页面置换竞争实验报告

17373492 刘取齐

测评结果(截止6.26 11:32)

```
remote: 切换到一个新分支 'racing'
remote: 分支 racing 设置为跟踪来自 origin 的远程分支 racing。
remote: [ find your pageReplace.c ]
remote: [ compile successfully ]
remote: 1
remote: [ you have upload 1 times ]
remote: [ your results are as follows ]
remote: your id is 17373492
remote: your cost is 254013
remote: your time is 0.521578
remote: your score is 0.994380
remote: your rank is 1 of 38
remote: [ You got 100 (of 100) this time. 2019年 06月 26日 星期三 11:26:17 CST ]
```

前言

在**地址映射过程**中，若在页面中发现所要访问的页面不在内存中，则产生**缺页中断** 。当发生缺页中断时，如果操作系统内存中没有空闲页面，则 操作系统必须在内存选择一个页面将其移出内存，以便为即将调入的页面让出空间。而用来选择淘汰哪一页的规则叫做**页面置换算法**。

页面置换考虑的问题

为了在**有限的页面容量**情况下能够尽可能的 **减少页面调换的时间成本**，我们要尽可能多地减少缺页中断的次数，所以要将最近即将使用的页面保留在缓存中，这就是我们的 **最佳算法(OPT)** 所表述的——每次将最近不会再使用的页面调换出去。但实际上最佳算法不可能实现，所以我们尽可能地取接近和模拟最佳算法，根据已有的数据去预测未来可能出现的常用页面。

主要考虑的两个问题：

1. Recency (最近使用)
    - 根据时间空间的局部性原理，最近使用的页面再未来很可能再次使用，如循环语句，会往复访问部分语句，因此需要保留在页面中
2. Frequency (最常使用)
    - 程序中存在一些在全局中需要访问的变量，如：页面的数量N等等，全局变量和静态变量所在的页面也需要保存在页面中

现有的页面置换算法

1. 最近最少使用算法(LRU)
    - 最近最少使用算法将每个页面使用的顺序进行排序，将最近使用的页面放到队列的头部，当发生缺页中断的时候淘汰尾部的页面，并将新进的页面放到队列的头部。
    - 缺点：单纯考虑了页面的Recency而没有考虑页面的Frequency性质，如全局变量可能会多次访问但是因为最近较少访问被淘汰，如 1、1、1、1、1、2、3、4、5、1、1、1、1 就有可能将全局变量所在的页面1淘汰。
2. 最近最常访问算法(LFU)
    - 最近最常访问算法将最少访问的页面进行淘汰，对每个页面维护一个使用次数的计数，每次访问进行加一，当发生缺页中断时，将当前容量中的最少访问次数的进行淘汰。
    - 缺点：单纯考虑了页面的Frequency而没有考虑页面的Recency，如初始化时候可能会对某一个数据进行反复访问，导致这一页的次数被增加到非常大，而一直保留在队列中，但后续不一定会访问。
3. 时钟算法(CLOCK)
    - 对于LRU算法的近似，同时对FIFO进行改进。将访问的页面构成一个循环队列，并对每个页面增加一位标志位，标记其是否被二次访问，并用指针指向下一个可能的被置换页面，当当前访问的页面存在于循环队列中，将该页面的标志位置1；当当前访问的页面不在队列中时，遍历

指针指向的页面，当指向的页面标志位为1时，将标志位置0并继续向前移动，直至遍历到标志位为0的位，将该页面置换出去。
    - CLOCK算法给了页面二次生存的机会，当某页面被二次访问时，粗略地将该页面视为经常访问的页面，并给予其一次机会再下次可能被置换时进行保留，使得全局变量存在的页不被调换出去。
4. 自适应置换算法(ARC)
    - 对LRU算法和LFU算法的结合，该算法维护两个四个队列
        1. LRU的队列T1
        2. 从LRU队列中被淘汰的队列B1
        3. LFU的队列T2
        4. 从LFU队列中被淘汰的队列B2
    - 对每个队列的容量进行自适应调整，具体算法为：
        1. 当B1的队列中有页面命中的话，增加T1的容量，将T2的队列队尾淘汰，将新页面加入到T1的队首中，视其为常访问页面
        2. 当B2的队列中有页面命中的话，减少T1的容量，将T1的队列队尾淘汰，将新页面加入到T2中，使其为常访问页面

