# 《操作系统原理》实验报告 3

| 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。

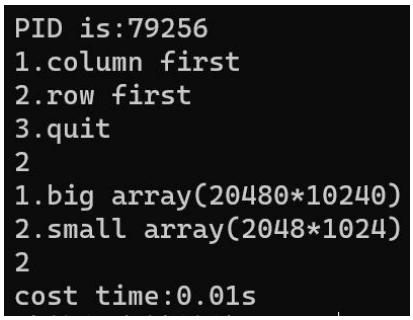
```
if (x == 1)
{ // 先列后行遍历数组
    clock_t s = clock();
    for (int i = 0; i < col * times; ++i)
    {
        for (int j = 0; j < row * times; ++j)
        {
            myArray[j][i] = 0;
        }
    }
    clock_t e = clock();
    cout << "cost time:" << (double)(e - s) / CLOCKS_PER_SEC << "s\n";
}
```

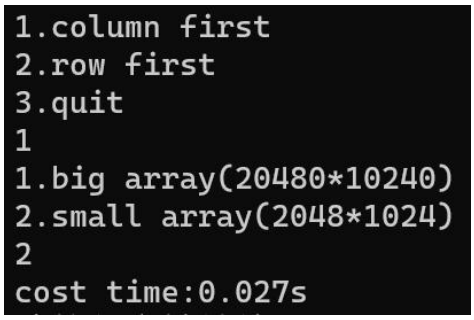
```
else if (x == 2)
{ // 先行后列遍历数组
    clock_t s = clock();
    for (int i = 0; i < row * times; ++i)
    {
        for (int j = 0; j < col * times; ++j)
        {
            myArray[i][j] = 0;
        }
    }
    clock_t e = clock();
    cout << "cost time:" << (double)(e - s) / CLOCKS_PER_SEC << "s\n";
}
```

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：

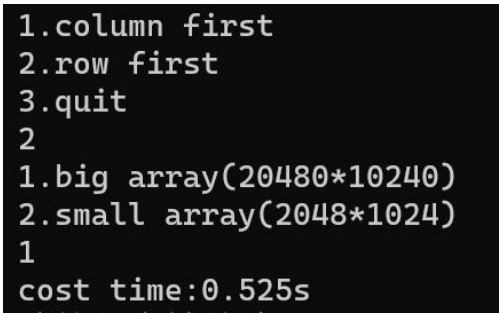
首先是小数组的运行情况，数组大小为 2048*1024，可以由输出结果知先行后列和先列后行的遍历效率差距较大，先行后列的局部性较好，遍历速度快了几倍。但是观察缺页率会发现两种方式一致。

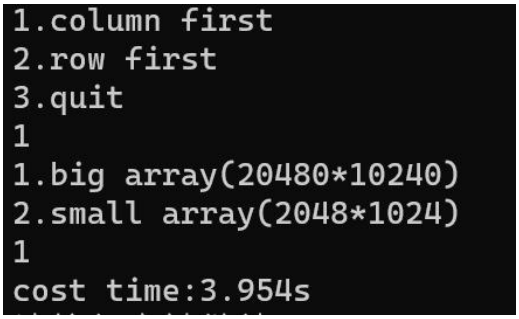



| 名称 | PID | 状态 | 用户名 | CPU | 内存(活动的... | 分页缓冲池 | 非分页缓冲池 | 页面错误 | 页面错误增量 | 体系结构 |
|---|---|---|---|---|---|---|---|---|---|---|
| array.exe | 79256 | 正在运行 | shiftw | 00 | 17,064 K | 1,710 K | 6 K | 5,803 | 0 | x64 |

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。



| 名称 | PID | 状态 | 用户名 | CPU | 内存(活动的... | 分页缓冲池 | 非分页缓冲池 | 页面错误 | 页面错误增量 | 体系结构 |
|---|---|---|---|---|---|---|---|---|---|---|
| array.exe | 79256 | 正在运行 | shiftw | 00 | 819,824 K | 1,710 K | 5 K | 206,508 | 0 | x64 |

