作业 10

贾城昊
2021K8009929010

10.1 假设一台计算机上运行的一个进程其地址空间有8个虚页（每个虚页大小为4KB，页号为1至8），操作系统给该进程分配了4个物理页框（每个页框大小为4KB），该进程对地址空间中虚页的访问顺序为 1 3 4 6 2 3 5 4 7 8。假设分配给进程的4个物理页框初始为空，请计算：
（1）如果操作系统采用CLOCK算法管理内存，那么该进程访存时会发生多少次page fault？当进程访问完上述虚页后，物理页框中保存的是哪些虚页？
（2）如果操作系统采用LRU算法管理内存，请再次回答（1）中的两个问题。请回答虚页保存情况时，写出LRU链的组成，标明LRU端和MRU端。

解：
(1)

- 访问虚页 1，发生 page fault，虚页 1 加载到物理页框 1 中（物理页框：[1] [ ] [ ] [ ]）。
- 访问虚页 3，发生 page fault，虚页 3 加载到物理页框 2 中（物理页框：[1] [3] [ ] [ ]）。
- 访问虚页 4，发生 page fault，虚页 4 加载到物理页框 3 中（物理页框：[1] [3] [4] [ ]）。
- 访问虚页 6，发生 page fault，虚页 6 加载到物理页框 4 中（物理页框：[1] [3] [4] [6]）。
- 访问虚页 2，发生 page fault，替换掉虚页 1（物理页框：[2] [3] [4] [6]）。
- 访问虚页 3，虚页 3 已经在物理页框中，将访问位设置为 1。
- 访问虚页 5，发生 page fault，替换掉虚页 4（物理页框：[2] [3] [5] [6]）。
- 访问虚页 4，发生 page fault，替换掉虚页 6（物理页框：[2] [3] [5] [4]）。

- 访问虚页 7，发生 page fault，替换掉虚页 2（物理页框：[7] [3] [5] [4]）。
- 访问虚页 8，发生 page fault，替换掉虚页 3（物理页框：[7] [8] [5] [4]）。

综上，共有 9 次 page fault，最后物理页框保存的虚页为 7 8 5 4

(2)

- 访问虚页 1，发生 page fault，虚页 1 加载到物理页框 1 中（物理页框：[1] [ ] [ ] [ ]，LRU 链表：[1]）。
- 访问虚页 3，发生 page fault，虚页 3 加载到物理页框 2 中（物理页框：[1] [3] [ ] [ ]，LRU 链表：[3, 1]）。
- 访问虚页 4，发生 page fault，虚页 4 加载到物理页框 3 中（物理页框：[1] [3] [4] [ ]，LRU 链表：[4, 3, 1]）。
- 访问虚页 6，发生 page fault，虚页 6 加载到物理页框 4 中（物理页框：[1] [3] [4] [6]，LRU 链表：[6, 4, 3, 1]）。
- 访问虚页 2，发生 page fault，替换掉 LRU 链表尾部的页面 1（物理页框：[2] [3] [4] [6]，LRU 链表：[2, 6, 4, 3]）。
- 访问虚页 3，虚页 3 已经在物理页框中，将其移到 LRU 链表头部（LRU 链表：[3, 2, 6, 4]）。
- 访问虚页 5，发生 page fault，替换掉 LRU 链表尾部的页面 4（物理页框：[2] [3] [5] [6]，LRU 链表：[5, 3, 2, 6]）。
- 访问虚页 4，发生 page fault，替换掉 LRU 链表尾部的页面 6（物理页框：[2] [3] [5] [4]，LRU 链表：[4, 5, 3, 2]）。
- 访问虚页 7，发生 page fault，替换掉 LRU 链表尾部的页面 2（物理页框：[7] [3] [5] [4]，LRU 链表：[7, 4, 5, 3]）。
- 访问虚页 8，发生 page fault，替换掉 LRU 链表尾部的页面 3（物理页框：[7] [8] [5] [4]，LRU 链表：[8, 7, 4, 5]）。

综上，共有 9 次 page fault，进程访问完毕后，物理页框中保存的是虚页 7、8、5、4。LRU 链表：[8, 7, 4, 5]，其中 MRU 端为 8，LRU 端为 4

10.2 假设一台计算机给每个进程都分配 4 个物理页框，每个页框大小为 512B。现有一个程序对一个二维整数数组（uint32 X[32][32]）进行赋值操作，该程序

