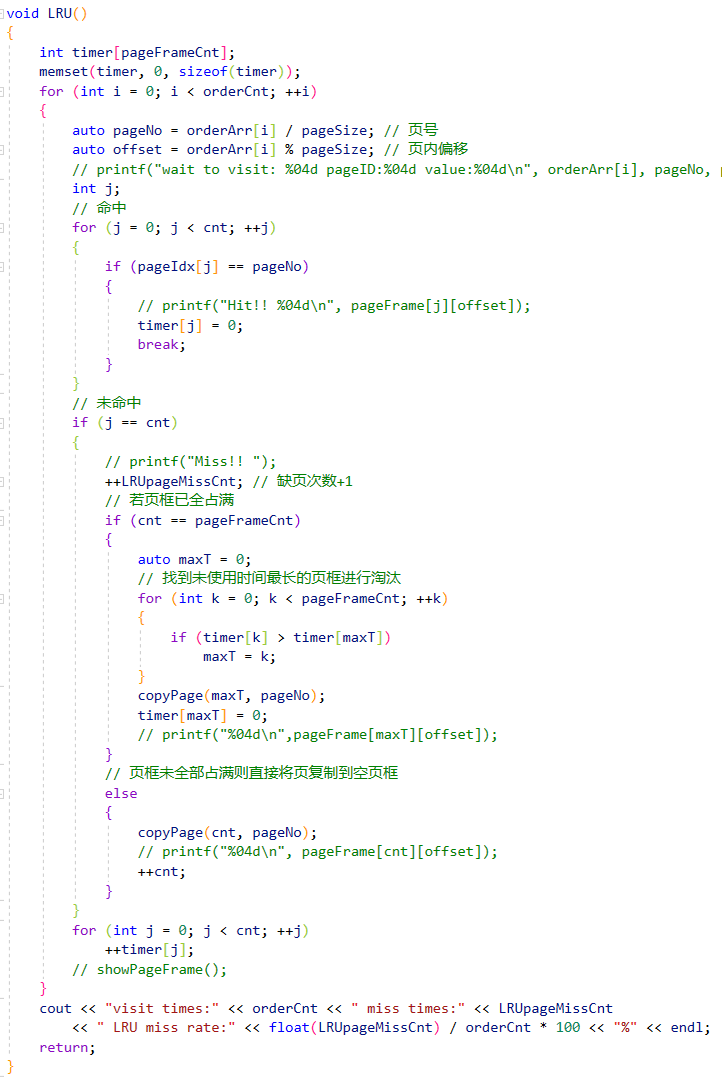
然后便是主要控制程序的编写，main函数首先询问用户访问指令序列个数，然后进入循环。每次循环可执行一次OPT模拟或者LRU模拟，每次都会重新初始化线程，并根据用户选择的指令数量和类型生成待访问指令序列，才开始进行模拟，具体代码如下：



接下来是核心淘汰算法的编写：

1. LRU算法

该算法的思路是淘汰内存中最长时间未被使用的页面。算法的实现思路也比较简单，只需要定义一个额外的辅助数组，记录内存中每个已装有页面的页框未被使用的时间。当指令命中时，则对其页面所在页框的计时器置零，同时每次访问对所有有页面的页框时间加 1，这样数组中最大数对应的页框，即为页面为最长时间未被使用的页面，在非命中且页框存满的情况下淘汰该页即可。



1. OPT算法

该算法的思想是淘汰以后不再使用或者最远的将来才会使用的页面。此实验中由于访问的序列有限，且可以提前读取访问序列，可以实现算法的模拟。

这部分我采用了一个待访问信息表来辅助算法的实现，表中记录所有需要访问的页面和其待访问信息，即该页的所有访问序号。这里我采用了映射实现，键名为页号，键值为一个堆栈，堆栈中将会存有该页面所有的访问序号。

算法首先要记录页面的待访问信息，通过逆序遍历访问序列记录。每遍历一个访问指令，计算出其对应页号，若该页号未在待访问信息表中，则表明此时该页面还未被访问过，则将该页号记录到待访问信息表中，同时在堆栈中添加当前访问序号，该序号是原始的顺序访问序号；若页号已经出现在表中，则直接将当前访问序号加入待访问信息的堆栈中。由此构建完成访问信息表，序列中所有会访问的页号都会记录在表中，而其对应的待访问信息堆栈会记录该页面所有的访问次序，由栈顶到栈底访问序号递增。

