《操作系统原理》实验报告

姓名	黄浩岩	学号		专业班级		时间	2020.12.9

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。

二、实验内容

1））编写二维数组遍历程序，理解程序运行局部性的原理。

2）Windows/Linux 模拟实现 OPT,FIFO,LRU 等淘汰算法。

3）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。

三、实验过程

1） 二维数组遍历（TravelArray.cpp）

定义一个容量较大的二维数组 myArray，并分别定义数组的行数 row 和列数 col，以及倍数 times。

定义两个函数 travel1 和 travel2，分别采用先行后列和先列后行的变量方式对数组 myArray 进行遍历。在主函数中调用 travel1 或 travel2，并在调用前后使用 clock 函数记录时间，最终作差求出遍历数组耗时并进行输出。具体代码如下：

```
#include <iostream>

#include <ctime>

using namespace std;



const int times = 1;    //倍数

const int row = 2048 * times;  //行数

const int col = 1024 * times;   //列数
```

```
int myArray[row][col];    //大数组
void travel1();
void travel2();


int main() {
    auto s = clock();
    travel2();
    auto e = clock();
    cout << "Cost " << e - s << "ms\n";     //遍历耗时
    return 0;
}
//先行后列遍历数组
void travel1() {
    for (int i = 0; i < row; ++i) {
        for (int j = 0; j < col; ++j) {
            myArray[i][j] = 0;
        }
    }
}
//先列后行遍历数组
void travel2() {
    for (int i = 0; i < col; ++i) {
        for (int j = 0; j < row; ++j) {
            myArray[j][i] = 0;
        }
```

```
    }
}
```

实验时将主函数分别调用函数 travel1（先行后列遍历）和 travel2（先列后行遍历）分别测试作为一组对照，观察两种不同遍历方式的耗时和操作系统的缺页次数的情况。然后将倍数 times 从 1 递增到 10，即数组 myArray 的大小从[2048][1024]变化到 HF[20480][10240]，纵向对照，观察不同组之间遍历耗时和缺页次数的情况。

注：运行环境为 Windows10 x64 处理器，页面大小为 4KB。

2） Windows 下模拟淘汰算法（Elimination.cpp）

该部分实验主要是通过编写 C 程序利用数组来模拟操作系统中的页面淘汰算法 OPT、FIFO 和 LRU。

首先进行一些全局常量的定义，包括：页面大小 pageSize，此处暂时设置为 16（字节）；进程中的指令数 instrCnt，此处设置为页面大小的 150 倍；物理内存中页框数 pageFrameCnt。以上三个变量均可根据需要进行调整。

然后定义以下全局变量用于记录模拟过程中的一些信息，包括用于记录当前物理内存中已经装有数据的页框数目 cnt；用于记录模拟进程执行过程中缺页次数 pageMissCnt；进程数组 processArr，其容量为 instrCnt，数组中每个元素都表示一个指令，值为一个随机数；带访问的指令序列 orderArr，其每一个元素记录了指令序列号，即在 processArr 数组中的索引；以及用于记录物理内存中每个页框存入页的页号的数组 pageIdx，其容量为页框数 pageFrameCnt；最后 order1 和 order2 分别为个人用于测试用的指令访问序列，运行时可以使用以上两个序列，也可以随机生成访问序列。该部分变量定义的代码如下：

```
const int pageSize = 16;   //页面大小
const int instrCnt = pageSize * 150;          //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 pageMissCnt = 0; //缺页数
int processArr[instrCnt];         //进程数组
```

```
int pageFrame[pageFrameCnt][pageSize];        //页框数组
int orderArr[100];            //指令访问序列
int pageIdx[pageFrameCnt];   //页框中存的页号
//访问序列
const int order1[] = {
    112,455,600,1100,112,456,2000,112,455,600,1100,2001
};
const int order2[] = {
    112,113,455,456,600,601,1100,1101,1102,112,113,114,115,
    456,457,458,2000,2001,2002,2003,112,113,455,456,600,601,
    1100,1101,1102,2001,2002,2003
};
```

接下来定义了一些模拟页面淘汰过程中用到的一些辅助函数，包括：初始化进程中的指令的函数 initProcess；将进程中的一页复制到内存的页框中的函数 copyPage；输出访问序列 orderArr 的函数 showOrderArray；初始化指令访问序列的函数 initProcessOrder，该函数可以设置访问序列为给定的序列，或者随机生成一组访问序列；输出页框状态的函数 showPageFrame，可以输出当前页框中分页的页号、内容。该部分具体代码如下：

```
//初始化进程
inline void initProcess() {
    for (int i = 0; i < instrCnt; ++i) {
        processArr[i] = rand() % 2000;
    }
}
//将页号为 pageNo 分页的数据复制到第 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
}
inline void showOrderArray() {
    cout << "[页号]访问序列:";
    for (int i = 0; i < orderCnt; ++i)
        cout << '[' << orderArr[i] / pageSize << ']'
        << orderArr[i] << ' ';
    putchar('\n');
}//初始化访问序列
void initInstrOrder(const int* srcArr,int size) {
    //若有传入访问序列则复制
    if (srcArr) {
        orderCnt = size;
        for (int i = 0; i < orderCnt; ++i) orderArr[i] = srcArr[i];
        return;
    }
    //否则随机生成
    orderCnt = 20;
    for (int i = 0; i < orderCnt; ++i) {
        orderArr[i] = rand() % instrCnt;
    }
}
```

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

接下来是 3 中淘汰算法 OPT、FIFO、LRU 的代码编写。需要说明的是，对于三种淘汰算法，主要的区别在于当内存中页框已经全部被装满数据时，选择其中一个页框将其中的页面淘汰并更新为新页面的策略是不同的。而对于当前页在页框中则直接从内存中读取数据；当页框未全部占满时直接将页面复制到空页框中；三种算法在处理这两种情况时操作是相同的。

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。一般说来该算法实践中无法实现，而此实验中由于访问的序列长度有限且并不是很长，所以在一定程度上可以实现。