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。

```
#define MAX_NUM 10000
const int pageSize = 10;       // 页面大小
const int instrCnt = MAX_NUM; // 指令数目
const int pageFrameCnt = 3;    // 页框数
int orderCnt;            // 访问次数
int cnt = 0;             // 当前页框数
int LRUpageMissCnt = 0; // 缺页数
int OPTpageMissCnt = 0;
int op;                                  // 选择页替换算法
int processArr[instrCnt];                // 进程数组
int pageFrame[pageFrameCnt][pageSize]; // 页框数组
int orderArr[MAX_NUM];                   // 指令访问序列
int pageIdx[pageFrameCnt];               // 页框中存的页号
int arraykind;                           // 访问数组类型，0为随机，1为顺序
```

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。

```
// 初始化进程
inline void initProcess()
{
    for (int i = 0; i < instrCnt; ++i)
    {
        processArr[i] = rand() % 1000;
    }
}

// 将页号为pageNo页面的数据复制到第pfIdx的页框中
inline void copyPage(int pfIdx, int pageNo)
{
    // 复制数据
    memcpy(pageFrame[pfIdx],
        processArr + pageNo * pageSize, pageSize * sizeof(int));
    pageIdx[pfIdx] = pageNo; // 记录页号
}
```

```
// 生成访问序列，kind为0随机，为1顺序
void initInstrOrder(int kind, int size)
{
    orderCnt = size;
    if (kind == 1)
    {
        for (int i = 0; i < orderCnt; ++i)
        {
            orderArr[i] = rand() % size;
        }
    }
    else
    {
        for (int i = 0; i < orderCnt; ++i)
        {
            orderArr[i] = i;
        }
    }
    return;
}

// 输出页框状态
void showPageFrame()
{
    cout << "pageframe situation:\n";
    int i;
    for (i = 0; i < cnt; ++i)
    {
        cout << "pageframe:" << i + 1 << "  page:" << pageIdx[i] << "  content:";
        for (int j = 0; j < pageSize; ++j)
        {
            cout << pageFrame[i][j] << ' ';
        }
        putchar('\n');
    }
    for (; i < pageFrameCnt; ++i)
    {
        cout << "pageframe:" << i + 1 << " NULL\n";
    }
    putchar('\n');
}
```

```
// 输出访问序列
inline void showOrderArray()
{
    for (int i = 0, j = 0; i < orderCnt; ++i, ++j)
    {
        printf("[%04d]%04d\t", orderArr[i] / pageSize, orderArr[i]);
        if ((j + 1) % 5 == 0)
            putchar('\n');
    }
    putchar('\n');
}
```

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：

```
int main()
{
    srand(time(nullptr));
    printf("array size:\n");
    cin >> orderCnt;
    while (1)
    {
        initProcess();
        printf("select array kind:\n1.Random\n2.Sequence\n");
        cin >> arraykind;
        initInstrOrder(arraykind, orderCnt);
        // showOrderArray();
        printf("1.OPT\n2.LRU\n3.quit\n");
        cin >> op;
        if (op == 1)
            OPTpageMissCnt = 0, OPT();
        else if (op == 2)
            LRUpageMissCnt = 0, LRU();
        else
            break;
    }
    system("pause");
    return 0;
}
```

接下来是核心淘汰算法的编写：

a）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。

```
void LRU()
{
    int timer[pageFrameCnt];
    memset(timer, 0, sizeof(timer));
    for (int i = 0; i < orderCnt; ++i)
    {
        auto pageNo = orderArr[i] / pageSize; // 页号
        auto offset = orderArr[i] % pageSize; // 页内偏移
        // printf("wait to visit: %04d pageID:%04d value:%04d\n", orderArr[i], pageNo,
        int j;
        // 命中
        for (j = 0; j < cnt; ++j)
        {
            if (pageIdx[j] == pageNo)
            {
                // printf("Hit!! %04d\n", pageFrame[j][offset]);
                timer[j] = 0;
                break;
            }
        }
        // 未命中
        if (j == cnt)
        {
            // printf("Miss!! ");
            ++LRUpageMissCnt; // 缺页次数+1
            // 若页框已全占满
            if (cnt == pageFrameCnt)
            {
                auto maxT = 0;
                // 找到未使用时间最长的页框进行淘汰
                for (int k = 0; k < pageFrameCnt; ++k)
                {
                    if (timer[k] > timer[maxT])
                        maxT = k;
                }
                copyPage(maxT, pageNo);
                timer[maxT] = 0;
                // printf("%04d\n",pageFrame[maxT][offset]);
            }
            // 页框未全部占满则直接将页复制到空页框
            else
            {
                copyPage(cnt, pageNo);
                // printf("%04d\n", pageFrame[cnt][offset]);
                ++cnt;
            }
        }
        for (int j = 0; j < cnt; ++j)
            ++timer[j];
        // showPageFrame();
    }
    cout << "visit times:" << orderCnt << " miss times:" << LRUpageMissCnt
        << " LRU miss rate:" << float(LRUpageMissCnt) / orderCnt * 100 << "%" << endl;
    return;
}
```

