



# 上海大学

SHANGHAI UNIVERSITY

## 操作系统（二）实验报告

组 号	第 6 组
学号姓名	20121034 胡才郁
实验序号	实验 3
日 期	2022 年 2 月 11 日

## 一 实验目的与要求

### 目的

提高存储器的效率始终是操作系统研究的重要课题之一，虚拟存储技术是用来扩大内存容量的一种重要方法。使用高级语言编写几个常用的存储分配算法，并设计一个存储管理的模拟程序，对各种算法进行分析比较，评测其性能优劣，从而加深对这些算法的了解。

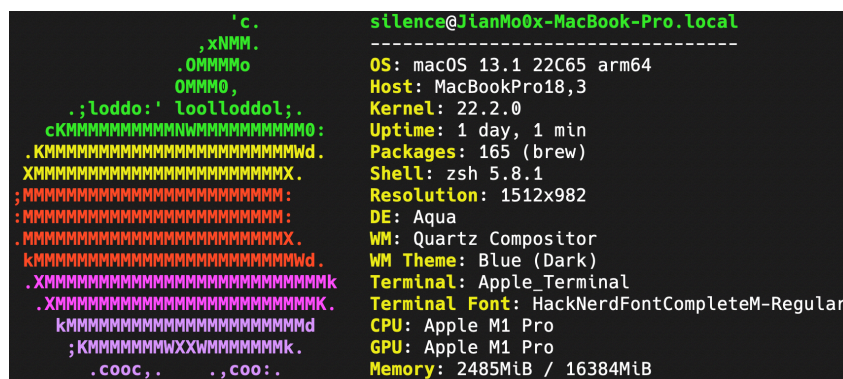
### 要求

为了比较真实地模拟存储管理，可预先生成一个大致符合实际情况的指令地址流。然后模拟这样一种指令序列的执行来计算和分析各种算法的访问命中率。

## 二 实验环境

- 操作系统：macOS
- 处理器架构：ARM

其余主要实验环境配置如图 1 所示。



```
'c.
,xNMM.
.OMMMMo
OHHMM,
.;loddol:' loolloddol;.
cKMMMMMMMMMMMMMMMMMMMMMMMMMM0:
.KMMMMMMMMMMMMMMMMMMMMMMMMMMd.
XMMMMMMMMMMMMMMMMMMMMMMMMMMX.
;MMMMMMMMMMMMMMMMMMMMMMMMMM:
;MMMMMMMMMMMMMMMMMMMMMMMMMM:
.MMMMMMMMMMMMMMMMMMMMMMMMMMX.
kMMMMMMMMMMMMMMMMMMMMMMMMMwd.
.XMMMMMMMMMMMMMMMMMMMMMMMMMMk
.XMMMMMMMMMMMMMMMMMMMMMMMMMMk.
kMMMMMMMMMMMMMMMMMMMMMMMMMdd
;KMMMMMMMMMMMMMMMMMMMMMMk.
.cooc,. .,coo:.
```

```
silence@JianMo0x-MacBook-Pro.local
-----
OS: macOS 13.1 22C65 arm64
Host: MacBookPro18,3
Kernel: 22.2.0
Uptime: 1 day, 1 min
Packages: 165 (brew)
Shell: zsh 5.8.1
Resolution: 1512x982
DE: Aqua
WM: Quartz Compositor
WM Theme: Blue (Dark)
Terminal: Apple_Terminal
Terminal Font: HackNerdFontCompleteM-Regular
CPU: Apple M1 Pro
GPU: Apple M1 Pro
Memory: 2485MiB / 16384MiB
```