本次实验采用的页面置换算法

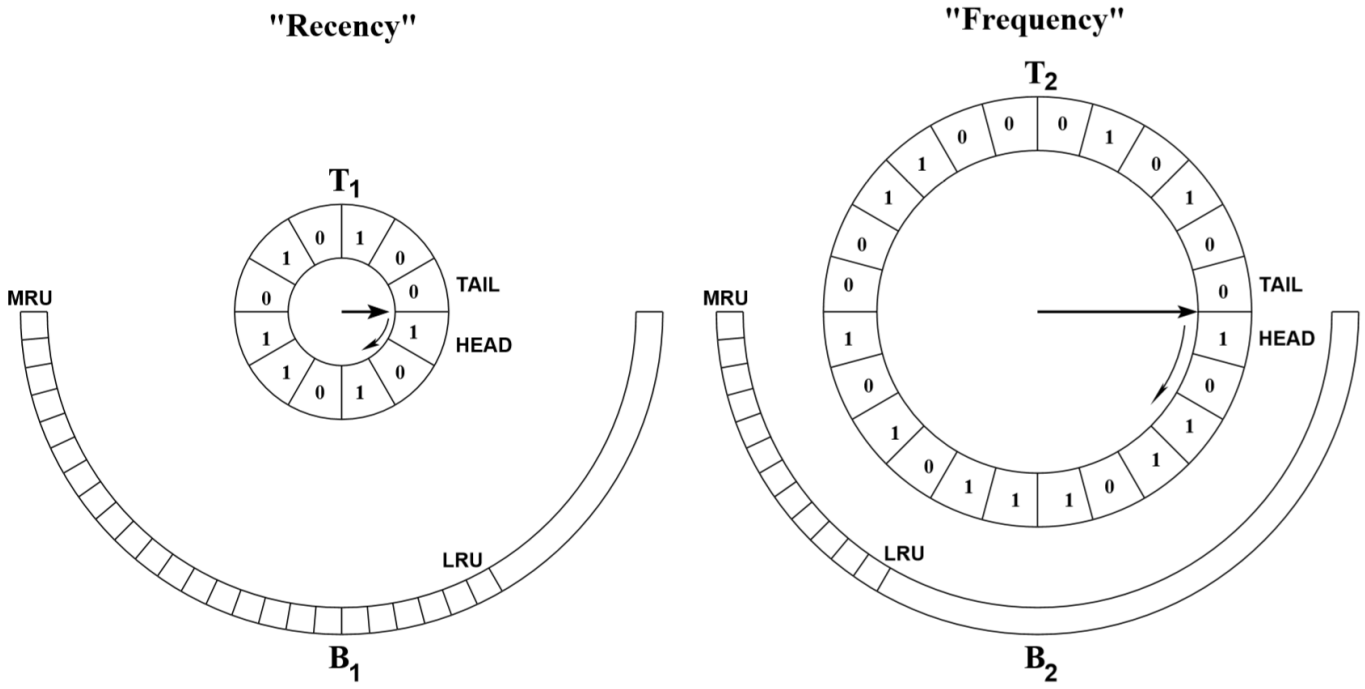
自适应的时钟算法(Clock with Adaptive Replacement)

本算法将时钟算法和自适应置换算法结合起来，得到了自适应的始终算法，既拥有始终算法的简易性，又包含了自适应算法兼顾Recency和Frequency的优点，具体算法如下：
算法包含四个集，分别是：

1. LRU的时钟队列T1
2. 从LRU淘汰的队列B1
3. LFU的时钟队列T2
4. 从LFU淘汰的队列B2

其中，T1和T2的时钟队列是一个循环队列，其中每个页面包括一个二次标志位，被一次以上访问的页面该标志位被置1；B1和B2是一个朴素的FIFO队列，先进入的页面在缺页中断时会被替换。