的代码段占用一个固定的页框，并一直存储在内存中。程序使用剩余 3 个物理页框存储数据。该程序操作的数组 X 以列存储形式保存在磁盘上，即 X[0][0]后保存的是 X[1][0]、X[2][0]...X[31][0]，然后再保存 X[0][1]，以此类推。当程序要赋值时，如果所赋值的数组元素不在内存中，则会触发 page fault，操作系统将相应元素以页框粒度交换至内存。如果该进程的物理页框已经用满，则会进行页换出。该程序有如下两种写法。

写法 1：

```
for(int i=0;i<32;i++)
    for(int j=0;j<32;j++)
        X[i][j] = 0
```

写法 2：

```
for(int j=0;j<32;i++)
    for(int i=0;i<32;j++)
        X[i][j] = 0
```

请分析使用这两种写法时，各自会产生多少次 page fault？（注：请写出分析或计算过程）

解：

(1)

一个 uint32 数据需要 4B 的空间，所以一个页框能存储$512 \div 4 = 128$个 uint32 的数据。也即如果导入一个新页时，会同时导入 128 个数组元素，而这 128 个数组元素等价于发生 4 次行改变。由于只有三个物理页可以用于进行页替换。数组访问某一个页后发生替换，每八次页错误会回到同一个页，但此时该页已经被替换掉了

所以对于写法一，初始时，数组元素均未在内存里，所以访问第一个元素触发一次 page fault，之后所以每访问完 4 行后，访问下一个元素会发生一次 page fault，所以一共会产生$8 \times 32 = 256$次 page fault。

(2)

同样，初始时，数组元素均未在内存里，所以访问第一个元素触发一次 page fault。由于此时是按列访问数组的，与数组的存储方式相同，所以每访问

128 个元素时，才会触发 page fault。即之后所以每访问 X[4*m][0]后，才会发生一次 page fault，所以一共会产生8次 page fault。

10.3 假设一个程序有两个段，其中段 0 保存代码指令，段 1 保存读写的数据。段 0 的权限是可读可执行，段 1 的权限是可读可写，如下所示。该程序运行的内存系统提供的虚址空间为 14-bit 空间，其中低 10-bit 为页内偏移，高 4-bit 为页号。

Segment 0		Segment 1	
Read/Execute		Read/Write	
Virtual Page #	Page frame #	Virtual Page #	Page frame #
0	2	0	On Disk
1	On Disk	1	14
2	11	2	9
3	5	3	6
4	On Disk	4	On Disk
5	On Disk	5	13
6	4	6	8
7	3	7	12

当有如下的访存操作时，请给出每个操作的实际访存物理地址或是产生的异常类型(例如缺页异常、权限异常等)

(1) 读取段 1 中 page 1 的 offset 为 3 的地址

(2) 向段 0 中 page 0 的 offset 为 16 的地址写入

(3) 读取段 1 中 page 4 的 offset 为 28 的地址

(4) 跳转至段 1 中 page 3 的 offset 为 32 的地址

解：

题目中没有给出物理页框的大小，但是一般而言是跟虚拟页大小一致，这里假设其为 2 也是 1KB 的页面大小

(1)

段 1 中 page 1 对应于第 14 个物理页框，同时段 1 的权限为可读可写，读操作不会发生异常。物理地址为 0x3800+3=0x3803；

(2)

段 0 中 page 0 已经对应于第 2 个物理页框，段 0 的权限为可读可执行，写操作会触发权限异常。

(3)

段 1 中 page 4 存储在磁盘上，会触发缺页异常

(4)

段 1 中 page 3 对应第 6 个物理页框，但段 1 的权限为可读可写，执行操作会触发权限异常。

10.4 假设一个程序对其地址空间中虚页的访问序列为 0,1,2，...,511,422,0,1,2,...,511，333,0,1,2,...，即访问一串连续地址（页 0 到页 511）后会随机访问一个页（页 422 或页 333)，且这个访问模式会一直重复。请分析说明：

（1）假设操作系统分配给该程序的物理页框为 500 个，那么，LRU，Second Chance 和 FIFO 这三种算法中哪一个会表现较好（即提供较高的缓存命中率)，或是这三种算法都表现不佳？为什么?

