最佳页面置换算法(Optimal)（非常专业）

|  |
| --- |
| 评价一个算法的优劣,可通过在一个特定的**存储访问序列**（页面走向）上运行它，并计算缺页数量来实现。**1 先入先出法（FIFO）**  最简单的页面置换算法是先入先出（FIFO）法。这种算法的实质是，总是选择在主存中停留时间最长（即最老）的一页置换，即**先进入内存的页，先退出内存**。理由是：最早调入内存的页，其不再被使用的可能性比刚调入内存的可能性大。建立一个FIFO队列，收容所有在内存中的页。被置换页面总是在队列头上进行。当一个页面被放入内存时，就把它插在队尾上。 这种算法只是在按线性顺序访问地址空间时才是理想的，否则效率不高。因为那些常被访问的页，往往在主存中也停留得最久，结果它们因变“老”而不得不被置换出去。 FIFO的另一个缺点是，它有一种异常现象，即在增加存储块的情况下，反而使缺页中断率增加了。当然，导致这种异常现象的页面走向实际上是很少见的。  现在来看下4块的情况：  0 1 2 3 2 1 3 2 5 2 3 6 2 1 4 2   【解答】 刚开始内存并没有这个作业，所以发生缺页中断一次。作业的0号页进入内存。(1次缺页中断)  而页1又不在内存，又发生缺页中断一次。作业页1进入内存。(2次缺页中断)  页2不在内存，发生缺页中断。页2进入内存。 (3次缺页中断)  页3不在内存，发生缺页中断。页3进入内存。 (4次缺页中断)  接下来调入页2，页1，页3，页2。由于都在内存中，并不发生缺页中断。  页5不在内存，发生缺页中断。页5进入内存，页5置换页0。 (5次缺页中断)  接下来调入页2，页3。由于都在内存中，并不发生缺页中断。  页6不在内存，发生缺页中断。页6进入内存。页6置换页1。 (6次缺页中断)  页2在内存，不发生缺页中断。  页1不在内存(在发生第6次缺页中断时被置换了)，发生缺页中断。  页1进入内存，页2被置换。 (7次缺页中断)  页4置换页3，页4进入内存。 (8次缺页中断)  现在调入页2，但页2在发生第7次缺页中断时被置换掉了。  现在页2进入内存，其置换页5。(因为这个时候是页5最先进入内存。)(9次缺页中断)  **2 最优置换算法（OPT）**  最优置换（Optimal Replacement）是在**理论上**提出的一种算法。其实质是：当调入新的一页而必须预先置换某个老页时，所选择的老页应是将来不再被使用，或者是在最远的将来才被访问。采用这种页面置换算法，保证有最少的缺页率。 但是最优页面置换算法的实现是困难的，因为它需要人们预先就知道一个进程整个运行过程中页面走向的全部情况。不过，这个算法可用来衡量（如通过模拟实验分析或理论分析）其他算法的优劣。  用最佳页面置换法计算缺页次数  6 5 4 3 5 4 3 6 5 4 5   －－－－－－－－－－－  6 6 6 3 3 3 3 6 6 6 6   5 5 5 5 5 5 5 5 5 5   4 4 4 4 4 4 4 4 4    仅仅第四列3和第八列6处,缺页.  第四列处:  opt算法中，页面发生冲突时，被替换的页面是未来访问最靠后的页面。  例子中，第4列处，6的再次访问最靠后，因而6被替换。  之后，第8列处，3被替换是因为3，4，5中未来被访问的页是4，5。  所以，3被替换。  **3 最久未使用算法（LRU）**  FIFO算法和OPT算法之间的主要差别是，FIFO算法利用页面进入内存后的时间长短作为置换依据，而OPT算法的依据是将来使用页面的时间。如果以最近的过去作为不久将来的近似，那么就可以把过去最长一段时间里不曾被使用的页面置换掉。它的实质是，当需要置换一页时，**选择在最近一段时间里最久没有使用过的页面予以置换**。这种算法就称为**最久未使用算法**（Least Recently Used，LRU）。 LRU算法是与每个页面最后使用的时间有关的。当必须置换一个页面时，LRU算法选择过去一段时间里最久未被使用的页面。 LRU算法是经常采用的页面置换算法，并被认为是相当好的，但是存在如何实现它的问题。LRU算法需要实际硬件的支持。其问题是怎么确定最后使用时间的顺序，对此有两种可行的办法： （1）计数器。最简单的情况是使每个页表项对应一个使用时间字段，并给CPU增加一个逻辑时钟或计数器。每次存储访问，该时钟都加1。每当访问一个页面时，时钟寄存器的内容就被复制到相应页表项的使用时间字段中。这样我们就可以始终保留着每个页面最后访问的“时间”。在置换页面时，选择该时间值最小的页面。这样做，不仅要查页表，而且当页表改变时（因CPU调度）要维护这个页表中的时间，还要考虑到时钟值溢出的问题。 （2）栈。用一个栈保留页号。每当访问一个页面时，就把它从栈中取出放在栈顶上。这样一来，栈顶总是放有目前使用最多的页，而栈底放着目前最少使用的页。由于要从栈的中间移走一项，所以要用具有头尾指针的双向链连起来。在最坏的情况下，移走一页并把它放在栈顶上需要改动6个指针。每次修改都要有开销，但需要置换哪个页面却可直接得到，用不着查找，因为尾指针指向栈底，其中有被置换页。 因实现LRU算法必须有大量硬件支持，还需要一定的软件开销。所以实际实现的都是一种简单有效的LRU近似算法。 一种LRU近似算法是**最近未使用算法**（Not Recently Used，NUR）。它在存储分块表的每一表项中增加一个引用位，操作系统定期地将它们置为0。当某一页被访问时，由硬件将该位置1。过一段时间后，通过检查这些位可以确定哪些页使用过，哪些页自上次置0后还未使用过。就可把该位是0的页淘汰出去，因为在最近一段时间里它未被访问过。  **4 第二次机会算法（SCR）**  第二次机会算法的基本思想是与FIFO相同的，但是有所改进，避免把经常使用的页面置换出去。当选择置换页面时，检查它的访问位。如果是0，就淘汰这页；如果访问位是1，就给它第二次机会，并选择下一个FIFO页面。当一个页面得到第二次机会时，它的访问位就清为0，它的到达时间就置为当前时间。如果该页在此期间被访问过，则访问位置1。这样给了第二次机会的页面将不被淘汰，直至所有其他页面被淘汰过（或者也给了第二次机会）。因此，如果一个页面经常使用，它的访问位总保持为1，它就从来不会被淘汰出去。 第二次机会算法可视为一个环形队列。用一个指针指示哪一页是下面要淘汰的。当需要一个存储块时，指针就前进，直至找到访问位是0的页。随着指针的前进，把访问位就清为0。在最坏的情况下，所有的访问位都是1，指针要通过整个队列一周，每个页都给第二次机会。这时就退化成FIFO算法了。  页面置换算法还有很多变种，如考虑到被置换页是否修改过、按FIFO算法选中的页正在使用等情况，都需要硬件、软件协同实现。  部分的页面在虚拟内存，部分在物理内存，操作系统需要访问的页面在物理内存找不到则会把物理内存的某个页面置换下来，最佳置换算法的解决方法就是看物理内存中的哪一个页面在将来最迟需要访问，就置换它。 