具体图例如下：



CAR算法中的T1和B1模拟了ARC算法中的T1和B1，表示只被访问过一次的页面，**这里需要注意的是：**若要严格遵守ARC算法中的T1和B1的性质，需要将T1中的所有标志位为1的页面加入到T2中，所以本应该是 T1(中页面标志位为0的页面)+B1 才是我们所说的**只被访问过一次的页面**。但是精确的计算会导致时间成本的增加，所以这里做了一个简单而大胆的近似，直接将T1+B1作为只访问过一次的页面。

每次有新的访问请求时：

1. 先检查T1和T2的队列是否含有该页面，若存在：
    1. 若存在于T1，则查找T1中的该页面：
        1. 若标志位为0，将标志位置1
        2. 若标志位为1，将该页放入到T2的队首，粗略表示最新的最常访问页面，同时将T2的队尾放入B2中
    2. 若存在于T2，则查找T2中的该页面：
        1. 若标志位为0，则标志位置1
        2. 若标志位为1，将该页放入到T2的队首，粗略表示最新的最常访问页面，同时将T2的队尾放入B2中
2. 若T1和T2中都不存在该页面：
    1. 若T1和T2加起来页面已满则需要找到一个牺牲的页面：
        1. 若T1的容量大于等于额定容量p，则从T1的队首开始遍历，寻找标志位为0的页面淘汰，其过程中若遇到标志位为1的位，将其插入到T2的队首。
        2. 若T1的容量小于额定容量，则从T2的队首开始遍历，寻找标志位为0的页面淘汰，其过程中若遇到标志位为1的位，将其插入到T2的队首
        3. 找到牺牲的页面后：
            1. 若新访问页面不在B1或B2中且T1和B1的总和为内存的大小则丢弃B1的队尾的页面

2. 若新访问页面不在B1或B2中且T1、T2、B1、B2的总和为内存大小的两倍，则丢弃B2队尾的页面

2. 若访问的页面不在B1或B2中，则直接将其插入到T1的队尾，并设置标志位为0
3. 若B1中存在该页面，则：
    1. 说明最近访问的页面数量较多，将T1的容量增加，更新p值为 p = min{PAGE,p+max{1,|B2|/|B1|}}
    2. 将其插入到T2的尾部，标志最新的常访问页面
4. 若B2中存在该页面，则：
    1. 说明经常访问的页面较多，将T2的容量增加，即减少T1的容量，更新p值为 p = max{p-max{1,|B1|/|B2|},0}
    2. 将其插入到T2的尾部，标志最新的常访问页面

具体实现：

更新于6月26日 11：33

利用**Hash**进行优化，在给定的空间限制下用**HashMap+DulNode**的方法代替了需要循环查找的Circle和按时间进出的B队列，具体优势如下：

1. DulNode链表可以使得增删更加简便，当新访问的页面存在于B队列需要插入到T2中时，只需要将其前后的节点进行连接即可，不需要像原来一样移动大部分的节点。

    e.g. 双向链表中插入节点

```
//B1Queue[B1tail] = temppage;
//B1tail = (B1tail+1) % PAGE;
B1tail->next = create(temppage);
B1tail->next->prev = B1tail;
B1tail = B1tail->next;
index2Bnode[B1tail->pagenum] = B1tail;
```

    e.g. 双向链表中删除节点

```
//how to move from B1 to T2
binode* tt;
tt = index2Bnode[pagenum];
tt->prev->next = tt->next;
if (B1tail==tt) {
    B1tail = tt->prev;
} else {
    tt->next->prev = tt->prev;
}
free(tt);
index2Bnode[pagenum] = null;
```