此处的算法实验个人借鉴了编译原理中“目标代码生成”中对变量记录“待用信息和活跃信息”的方法。使用一个待用信息表，表中记录所有需要访问的页面和其待用信息，该待

用信息即该页面的所有访问序号。此处采用一个映射实现，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用信息：逆序遍历一遍指令访问序列，计算出其对应页号，若该页号未在待用信息表中，则表明此时该页面还未被访问过，则将该页号记录到待用信息表中，同时在堆栈中添加当前访问序号；若页号已经出现在表中，则直接将当前访问序号加入待用信息的堆栈中。遍历一遍访问序列后，整个待用信息表就构建完成，序列中所有会访问的页号都会记录在表中，而其对应的待用信息堆栈会记录该页面所有的访问次序，且由栈顶到栈底访问序号递增。

每次进行指令访问时都会先将其页面在待用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。

可以看到，对于该算法的上述实现方法，首先要变量一般指令访问序列来构建待用信息表，而对于指令较多的程序，该步骤是十分耗时的，而且在运行过程中要对表进行实时维护，开销较大，猜测这也可能是 OPT 算法实际无法实现的原因之一。

具体代码如下：

```
//OPT 淘汰算法
void eliminateOPT() {
    //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
        cout << "当前待访问序号: " << orderArr[i]
            << "\t 所在页: " << pageNo
            << "\t 值: " << processArr[orderArr[i]] << endl;
        //每次执行完将该页栈顶的待用信息出栈
        ms.at(pageNo).pop();
        int j;
        //遍历页框若命中
        for (j = 0; j < cnt; ++j) {
            if (pageIdx[j] == pageNo) {
                cout << "命中  " << pageFrame[j][offset] << endl;
                break;
            }
        }
        //若未命中
        if (j == cnt) {
```

```
cout << "缺页  ";
++pageMissCnt; //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);
    cout << pageFrame[maxT][offset] << endl;
}
//页框为全部占满则直接将页复制到空页框
else {
    copyPage(cnt, pageNo);
    cout << pageFrame[cnt][offset] << endl;
    ++cnt;
}
```

```
        }
        //输出每次的页框信息
        showPageFrame();
    }
    cout << "访问总次数:" << orderCnt << "  缺页次数:" << pageMissCnt
        << "  缺页率:" << float(pageMissCnt) / orderCnt << endl;
}
```

FIFO 算法：

该算法即淘汰在内存中停留时间最长的页面。进一步讲，类似于将内存看成一个循环队列，队首元素为停留内存最长时间的页框。因为最初页框中页面是顺序加载进去的，而下次淘汰时 0 号页框停留内存时间最长，作为队首被淘汰；此时 1 号页框的页面停留时间变成了最长，下次就要淘汰该页面，就变成了队首；同理下次淘汰 2 号页框，2 号页面就作为队首。综上分析可知，队首是递增且循环的，由 0 号页框到 pageFrameCnt-1 号页框，再到 0 号页框……

因此，算法实现的过程中只需设置一个变量 front 初值为 0，用于记录当前队列队首的页框号。每次需要淘汰页面时，即将 front 记录的队首页框中的页面进行淘汰，同时队首页框号加 1 即可。

具体代码如下所示：

```
//FIFO 淘汰算法
void eliminateFIFO() {
    int front = 0; //记录队首页框号
    for (int i = 0; i < orderCnt; ++i) {
        auto pageNo = orderArr[i] / pageSize;  //页号
        auto offset = orderArr[i] % pageSize;   //页内偏移
        cout << "当前待访问序号: " << orderArr[i]
            << "\t 所在页: " << pageNo
```

```
        << "\t 值: " << processArr[orderArr[i]] << endl;
    int j;
    //遍历页框若命中
    for (j = 0; j < cnt; ++j) {
        if (pageIdx[j] == pageNo) {
            cout << "命中  " << pageFrame[j][offset] << endl;
            break;
        }
    }
    //若未命中
    if (j == cnt) {
        cout << "缺页  ";
        ++pageMissCnt; //缺页次数+1
        //若页框已全占满
        if (cnt == pageFrameCnt) {
            copyPage(front, pageNo);
            cout << pageFrame[front][offset] << endl;
            front = (front + 1) % pageFrameCnt;
        }
        //页框未全部占满则直接将页复制到空页框
        else {
            copyPage(cnt, pageNo);
            cout << pageFrame[cnt][offset] << endl;
            ++cnt;
        }
```

```
        }
        showPageFrame();
    }
    cout << "访问总次数:" << orderCnt << "  缺页次数:" << pageMissCnt
        << "  缺页率:" << float(pageMissCnt) / orderCnt << endl;
}
```

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，即定义一个数组 timer，用于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 timer 中最大数对应的页框，其页面为最长时间未被使用的页面，则被淘汰。

具体代码如下：

```
//LRU 淘汰算法
void eliminateLRU() {
    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
        cout << "当前待访问序号: " << orderArr[i]
            << "\t 所在页: " << pageNo
            << "\t 值: " << processArr[orderArr[i]] << endl;
        int j;
        //遍历页框若命中
```

```
for (j = 0; j < cnt; ++j) {
    if (pageIdx[j] == pageNo) {
        cout << "命中  " << pageFrame[j][offset] << endl;
        timer[j] = 0;
        break;
    }
}
//若未命中
if (j == cnt) {
    cout << "缺页  ";
    ++pageMissCnt; //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;
        cout << pageFrame[maxT][offset] << endl;
    }
    //页框未全部占满则直接将页复制到空页框
    else {
        copyPage(cnt, pageNo);
```

```
                cout << pageFrame[cnt][offset] << endl;
                ++cnt;
            }
        }
        for (int j = 0; j < cnt; ++j) ++timer[j];
        showPageFrame();
    }
    cout << "访问总次数:" << orderCnt << "  缺页次数:" << pageMissCnt
        << "  缺页率:" << float(pageMissCnt) / orderCnt << endl;
}
```

最后是主函数，其组成较为简单，即首先进行进程数组的初始化和指令访问序列的初始化并输出，然后即使用淘汰算法模拟页面淘汰过程。该部分具体代码如下：

```
int main() {
    srand(time(nullptr));
    initProcess();      //初始化进程
    //初始化指令访问序列
    initInstrOrder(order1, sizeof(order1)/sizeof(int));
    //输出访问序列
    showOrderArray();
    //淘汰算法
    //eliminateFIFO();
    //eliminateLRU();
    eliminateOPT();
    return 0;
```

```
}
```

