南京邮电大学

实 验 报 告

（ 2025/ 2026学年 第 2 学期）

课程名称	操作系统基础
实验名称	页面调度算法模拟
实验时间	2026 年 6 月 5 日
指导单位	计算机学院
指导教师	

学生姓名		班级学号	
学院(系)	计算机学院	专 业	软件工程

实 验 报 告

实验名称	页面调度算法模拟			指导教师	
实验类型	验证	实验学时	2	实验时间	2026 6 5

一. 实验目的和要求

1. 理解 FIFO、LRU、OPT 三种页面置换算法的工作原理;
2. 能用 C 语言实现这三种算法并计算缺页次数与缺页率;
3. 对比三种算法在不同物理块数下的性能差异。

二、实验环境(实验设备)

Windows + VMWare + Ubuntu

三、实验原理及内容

本次实验主要内容如下:

(1) 掌握 FIFO、LRU、OPT 三种页面置换算法的原理:FIFO 换最早进入的页面,LRU 换最久未访问的页面,OPT 换将来最久不会被访问的页面。

(2) 选择合适的存储结构,用 C 语言分别实现这三种算法,并计算缺页次数与缺页率。

(3) 打印每个时刻内存中页面的变化情况,直观对比三种算法的不同决策。

(4) 改造程序使其能从外部 txt 文件读取请求序列和物理块数,并通过改变物理块数观察缺页变化趋势。

实 验 报 告

2. 三种页面置换算法的核心思想。

三种算法都是在物理内存已满、再来一个不在内存中的页面请求时,需要从内存中挑一个页面换出去。区别在于挑哪一个:

FIFO:挑进入内存最早的那个,看的是"过去"。

LRU:挑最久没有被访问过的那个,看的也是"过去",但比 FIFO 更聪明一点 —— 它考虑的是访问行为而不是进入时间。

OPT:挑未来最久才会被用到的那个(如果以后永远不用就直接换走),看的是"未来"。OPT 需要知道整个请求序列,只能在模拟中使用,实际系统中无法实现。

3. 程序设计。

三种算法的整体框架几乎相同:维护一个长度为 mnum 的内存数组 mem,以及一个长度相同的辅助数组 time 用于记录相关时间。区别只在 time 的含义和更新时机不同。

(1) FIFO 算法核心代码:

```
int fifo(int req[], int rnum, int mnum) {
    int *mem = malloc(sizeof(int) * mnum);
    int *time = malloc(sizeof(int) * mnum);
    setarray(mem, mnum, -1);
    setarray(time, mnum, M);
    int count = 0;
    for (int i = 0; i < rnum; ++i) {
        if (findexist(mem, mnum, req[i]) != -1) continue;
        count++;
        int pos = findempty(mem, mnum);
        if (pos == -1) {                /* 找进入最早的 */
            pos = 0;
            for (int j = 1; j < mnum; ++j)
                if (time[j] < time[pos]) pos = j;
        }
        mem[pos] = req[i];
        time[pos] = i;
    }
    return count;
}
```

(2) LRU 与 FIFO 唯一的差别:命中时也要更新 time。FIFO 命中时是 continue,LRU 命中时要先 time[pos] = i 再 continue,这样 time 记录的就是"最近一次访问的时刻"而不是"进入时刻"。

```
/* LRU 在命中分支里多一句更新 */
if ((pos = findexist(mem, mnum, req[i])) != -1) {
    time[pos] = i;        /* 更新最近访问时间 */
    continue;
}
```

(3) OPT 算法需要每次缺页时往后扫描请求序列,找每个页面下一次出现的位置;若以后不再出现就标记为 M(大于任何下标的值,表示"无穷远"),然后选 time 值最大的页面替换:

```
/* OPT 的关键：找未来最远的页面 */
setarray(time, mnum, M);
for (int j = 0; j < mnum; ++j) {
    for (int k = i + 1; k < rnum; ++k) {
        if (mem[j] == req[k]) { time[j] = k; break; }
    }
}
pos = 0;
for (int j = 1; j < mnum; ++j)
    if (time[j] > time[pos]) pos = j;    /* 注意：这里是 > 不是 < */
```

4. 简单测试。

使用参考文档给出的请求序列 2 3 2 1 5 2 4 5 3 2 5 2,物理块数 3。基本版只输出缺页次数:

```
$ gcc memrep.c -o memrep.out
$ ./memrep.out
fifo: 9
lru:  7
opt:  6
$
```

