

# 《操作系统原理》实验报告

姓名	侯皓斐	学号	U202010851	专业班级	软工 2003 班	时间	2022.5.24
----	-----	----	------------	------	-----------	----	-----------

## 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 程序局部性实验

开发环境：银河麒麟操作系统 v10 桌面版

根据程序局部性原理，我们知道程序局部性越好，缺页中断次数越少，运行效率也越高，运行时间也越短。

而 C 语言中的二维数组以行序为先，则易知按行遍历其程序局部性较好。

```
11     for(int i = 0; i < 10240; i++)
12         for(int j = 0; j < 20480; j++)
13             MyArray[i][j] = 0;
```

按列编译则程序局部性较差。

```

11     for(int i = 0; i < 20480; i++)
12         for(int j = 0; j < 10240; j++)
13             MyArray[j][i] = 0;

```

可用 C 语言中自带的 `clock()` 函数测量时间。使用 `/proc` 文件系统，`cat /proc/pid/stat`，其第 10 位到第 14 位依次是 `min_flt`，`cmin_flt`，`maj_flt`，`cmaj_flt`，分别表示该任务不需要从硬盘拷数据而发生的缺页(次缺页)的次数，累计的该任务的所有的 `waited-for` 进程曾经发生的次缺页的次数目，该任务需要从硬盘拷数据而发生的缺页(主缺页)的次数，累计的该任务的所有的 `waited-for` 进程曾经发生的主缺页的次数目，累计求和获得总缺页次数。

```

moonlight@moonlight-VMware: ~/test3/Locality
文件(F) 编辑(E) 视图(V) 搜索(S) 终端(T) 帮助(H)
moonlight@moonlight-VMware:~/test3/Locality$ ./good.out
now process id is 3999
time=0.570188
[]

moonlight@moonlight-VMware:~/test3/Locality
文件(F) 编辑(E) 视图(V) 搜索(S) 终端(T) 帮助(H)
moonlight@moonlight-VMware:~/test3/Locality$ cat /proc/3999/stat
3999 (good.out) R 3945 3999 3945 34816 3999 4194304 204882 0 0 0 7040 31 0 0 20
0 1 0 139064 841416704 205159 18446744073709551615 94555322425344 94555322426069
140723753947200 0 0 0 0 0 0 0 0 17 2 0 0 0 0 94555322437032 94555322437648
94556178980864 140723753952169 140723753952180 140723753952180 140723753955309 0
moonlight@moonlight-VMware:~/test3/Locality$

```

采用控制变量法，多次测试数组大小，二维数组遍历方式，64/32 位编译方式对时间和缺页次数的影响。获得一系列数据。

```

moonlight@moonlight-VMware: ~/test3/Locality
文件(F) 编辑(E) 视图(V) 搜索(S) 终端(T) 帮助(H)
moonlight@moonlight-VMware:~/test3/Locality$ gcc -m32 good.c -o good32.out
moonlight@moonlight-VMware:~/test3/Locality$ ./good32.out
now process id is 24512
time=0.720880
[]

moonlight@moonlight-VMware:~/test3/Locality
文件(F) 编辑(E) 视图(V) 搜索(S) 终端(T) 帮助(H)
moonlight@moonlight-VMware:~/test3/Locality$ cat /proc/24512/stat
24512 (good32.out) R 24457 24512 24457 34816 24512 4194304 204873 0 0 0 3139 81
0 0 20 0 1 0 240932 841379840 205117 18446744073709551615 1448775680 1448776564
4288911648 0 0 0 0 0 0 0 17 1 0 0 0 0 1448787664 1448787976 2299924480 428
8914339 4288914352 4288914352 4288917483 0
moonlight@moonlight-VMware:~/test3/Locality$