3）Linux 下计算虚拟地址对应的物理地址（GetPA.c）

该部分实验主要通过 Linux 下的 pagemap 文件获取虚拟地址与物理地址的对应关系，从而计算某个变量或函数对应的虚拟地址。

Linux 的/proc/self/pagemap 文件运行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，首先通过 Linux 提供的 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，进而计算出虚拟地址对应的物理地址。具体代码如下:

```
//获取物理地址
void getMemAddr(unsigned long va) {        //va 为虚拟内存地址
    size_t pageSize = getpagesize();       //页面大小
    unsigned long pageIdx = va / pageSize;    //页号
    unsigned long offset = va % pageSize;     //页内偏移
    uint64_t it;
    FILE* fp;
    printf("Virtual Addr: 0x%lx\n",va);
    printf("Page Index: 0x%lx\nPage Offset: 0x%lx\n", pageIdx, offset);
    //打开当前进程的 pagemap 文件
    if((fp=fopen("/proc/self/pagemap", "rb")) == NULL) {
        printf("Open /proc/self/pagemap Error.\n");
        return;
    }
```

```
    //每项记录为 8 字节，找到当前页号对应的记录
    unsigned long fileOffset = pageIdx * sizeof(uint64_t);
    if(fseek(fp,fileOffset,SEEK_SET)!=0) {
        printf("fSeek Error.\n");
        return;
    }
    //读取记录
    if(fread(&it,sizeof(uint64_t),1,fp) != 1) {
        printf("fread Error.\n");
        return;
    }
    fclose(fp);
    printf("item: 0x%" PRIx64 "\n", it);
    //记录的第 63 位记录当前页面位置：1 为在物理内存中，0 表示不在物理内存中
    if((it >> 63) & 1 == 0) {
        printf("Page Present is 0.\nThe present page isn't in the Physical Memory.\n");
        return;
    }
    //记录的 0-54 位位物理页号
    uint64_t phyPageIdx = (((uint64_t)1 << 55) - 1) & it;
    printf("Physical Page Index: 0x%" PRIx64 "\n",phyPageIdx);
    //物理地址=物理页号*页大小+页内偏移
    unsigned long pa = phyPageIdx * pageSize + offset;
    printf("Physical Addr: 0x%lx\n\n", pa);
}
```

四、实验结果

1） 二维数组遍历（TravelArray.cpp）

通过由小到大修改数组的大小以及分别使用先行后列和先列后行两种方法遍历数组，查看遍历的效率和缺页次数。

以下列出了测试过程中的部分截图，如图 4-1、图 4-2 分别为对 2048*1024 大小的数组先行后列遍历和先列后行的遍历；图 4-3、图 4-4 分别为对 8192*4096 大小的数组先行后列遍历和先列后行遍历；图 4-5、图 4-6 分别为对 16384*8192 大小的数组先行后列遍历和先列后行遍历。