如物理内存里是0，7，6，访问到5时产生缺页中断，检查物理内存，发现0在将来第14个访问到，显然置换0是最佳方案！  using namespace std; #include <iostream> #define MAX 20 int arr[MAX]={0,7,6,5,7,4,7,3,5,4,7,4,5,6,5,7,6,0,7,6}; int tarr[MAX]; int Num=0;  class Templist {      friend class Opclass; private:      Templist\* next;      int data;      int count; public:      Templist(){next=NULL;}      Templist(int data){this->data=data;next=NULL;}      ~Templist(){} public:      int GetCount(){return count;}      int GetData(){return data;}      Templist\* GetNext(){return next;} };  class Opclass { private:      Templist\* head; public:      Opclass()      {          head=new Templist;      }      Opclass(int size)      {          head=new Templist;          for(int i=0;i<size;i++)          {              Templist\* newnode=new Templist;              newnode->data=-1;              newnode->next=head->next;              head->next=newnode;          }      }      ~Opclass(){}           void Optimal();      int Count(int data,int n);      void Display(Templist\* temp); };  int Opclass::Count(int data,int n) {      int count=0;      for(int i=n;i<MAX;i++)      {          count++;          if(arr[i]==data)              break;      }      return count; } void Opclass::Optimal() {      int Max=0;      bool bl=false;      Templist \*temp=head->next,\*p=NULL;      for(int i=0;i<MAX;i++)      {              if(temp==NULL)          {              p=head->next;              while(p!=NULL)              {                  if(p->data==arr[i])                  {                      bl=true;                      break;                  }                  p=p->next;              }              if(bl==true)                  continue;              p=head->next;              while(p!=NULL)              {                  p->count=Count(p->data,i+1);                  if(Max<p->count)                      Max=p->count;                  p=p->next;              }              p=head->next;              while(p!=NULL)              {                  if(p->count==Max)                  {                      p->data=arr[i];                      tarr[Num++]=i+1;                      break;                  }              }          }          else          {              temp->data=arr[i];              temp=temp->next;          }          cout<<"物理块状态：";          p=head->next;          Display(p);      } }  void Opclass::Display(Templist\* temp) {      while(temp!=NULL)      {          cout<<temp->data<<" ";          temp=temp->next;      }      cout<<endl; } int main(int argc, \_TCHAR\* argv[]) {      Opclass opclass(3);      cout<<"分配了3个物理块"<<endl;      cout<<"页面访问顺序：";      for(int i=0;i<MAX;i++)          cout<<"第"<<i+1<<"："<<arr[i]<<" ";      cout<<endl;      opclass.Optimal();      for(int i=0;i<Num;i++)          cout<<"在访问第"<<tarr[i]<<"个页面："<<arr[tarr[i]-1]<<"时发生缺页中断。"<<endl;      cout<<"共发生"<<Num<<"次页面置换!"<<endl;      return 0; } |