**鲁东大学信息与电气工程学院**

**实验报告**

（ 2022 — 2023 学年第 1 学期）

2220180106

**课程名称**  操作系统

**实验题目** 内存管理

**专 业** 软件工程

**班 级**  软工2101

**姓 名**  管恩浩

**学 号**  20212203357

2023 年 5 月 22 日

|  |  |  |  |
| --- | --- | --- | --- |
| 实验题目 | 实验4 内存管理 | | |
| 实验类型 | 验证、设计 | 实验日期 | 2023 年 4 月 22 日 |
| 题目来源 | ✔1.必修 2.选修 3.自拟(设计) 4.专题 | | |
| **一、实验目的及要求**  1．**掌握**动态分区分配方式使用的数据结构和分配算法(首次/最佳/最坏适应算法)；  2．进一步加深对动态分区分配管理方式及其实现过程的**理解**；  3．通过模拟实现请求页式存储管理的几种基本页面置换算法，**了解**虚拟存储技术的特点；  4．**掌握**虚拟存储请求页式存储管理中OPT、FIFO和LRU三种基本页面置换算法的基本思想和实现过程，并比较它们的效率；  5. 通过页面访问序列随机发生器实现对上述算法的测试及**性能比较**。  **二、实验仪器设备与软件环境**  1．一台装有Windows的微型计算机（Win7或Win10）；  2．在微型计算机上（通过虚拟机）安装了Linux环境；  3．Linux环境下的编辑器（vi/vim、gedit或vscode），编译器（gcc），调试器（gdb）。  **三、实验基础**  在单道批处理系统阶段，一个系统在一个时间段内只执行一个程序，内存的分配极其简单，仅分配给当前运行的进程。引入多道程序的并发执行后，进程之间共享的不仅仅是处理机，还有主存储器。然而，共享主存会形成一些特殊的挑战。若不对内存进行管理，则容易导致内存数据的混乱，以至于限制进程的并发执行。因此为了更好地支持多道程序并发执行，必须进行内存管理。  内存管理需要考虑内存共享、内存保护和内存分配和回收等多个方面的内容。    **（一）内存连续分配方式**  连续分配方式是指为每个用户进程分配一个连续的内存空间。分为单一连续分配、固定分区分配和动态分区分配（可变分区分配）三种方案。        在上面三种方案中，需要注意对比、理解固定分区、可变分区和可重定位分区分配：  1、固定分区分配  原理：又称定长分区或静态分区模式，是满足多道程序设计需要的最简单的存储管理技术。基本思想：给进入主存的用户作业划分一块连续存储区域，把作业装入该连续存储区域，若有多个作业装入主存，则它们可并发执行。  使用大小相等的固定分区有两个难点：程序可能太大而不能放到一个分区中，内存的，利用率很低。由于被装入的数据块小于分区大小，从而导致分区内部有浪费现象，成为“内部碎片”。对与大小不等的分区策略，最简单的方法就是把每个进程分配到能够容纳它的最小分区中。  目前已经基本上没有什么场合使用固定分区。  优势：实现简单，只需要极少的操作系统开销  缺点：有内部碎片，对内存的使用不充分，活动进程的最大数目是固定的。  2、可变分区分配  可变分区存储管理不是预先把内存中的用户区域划分成若干固定分区，而是在作业要求装入内存时，根据用户作业的大小和当时内存空间使用情况决定是否为该作业分配一个分区。因此分区大小不是预先固定的，而是按作业需求量来划分的；分区的个数和位置也不是预先确定的。它有效地克服了固定分区方式中，由于分区内部剩余内存空置造成浪费的问题。  3、可重定位分区分配  由于若干次内存分配与回收之后，各个空闲的内存块不连续了。通过“重定位”，将已经分配的内存“紧凑”在一块（就类似于JVM垃圾回收中的复制算法）从而空出一大块空闲的内存出来，也就是增加了内存移动的功能。  紧凑是需要开销的，比如需要重新计算地址，这也为什么JVM垃圾回收会导致STW的原因。而离散分配方式不管是分页还是分段，都是直接将程序放到各个离散的页中，就不存在“紧凑”一说。  可变分区方式常用的内存分配算法有以下几种：  **1）最先适应分配算法(FF)**  每次分配总是顺序查找空闲区表，找到能满足长度要求的空闲区就分配。优点是实现简单，缺点是可能将大的空闲区分割成许多小的空闲区，形成许多不连续的“碎片”。碎片长度可能不能满足作业要求，降低了内存利用率。  改进方法，可把空闲区按地址顺序从小到大登记在空闲区表中，有利于大作业。问题是归还空区时须按地址插入表中适当位置。  **2）最优适应分配算法(BF)**  按作业要求从所有空闲区中挑选一个能满足要求的最小空闲区，这样保证不去分割一个更大的区域，使装入大作业时比较容易得到满足。实现办法：将空闲区按长度以递增次序登记在表中，分配时按空闲区表顺序查找即可。缺点是可能碎片更小而无法使用。回收时也要按长度扦入。  **3）最坏适应分配算法(WF)**  这种算法总是挑选一个最大的空闲区分割一部分给作业使用，使剩下部分不致太小，仍可供分配使用。