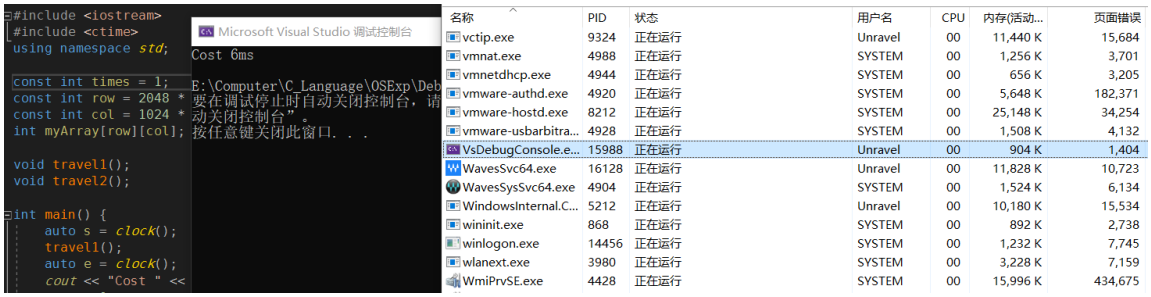

图 4-1 对 2048*1024 大小的数组先行后列遍历

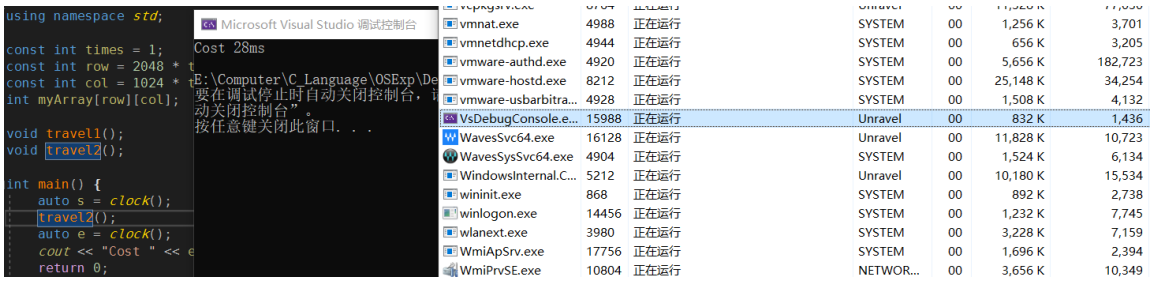

图 4-2 对 2048*1024 大小的数组先列后行遍历

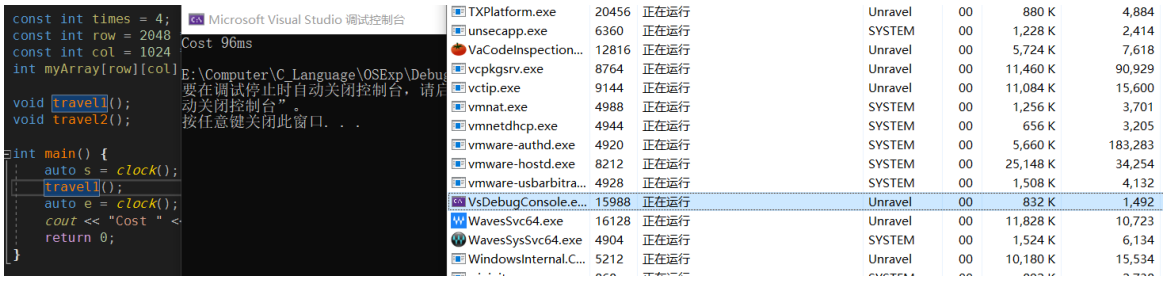

图 4-3 对 8192*4096 大小的数组先行后列遍历

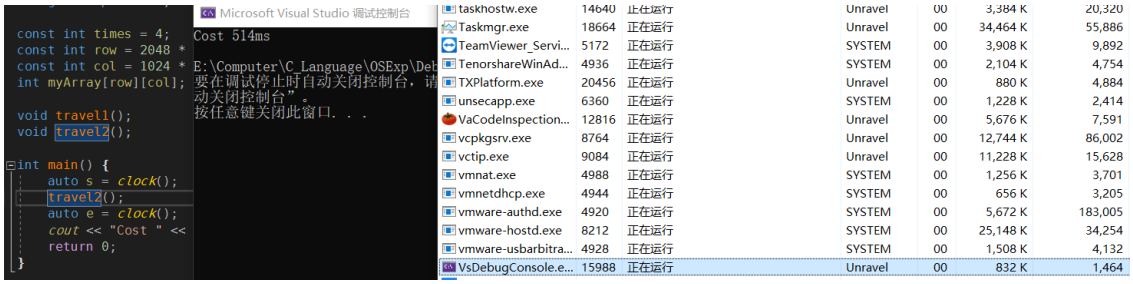

图 4-4 对 8192*4096 大小的数组先列后行遍历

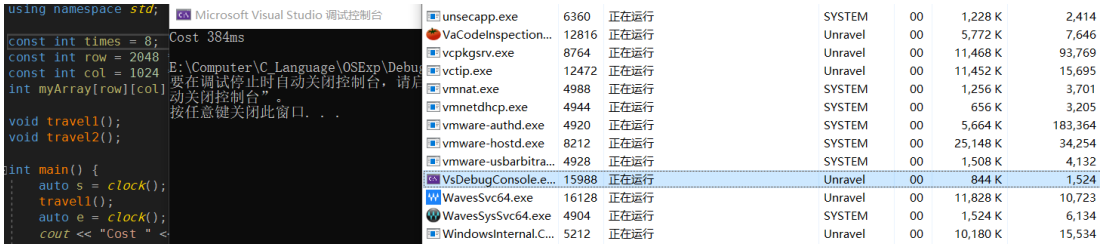

图 4-5 对 16384*8192 大小的数组先行后列遍历

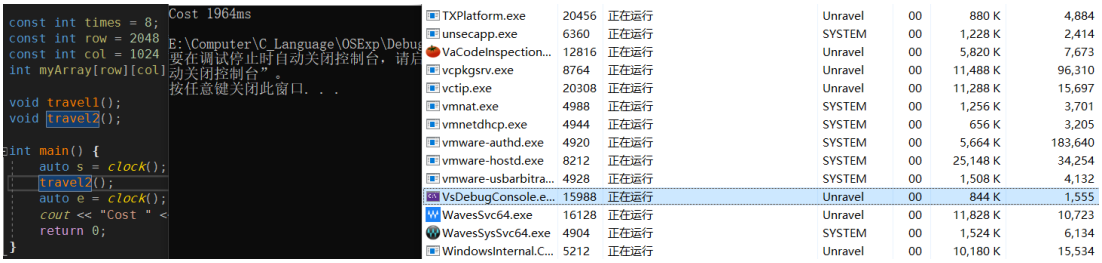

图 4-6 对 16384*8192 大小的数组先列后行遍历

通过比较发现，对于相同大小的数组，先行后列遍历数组的速度明显要快于先列后行遍历数组的速度，用时明显更少。但是缺页次数（任务管理器页面错误项）两者的差距并不大，有时先列后行遍历时缺页次数反而要少一点。而随着数组大小变大，可以看到遍历数组的时间明显变长，但是缺页次数的变化并不是很明显，仅是有少量增加，在数组容量扩大了 8*8=64 倍后也仅比原来缺页次数增加了 100 多次。

2） Windows 下模拟淘汰算法（Elimination.cpp）

淘汰算法模拟时以下均选择 order1 命令访问序列，该序列与课程 PPT 中“A，B，C，D，A，B，E，A，B，C，E，D”的页面访问序列相对应，页框数设定为 3，其缺页次数和缺页率计算结果应与 PPT 中给定结果一致，此处仅为方便验证算法的正确性。