符合预期:OPT 是理论最优,缺页最少(6);LRU 居中(7);FIFO 最差(9)。

5. 打印内存中页面的变化。

为了直观地看清每个时刻内存中放了哪些页面,把内存数组改成二维:mem[mnum][rnum],每发生一次访问就把当前内存状态拷贝到 mem[*][i]。注意作为函数参数时二维数组必须指明列宽。

```
void initarr(int arr[][M], int rnum, int cnum) {
    for (int r = 0; r < rnum; ++r)
        for (int c = 0; c < cnum; ++c)
            arr[r][c] = -1;
}
```

运行结果如下:

```
$ gcc memrepdisp.c -o memrepdisp.out
$ ./memrepdisp.out
Request: 2 3 2 1 5 2 4 5 3 2 5 2
Frames : 3

fifo: 9
  2  2  2  2  5  5  5  5  3  3  3  3
 -1  3  3  3  3  2  2  2  2  2  5  5
 -1 -1 -1  1  1  1  4  4  4  4  4  2

lru:  7
  2  2  2  2  2  2  2  2  3  3  3  3
 -1  3  3  3  5  5  5  5  5  5  5  5
 -1 -1 -1  1  1  1  4  4  4  2  2  2

opt:  6
  2  2  2  2  2  2  4  4  4  2  2  2
 -1  3  3  3  3  3  3  3  3  3  3  3
 -1 -1 -1  1  5  5  5  5  5  5  5  5
$
```

拿请求序列的第 5 列(请求 5)来对比三种算法:这是第一次出现物理内存满的情况,FIFO 换掉了最早进入的页面 2(因为 2 进入最早),LRU 换掉了最久未用的 3(2 刚在上一步被访问过),OPT 则会向前看到 1 以后再也不会出现,所以换掉了 1。可见三种算法换页时"瞄准"的页面就是不一样的。

6. 从外部文件读取请求序列。

把请求序列硬编码在 main 函数里测试很不方便,改成从 txt 文件读入。文件格式约定:第一行 rnum,第二行 rnum 个空格分隔的页号,第三行 mnum。

```
FILE *fp = fopen(filename, "r");
fscanf(fp, "%d", &rnum);
int *req = malloc(sizeof(int) * rnum);
for (int i = 0; i < rnum; i++) fscanf(fp, "%d", &req[i]);
fscanf(fp, "%d", &mnum);
fclose(fp);
```

同时改造统计部分,把缺页率(缺页次数 / 请求总数)也算出来打印:

```
$ cat request.txt
12
2 3 2 1 5 2 4 5 3 2 5 2
3
$ gcc memrepfile.c -o memrepfile.out
$ ./memrepfile.out request.txt
request seq: 2 3 2 1 5 2 4 5 3 2 5 2
frame num  : 3
fifo: faults = 9   miss rate = 75.00%
lru : faults = 7   miss rate = 58.33%
opt : faults = 6   miss rate = 50.00%
$
```

7. 比较不同物理块数下三种算法的表现。

保持请求序列不变,把物理块数从 2 一路调到 5,观察缺页次数的变化趋势:

```
$ ./memrepfile.out request.txt 2
frames = 2 : fifo=10  lru=10  opt= 7
$ ./memrepfile.out request.txt 3
frames = 3 : fifo= 9  lru= 7  opt= 6
$ ./memrepfile.out request.txt 4
frames = 4 : fifo= 6  lru= 6  opt= 5
$ ./memrepfile.out request.txt 5
frames = 5 : fifo= 5  lru= 5  opt= 5
$
```

从数据可以看出几个规律:

a) 同一物理块数下,三者关系基本满足 FIFO ≥ LRU ≥ OPT,与理论吻合。

b) 物理块数从 2 增加到 3 时缺页次数下降明显,从 3 到 4 还有改善,但从 4 增加到 5 时三种算法的缺页次数都收敛到 5(刚好等于请求序列中不同页面的个数,即每个页面只缺页一次,所有缺页都是"冷启动"),已经无法再降。

c) 当物理块数足够大时,三种算法没有区别。但内存是稀缺资源,真正起决定性作用的是物理块数偏紧的时候,这时算法选择就很重要 —— 比如 3 块时 FIFO 缺页率 75%,OPT 只有 50%,差距相当大。

8. 思考:增加物理块数,缺页次数一定下降吗?