b）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。

```
void OPT()
{
    // 待用信息表：键名为待访问的页号,键值为一个存有该页号访问次序的堆栈
    map<int, stack<int>> ms;
    // 逆序变量访问序列
    for (int i = orderCnt - 1; i >= 0; --i)
    {
        auto pageNo = orderArr[i] / pageSize; // 页号
        // 若该页未被访问
        if (ms.count(pageNo) == 0)
        {
            stack<int> tmp; // 创建堆栈
            tmp.push(i);    // 在堆栈中添加访问次序
            ms.insert(pair<int, stack<int>>(pageNo, tmp));
        }
        else
        {
            ms.at(pageNo).push(i); // 在堆栈中添加访问次序
        }
    }
    // 顺序执行
    for (int i = 0; i < orderCnt; ++i)
    {
        auto pageNo = orderArr[i] / pageSize; // 页号
        auto offset = orderArr[i] % pageSize; // 页内偏移
        // printf("wait to visit: %04d pageID:%04d value:%04d\n", orderArr[i], pageNo, processArr[orderArr[i]]);
        // 每次执行完将该页栈顶的待用信息出栈
        if (ms.at(pageNo).size())
            ms.at(pageNo).pop();
        int j;
        // 遍历页框若命中
        for (j = 0; j < cnt; ++j)
        {
            if (pageIdx[j] == pageNo)
            {
                // printf("Hit!! %04d\n", pageFrame[j][offset]);
                break;
            }
        }
```

```
    // 若未命中
    if (j == cnt)
    {
        // cout << "Miss!! ";
        ++OPTpageMissCnt; // 缺页次数+1
        // 若页框已全占满
        if (cnt == pageFrameCnt)
        {
            auto maxT = 0;
            // 遍历页框根据待用信息寻找不在需要或最远的将来才用的页面
            for (int k = 0; k < pageFrameCnt; ++k)
            {
                if (ms.at(pageIdx[k]).size() == 0)
                {
                    maxT = k;
                    break;
                }
                else if (ms.at(pageIdx[k]).top() > ms.at(pageIdx[maxT]).top())
                {
                    maxT = k;
                }
            }
            // 淘汰页面复制新数据
            copyPage(maxT, pageNo);
            // printf("%04d\n",pageFrame[maxT][offset]);
        }
        // 页框未全部占满则直接将页复制到空页框
        else
        {
            copyPage(cnt, pageNo);
            // printf("%04d\n", pageFrame[cnt][offset]);
            ++cnt;
        }
    }
    // 输出每次的页框信息
    // showPageFrame();
}
cout << "visit times:" << orderCnt << " miss times:" << OPTpageMissCnt
    << " OPT miss rate:" << float(OPTpageMissCnt) / orderCnt * 100 << "%" << endl;
return;
```

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。

```
array size:
1000
select array kind:
1.Random
2.Sequence
1
1.OPT
2.LRU
3.quit
1
visit times:1000 miss times:851 OPT miss rate:85.1%
select array kind:
1.Random
2.Sequence
1
1.OPT
2.LRU
3.quit
2
visit times:1000 miss times:966 LRU miss rate:96.6%
```

```
select array kind:
1.Random
2.Sequence
2
1.OPT
2.LRU
3.quit
1
visit times:1000 miss times:98 OPT miss rate:9.8%
select array kind:
1.Random
2.Sequence
2
1.OPT
2.LRU
3.quit
2
visit times:1000 miss times:100 LRU miss rate:10%
```