2. HashMap的维护使得访问节点的时间复杂度降至O(1)，不需要每次遍历64个位置进行查找，节约了很多循环的时间。

本地测试

测试环境

Windows10 64bit - WSL Ubuntu18.04 LTS

测试方法

```
int main() {
    long physic_memory[PAGE];
    int i;
    for (i = 0; i < PAGE; i++)
    {
        physic_memory[i] = 0;
    }
    long page;
        FILE *fp;
        int found = 0;
        int cost = 0;
        char name[20];
        scanf("%s",name);
        fp = fopen(name,"r");
    while (EOF!=fscanf(fp,"%ld",&page)) {
                for (int i = 0; i < PAGE; i++)
        {
                        if (physic_memory[i] == page>>12){
                                found = 1;
                                break;
                        }
        }
                if (found == 0) {
                        cost+=3;
                } else {
                        found = 0;
                        continue;
                }
        pageReplace(physic_memory, page);
        for (int i = 0;i < PAGE; i++)
        {
                        if (physic_memory[i] == page>>12){
                                found = 1;
                                break;
                        }
        }
        if (!found)
                        while(1);
    }
        fseek(fp,0L,SEEK_SET);
        long startTime,endTime;
        startTime = clock();
        while (EOF!=fscanf(fp,"%ld",&page)) {
        pageReplace(physic_memory, page);
    }
        endTime = clock();
        printf("Local: your id is 17373492\n");
        printf("Local: your cost is %d\n",cost);
        printf("Local: your real runtime is %ld ms\n",endTime-startTime);
```

```
    return 0;
}
```

读入要输入的数据的文件名，先运行一次**保证每次置换成功**，再运行第二次计算**实际运行时间**

测试数据

随机数据

随机数据生成使用的是c语言中的随机数生成函数 `rand()` 搭配随机种子选择函数 `srand()`

```
cookie@DESKTOP-SA554RQ:/mnt/c/shared$ cat generator.c
#include<stdio.h>
#include<stdlib.h>
#include<time.h>


int main(){
        srand(time(0));
        int i=0;
        for (i=0;i<170000;i++)
                printf("%d\n",rand()%2147483648);
}
```

随机数据测试结果

```
cookie@DESKTOP-SA554RQ:/mnt/c/shared$ ./CAR_ALGO
data1.in
Local: your id is 17373492
Local: your cost is 218574
Local: your real runtime is 62500 us
cookie@DESKTOP-SA554RQ:/mnt/c/shared$ ./CAR_ALGO
data2.in
Local: your id is 17373492
Local: your cost is 218640
Local: your real runtime is 78125 us
cookie@DESKTOP-SA554RQ:/mnt/c/shared$ ./CAR_ALGO
data3.in
Local: your id is 17373492
Local: your cost is 218646
Local: your real runtime is 62500 us
cookie@DESKTOP-SA554RQ:/mnt/c/shared$ ./CAR_ALGO
data4.in
Local: your id is 17373492
Local: your cost is 218565
Local: your real runtime is 78125 us
cookie@DESKTOP-SA554RQ:/mnt/c/shared$ ./CAR_ALGO
data5.in
Local: your id is 17373492
Local: your cost is 218331
Local: your real runtime is 78125 us
cookie@DESKTOP-SA554RQ:/mnt/c/shared$ ./CAR_ALGO
data6.in
Local: your id is 17373492
Local: your cost is 218334
Local: your real runtime is 78125 us
cookie@DESKTOP-SA554RQ:/mnt/c/shared$ ./CAR_ALGO
data7.in
Local: your id is 17373492
Local: your cost is 218184
Local: your real runtime is 62500 us
cookie@DESKTOP-SA554RQ:/mnt/c/shared$ ./CAR_ALGO
data8.in
Local: your id is 17373492
Local: your cost is 218766
Local: your real runtime is 62500 us
```

真实数据

根据17级数据结构大作业“马尔科夫链生成文本”来获取构建马尔科夫链时的访问内存地址的数据，此次使用的数据量经过缩小，从原来的500MB缩减到现在的60MB输入，输入包含 5643718 次查询：