```

应注意，Linux 环境下 `gcc` 默认编译为 64 位程序，使用 `-m32` 参数编译 32 位程序。

### 3.2 模拟页式地址映射（页面淘汰算法）

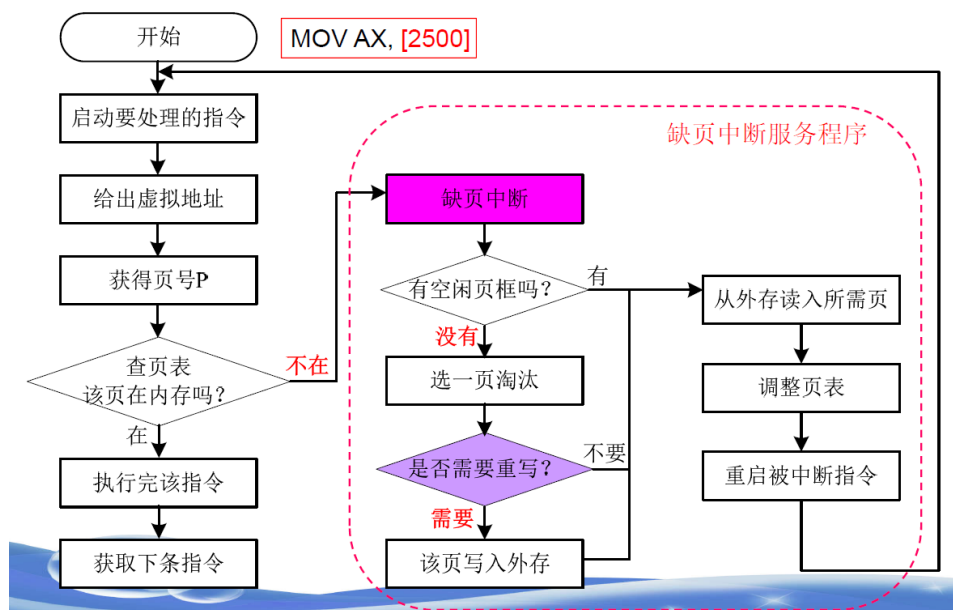
开发环境：银河麒麟操作系统 v10 桌面版

我们使用 C 语言编写的小程序模拟指令执行过程中的页式地址映射机制（采用遍历数组的操作来模拟）。

我们用 1 个较大数组 A（例如 2400 个元素）模拟进程，数组里面放的是随机数，按照某种顺序，如顺序，跳转，分支，循环，或随机，来访问数组 A（不直接访问 A，而是通过页式地址映射），模仿指令的执行。而用二维数组 B 模拟内存中的物理存储单元，B[m][n] 其中 m 为页框号，n 为偏移地址，访问元素时应该先检查其是否在物理内存中，若不在需要将元素所在页面复制进入 B 数组。我们因此需要构建一个页表（与真正的页表不同，我们存放的是页框号和页号的对应关系，其中也多存储了页框号最近被访问的时间，页框号被复制进入 B 数组的时间等一系列辅助信息）

```
int N = 2400, M = 10, pagesize = 10, Algori = 0; FILE *p; //访问数目, 页框数目, 页面大小
int displayAlgori[4] = {0, 0, 0, 0};
int displayOrder[6] = {0, 0, 0, 0, 0, 0};
int A[1000000], B[100][200], PG[100][3], visorder[1000000], totVistime; // 0 页框号对应的装填
void init(); //初始化各种参数, 指令数目, 页面大小, 页框数, 访问次序, 淘汰算法
int OPT(int nowVistime); int FIFO(int nowVistime); int LRU(int nowVistime);
double sequential(); double jump(); double branch(); double loop(); double rand0m();
int (*Algorithm[4])(int) = {NULL, OPT, FIFO, LRU};
double (*Run[6])() = {NULL, sequential, jump, branch, loop, rand0m};
char* eliminateStr[4] = {"", "OPT", "FIFO", "LRU"};
char* visitStr[6] = {"", "顺序", "跳转", "分支", "循环", "随机"};
```

我们指定顺序访问数组。访问数组时发生了如下过程。



- 1.通过 `pagesize` 得到虚拟地址对应的页号与偏移地址。
- 2.检查页表中是否存在对应的页号。若有则得到页框号更新信息，并跳转。
- 3.检查页表中是否有空白的页框，若有直接将页复制进入 **B** 中对应页框，得到页框号更新信息，并跳转。
- 4.按照指定的算法选择页面淘汰，并将页复制进入 **B** 中对应页框，得到页框号更新信息，并跳转。
- 5.得到页框号 `m` 与偏移地址 `n`，直接读出 `B[m][n]`。

因此我们可以编写如下页式地址映射机制下的访存过程。（未显式包含缺页中断）

```
int visArray(int id, int lastvis) {
    int page = id / M, bias = id % M, nopage = 1;
    for(int i = 0; i < M; i++)
        if(PG[i][0] == page) {
            nopage = 0;
            page = i;
            PG[i][1] = lastvis;
            break;
        }
    if(nopage == 1) {
        int hasfreepage = 0; int i = 0;
        for(i = 0; i < M; i++) {
            if(PG[i][0] == -1) {
                hasfreepage = 1;
                break;
            }
        }
        if(hasfreepage == 0)
            i = Algorithm[Algori](lastvis);
        PG[i][0] = page;
        PG[i][1] = lastvis;
        PG[i][2] = lastvis;
        for(int j = 0; j < M; j++)
            B[i][j] = A[page*M + j];
        page = i;
    }
    fprintf(p, "%d : %d %d\n", id, B[page][bias], nopage);
    return nopage;
}
```

其中一句 `i = Algorithm[Algori](lastvis);` 便通过规定的算法获得了被淘汰的页面。我们下面逐个实现 OPT, FIFO, LRU 算法。

其中 OPT 淘汰算法，淘汰不再需要或最远将来才会用到的页面，其需要知道未来访问的页面序列，为此需要在每个访问方法中做出调整，事先规定访问序列，得到页面序列后，再度按序访问数组。

```

int OPT(int nowVistime) {
    int finalvistime[1000];
    for(int i = 0; i < M; i++)
        finalvistime[i] = totVistime;
    for(int i = 0; i < M; i++) {
        int j = totVistime-1;
        for( ; j > nowVistime; j--)
            if(PG[i][0] == visorder[j]) {
                finalvistime[i] = j;
                break;
            }
    }
    int despage = 0;
    for(int i = 0; i < M; i++)
        if(finalvistime[despage] < finalvistime[i])
            despage = i;
    return despage;
}

```

FIFO 算法，淘汰在内存中**停留时间最长**的页面。我们借助页表中的信息。

```

int FIFO(int nowVistime) {
    int despage = 0;
    for(int i = 0; i < M; i++)
        if(PG[despage][2] > PG[i][2])
            despage = i;
    return despage;
}

```

LRU 算法，淘汰**最长时间未被使用**的页面。我们借助页表中的信息。

```

int LRU(int nowVistime) {
    int despage = 0;
    for(int i = 0; i < M; i++)
        if(PG[despage][1] > PG[i][1])
            despage = i;
    return despage;
}

```

然后实现了各种访问序列，这里以顺序访问为例。其为保证 OPT 算法的正常运行应事先规定顺序，而后保存 visorder，再按顺序访问。最后返回了缺页率。

```

double sequential() {
    int nowVistime = 0, nopagefault = 0; totVistime = 0;
    fprintf(p, "顺序访问过程\n");
    for(int i = 0; i < N; i++) {
        visorder[totVistime] = i/M;
        totVistime++;
    }
    for(int i = 0; i < N; i++) {
        nopagefault += visArray(i, nowVistime);
        nowVistime++;
    }
    return (double)nopagefault/totVistime;
}