### 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

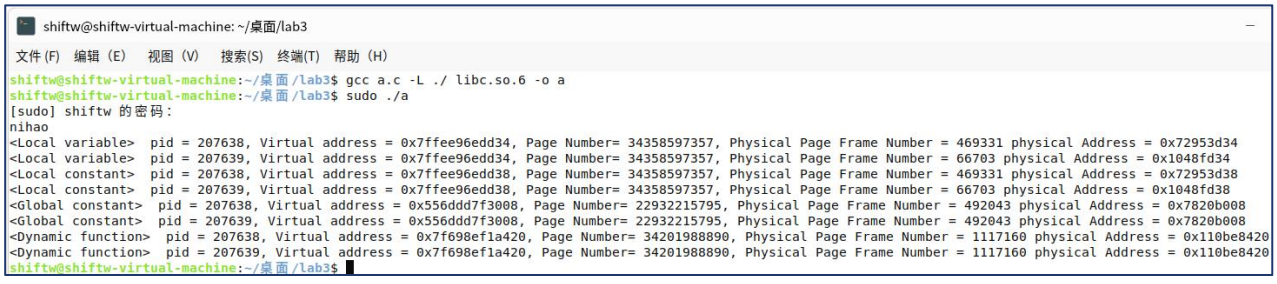
对应的物理页号，进而加上页内偏移量计算出虚拟地址对应的物理地址。

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：

```
#define UL unsigned long
#define U64 uint64_t
char buf[100];
const int a = 20;
const char* name = "nihao";
void fun(char* str, UL pid, UL viraddress, UL* phyaddress)
{
    U64 temp = 0;
    int pageSize = getpagesize();
    UL vir_pageIndex = viraddress / pageSize;
    UL vir_offset = vir_pageIndex * sizeof(U64);
    UL page_offset = viraddress % pageSize; // 虚拟地址在页面中的偏移量
    sprintf(buf, "%s%lu%s", "/proc/", pid, "/pagemap");
    int fd = open(buf, O_RDONLY);                          // 只读打开
    lseek(fd, vir_offset, SEEK_SET);                       // 游标移动
    read(fd, &temp, sizeof(U64));                          // 读取对应项的值
    U64 phy_pageIndex = (((U64)1 << 55) - 1) & temp;       // 物理页号
    *phyaddress = (phy_pageIndex * pageSize) + page_offset; // 加页内偏移量得物理地址
    printf("<%s>  pid = %lu, Virtual address = 0x%lx, Page Number= %lu, Physical Page Frame Number
    sleep(1);
    return;
}
int main()
{
    int b = 1;
    const int d = 3;
    UL phy = 4;
    puts(name);
    int pid = fork();
    fun("Local variable", getpid(), (UL)&b, &phy);
    fun("Local constant", getpid(), (UL)&d, &phy);
    fun("Global constant", getpid(), (UL)&a, &phy);
    fun("Dynamic function", getpid(), (UL)&puts, &phy);
    return 0;
```

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：



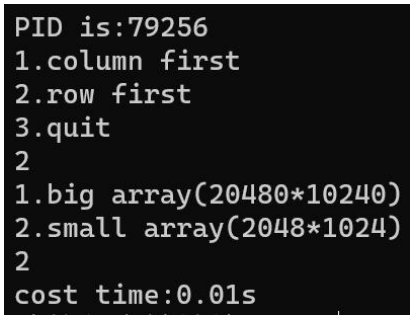

可以观察到局部变量和全局变量在父子进程中映射的物地址不同，但是全局常量是相同的，说明子进程会重新创建变量，但是仍使用父进程的常量。而对于动态链接库，main 中调用 puts 成功打印"nihao"字符串，且父子进程调用的 puts 函数映射到的物理地址相同。

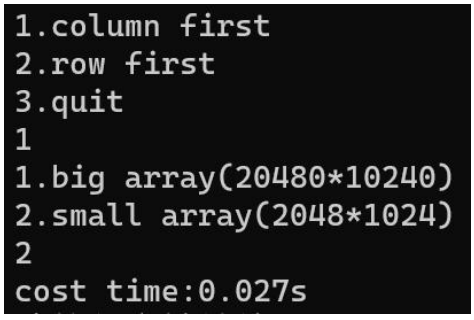
## 四、实验结果

### 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：

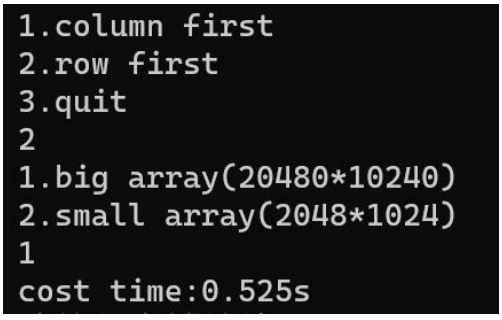
首先是小数组的运行情况，数组大小为 2048*1024，可以由输出结果知先行后列和先列后行的遍历效率差距较大，先行后列的局部性较好，遍历速度快了几倍。但是观察缺页率会发现两种方式一致。