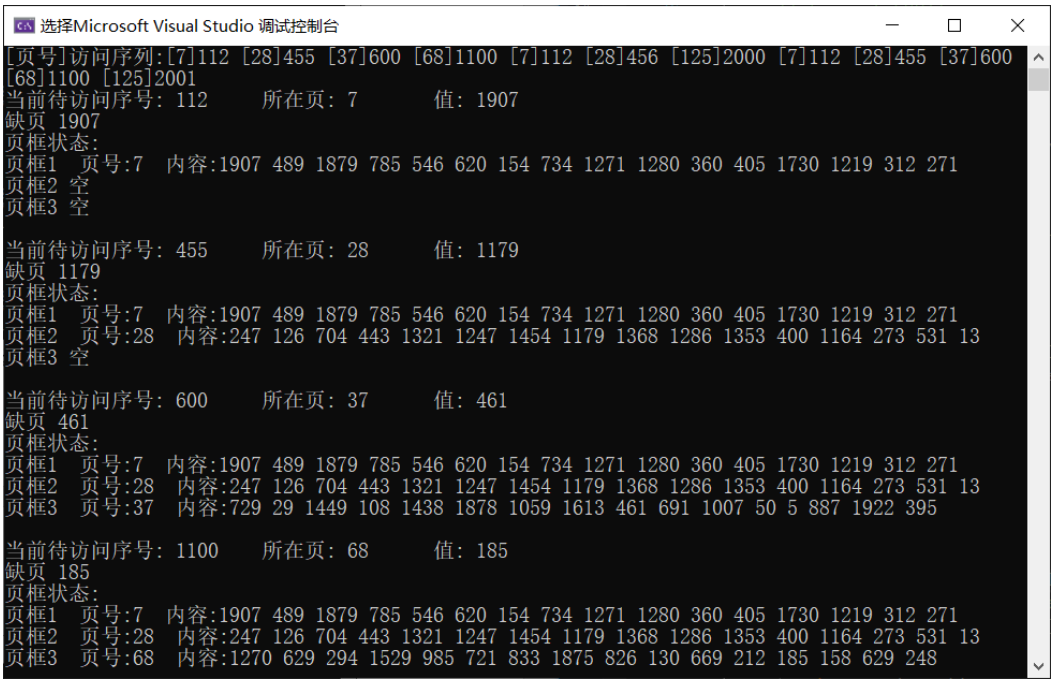

图 4-7 OPT 淘汰算法模拟部分截图 1

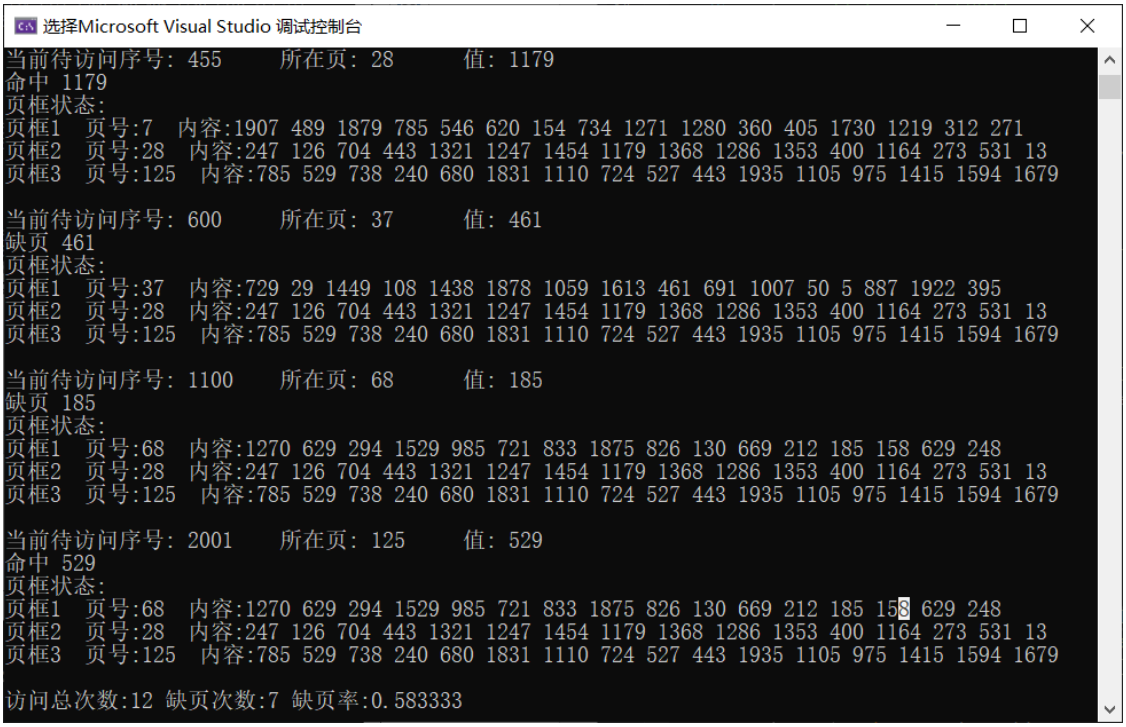

图 4-8 OPT 淘汰算法模拟部分截图 2

如图 4-7 和 4-8 列出了使用 OPT 淘汰算法模拟的部分截图，在第一张图可以看到，当 3 个页框都被占满时， 由于第 7 页和第 28 页接下来就会被访问，而第 37 页要过一段时间才会被访问，因此页框 3 中的 38 页淘汰被 68 页覆盖。