```

### 3.4 Linux 计算物理地址

开发环境：银河麒麟操作系统 v10 桌面版

借助 Linux 的 `/proc/pid/maps` 文件允许用户查看当前进程虚拟页的物理地址等相关信息。从而实现虚拟地址到物理地址的转换。我们通过

```
unsigned long v_offset = v_pageIndex * sizeof(uint64_t);
lseek(fd, v_offset, SEEK_SET);
read(fd, &item, sizeof(uint64_t));
```

这样几句指令便可以实现寻找虚拟页号对应的物理页框号的过程。

则我们可以写下如下函数以得到 Linux 虚拟地址对应的物理地址。

注意运行时，使用 `sudo` 获得权限。

```
void getphysicaladdr(unsigned long pid, unsigned long vaddr) {
    unsigned long paddr = 0;
    int pageSize = getpagesize();

    unsigned long v_pageIndex = vaddr / pageSize;
    unsigned long v_offset = v_pageIndex * sizeof(uint64_t);
    unsigned long page_offset = vaddr % pageSize;
    uint64_t item = 0;
    sprintf(buf, "%s%lu%s", "/proc/", pid, "/pagemap");
    int fd = open(buf, O_RDONLY);
    lseek(fd, v_offset, SEEK_SET);
    read(fd, &item, sizeof(uint64_t));

    uint64_t phy_pageIndex = (((uint64_t)1 << 55) - 1) & item;
    paddr = (phy_pageIndex * pageSize) + page_offset; //再加上页内偏移量就得到了物理地址
    printf("pid = %lu, 虚拟地址 = 0x%lx, 所在页号 = %lu, 物理地址 = 0x%lx, 所在物理页框号 = %lu\n",
        pid, vaddr, v_pageIndex, paddr, phy_pageIndex);
    return ;
}
```

(1) 如何通过扩充实验展示不同进程的同一虚拟地址对应不同的物理地址？

创建父子进程，使父子进程的区别仅在于输出不同，保证其局部变量，全局变量等虚拟地址保持相同，输出他们对应的物理地址，从而验证同一虚拟地址对应不同的物理地址。

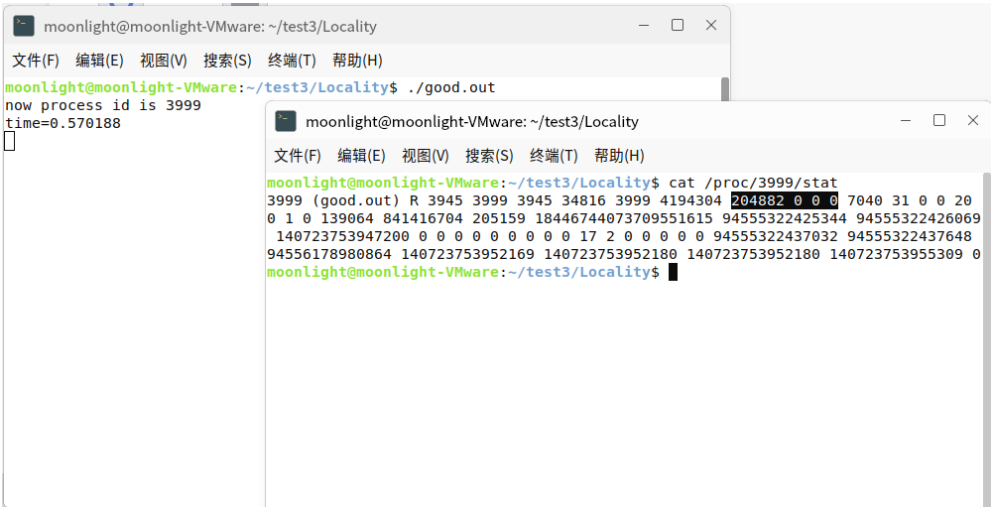
(2) 如何扩充实验验证不同进程的共享库具有同一物理地址。

同时运行两个已知 `pid` 的进程。利用 `/proc/pid/maps` 文件系统找到其共享库，然后根据 `pid` 和虚拟地址检查是否物理地址相同。

## 四、实验结果

### 4.1 程序局部性实验

一次运行局部性好的（按行遍历）的程序遍历较大的数组，具体的运行情况如下。



我们采用控制变量法，多次测试数组大小，二维数组遍历方式，64/32 位编译方式对时间和缺页次数的影响。获得一系列数据。

64 位程序	局部性好	局部性差	32 位程序	局部性好	局部性差
10240×20480	0.570188 204882	2.990578 204882	10240×20480	0.720880 204873	3.232463 204873
2048×2048	0.011767 4180	0.053986 4179	2048×2048	0.010280 4170	0.055174 4168

综上可看到，64 位进程和 32 位进程对缺页次数没有影响，而 32 位程序大部分情况下会慢一些，因为我们使用的设备是 64 位的 PC，32 位的程序在 64 位的系统下运行不仅不能发挥系统和硬件的性能，还迫使系统采用“虚拟”的方式营造出一个 32 位的环境给程序，所以效率更低。

