# 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。**

表格

描述已自动生成

**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

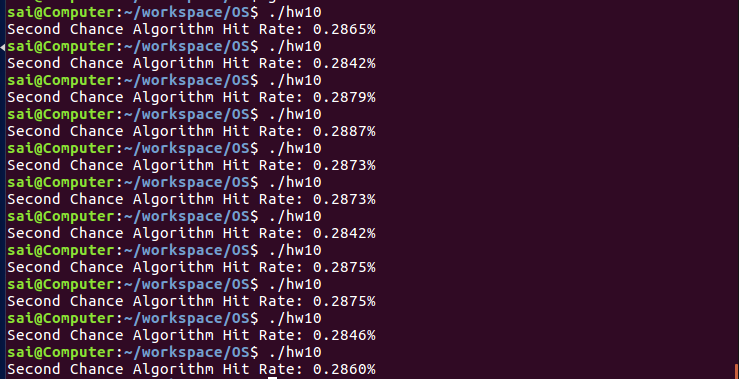
LRU：对于FIFO算法，当程序第一轮访问结束时，物理页对应的虚页为12，13，14，…….，511，LRU的链表为12，13，14，…….，511。此时如果随机访问的是0-11，未命中，但会把对应的虚页号移动到MRU端，在之后的循环中，该虚页会命中一次，同样其它的虚页均无法命中（不考虑随机，每513次才能访问到相同的物理页，所以必定不会命中）；如果随机访问的是12-499，这次命中，并把对应的虚页移动到链表的MRU端，且之后循环该虚页还会命中一次；如果为500-511，这次命中，并把对应的虚页移动到链表的MRU端，但后续循环均不会命中。而第二轮访问结束后， 物理页对应的虚页还是12，13，14，…….，511，LRU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:

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:

|  |
| --- |
| #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;  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。