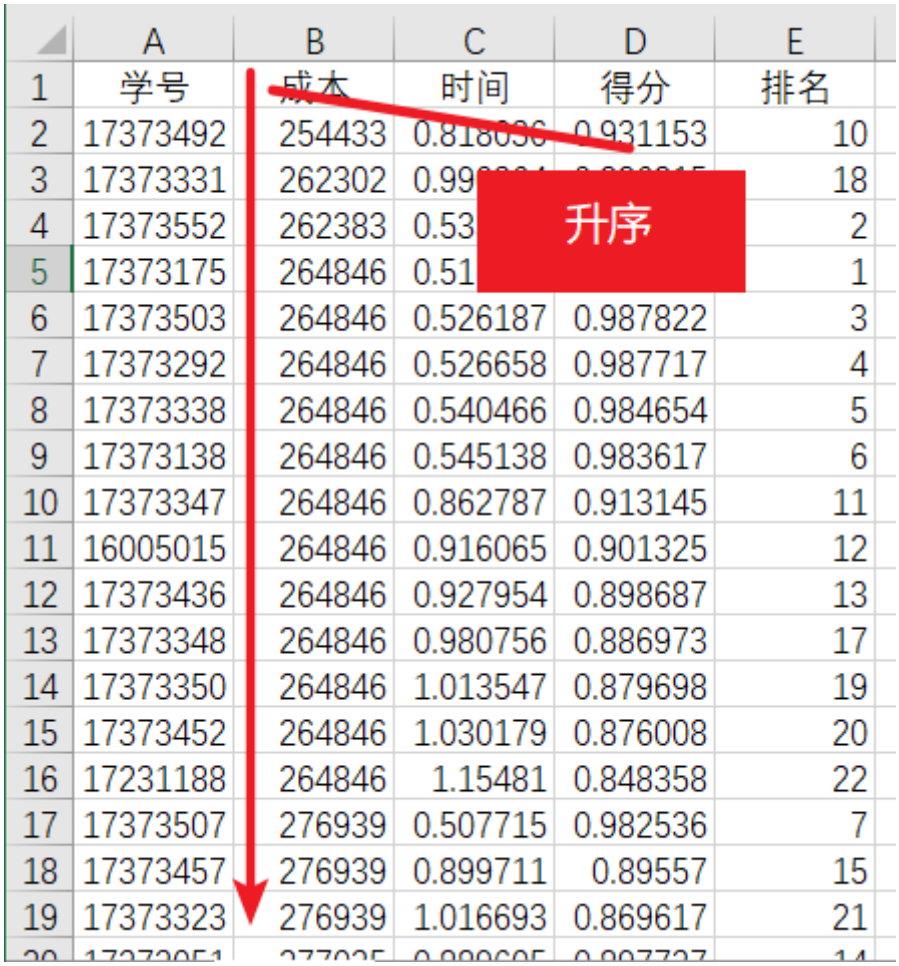
```
5643711 3805902408
5643712 3805902400
5643713 3944609280
5643714 3805902408
5643715 3805902400
5643716 3944609280
5643717 3805902408
5643718 3805902400
[/mnt/c/shared/realdata][txt]
```

得到的结果较为满意：

```
cookie@DESKTOP-SA554RQ:/mnt/c/shared$ ./CAR_ALGO
realdata
Local: your id is 17373492
Local: your cost is 160359
Local: your real runtime is 2890625 us
cookie@DESKTOP-SA554RQ:/mnt/c/shared$
```