而程序局部性对缺页次数也没有较大的影响（？），却对运行时间产生了较大的影响。  
而数组越大，遍历所需时间和缺页次数自然会变多。

4.2 模拟页式地址映射（页面淘汰算法）

编译并运行我们的模拟程序，按照默认的参数 2400 个数组单元，页面大小为 10，页框为 10 个运行。



```

moonlight@moonlight-VMware:~/test3/Algorithm$ ./a.out
输入0默认模拟2400个指令，10个页框，页面大小为10。输入1调整指令数目和页框数目和页面大小
0
输入0展示全部实现的淘汰算法。输入N>0，然后输入N个数(1-3)分别表示想要得到展示的淘汰算法。其中1表示OPT算法，2表示FIFO，3表示LRU算法。
0
输入0展示全部实现的访问次序。输入N>0，然后输入N个数(1-5)分别表示想要得到展示的淘汰算法。其中1-5分别表示顺序，跳转，分支，循环，或随机
0

```

运行结果如下：

```

[顺序访问]:
OPT算法缺页率 = 0.100000
FIFO算法缺页率 = 0.100000
LRU算法缺页率 = 0.100000
[跳转访问]:
OPT算法缺页率 = 0.096298
FIFO算法缺页率 = 0.100748
LRU算法缺页率 = 0.100748
[分支访问]:
OPT算法缺页率 = 0.105201
FIFO算法缺页率 = 0.105201
LRU算法缺页率 = 0.105201
[循环访问]:
OPT算法缺页率 = 0.011158
FIFO算法缺页率 = 0.011158
LRU算法缺页率 = 0.011158
[随机访问]:
OPT算法缺页率 = 0.860000
FIFO算法缺页率 = 0.947500
LRU算法缺页率 = 0.952500

```

我们可以看到在大多数的情况下三个算法的缺页率并无较大差别，而 OPT 算法一定是缺页率最低的算法。由于随机访问毫无局部性，其缺页率自然最高。而且顺序访问的缺页率一定为  $1/10=0.100000$ ，符合常理。

#### 4.4 Linux 计算物理地址.

我们使用我们编写的 phyaddrget.c 中的函数为基础探究这两个问题。

(1) 如何通过扩充实验展示不同进程的同一虚拟地址对应不同的物理地址？

创建父子进程，使父子进程的区别仅在于输出不同，保证其局部变量，全局变量等虚拟地址保持相同，输出他们对应的物理地址，从而验证同一虚拟地址对应不同的物理地址。我们得到了运行结果如下：



```
moonlight@moonlight-VMware:~/test3/pagemap$ sudo ./a.out
[进程 1]
[全局常量]pid = 99783, 虚拟地址 = 0x558ca69b908c, 所在页号 = 22964496825, 物理地址 = 0x6eadb08c, 所在物理页框号 = 453339
[全局变量]pid = 99783, 虚拟地址 = 0x558ca69bb010, 所在页号 = 22964496827, 物理地址 = 0x5a06b010, 所在物理页框号 = 368747
[全局函数]pid = 99783, 虚拟地址 = 0x558ca69b842f, 所在页号 = 22964496824, 物理地址 = 0x5dc2442f, 所在物理页框号 = 384036
[局部变量]pid = 99783, 虚拟地址 = 0x7ffe6595edc4, 所在页号 = 34358057310, 物理地址 = 0x31718dc4, 所在物理页框号 = 202520
[局部静态变量]pid = 99783, 虚拟地址 = 0x558ca69bb024, 所在页号 = 22964496827, 物理地址 = 0x5a06b024, 所在物理页框号 = 368747
[局部常量]pid = 99783, 虚拟地址 = 0x7ffe6595edc8, 所在页号 = 34358057310, 物理地址 = 0x31718dc8, 所在物理页框号 = 202520
[进程 2]
[全局常量]pid = 99782, 虚拟地址 = 0x558ca69b908c, 所在页号 = 22964496825, 物理地址 = 0x6eadb08c, 所在物理页框号 = 453339
[全局变量]pid = 99782, 虚拟地址 = 0x558ca69bb010, 所在页号 = 22964496827, 物理地址 = 0x1c281010, 所在物理页框号 = 115329
[全局函数]pid = 99782, 虚拟地址 = 0x558ca69b842f, 所在页号 = 22964496824, 物理地址 = 0x5dc2442f, 所在物理页框号 = 384036
[局部变量]pid = 99782, 虚拟地址 = 0x7ffe6595edc4, 所在页号 = 34358057310, 物理地址 = 0x4c822dc4, 所在物理页框号 = 313378
[局部静态变量]pid = 99782, 虚拟地址 = 0x558ca69bb024, 所在页号 = 22964496827, 物理地址 = 0x1c281024, 所在物理页框号 = 115329
[局部常量]pid = 99782, 虚拟地址 = 0x7ffe6595edc8, 所在页号 = 34358057310, 物理地址 = 0x4c822dc8, 所在物理页框号 = 313378
```