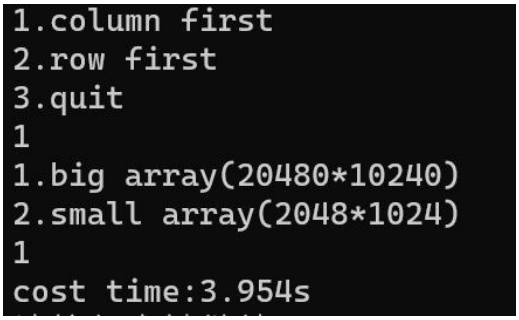



| 名称 | PID | 状态 | 用户名 | CPU | 内存(活动的... | 分页缓冲池 | 非分页缓冲池 | 页面错误 | 页面错误增量 | 体系结构 |
|---|---|---|---|---|---|---|---|---|---|---|
| array.exe | 79256 | 正在运行 | shiftw | 00 | 17,064 K | 1,710 K | 6 K | 5,803 | 0 | x64 |

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。




| 名称 | PID | 状态 | 用户名 | CPU | 内存(活动的... | 分页缓冲池 | 非分页缓冲池 | 页面错误 | 页面错误增量 | 体系结构 |
|---|---|---|---|---|---|---|---|---|---|---|
| array.exe | 79256 | 正在运行 | shiftw | 00 | 819,824 K | 1,710 K | 5 K | 206,508 | 0 | x64 |

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。

```
array size:
1000
select array kind:
1.Random
2.Sequence
1
1.OPT
2.LRU
3.quit
1
visit times:1000 miss times:851 OPT miss rate:85.1%
select array kind:
1.Random
2.Sequence
1
1.OPT
2.LRU
3.quit
2
visit times:1000 miss times:966 LRU miss rate:96.6%
```

```
select array kind:
1.Random
2.Sequence
2
1.OPT
2.LRU
3.quit
1
visit times:1000 miss times:98 OPT miss rate:9.8%
select array kind:
1.Random
2.Sequence
2
1.OPT
2.LRU
3.quit
2
visit times:1000 miss times:100 LRU miss rate:10%
```

### 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：

```
shiftw@shiftw-virtual-machine: ~/桌面/lab3
文件(F)  编辑(E)  视图(V)  搜索(S)  终端(T)  帮助(H)
shiftw@shiftw-virtual-machine:~/桌面/lab3$ gcc a.c -L ./ libc.so.6 -o a
shiftw@shiftw-virtual-machine:~/桌面/lab3$ sudo ./a
[sudo] shiftw 的密码：
nihao
<Local variable>  pid = 207638, Virtual address = 0x7ffee96edd34, Page Number= 34358597357, Physical Page Frame Number = 469331 physical Address = 0x72953d34
<Local variable>  pid = 207639, Virtual address = 0x7ffee96edd34, Page Number= 34358597357, Physical Page Frame Number = 66703 physical Address = 0x1048fd34
<Local constant>  pid = 207638, Virtual address = 0x7ffee96edd38, Page Number= 34358597357, Physical Page Frame Number = 469331 physical Address = 0x72953d38
<Local constant>  pid = 207639, Virtual address = 0x7ffee96edd38, Page Number= 34358597357, Physical Page Frame Number = 66703 physical Address = 0x1048fd38
<Global constant>  pid = 207638, Virtual address = 0x556ddd7f3008, Page Number= 22932215795, Physical Page Frame Number = 492043 physical Address = 0x7820b008
<Global constant>  pid = 207639, Virtual address = 0x556ddd7f3008, Page Number= 22932215795, Physical Page Frame Number = 492043 physical Address = 0x7820b008
<Dynamic function>  pid = 207638, Virtual address = 0x7f698ef1a420, Page Number= 34201988890, Physical Page Frame Number = 1117160 physical Address = 0x110be8420
<Dynamic function>  pid = 207639, Virtual address = 0x7f698ef1a420, Page Number= 34201988890, Physical Page Frame Number = 1117160 physical Address = 0x110be8420
shiftw@shiftw-virtual-machine:~/桌面/lab3$
```

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