按照随机数据的运行比例应该是 `cost` 的级别为百万级别，但因为真实程序是具有强局部性访问的，所以经常访问命中， `cost` 降到了10w级别。

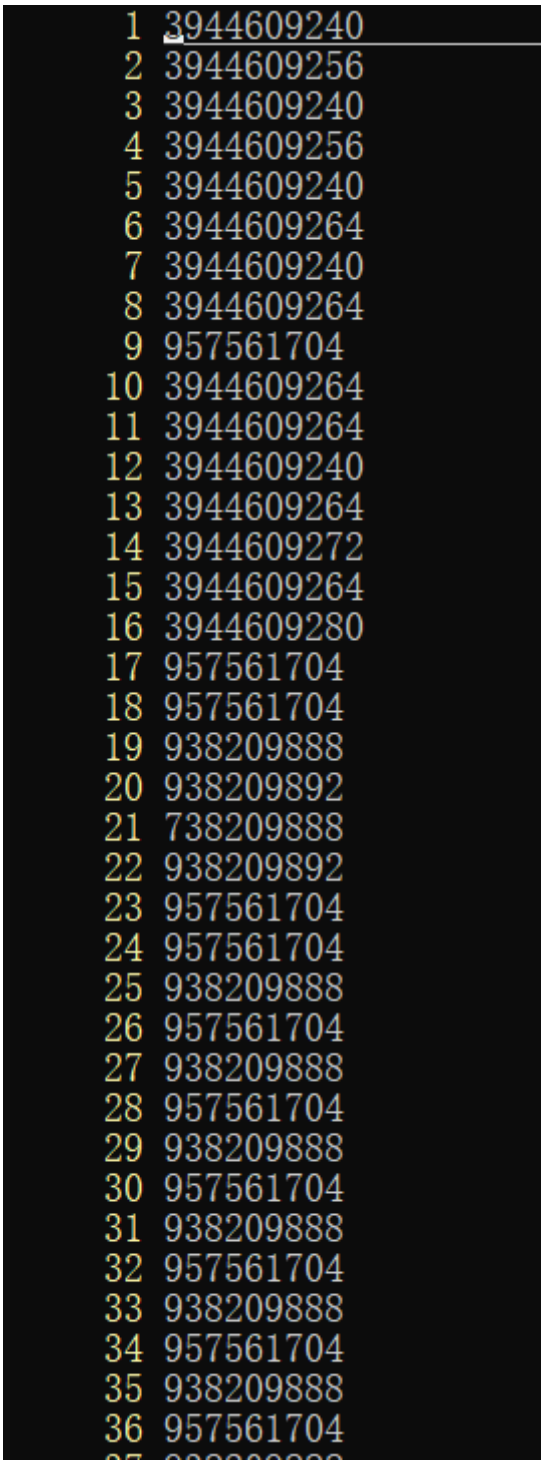
根据6.24公布的排名：我采用的CAR算法是所有参与页面竞争算法中cost最小的：



	A	B	C	D	E
1	学号	成本	时间	得分	排名
2	17373492	254433	0.818036	0.931153	10
3	17373331	262302	0.99		18
4	17373552	262383	0.53		2
5	17373175	264846	0.51		1
6	17373503	264846	0.526187	0.987822	3
7	17373292	264846	0.526658	0.987717	4
8	17373338	264846	0.540466	0.984654	5
9	17373138	264846	0.545138	0.983617	6
10	17373347	264846	0.862787	0.913145	11
11	16005015	264846	0.916065	0.901325	12
12	17373436	264846	0.927954	0.898687	13
13	17373348	264846	0.980756	0.886973	17
14	17373350	264846	1.013547	0.879698	19
15	17373452	264846	1.030179	0.876008	20
16	17231188	264846	1.15481	0.848358	22
17	17373507	276939	0.507715	0.982536	7
18	17373457	276939	0.899711	0.89557	15
19	17373323	276939	1.016693	0.869617	21

不足与未来的改进方向

1. 对页面的经常访问的判定较为粗略，在CAR中，我们假定被两次访问就视其为“**经常访问页面**”，但其实对一个数据的更新就有对其两次访问的操作，如我们的真实数据中可以看出很多访问超过两次的页面实际上是在访问同一个地址，即对一个变量的操作：



致谢

感谢OS课程组的老师和助教们，给我们带来丰富有趣的Lab1~6上机实验，感谢为OS实验课程的代码进行了改进升级和完善，让同学们真正接触到了操作系统是如何工作运转的，将我们引进了计算机底层的大门。
就我个人而言，我喜欢OS这门课程多余喜欢OO的课程，可以一天就看OS的某一个文件夹甚至一个文档，里面的汇编不遗余力地体现出了计算机在设计时的巧妙与精密，着实令人着迷。

题外话：指导书在实验过程中确实帮助了我们许多，但不可否认的是，指导书仍有进步的空间，有的地方确实叙述的不是很详细，若是让同学们自主学习也最好加上一些搜索的关键词，有时甚至有一些纰漏需要去修正，在实验的代码中仍有不完善的地方，如这次课程组改革的轮转时间片调度算法，确实是一个重大的革新，但在插入的时候严格意义上颠倒了调用的顺序，也最好在后面的Lab中能够向大家说明是否可能造成影响，减少大家的疑惑。希望能够加入OS课程组，为这门课做出一些贡献吧。