我们可以得到结论如下：

全局常量和全局函数的物理地址在父子进程中一致。

而全局变量，局部变量，局部静态变量，局部常量在父子进程中均不一致。

（2）如何扩充实验验证不同进程的共享库具有同一物理地址。

同时运行两个已知 pid 的进程。利用 /proc/pid/maps 文件系统找到其共享库，然后根据 pid 和虚拟地址检查是否物理地址相同。

```
moonlight@moonlight-VMware:~/test3/pagemap
文件(F) 编辑(E) 视图(V) 搜索(S) 终端(T) 帮助(H)
moonlight@moonlight-VMware:~/test3/pagemap$ sudo cat /proc/66213/maps
[sudo] moonlight 的密码:
561040e9e000-561040e9f000 r--p 00000000 08:07 6685285 /home/moonlight/test3/pagemap/b.out
561040e9f000-561040ea0000 r-xp 00001000 08:07 6685285 /home/moonlight/test3/pagemap/b.out
561040ea0000-561040ea1000 r--p 00002000 08:07 6685285 /home/moonlight/test3/pagemap/b.out
561040ea1000-561040ea2000 r--p 00002000 08:07 6685285 /home/moonlight/test3/pagemap/b.out
561040ea2000-561040ea3000 rw-p 00003000 08:07 6685285 /home/moonlight/test3/pagemap/b.out
56104130f000-561041330000 rw-p 00000000 08:00 0 [heap]
7fdbbe679000-7fdbbe69e000 r--p 00000000 08:05 4599838 /usr/lib/x86_64-linux-gnu/libc-2.31.so
7fdbbe69e000-7fdbbe816000 r-xp 00025000 08:05 4599838 /usr/lib/x86_64-linux-gnu/libc-2.31.so
7fdbbe816000-7fdbbe860000 r--p 0019d000 08:05 4599838 /usr/lib/x86_64-linux-gnu/libc-2.31.so
7fdbbe860000-7fdbbe861000 ---p 001e7000 08:05 4599838 /usr/lib/x86_64-linux-gnu/libc-2.31.so
7fdbbe861000-7fdbbe864000 r--p 001e7000 08:05 4599838 /usr/lib/x86_64-linux-gnu/libc-2.31.so
7fdbbe864000-7fdbbe867000 rw-p 001ea000 08:05 4599838 /usr/lib/x86_64-linux-gnu/libc-2.31.so

moonlight@moonlight-VMware:~/test3/pagemap
文件(F) 编辑(E) 视图(V) 搜索(S) 终端(T) 帮助(H)
moonlight@moonlight-VMware:~/test3/pagemap$ sudo cat /proc/66214/maps
[sudo] moonlight 的密码:
561040e9e000-561040e9f000 r--p 00000000 08:07 6685285 /home/moonlight/test3/pagemap/b.out
561040e9f000-561040ea0000 r-xp 00001000 08:07 6685285 /home/moonlight/test3/pagemap/b.out
561040ea0000-561040ea1000 r--p 00002000 08:07 6685285 /home/moonlight/test3/pagemap/b.out
561040ea1000-561040ea2000 r--p 00002000 08:07 6685285 /home/moonlight/test3/pagemap/b.out
561040ea2000-561040ea3000 rw-p 00003000 08:07 6685285 /home/moonlight/test3/pagemap/b.out
56104130f000-561041330000 rw-p 00000000 08:00 0 [heap]
7fdbbe679000-7fdbbe69e000 r--p 00000000 08:05 4599838 /usr/lib/x86_64-linux-gnu/libc-2.31.so
7fdbbe69e000-7fdbbe816000 r-xp 00025000 08:05 4599838 /usr/lib/x86_64-linux-gnu/libc-2.31.so
7fdbbe816000-7fdbbe860000 r--p 0019d000 08:05 4599838 /usr/lib/x86_64-linux-gnu/libc-2.31.so
7fdbbe860000-7fdbbe861000 ---p 001e7000 08:05 4599838 /usr/lib/x86_64-linux-gnu/libc-2.31.so
7fdbbe861000-7fdbbe864000 r--p 001e7000 08:05 4599838 /usr/lib/x86_64-linux-gnu/libc-2.31.so
7fdbbe864000-7fdbbe867000 rw-p 001ea000 08:05 4599838 /usr/lib/x86_64-linux-gnu/libc-2.31.so
7fdbbe867000-7fdbbe86d000 rw-p 00000000 08:00 0 /usr/lib/x86_64-linux-gnu/ld-2.31.so
7fdbbe86d000-7fdbbe88e000 r--p 00000000 08:05 4599574 /usr/lib/x86_64-linux-gnu/ld-2.31.so
7fdbbe88e000-7fdbbe8b1000 r-xp 00001000 08:05 4599574 /usr/lib/x86_64-linux-gnu/ld-2.31.so
7fdbbe8b1000-7fdbbe8b9000 r--p 00024000 08:05 4599574 /usr/lib/x86_64-linux-gnu/ld-2.31.so
7fdbbe8b9000-7fdbbe8bb000 r--p 0002c000 08:05 4599574 /usr/lib/x86_64-linux-gnu/ld-2.31.so

moonlight@moonlight-VMware:~/test3/pagemap
文件(F) 编辑(E) 视图(V) 搜索(S) 终端(T) 帮助(H)
moonlight@moonlight-VMware:~/test3/pagemap$ gcc test.c -o a.out
moonlight@moonlight-VMware:~/test3/pagemap$ ./a.out
[进程 2]运行 pid = 66213
[进程 1]运行 pid = 66214
```

我们注意到父子进程（pid 分别为 66213, 66214）共享着 /usr/lib/x86-64-linux-gnu/libc-2.31.so 共享库。其虚拟地址在其左侧为相等，使用程序验证得物理地址页相同。

## 五、实验错误排查和解决方法

### 5.1 程序局部性实验

我们采用控制变量法，多次测试数组大小，二维数组遍历方式，64/32 位编译方式对时

间和缺页次数的影响。获得一系列数据。

64 位程序	局部性好	局部性差	32 位程序	局部性好	局部性差
10240× 20480	0.570188 204882	2.990578 204882	10240× 20480	0.720880 204873	3.232463 204873
2048×2048	0.011767 4180	0.053986 4179	2048×2048	0.010280 4170	0.055174 4168