通常情况下增加物理块意味着内存能装更多页面,缺页次数应当单调下降。但 FIFO 算法存在一个反直觉的现象,叫做 Belady 异常 —— 在某些特殊的请求序列下,FIFO 在物理块数变多时缺页次数反而变多。比如经典的反例 1, 2, 3, 4, 1, 2, 5, 1, 2, 3, 4, 5,3 块时缺页 9 次,4 块时缺页 10 次。LRU 和 OPT 都不会出现这种异常(LRU 和 OPT 都属于栈算法,具备"块数增加缺页次数不增"的性质)。

尝试用程序验证这个例子(请求序列改为 1 2 3 4 1 2 5 1 2 3 4 5,物理块分别取 3 和 4):FIFO 确实出现了缺页次数从 9 增到 10 的反常情况。这从实验角度直观看到了 Belady 异常的存在,也理解了"FIFO 不是栈算法"这句教材里的话到底是什么意思。

实 验 报 告

四、实验小结（包括问题和解决方法、心得体会、意见与建议等）

本次实验是设计型实验,要求自己用 C 语言把 FIFO、LRU、OPT 这三种页面置换算法实现出来。前两个实验更多是"照着写、跑通、看现象",这次则真正进入了"我要把书上的算法描述变成可运行的代码"的状态。整体做下来收获不少:

1. 把课本上的概念落到了代码上。课堂上听 FIFO/LRU/OPT 时感觉很简单 —— "换最老的"、"换最久没用的"、"换将来最久才用的",一句话就能说清。但真正动手写,才发现"换"这个动作背后藏着不少细节:用什么数据结构记录页面进入顺序?LRU 在命中时要不要更新时间?OPT 的 "未来最久" 怎么实现?当我意识到三种算法可以共用一个框架,只在 time 的语义和更新时机上有区别时,对它们之间的关系一下子清楚了很多。

2. 真正理解了 LRU 与 FIFO 的差别。教材上常说 LRU "比 FIFO 更好",我之前的理解就是一句模糊的口号。这次自己写,看到 LRU 跟 FIFO 的代码差别仅仅是"命中时多更新一句 time",但缺页次数从 9 降到了 7,这种"小修改 → 大改善"的对比让我对"局部性原理 (locality)"有了更直观的体会 —— LRU 之所以好,就是因为它利用了"刚被访问的页面很可能再被访问"这一规律,而 FIFO 完全无视访问行为。

3. 体会到 OPT 的双重身份。OPT 在实际系统里是不可能实现的(没有人能预知未来的请求序列),但它给所有可实现算法划了一道"性能上界"。在实验里把 OPT、LRU、FIFO 三者放一起跑,可以清楚地看到 LRU 离 OPT 有多远 —— 这正是衡量一个可实现算法"还有多大改进空间"的标尺。

4. 通过改变物理块数,看到了"算法重要性"和"硬件够用"之间的关系。物理块数=5 时三种算法结果完全一样;物理块数=3 时 FIFO 和 OPT 差距很大。这说明算法选择只在"内存吃紧"时才有价值,反过来如果系统资源很宽裕,简单的算法反而更划算 —— 这种"权衡"的思路在系统课程里反复出现,这次有了第一次具体的体验。

5. 接触到了 Belady 异常这个有意思的反例。一开始很难相信"块数多反而缺页多",直到自己跑出来 3 块 9 次、4 块 10 次的结果。后来查资料才明白,栈算法 (stack algorithm) 是不存在 Belady 异常的,LRU 和 OPT 都是栈算法,而 FIFO 不是。从这个具体反例反推出抽象的"栈性质",感觉比直接背定义印象深得多。

6. 在编码层面也有几个小收获:二维数组作函数参数必须指定列宽;不在算法核心的辅助函数(findexist、findempty、setarray)拆出来后,三种算法主体非常清晰,这也是一种简单的"代码抽象"练习。

问题及解决:

1. 第一次写 OPT 时,在选替换页面的循环里习惯性地写了 time[j] < time[pos],跟 FIFO 一样。结果跑出来 OPT 的缺页次数比 LRU 还多,非常奇怪。仔细想了想 OPT 的语义:"未来最久才用的",应该是 time 值最大的,所以条件应该是 > 不是 <。改过来后结果就对了。这件事告诉我 "算法对称结构 + 一个符号取反" 的代码经常出错,以后写这类对称代码要特别警惕。

2. 调试 OPT 时一开始 future 数组没有初始化为 M,内存中残留旧值,导致选择替换页面时结果不稳定。每次缺页前都重新 setarray(time, mnum, M) 之后就正常了。

3. 想看完整内存变化时,二维数组传参直接写 int arr[][] 编译报错。查资料才知道 C 语言要求函数参数里的多维数组必须指定除第一维以外所有维度的大小,所以要写成 int arr[][M],这个 M 是宏定义的列宽。

意见与建议:

1. 这次实验给出的请求序列只有 12 个数,缺页次数变化不大,不容易看出长尾趋势。建议补充一个稍长一点(比如 50~100 个)的请求序列作为额外测试,这样三种算法的差异会更明显。

实 验 报 告

五、指导教师评语					
成　绩		批阅人		日　期	