解：

FIFO：对于 FIFO 算法，当程序第一轮访问结束时，物理页对应的虚页为 12，13，14，.....，511。此时随机访问，如果为 0-11，虽然这次没有命中，但之后循环中会命中一次，且其它虚页无法命中（不考虑随机，每 514 次才能访问到相同的物理页，所以必定不会命中)；如果为 12-511，这次命中，但后续循环均不会命中。而第二轮访问结束后， 物理页对应的虚页还是 12，13，14，.....，511，这样与第一次循环后续的情况相同。所以对于 FIFO 算法，命中率是固定的，为

$$\frac{1}{513} = 0.1949\%$$

LRU：对于 FIFO 算法，当程序第一轮访问结束时，物理页对应的虚页为 12，13，14，.....，511，LRU 的链表为 12，13，14，.....，511。此时如果随机访问的是 0-11，未命中，但会把对应的虚页号移动到 MRU 端，在之后的循环中，该虚

页会命中一次，同样其它的虚页均无法命中（不考虑随机，每 513 次才能访问到相同的物理页，所以必定不会命中）；如果随机访问的是 12-499，这次命中，并把对应的虚页移动到链表的 MRU 端，且之后循环该虚页还会命中一次；如果为 500-511，这次命中，并把对应的虚页移动到链表的 MRU 端，但后续循环均不会命中。而第二轮访问结束后， 物理页对应的虚页还是 12，13，14，……，511，LRU 的链表还为 12，13，14，……，511。这样与第一次循环后续的情况相同。所以对于 LRU 算法，我们可以计算其命中率的期望，为:

$$\frac{1}{513} \times \frac{12}{512} + \frac{2}{513} \times \frac{488}{512} + \frac{1}{513} \times \frac{12}{512} = 0.3807\%$$

Second Chance：该算法维护了一个循环队列，当一个页面首次被加载到内存时，它会被放到队列的尾部。当这个页面再次被访问时，它的引用位会被设置为 1，但它的位置不会改变。当需要替换页面时，Second Chance 算法会查看队列头部的页面。如果这个页面的引用位为 0，那么它会被替换掉；如果引用位为 1，那么这个页面会被放到队列的尾部，引用位被设置为 0，然后算法会继续查看下一个页面。

考虑到以上前提，对于上述序列，所导致的情况较为复杂，因为第二次循环开始后与上一次循环的状态并不相同，并且随机访问也会导致很多的不确定性。所以本人和室友李金明进行合作写了一份代码来模拟 Second Chance 的情况，代码如下:

```c
#include <stdio.h>
#include <stdlib.h>
#include <time.h>

#define PAGE_TABLE_SIZE 512
#define PHYSICAL_FRAMES 500
#define CIRCULATION_NUM 1000

typedef struct PageTableEntry {
    int pageNumber;
    int referenced;
    struct PageTableEntry* next;
} PageTableEntry;
```