我们观察到而程序局部性对缺页次数也没有较大的影响（？），却对运行时间产生了较大的影响。这对于我来说较难理解？实际上经过查阅相关资料，我得到了几种可能的原因。一是编译器的自然优化，编译器对一些行为有着优化，通过调整（例如主动反转矩阵）可能会保证缺页率的降低。二是多级存储体系的参与使程序局部性的分析更加复杂，cache 内存取的信息有着复杂的管理机制，提高了访存效率。三是/proc 文件系统的缺页数统计并不完全，或者是我对资料的理解出现了偏差，导致了错误的结果。

而后面的几次测试中也偶尔出现了令人诧异的效果。32 位进程也不一定就要比 64 位进程慢，这个时间分析较为复杂。

```
moonlight@moonlight-VMware:~/test3/Locality$ ./good.out
now process id is 25149
time=0.572343
^C
moonlight@moonlight-VMware:~/test3/Locality$ ./good.out
now process id is 25151
time=0.570656
^C
moonlight@moonlight-VMware:~/test3/Locality$ ./good32.out
now process id is 25154
time=0.487492
^C
moonlight@moonlight-VMware:~/test3/Locality$ ./good32.out
now process id is 25156
time=0.484286
^C
```

5.2 模拟页式地址映射（页面淘汰算法）

一开始我对题目出现了错误的理解，本以为老师只是想我们去模拟考试中可能会出现

的填表题目，实现过程中发现不对劲，便重新利用函数指针实现了较好的逻辑结构和重构。

后来想要完成一整套的模拟，我首先将问题拆分，一个重要的地方就是访存过程，借助页式地址映射机制，完成了如下流程。

- 1.通过 pagesize 得到虚拟地址对应的页号与偏移地址。

2.检查页表中是否存在对应的页号。若有则得到页框号更新信息，并跳转。

3.检查页表中是否有空白的页框，若有直接将页复制进入 B 中对应页框，得到页框号更新信息，并跳转。

4.按照指定的算法选择页面淘汰，并将页复制进入 B 中对应页框，得到页框号更新信息，并跳转。

5.得到页框号 m 与偏移地址 n，直接读出 B[m][n]。

编写代码的过程中也出现了一些小 bug。一开始时，顺序访问各算法的缺页率均不为 0.1。

经过 debug 发现了两处错误。

<pre>if(nopage == 1) {     int hasfreepage = 0;int i = 0;     for(i = 0; i &lt; M; i++) {         if(PG[i][0] == 0) {             hasfreepage = 1;             break;         }     }     if(hasfreepage == 0)         i = Algorithm[Algori](lastvis); }</pre>	<pre>if(nopage == 1) {     int hasfreepage = 0;int i = 0;     for(i = 0; i &lt; M; i++) {         if(PG[i][0] == -1) {             hasfreepage = 1;             break;         }     }     if(hasfreepage == 0)         i = Algorithm[Algori](lastvis); }</pre>
----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------	-----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------

模仿缺页中断的程序部分中，遍历全部页表查看是否有空页框，而我们实际上页号是有 0 的，我们不能将 0 作为空页框的标志，我们将 PG[i][0]初始值设为-1 即可。

<pre>int OPT(int nowVistime) {     int finalvistime[1000];     for(int i = 0; i &lt; M; i++)         finalvistime[i] = totVistime;     for(int i = 0; i &lt; M; i++) {         int j = totVistime-1;         for(; j &gt;= nowVistime; j--)             if(PG[i][0] == visorder[j]) {                 finalvistime[i] = j;                 break;             }     }     int despage = 0;     for(int i = 0; i &lt; M; i++)         if(finalvistime[despage] &lt; finalvistime[i])             despage = i;     return despage; }</pre>	<pre>int OPT(int nowVistime) {     int finalvistime[1000];     for(int i = 0; i &lt; M; i++)         finalvistime[i] = totVistime;     for(int i = 0; i &lt; M; i++) {         int j = totVistime-1;         for(; j &gt; nowVistime; j--)             if(PG[i][0] == visorder[j]) {                 break;             }         finalvistime[i] = j;     }     int despage = 0;     for(int i = 0; i &lt; M; i++)         if(finalvistime[despage] &lt; finalvistime[i])             despage = i;     return despage; }</pre>
------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------	---------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------

OPT 算法的实现如上，我当时一时脑抽，将>错打为>=，按照我们对于 OPT 算法的理解，其中 OPT 淘汰算法，淘汰不再需要或最远将来才会用到的页面，其固然这次访问不会被计算，如果一个元素这次访问完便也不再被访问，这就是错误的一种情况。

### 5.3 Linux 缺页处理程序增加调试信息

这个任务难度十分大，最后并没有完成。但我仔细阅读了 Linux5.17.5 内核的缺页处理相关程序，对进程调度，缺页终端等概念有了更深的理解。

我们还是先回到这个任务。新进的 Linux 内核中缺页处理程序在/arch/x86/mm/fault.c 中。其名为 handle\_page\_fault。其会区分中断发生于用户态还是核态，然后对应执行相应处理。

```
1470 static __always_inline void
1471 handle_page_fault(struct pt_regs *regs, unsigned long error_code,
1472                  unsigned long address)
1473 {
1474     trace_page_fault_entries(regs, error_code, address);|
1475
1476     if (unlikely(kmmio_fault(regs, address)))
1477         return;
1478
1479     /* Was the fault on kernel-controlled part of the address space? */
1480     if (unlikely(fault_in_kernel_space(address))) {
1481         do_kern_addr_fault(regs, error_code, address);
1482     } else {
1483         do_user_addr_fault(regs, error_code, address);
1484         /*
1485          * User address page fault handling might have reenabled
1486          * interrupts. Fixing up all potential exit points of
1487          * do_user_addr_fault() and its leaf functions is just not
1488          * doable w/o creating an unholy mess or turning the code
1489          * upside down.
1490          */
1491         local_irq_disable();
1492     }
1493 }
```