图 1: 实验环境

## 三 实验内容及其设计与实现

在本实验中，采用页式分配存储管理方案，并通过分析计算不同页面淘汰算法情况下的访问命中率来比较各种算法的优劣。

另外也考虑到改变页面大小和实际存储器容量对计算结果的影响，从而可为算则好的算法、合适的页面尺寸和实存容量提供依据。

对于随机生成一个页面序列  $(P_1, P_2, \dots, P_n)$  分别编程实现 OPT、FIFO、LRU 算法三种算法，同时计算三种算法的缺页率。

缺页率，即缺页次数与访问页面次数之比。假设页面序列长为  $n$ ，系统为进程分配  $m$  个物理块，如果缺页  $n'$  次，则缺页率为  $n'/n * 100\%$ 。对于同一算法，应比较分配的物理块数不同时的缺页率；不同的算法，应比较分配的物理块数相同时的缺页率。

## 1 页面序列的构建

为了比较真实地模拟存储管理，可预先生成一个大致符合实际情况的指令地址流。然后模拟这样一种指令序列的执行来计算和分析各种算法的访问命中率。在本实验中，采用如下的方法生成指令地址流。

- 50% 的指令是顺序执行的。
- 25% 的指令均匀散布在前地址部分。
- 25% 的指令均匀散布在后地址部分。

对于均匀散布在前后地址部分的要求，具体实现方式如代码 1 所示。 $i$  为由 0 至 1 均匀采样的随机数，当其小于 0.25 时，生成的指令地址均匀分布在前地址部分，当其大于 0.75 时，生成的指令地址均匀分布在后地址部分。因为 `short` 类型的取值范围为  $-2^{15} \sim 2^{15} - 1$ ，则对于  $2^{15} = 16384$  取模运算可以将生成的地址均匀分布在前地址部分。后地址部分的均匀分布生成方式同理。

代码 1: 均匀地址生成

```
1 while (i <= N * 0.25) {
2     // 16384 = 2^15 即 0 _____ 均匀分布
3     x = rand() % 16384;
4     A.push_back(x);
5     i++;
6 }
```

而对于顺序指令的生成，每次生成的指令地址为上一条指令地址加 1 即可。

## 2 最佳淘汰算法 OPT

OPT 算法是通过预测将来每个页面是否被访问到，以及未来何时被访问，来选择最佳的页面替换。在理论上，OPT 算法是最优的；但是，OPT 算法并不实际使用，因为它无法在实时计算机系统中实现，相反，OPT 算法通常用作其他算法的基准。在本实验中采用模拟的方式，预先生成了全部的指令地址流，有了访问序列这一先验知识，在此基础上进行计算。

该算法的准则是淘汰已满页表中不再访问或是最迟访问的页，这就要求将页表中的页逐个与后继指令访问的所有页比较，如后继指令不在访问该页，则把此页淘汰，不然得找出后继指令中最迟访问的页面淘汰。查找过程使用 STL 中的 `find` 函数，若找到则返回对应下标，否则返回 `end` 迭代器。

OPT 算法的核心实现部分如代码 2 所示，

代码 2: OPT 核心代码实现

```
1 auto index = find(memory.begin(), memory.end(), pageno[i]);
2 // index 遍历 memory 找到 pageno[i] 的位置， 如果没找到，最终会指向 memory.end()
3 if (index != memory.end()) { // 该页命中
4     cout << "命中 ";
5 } else { // 没命中的时候
6     int maxindex = i;
7     int position = 0;
8     for (int j = 0; j < memory.size(); j++) { // 用 j 来找出要被换走的页面
9         int k = find(pageno.begin() + i + 1, pageno.end(), memory[j]) - pageno.
            begin();
10        // k 表示找到 memory[j] 所表示页面的位置
11        if (k == pageno.end() - (pageno.begin() + i + 1)) {
12            // 表示 memory[j] 表示的页是画物理内存最后一页了
```

```

13         position = j;
14         break;
15     } else{
16         // 表示找到了memory[j]表示的页
17         if (k > maxindex){ // i比当前的maxindex大 所以更新maxindex
18             maxindex = k;
19             position = j;
20         }
21     }
22 }
23 memory[position] = pageno[i]; // 置换完毕
24 miss++; // 缺页次数+1
25 }