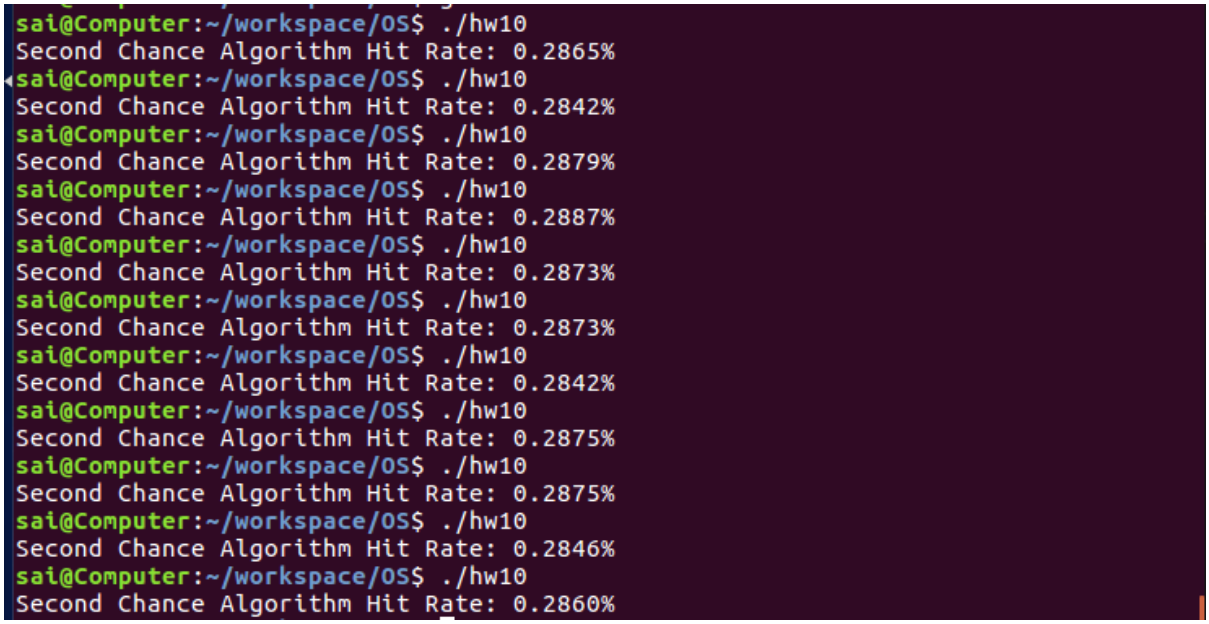
```c
typedef struct PageTable {
    PageTableEntry* head;
    PageTableEntry* tail;
    int num;
} PageTable;

// Function prototypes
void initializePageTable(PageTable* pageTable);
int accessPage(PageTable* pageTable, int pageNumber);
void secondChance(PageTable* pageTable);

int main() {
    srand(time(NULL));

    PageTable pageTable;
    initializePageTable(&pageTable);

    int pageSequence[(PAGE_TABLE_SIZE + 1) * CIRCULATION_NUM];
    int idx = 0;
    for(int i=0;i<CIRCULATION_NUM;i++){
        for(int j=0;j<PAGE_TABLE_SIZE;j++){
            pageSequence[idx] = j;
            idx++;
        }
        pageSequence[idx++] = rand() % PAGE_TABLE_SIZE;
    }
    int sequenceLength = sizeof(pageSequence) /
sizeof(pageSequence[0]);

    int hits = 0;

    for (int i = 0; i < sequenceLength; ++i) {
        int pageNumber = pageSequence[i];
        int hit = accessPage(&pageTable, pageNumber);
        if (hit) {
            hits++;
        }
    }
```

```
    double hitRate = (double)hits / sequenceLength;
    printf("Second Chance Algorithm Hit Rate: %.4f%% \n",
hitRate*100);

    return 0;
}

void initializePageTable(PageTable* pageTable) {
    pageTable->head = pageTable->tail = NULL;
    pageTable->num = 0;
}

int accessPage(PageTable* pageTable, int pageNumber) {
    PageTableEntry* current = pageTable->head;

    while (current != NULL) {
        if (current->pageNumber == pageNumber) {
            // Page hit
            current->referenced = 1;
            return 1;
        }

        current = current->next;
    }

    // Page fault
    if (pageTable->head == NULL) {
        // Page table is empty, insert the first page
        PageTableEntry* newEntry =
(PageTableEntry*)malloc(sizeof(PageTableEntry));
        newEntry->pageNumber = pageNumber;
        newEntry->referenced = 1;
        newEntry->next = NULL;
        pageTable->head = pageTable->tail = newEntry;
        pageTable->num++;
    }
    else if(pageTable->num <= PHYSICAL_FRAMES){
        PageTableEntry* newEntry =
(PageTableEntry*)malloc(sizeof(PageTableEntry));
```

```
        newEntry->pageNumber = pageNumber;
        newEntry->referenced = 1;
        newEntry->next = NULL;
        pageTable->tail->next = newEntry;
        pageTable->tail = newEntry;
        pageTable->num++;
    }
    else {
        // Page table is not empty, use Second Chance algorithm
to replace a page
        secondChance(pageTable);
        PageTableEntry* newEntry =
(PageTableEntry*)malloc(sizeof(PageTableEntry));
        newEntry->pageNumber = pageNumber;
        newEntry->referenced = 1;
        newEntry->next = NULL;
        pageTable->tail->next = newEntry;
        pageTable->tail = newEntry;
    }

    return 0;
}

void secondChance(PageTable* pageTable) {
    PageTableEntry* current = pageTable->head;

    while (current != NULL) {
        if (current->referenced) {
            // Give the page a second chance
            current->referenced = 0;
            pageTable->head = current->next;

            current->next = NULL;
            pageTable->tail->next = current;
            pageTable->tail = current;

        } else {
            // Remove the page with no second chance
```

```
            pageTable->head = current->next;

            free(current);
            return;
        }

        current = pageTable->head;
    }
}
```

代码会将上面的序列循环 1000 次，多次运行的结果如下：



可以看到，此时 Second Chance 的正确率大概为 0.2860%左右。

总的来说，这种这三种算法都表现不佳，这是因为大多数情况下，每隔 513 次才能必定访问到相同的页，而物理页框只有 500 个，所以大概率下次访问的时候已经替换了。但相对而言，LRU 算法略优与 Second Chance 算法，Second Chance 算法略优于 FIFO 算法。