为了采用 printk 方式添加调试信息，打印特定进程（以可程序的程序名为过滤条件）的缺页信息，统计缺页次数。我们需要从其参数中获得缺页进程相关信息，而 pt\_regs 结构体就存储着这样的信息。

```
struct pt_regs {
    long ebx;           //可执行文件路径的指针 (regs.ebx中)
    long ecx;           //命令行参数的指针 (regs.ecx中)
    long edx;           //环境变量的指针 (regs.edx中)。
```

如何调出其中信息，网友提供了如下方法：

```
1471 handle_page_fault(struct pt_regs *regs, unsigned long error_code,
1472                  unsigned long address)
1473 {
1474     trace_page_fault_entries(regs, error_code, address);
1475
1476     char *filename = (char *)PT_REGS_PARM2(regs);|
1477     char rightname[50] = "/home/moonlight/test3/testKernel/test.out";
1478     unsigned long allsame = 1, i = 0;
1479     while(i <= 41) {
1480         if(filename[i] != rightname[i]) {
1481             allsame = 0;
1482             break;
1483         }
1484         i = i + 1;
1485     }
1486     if(allsame == 1)
1487         printk("testpage.out page fault!");
```

```

arch/x86/mm/fault.c: In function 'handle_page_fault':
arch/x86/mm/fault.c:1476:27: error: implicit declaration of function 'PT_REGS_PARM2'
RM2' [-Werror=implicit-function-declaration]
1476 | char *filename = (char *)PT_REGS_PARM2(regs);
    |                               ^~~~~~
arch/x86/mm/fault.c:1476:19: warning: cast to pointer from integer of different
size [-Wint-to-pointer-cast]
1476 | char *filename = (char *)PT_REGS_PARM2(regs);
    |                               ^
arch/x86/mm/fault.c:1476:2: warning: ISO C90 forbids mixed declarations and code
[-Wdeclaration-after-statement]
1476 | char *filename = (char *)PT_REGS_PARM2(regs);
    | ^~~~~
cc1: some warnings being treated as errors
make[2]: *** [scripts/Makefile.build:288: arch/x86/mm/fault.o] 错误 1
make[1]: *** [scripts/Makefile.build:550: arch/x86/mm] 错误 2
make: *** [Makefile:1831: arch/x86] 错误 2

```

但其具体实现应该是依赖于特定的环境，当前 Linux 内核中缺少这样的宏。

我按照文档的描述尝试。却出现了编译错误。

```

1471 handle_page_fault(struct pt_regs *regs, unsigned long error_code,
1472                    unsigned long address)
1473 {
1474     trace_page_fault_entries(regs, error_code, address);
1475
1476     char *filename = (char *)regs->ebx;
1477     char rightname[50] = "/home/moonlight/test3/testKernel/test.out";
1478     unsigned long allsame = 1, i = 0;
1479     while(i <= 41) {
1480         if(filename[i] != rightname[i]) {
1481             allsame = 0;
1482             break;
1483         }
1484         i = i + 1;
1485     }
1486     if(allsame == 1)
1487         printk("testpage.out page fault!");

```

```

arch/x86/mm/fault.c: In function 'handle_page_fault':
arch/x86/mm/fault.c:1476:33: error: 'struct pt_regs' has no member named 'ebx';
did you mean 'bx'?
1476 | char *filename = (char *)regs->ebx;
    |                               ^~~
    |                               bx
arch/x86/mm/fault.c:1476:2: warning: ISO C90 forbids mixed declarations and code
[-Wdeclaration-after-statement]
1476 | char *filename = (char *)regs->ebx;
    | ^~~~~
make[2]: *** [scripts/Makefile.build:288: arch/x86/mm/fault.o] 错误 1
make[1]: *** [scripts/Makefile.build:550: arch/x86/mm] 错误 2
make: *** [Makefile:1831: arch/x86] 错误 2
make: *** 正在等待未完成的任务....
^Cmake[2]: *** [scripts/Makefile.build:550: drivers/nfc/pn544] 中断
make[1]: *** [scripts/Makefile.build:550: drivers/nfc] 中断
make[1]: *** [scripts/Makefile.build:550: fs/xfs] 中断

```

遵循编译器修改后，甚至简单的 `printk` 一下路径信息。系统更换内核后都会无法启动。

```

1471 handle_page_fault(struct pt_regs *regs, unsigned long error_code,
1472                    unsigned long address)
1473 {
1474     trace_page_fault_entries(regs, error_code, address);
1475
1476     printk("%s", (char*)regs->bx);
1477
1478     if (unlikely(kmmio_fault(regs, address)))
1479         return;
1480
1481     /* Was the fault on kernel-controlled part of the address space? */
1482     if (unlikely(fault_in_kernel_space(address))) {
1483         do_kern_addr_fault(regs, error_code, address);
1484     } else {
1485         do_user_addr_fault(regs, error_code, address);
1486     }
1487     /*
1488     * User address page fault handling might have reenabled
1489     * interrupts. Fixing up all potential exit points of
1490     * do_user_addr_fault() and its leaf functions is just not
1491     * doable w/o creating an unholy mess or turning the code
1492     * upside down.
1493     */
1494     local_irq_disable();
1495 }