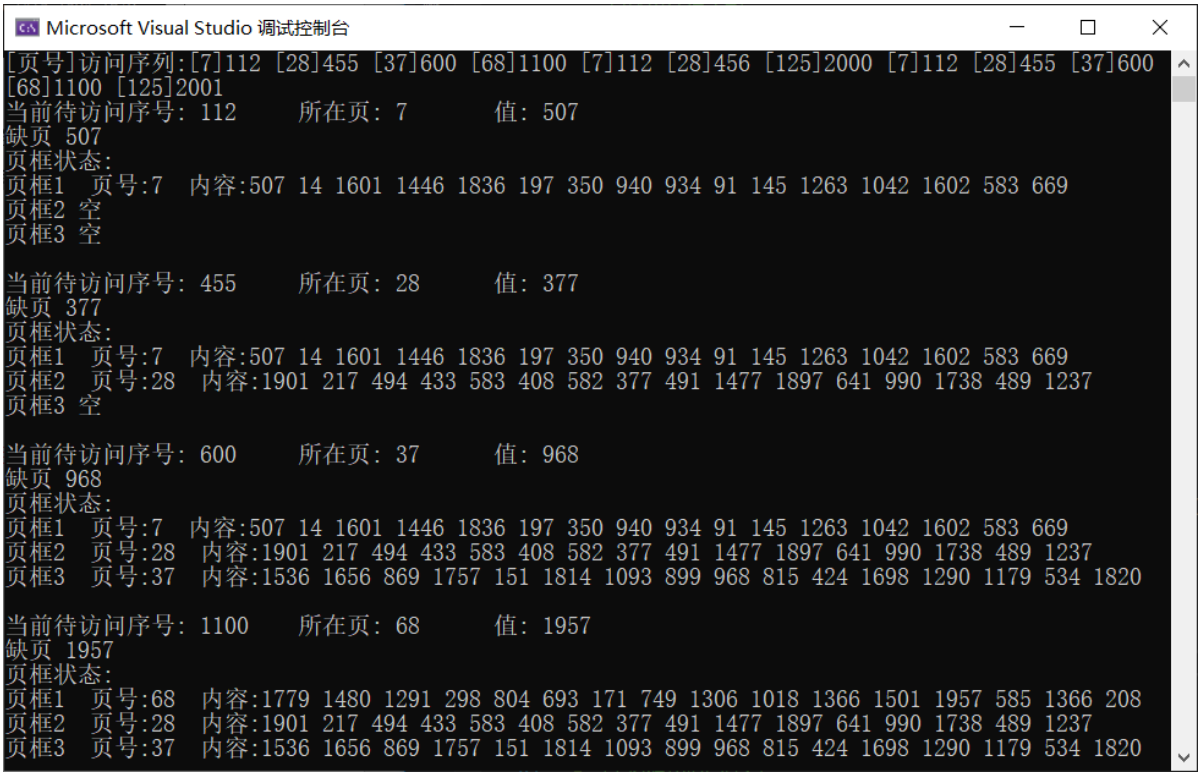

图 4-9 FIFO 淘汰算法模拟部分截图 1

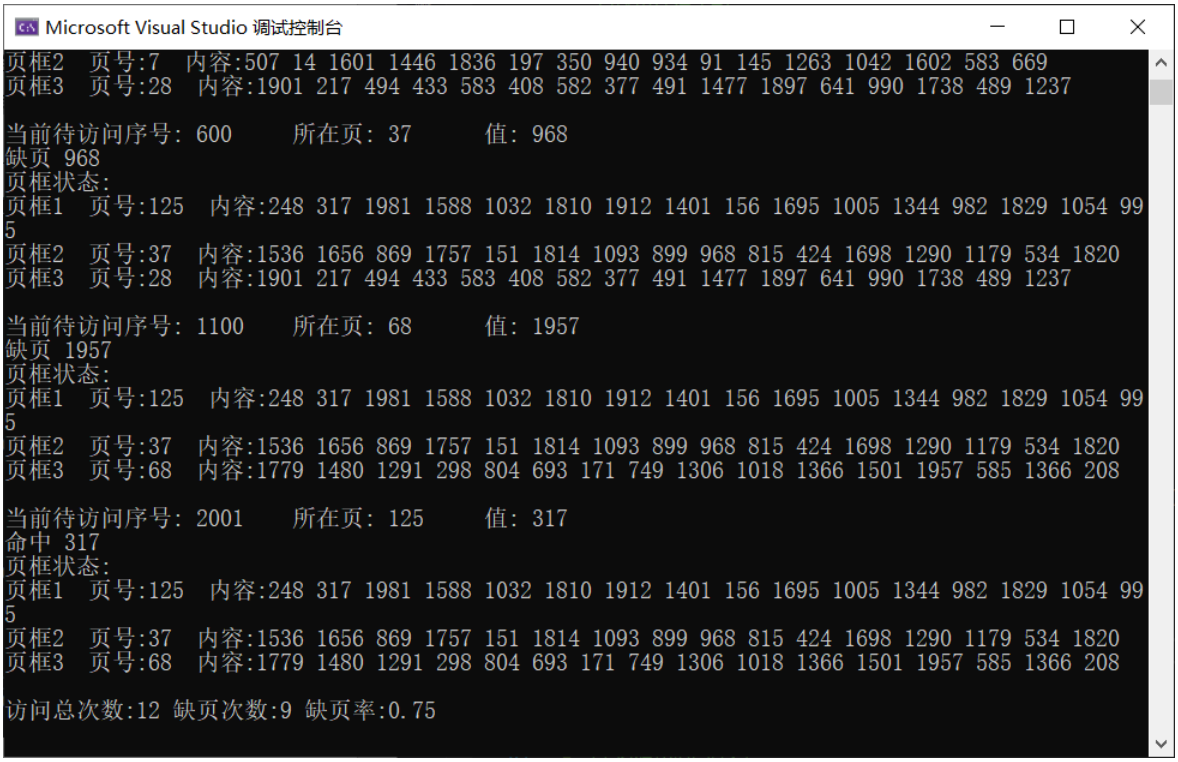

图 4-10 FIFO 淘汰算法模拟部分截图 2

如图 4-9 和 4-10 列出了使用 FIFO 淘汰算法模拟的部分截图，在第一张图可以看到，当 3 个页框都被占满时，由于第 7 页在内存中停留时间最长，因此淘汰，被 68 页覆盖。

如图 4-11 和 4-12 列出了使用 LRU 淘汰算法模拟的部分截图，在第一张图可以看到，当 3 个页框都被占满时，由于第 7 页在内存中最长时间未使用，因此淘汰，被 68 页覆盖。