```

### 3 先进先出算法 FIFO

先进先出算法总是会淘汰最先进入内存的页，即选择在内存中驻留时间最久的页予以淘汰。FIFO 算法只是在应用程序按线性顺序访问地址空间时效果才好，否则效率不高。并且 FIFO 算法不会考虑页面的使用频率，那些常被访问的页，往往在内存中也停留得最久，结果它们因变“老”而不得不被置换出去。

对于具体的实现而言，需要已调入内存的页按先后次序链接成一个队列，队列头指向内存中驻留时间最久的页，队列尾指向最近被调入内存的页。这样需要淘汰页时，从队列头很容易查找到需要淘汰的页。

当旧页被淘汰时，新页插入队尾，这样就能保证队列头指向的页是最早调入内存的页，而队列尾指向的页是最近调入内存的页。具体实现如代码 3，使用数组移动的方式模拟队列出队，实现难度相比其余两个算法较低。

代码 3: FIFO 核心代码实现

```

1 for (int k = 1; k < memory.size(); k++) {
2     // 将所有页向前移动一位
3     memory[k - 1] = memory[k];
4 }
5 // 将新页放到最后
6 memory[memory.size() - 1] = i;

```

### 4 最近最少使用算法 LRU

这是一种经常使用的方法，这种算法有各种不同的实施方案，本实验中采用的是不断调整页表链的方法，即总是淘汰页表链链首的页，而把新访问的页插入链尾。如果当前调用页已在页表内，则把它再次调整到链尾。这样就能保证最近使用的页，总是处于靠近链尾部分，而不常使用的页就移到链首，逐个被淘汰，在页表较大时，调整页表链的代价也是不小的。

有趣的一点是，LRU 算法不仅在页面置换算法之中得到了很好的应用，现实生活之中也有着 LRU 的算法思想。在微信群管理中，当一个群聊人数达到上限显示，可以使用 LRU 算法来淘汰（踢出）不活跃的群成员，这样可以保证群聊的活跃度。

LRU 算法需要系统的硬件的支持，为了明确一个进程在内存中各个页面各有多少的时间没有被进程访问，以及快速知道哪一页最近未被使用，需要有寄存器和栈两种硬件之一的支持。

## 4.1 寄存器

为了记录进程在内存中各页使用，给内存中的页面配置一个移位寄存器，可以表示为：

$$R = R_{n-1}R_{n-2}R_{n-3} \dots R_2R_1R_0$$

当进程访问某个物理块的时，将内存块对应的寄存器  $R_{n-1}$  位置为 1。此时，定时信号会每隔一定时间将寄存器右移 1 位。如果把  $n$  位寄存器的数看作整体，那么具有最小数值的寄存器对应的页面就是最久没有使用的页面，在访问内存发生缺页的时候应该最先将其置换出去。

## 4.2 栈

可以利用特殊的栈来表示各个页面的页面号。当进程访问页面的时候，将该页面的页面号从栈中移出，将它压入栈的顶部。因此栈顶总是最新被访问的页面号，而栈底是最近最久未使用的页面号。

在本次实验中，使用一个 `vector<int>` 的 `clocks_stack` 数组来记录页面最近一次被访问/使用的时间。每一次操作后 `clock++` 来表示时间的增长。如代码 4 所示。

代码 4: LRU 核心代码实现

```
1 int minclocks_stack = clocks_stack.front(); // 栈底是最近未使用的页面
2 clocks_stack.erase(clocks_stack.begin());
3 // 找到最近未使用的页面在内存中的位置
4 auto pos = find(memory.begin(), memory.end(), minclocks_stack);
5 pos = memory.erase(pos);
6 memory.insert(pos, pageno[i]);
7 clocks_stack.push_back(pageno[i]);
```