```



```

003, bcdDevice= 1.03
[ 1.335948] T67] usb 2-1: New USB device strings: Mfr=1, Product=2, Serial
Number=0
[ 1.336020] T67] usb 2-1: Product: VMware Virtual USB Mouse
[ 1.336074] T67] usb 2-1: Manufacturer: VMware
[ 1.475592] T67] usb 2-2: new full-speed USB device number 3 using uhci_hc
d
[ 1.659276] T67] usb 2-2: New USB device found, idVendor=0e0f, idProduct=0
002, bcdDevice= 1.00
[ 1.659376] T67] usb 2-2: New USB device strings: Mfr=1, Product=2, Serial
Number=0
[ 1.659448] T67] usb 2-2: Product: VMware Virtual USB Hub
[ 1.659500] T67] usb 2-2: Manufacturer: VMware, Inc.
[ 1.673248] T67] hub 2-2:1.0: USB hub found
[ 1.679260] T67] hub 2-2:1.0: 7 ports detected
[ 2.020885] T62] usb 2-2.1: new full-speed USB device number 4 using uhci_
hcd
[ 2.189741] T62] usb 2-2.1: New USB device found, idVendor=0e0f, idProduct
=0000, bcdDevice= 1.00
[ 2.189880] T62] usb 2-2.1: New USB device strings: Mfr=1, Product=2, Seri
alNumber=3
[ 2.189966] T62] usb 2-2.1: Product: Virtual Bluetooth Adapter
[ 2.190019] T62] usb 2-2.1: Manufacturer: VMware
[ 2.190063] T62] usb 2-2.1: SerialNumber: 000650260328

```

与苏曙光老师交流后决定暂时搁置这个问题。

## 5.4 Linux 计算物理地址.

/proc 文件系统的访问需要较高特权级。一定要加 **sudo**!! 不加 **sudo** 不能产生正确结果。

```

moonlight@moonlight-VMware:~/test3/pagemap$ gcc pagemap.c -o a.out
moonlight@moonlight-VMware:~/test3/pagemap$ ./a.out
[进程 1]
[全局常量]pid = 96954, 虚拟地址 = 0x55e64e049010, 所在页号 = 23058505801, 物理地址 = 0x10, 所在物理页框号 = 0
[全局函数]pid = 96954, 虚拟地址 = 0x55e64e04642f, 所在页号 = 23058505798, 物理地址 = 0x42f, 所在物理页框号 = 0
[局部变量]pid = 96954, 虚拟地址 = 0x7ffc63714794, 所在页号 = 34355951380, 物理地址 = 0x794, 所在物理页框号 = 0
[局部静态变量]pid = 96954, 虚拟地址 = 0x55e64e049024, 所在页号 = 23058505801, 物理地址 = 0x24, 所在物理页框号 = 0
[局部常量]pid = 96954, 虚拟地址 = 0x7ffc63714798, 所在页号 = 34355951380, 物理地址 = 0x798, 所在物理页框号 = 0
[进程 2]
[全局常量]pid = 96953, 虚拟地址 = 0x55e64e049010, 所在页号 = 23058505801, 物理地址 = 0x10, 所在物理页框号 = 0
[全局函数]pid = 96953, 虚拟地址 = 0x55e64e04642f, 所在页号 = 23058505798, 物理地址 = 0x42f, 所在物理页框号 = 0
[局部变量]pid = 96953, 虚拟地址 = 0x7ffc63714794, 所在页号 = 34355951380, 物理地址 = 0x794, 所在物理页框号 = 0
[局部静态变量]pid = 96953, 虚拟地址 = 0x55e64e049024, 所在页号 = 23058505801, 物理地址 = 0x24, 所在物理页框号 = 0
[局部常量]pid = 96953, 虚拟地址 = 0x7ffc63714798, 所在页号 = 34355951380, 物理地址 = 0x798, 所在物理页框号 = 0

```

## 六、实验参考资料和网址

- (1) 教学课件：感谢华中科技大学软件学院苏曙光老师
- (2) 苏曙光. 操作系统原理（慕课版）[M]. 北京：人民邮电出版社，2022.
- (3) [https://blog.csdn.net/qq\\_46106285/article/details/121768610](https://blog.csdn.net/qq_46106285/article/details/121768610)
- (4) [https://blog.csdn.net/fz\\_ywj/article/details/8109368](https://blog.csdn.net/fz_ywj/article/details/8109368)
- (5) <https://www.cnblogs.com/arnoldlu/p/10272466.html>
- (7) <https://zhuanlan.zhihu.com/p/55371024>
- (8) [https://blog.csdn.net/q\\_l\\_s/article/details/52749657](https://blog.csdn.net/q_l_s/article/details/52749657)
- (9) <https://zhidao.baidu.com/question/369342402938620044.html>

- (10) [https://blog.csdn.net/weixin\\_33933118/article/details/94033705](https://blog.csdn.net/weixin_33933118/article/details/94033705)
- (11) <https://blog.csdn.net/cunfen6312/article/details/107682994>
- (12) <https://blog.csdn.net/astonqa/article/details/7613091>
- (13) [https://blog.csdn.net/weixin\\_45948376/article/details/121707692](https://blog.csdn.net/weixin_45948376/article/details/121707692)
- (14) <https://blog.csdn.net/h2763246823/article/details/122559742>
- (15) <https://blog.csdn.net/helloworlddm/article/details/76785294>
- (16) <https://cloud.tencent.com/developer/article/1459526>
- (17) <https://www.cnblogs.com/wanghetao/archive/2011/11/06/2238310.html>
- (18) <https://www.cnblogs.com/liuhailong0112/p/14916456.html>
- (19) <https://www.coder.work/article/1547461>
- (20) <https://blog.csdn.net/qq78442761/article/details/81843631>
- (21) <https://mozillazg.com/2022/05/ebpf-libbpfgo-use-perfbuf-map.html>
- (22) <https://www.cnblogs.com/botoo/p/9667594.html>