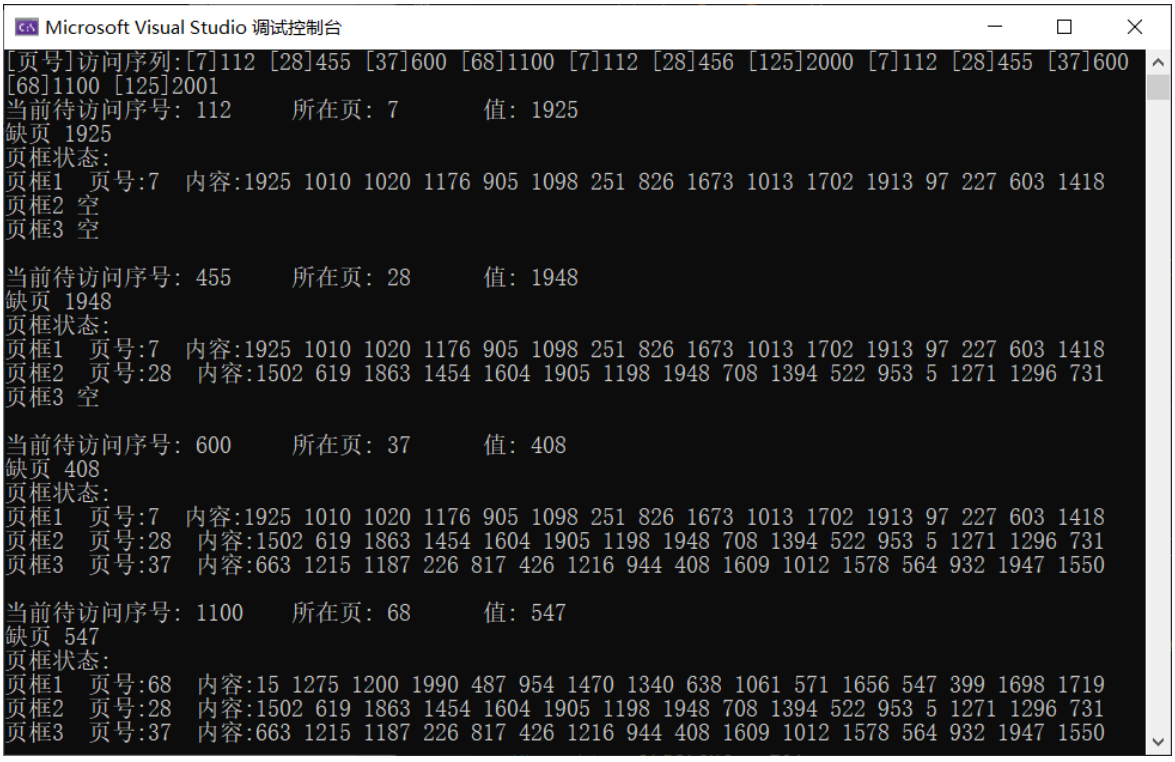

图 4-11 LRU 淘汰算法模拟部分截图 1

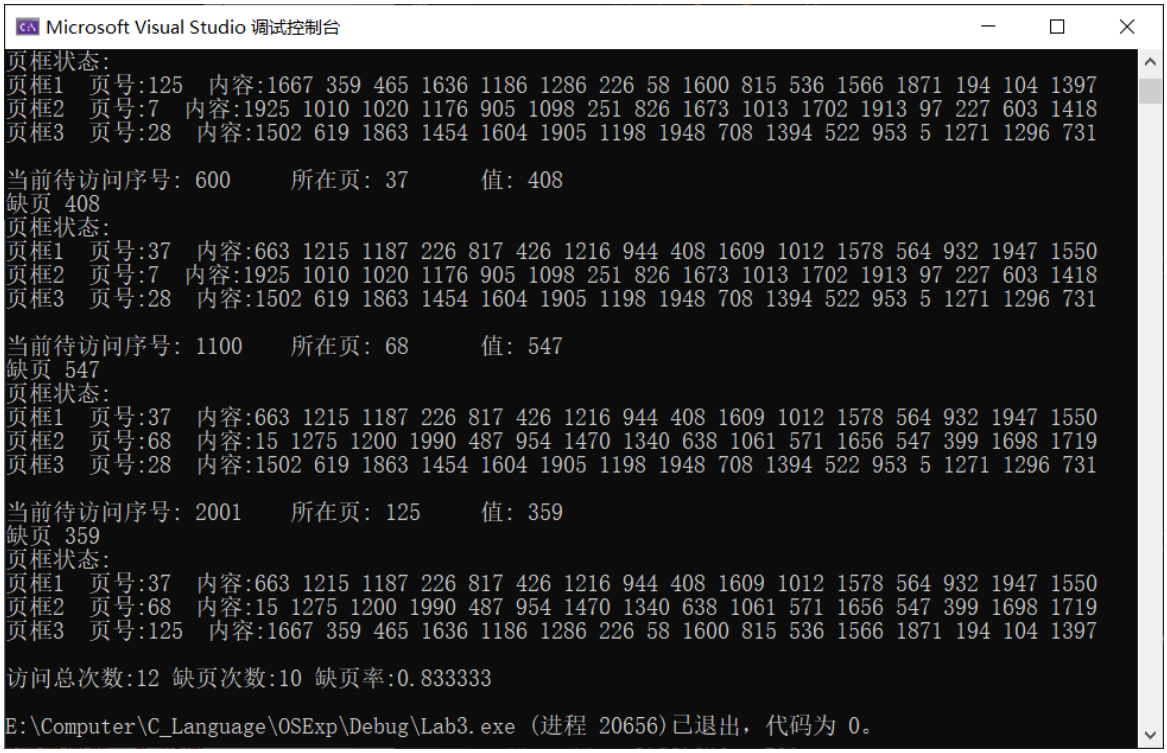

图 4-12 LRU 淘汰算法模拟部分截图 2

3）Linux 下计算虚拟地址对应的物理地址（GetPA.c）

如下代码所示，定义了函数 max，全局变量 a，局部变量 b、c，并使用实验中封装的 getMemAddr 函数获取物理地址。此外，在主函数中使用了一个函数 fork 创建了一个子进程，同父进程一样输出变量 a 和函数 fork 的物理地址。

```
int a = 100;

int max(int a, int b) {

    return a >= b ? a : b;

}


int main() {

    int b = 10, c = 20;

    int pid = fork();

    if(pid == 0) {

        printf("ChildProcess: pid = %d\n", getpid());
```

```
        printf("Global variable - a:\n");
        // printf("a=%d\n", a+=10);
        getMemAddr(&a);
        printf("Function - fork:\n");
        getMemAddr(&fork);
    }
    else {
        printf("Local variable - b:\n");
        getMemAddr(&b);
        printf("Local variable - c:\n");
        getMemAddr(&c);
        printf("Function - max:\n");
        getMemAddr(&max);
        printf("ParentProcess: pid = %d\n", getpid());
        printf("Global variable - a:\n");
        // printf("a=%d\n", a+=5);
        getMemAddr(&a);
        printf("Function - fork:\n");
        getMemAddr(&fork);
    }
    return 0;
}
```

对代码编译并使用 Root 权限运行后，如图 4-13 显示了局部变量 b、c 和函数 max 在物理内存中的地址。如图 4-14 显示了父子进程中对全局变量 a 和 fork 函数的地址，可以看到不仅是 fork 函数的物理地址，两个进程的全局变量 a 的物理地址均相同，这一点与理论上不相符。但当父子进程中分别对全局变量 a 进行修改时（将上述代码中两个 printf 的注释取

消），在运行程序，可以看到如图 4-15 的结果，两个进程的全局变量 a 又拥有了不同的物理地址。此处个人猜测是可能是操作系统为了节省空间，仅当不同进程对变量有不同修改时才会将变量存于新的物理内存中。

```
hhy@hhy-virtual-machine:~/os$ sudo ./GetPA
Local variable - b:
Virtual Addr: 0x7ffd8c746dac
Page Index: 0x7ffd8c746
Page Offset: 0xdac
item: 0x8180000000001da17
Physical Page Index: 0x1da17
Physical Addr: 0x1da17dac

Local variable - c:
Virtual Addr: 0x7ffd8c746db0
Page Index: 0x7ffd8c746
Page Offset: 0xdb0
item: 0x8180000000001da17
Physical Page Index: 0x1da17
Physical Addr: 0x1da17db0

Function - max:
Virtual Addr: 0x55a7c9c1fa98
Page Index: 0x55a7c9c1f
Page Offset: 0xa98
item: 0xa18000000004d953
Physical Page Index: 0x4d953
Physical Addr: 0x4d953a98
```