接下来顺序遍历访问序列，正式开始模拟。每次进行指令访问时都会先将其页面在待访问信息表中堆栈的栈顶元素（即当前访问序号）出栈。当该指令的页面未命中且内存页框全部占满时，则在待用信息表中查找页框中所有页面的待用信息，若有页面的堆栈为空，则证明该页面在之后不会被需要，则直接选择该页面进行淘汰；否则比较待用信息堆栈的栈顶元素，该栈顶元素即为该页面下次访问的序号，找出序号最大者即为最远的将来才会被访问的页面，然后将其淘汰即可。

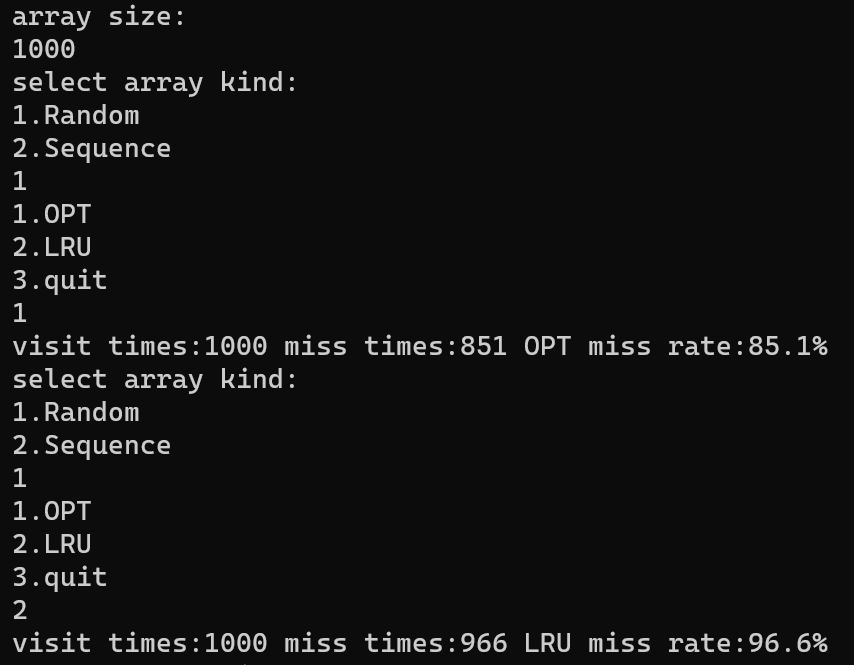


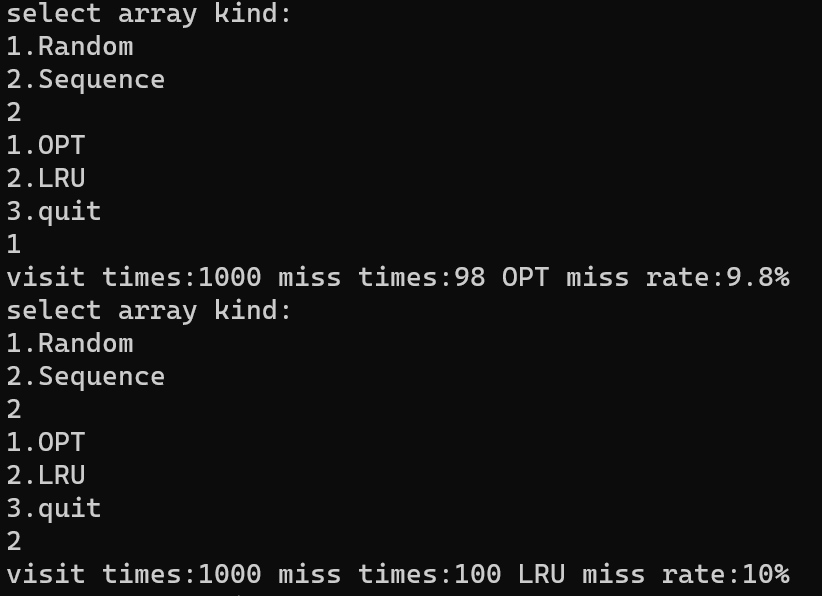


2）测试程序

在Windows11下编译运行该程序。

可观察到在访问序列数量为1000且随机的情况下，无论是OPT还是LRU算法的缺页率都较高，百分之九十上下的缺页率，但OPT的表现相对于LRU要更优。而当顺序访问时，两者表现情况相差不大。





**3.3 Linux下计算虚拟地址对应的物理地址**

1）程序编写

Linux的/proc/self/pagemap文件允许用户查看当前进程的虚拟页的物理地址相关信息，其中每个记录均为8字节64位，最高位记录了当前虚拟页是否在内存中，1表示在物理内存中，0表示不在物理内存中。当最高位为1即虚拟页在物理内存时，0~54位则记录了该虚拟页的物理页号。