## 四 实验结果

对比 OPT、FIFO、LRU 三种算法的缺页率，如图 2 所示。

<div><div>PAGE NUMBER WITH SIZE 128k EACH ADDRESS IS: 18 命中率: 0.999575</div><div>PAGE NUMBER WITH SIZE 256k EACH ADDRESS IS: 20 命中率: 0.999587</div><div>PAGE NUMBER WITH SIZE 512k EACH ADDRESS IS: 22 命中率: 0.999600</div><div>PAGE NUMBER WITH SIZE 1024k EACH ADDRESS IS: 24 命中率: 0.999613</div><div>PAGE NUMBER WITH SIZE 2048k EACH ADDRESS IS: 26 命中率: 0.999625</div><div>PAGE NUMBER WITH SIZE 4096k EACH ADDRESS IS: 28 命中率: 0.999637</div><div>PAGE NUMBER WITH SIZE 8192k EACH ADDRESS IS: 30 命中率: 0.999650</div><div>PAGE NUMBER WITH SIZE 16384k EACH ADDRESS IS: 32 命中率: 0.999663</div></div> <div>OPT</div>	<div><div>PAGE NUMBER WITH SIZE 128k EACH ADDRESS IS: 18 命中率: 0.999463</div><div>PAGE NUMBER WITH SIZE 256k EACH ADDRESS IS: 20 命中率: 0.999463</div><div>PAGE NUMBER WITH SIZE 512k EACH ADDRESS IS: 22 命中率: 0.999481</div><div>PAGE NUMBER WITH SIZE 1024k EACH ADDRESS IS: 24 命中率: 0.999481</div><div>PAGE NUMBER WITH SIZE 2048k EACH ADDRESS IS: 26 命中率: 0.999481</div><div>PAGE NUMBER WITH SIZE 4096k EACH ADDRESS IS: 28 命中率: 0.999481</div><div>PAGE NUMBER WITH SIZE 8192k EACH ADDRESS IS: 30 命中率: 0.999481</div><div>PAGE NUMBER WITH SIZE 16384k EACH ADDRESS IS: 32 命中率: 0.999481</div></div> <div>LRU</div>	<div><div>PAGE NUMBER WITH SIZE 4k EACH ADDRESS IS: 18 命中率: 0.501966</div><div>PAGE NUMBER WITH SIZE 8k EACH ADDRESS IS: 20 命中率: 0.504016</div><div>PAGE NUMBER WITH SIZE 16k EACH ADDRESS IS: 22 命中率: 0.505972</div><div>PAGE NUMBER WITH SIZE 32k EACH ADDRESS IS: 24 命中率: 0.507800</div><div>PAGE NUMBER WITH SIZE 64k EACH ADDRESS IS: 26 命中率: 0.509912</div><div>PAGE NUMBER WITH SIZE 128k EACH ADDRESS IS: 28 命中率: 0.511788</div><div>PAGE NUMBER WITH SIZE 256k EACH ADDRESS IS: 30 命中率: 0.513800</div><div>PAGE NUMBER WITH SIZE 512k EACH ADDRESS IS: 32 命中率: 0.515659</div></div> <div>FIFO</div>
--	--	--

图 2: 实验结果

观察数据可知，对于三种算法中的任意一种，缺页率都会随着块数的增加而下降，并且下降速度并非线性，缺页率下降的速度会越来越慢。

其中，无论物理块数量，OPT 算法的缺页率总是最低的，都小于 LRU 算法和 FIFO 算法的缺页率，是理想状态下的最优页面置换算法；FIFO 算法虽然实现难度最低，但它的缺页率总是最高的，因为它总是淘汰最早进入内存的页面，并没有考虑使用频率的因素；并未而 LRU 算法的缺页率介于两者之间。

## 五 收获与体会

在本次实验中完成了三种页面置换算法的编写，并且对比了算法之间的执行效率。我对 C++ 语言的 STL 库有了更深的了解，也对 C++ 语言的一些特性有了更深的了解，如 `vector` 等。操作系统中的许多算法都离不开基本数据结构，如栈、队列、等等，这些数据结构的使用在本次实验中得到了很好的锻炼。