图 4-13 局部变量 b、c 和函数 max 对应的物理地址

```
hhy@hhy-virtual-machine: ~/os
文件(F)  编辑(E)  查看(V)  搜索(S)  终端(T)  帮助(H)
ParentProcess: pid = 57977
Global variable - a:
Virtual Addr: 0x55d0f84d2010
Page Index: 0x55d0f84d2
Page Offset: 0x10
item: 0x808000000000ee7c
Physical Page Index: 0xee7c
Physical Addr: 0xee7c010

Function - fork:
Virtual Addr: 0x7f0f15eb7820
Page Index: 0x7f0f15eb7
Page Offset: 0x820
item: 0xa0800000000400aa
Physical Page Index: 0x400aa
Physical Addr: 0x400aa820

hhy@hhy-virtual-machine:~/os$ ChildProcess: pid = 57978
Global variable - a:
Virtual Addr: 0x55d0f84d2010
Page Index: 0x55d0f84d2
Page Offset: 0x10
item: 0x818000000000ee7c
Physical Page Index: 0xee7c
Physical Addr: 0xee7c010

Function - fork:
Virtual Addr: 0x7f0f15eb7820
Page Index: 0x7f0f15eb7
Page Offset: 0x820
item: 0xa0800000000400aa
Physical Page Index: 0x400aa
Physical Addr: 0x400aa820
```

图 4-14 父子进程的全局变量 a 和 fork 函数对应的物理地址

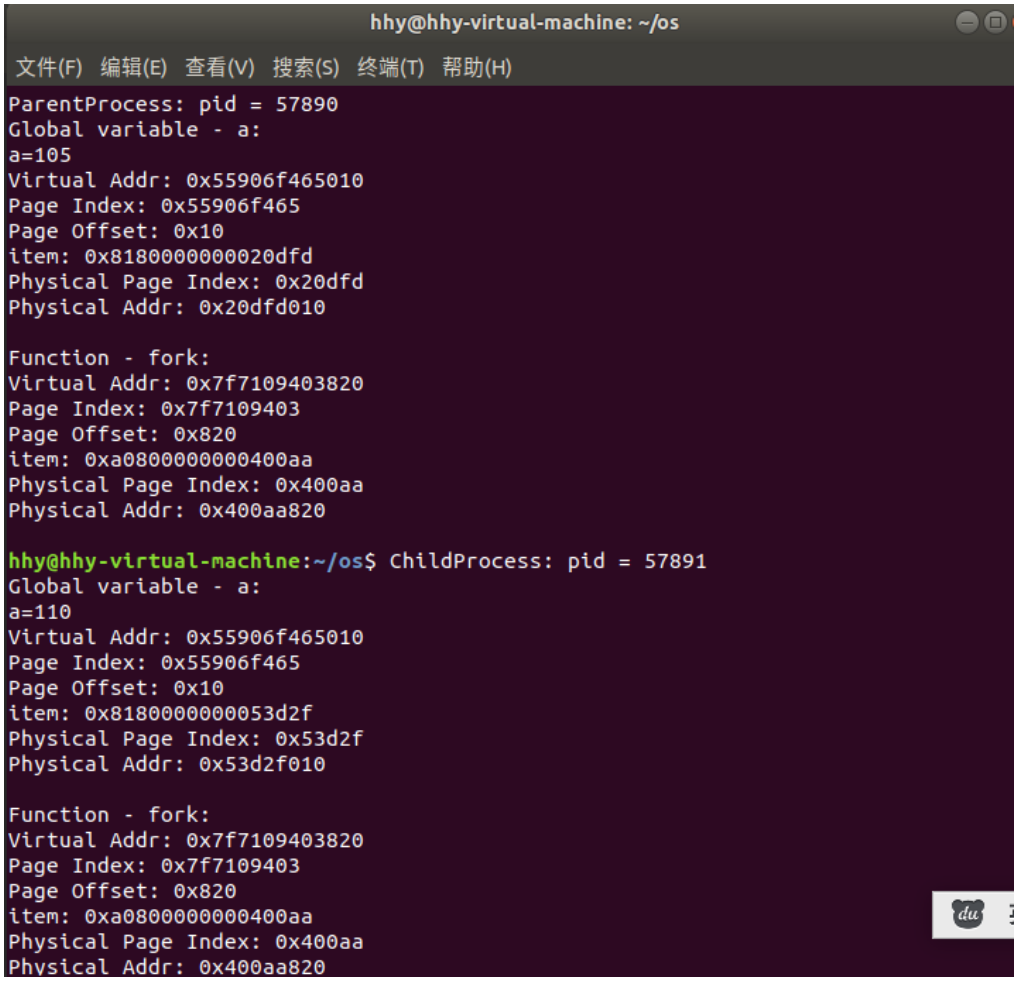

图 4-15 父子进程的全局变量 a 在修改后对应的物理地址

五、体会

1. 本次实验通过以不同方式遍历数组比较效率和缺页次数，进一步理解和验证了程序运行的局部性原理。其中，先行后列遍历数组的速度明显比先列后行遍历数组的速度要快，但是在缺页次数上面，两种方式的差异并不明显，而且即便数组大小发生变化，差异也并不大，这一点与预期结果有较大的出入，且经过验证个人运行的 Win10 环境操作系统的分页大小为 4KB，而实际程序运行的缺页次数均在 1400 上下，感觉比较迷惑。

2. 加深理解了操作系统中的页面淘汰算法的原理，用程序模拟了 OPT、FIFO、LRU 算法页面淘汰的过程。在这个过程中也体会到了 OPT 算法虽然性能最高但是在实际实现过程中的难点和不足，以及 FIFO、LRU 算法各自在选择淘汰页面的特点。

3. 了解学习了 Linux 中用于用户查看进程虚拟页的物理地址信息的 pagemap，通过实验将虚拟地址转换为物理地址，加深了对操作系统分页机制的了解以及如何将虚拟地址转换物理地址的方法。同时在过程中也发现了对于不同进程同一变量的虚拟地址，在变量值

未被修改时对应同一物理地址；而两进程对变量进行不同修改时，其物理地址就会不同。

4. 遇到问题及解决：

1）问题：实验 1 如何查看程序的缺页次数。

解决：经过查阅资料，在 Win10 打开任务管理器，选择“详细信息”一栏，在列表的标题栏上右键“选择列”，对任务管理器中进程显示的信息进行调整，从中找到“页面错误”一项选择并确定，即可在任务管理器查看每个进程的页面错误即缺页次数的统计。

2）问题：实验 3 运行编写的程序后得到的物理页号总是 0。

解决：经过仔细阅读文档“Documentation/vm/pagemap.txt”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。因此需要在执行程序时加上“sudo”才能够正确运行程序得到变量正确的物理页号。