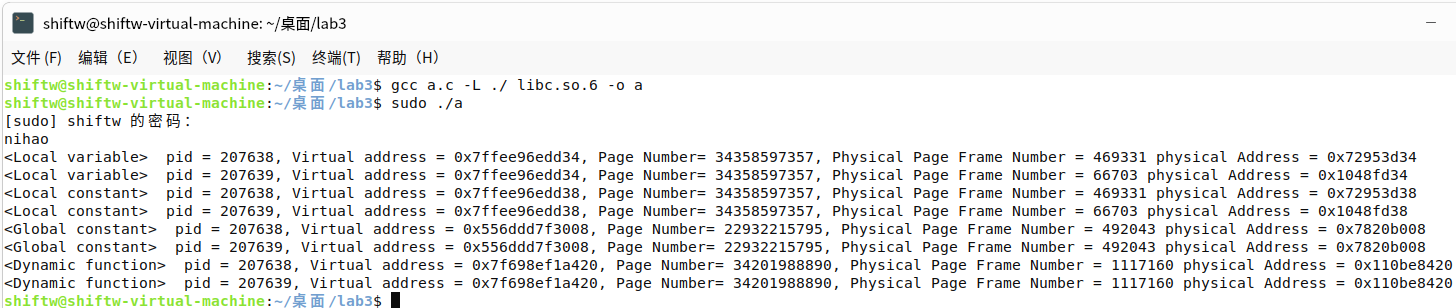
因此，对于给定的虚拟地址，先使用getpagesize()函数获取页面大小，利用虚拟地址除以页面大小可以得到虚拟页号，虚拟地址对页面大小取模可以得到页内偏移。利用pagemap文件，找到虚拟页号对应的8字节记录，并分析其最高位和第55位，即可找到该虚拟页号对应的物理页号，进而加上页内偏移量计算出虚拟地址对应的物理地址。

同时需要声明局部和全局变量，以及调用动态库函数。并fork子线程以便对比父子进程的物理地址映射，最终具体代码如下：



2）测试运行

在vmware的ubuntukylin虚拟机中链接动态库编译运行c程序，结果如下所示：

****

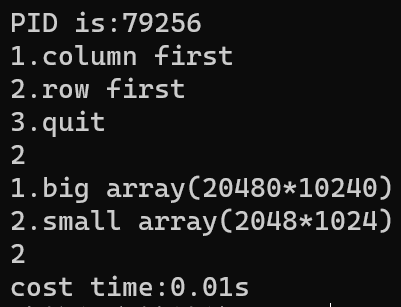
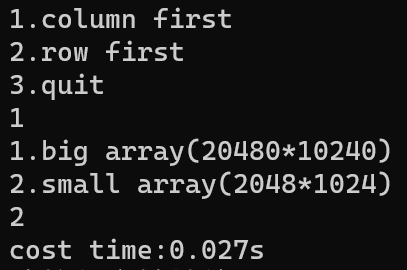
可以观察到局部变量和全局变量在父子进程中映射的物地址不同，但是全局常量是相同的，说明子进程会重新创建变量，但是仍使用父进程的常量。而对于动态链接库，main中调用puts成功打印"nihao"字符串，且父子进程调用的puts函数映射到的物理地址相同。

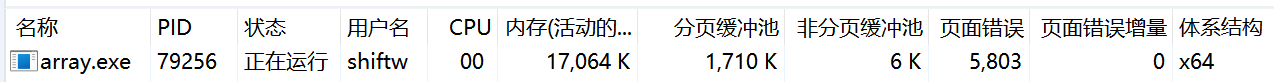
**四、实验结果**

**4.1 Windows遍历二维数组**

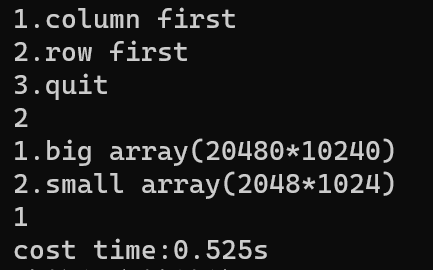
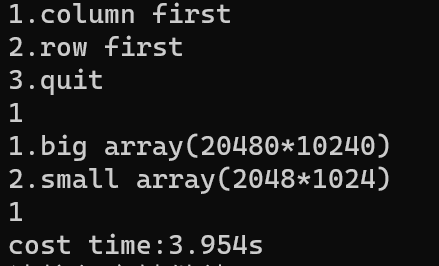
Windows11下编译运行cpp程序，分别测试四种情况下的遍历过程，同时在任务管理器中检测程序缺页情况，结果如下：

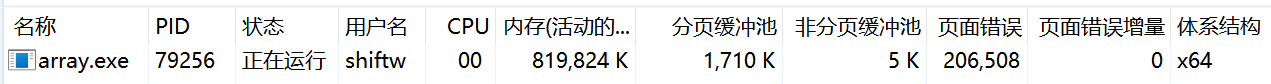
首先是小数组的运行情况，数组大小为2048\*1024，可以由输出结果知先行后列和先列后行的遍历效率差距较大，先行后列的局部性较好，遍历速度快了几倍。但是观察缺页率会发现两种方式一致。



然后是大数组的运行情况，数组大小为20480\*10240，可以观察到两种运行方式的效率差异更大了，先列后行的方式局部性较低，在数据量增大的情况下运行时间急剧增大，而先行后列的方式运行效率则比小数据量情况下更显著地优于先列后行，从几倍提高到了几十倍的差异。不过两者的缺页率一致。

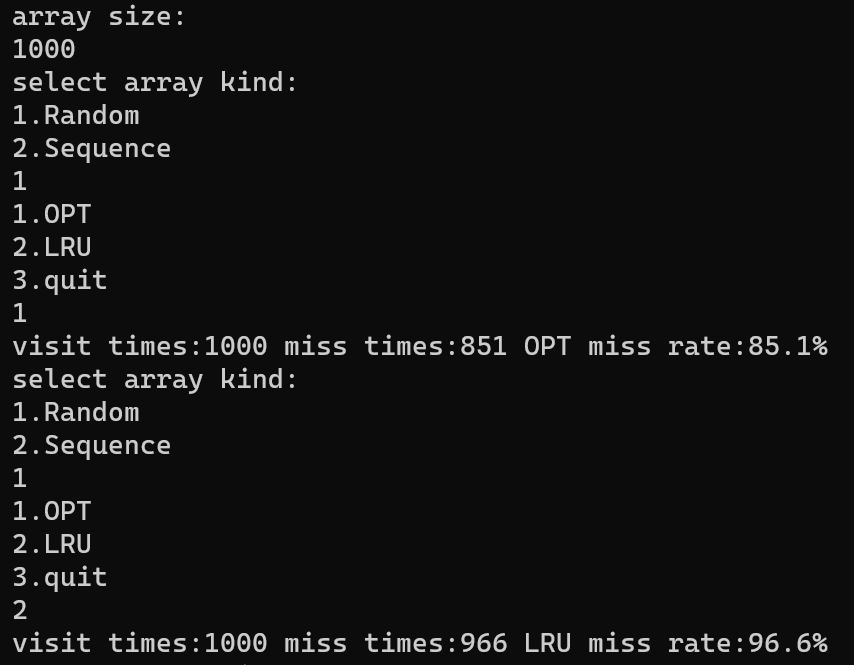
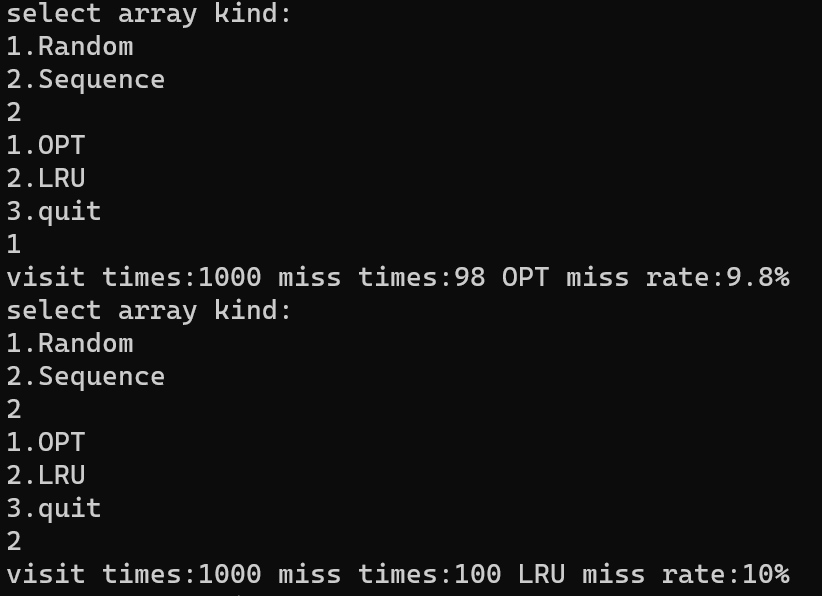


综上所述，访问数组时会将部分页面调到内存中，占用内存增加、页面错误增加。所以数组更大的情况下缺页次数更多，遍历耗时也更长。但是同规模数组情况下，遍历过程的缺页次数一致，执行效率却差别很大，这是因为程序局部性的差异不影响缺页率但影响执行效率，先行后列的遍历方式耗时更短。

**4.2 Windows下模拟OPT和LRU算法**

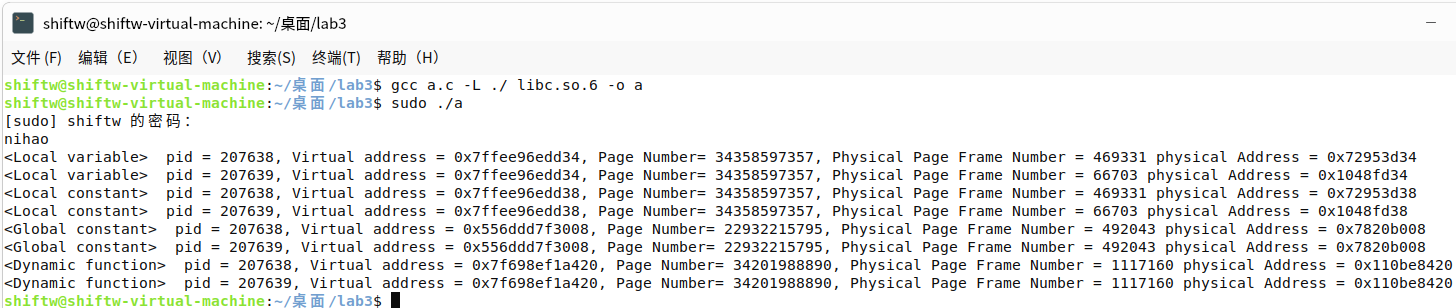
在Windows11下编译运行该程序。

可观察到在访问序列数量为1000且随机的情况下，无论是OPT还是LRU算法的缺页率都较高，百分之九十上下的缺页率，但OPT的表现相对于LRU要更优。而当顺序访问时，两者表现情况相差不大。

**4.3 Linux下计算虚拟地址对应的物理地址**

在vmware的ubuntukylin虚拟机中链接动态库编译运行c程序，结果如下所示：



可以观察到局部变量和全局变量在父子进程中虚拟地址虽然相同，但映射的物地址不同，而全局常量物理地址是相同的，说明子进程会重新创建变量，但仍使用父进程的常量。而对于动态链接库，main中调用puts成功打印"nihao"字符串，且父子进程调用的puts函数映射到的物理地址相同。

**五、实验错误排查和解决方法**

**5.1 Windows遍历二维数组**

Windows11下运行程序时发现同样的数组同样的遍历方式每一次的运行结果都有一点小差异，在Linux虚拟机中跑也是一样的。目前该问题未得到解答，猜测可能是该程序编译时初始的一些要调用的数据在内存中的缺页情况不同，比如刚运行过一次的一些数据仍存在内存中，或过一段时间已经全部被淘汰。

**5.2 Windows下模拟OPT和LRU算法**

FIFO和LRU的实现思路比较容易想到，但是如何实现OPT算法则不是那么容易想到。一开始想直接顺序运行，需要淘汰的时候直接往后遍历访问序列并做标记，最后寻找没有标记或者标记比较大的，但是这个操作的时间复杂度显然比较高，因为每次都要重新向后遍历。之后想到先逆序遍历预处理的方式，直接将访问顺序用堆栈的形式预先记录下来，这样每次淘汰的时候只需要查询即可，复杂度降低。

**5.3 Linux下计算虚拟地址对应的物理地址**

1）获取物理页号失败

运行程序得到的物理页号总为0，经搜索资料发现，自 Linux4.0 之后，只有系统管理员才能获取到 PFNs 即物理页号，而普通用户只能得到物理页号为 0。因此需要使用管理员权限执行程序才可以获得正确的物理页号。

2）动态库链接失败

动态库是直接拷贝现有的libc.so.6到c源代码文件夹中，并在程序中调用。一开始按照网上教程用gcc a.c -L ./ -l libc.so.6 -o a编译失败，显示找不到动态库，搜索解决方案是要在/usr/bin/路径下复制一份动态库，但仔细查看报错信息后发现是-l参数失效，去掉-l即可成功链接编译。

**六、实验参考资料和网址**

**（1）教学课件**

**（2）https://www.jianshu.com/p/544ee20e307c**

**（3）https://www.cnblogs.com/wasi-991017/p/13072328.html**

**（4）https://blog.csdn.net/qq\_42615643/article/details/97512361**

**（5）https://www.cnblogs.com/pengdonglin137/p/6802108.html**