实现办法：空闲区表中的登记项按空闲区长度递减顺序排列，按序查找分配。    连续内存分配方式涉及两种操作：内存分配操作和内存回收操作。内存的回收也可以分为下面4种情况：  当进程运行完毕释放内存时，系统根据回收区的首址，从空闲区链（表）中找到相应的插入点，此时可能出现以下四种情况之一：  1）回收区与插入点的前一个空闲分区F1相邻接  见图(a)。此时应将回收区与插入点的前一分区合并，不必为回收分区分配新表项，而只需修改其前一分区F1的大小。  2）回收分区与插入点的后一空闲分区F2相邻接  见图(b)。此时也可将两分区合并，形成新的空闲分区，但用回收区的首址作为新空闲区的首址，大小为两者之和。  3）回收区同时与插入点的前、后两个分区邻接  见图(c)。此时将三个分区合并，使用F1的表项和F1的首址，取消F2的表项，大小为三者之和。    4）回收区既不与F1邻接，又不与F2邻接。这时应为回收区单独建立一个新表项，填写回收区的首址和大小，并根据其首址插入到空闲链中的适当位置。  下面的图示出了内存回收时的流程。  **（二）离散内存分配方式**  由于在多道批处理系统中的实际的进程执行过程中，并非那么容易的就能找到连续的内存空间进行内存的分配，因此便诞生了离散内存分配方式：页式管理、段式管理和段页式管理。  **1、虚拟存储系统**  内存是有限的，作业初始时保存在磁盘上的，如果要运行，必须得将相应的程序(数据)加载到内存中。那如果要运行的作业特别多，无法一下子装入内存，解决方法的一种方式是加内存条，这是从物理上扩充内存的容量。另一种方式是：先把作业的一部分程序(数据)装入内存，先让它运行着，运行过程中发现还需要其他的数据，而这些数据还未装入内存，因此就产生中断(缺页中断)再将数据加载到内存。采用这种方式，系统一次就可以将很多作业装入内存运行了。这时，从物理上看，内存还是原来的大小，但是它能运行的作业多了，因此说从逻辑上扩充了内存。  将虚拟存储器这种思想与分页存储管理结合，一次只将作业的部分页面加载到内存中，形成了一个强大的内存分配与管理系统。引入了虚拟存储器，就需要有页表来记录逻辑地址到物理地址的映射，只不过此时的页表更复杂了，因为有些页可能还在磁盘上，需要有缺页中断处理机构，因为毕竟只将一部分数据装入内存，会引起缺页中断，这时需要处理中断；还需要地址变换机构，它的功能更多，因为需要处理中断情况下的地址变换。  UNIX中，为了提高内存利用率，提供了内外存进程对换机制；内存空间的分配和回收均以页为单位进行；一个进程只需将其一部分（段或页）调入内存便可运行；还支持请求调页的存储管理方式。      当进程在运行中需要访问某部分程序和数据时，发现其所在页面不在内存，就立即提出请求（向CPU发出缺中断），由系统将其所需页面调入内存。这种页面调入方式叫请求调页。  为实现请求调页，核心配置了四种数据结构：页表、页框号、访问位、修改位、有效位、保护位等。    **2、请求分页系统中的页面置换算法**  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CPU接收到缺页中断信号，中断处理程序先保存现场，分析中断原因，转入缺页中断处理程序。该程序通过查找页表，得到该页所在外存的物理块号。如果此时内存未满，能容纳新页，则启动磁盘I/O将所缺之页调入内存，然后修改页表。如果内存已满、没有空闲页，则须按某种置换算法从内存中选出一页准备换出，是否重新写盘由页表的修改位决定，然后将缺页调入，修改页表。利用修改后的页表，去形成所要访问数据的物理地址，再去访问内存数据。整个页面的调入过程对用户是透明的。  用来选择淘汰哪一页的规则叫做页面置换算法。常用的页面置换算法有：  1）最佳置换算法（Optimal）  这是一种理想情况下的页面置换算法，但实际上是不可能实现的。该算法的基本思想是：发生缺页时，选择的被淘汰页面，将是以后永远不使用的，或许是在最长（未来）时间内不再被访问的页面，每个页面都可以用在该页面首次被访问前所要执行的指令数进行标记。最佳页面置换算法只是简单地规定：标记最大的页应该被置换。采用该算法，通常可保证获得最低的缺页率。但由于当缺页发生时，操作系统无法知道各个页面下一次是在什么时候被访问，也就是人们目前还无法预知一个进程在内存的若干个页面中哪一个页面是未来最长时间内不再被访问的，因而该算法是无法实现的，但可以用于对可实现算法的性能进行衡量比较。  2）先进先出法（Fisrt In First Out）  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。该算法实现简单，只需建立一个 FIFO队列，收容所有在内存中的页，并设置一个指针，称为替换指针，使它总是指向最老的页面。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。  3）最近最久未使用（Least Recently Used）  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，选择在之前一段时间里最久没有使用过的页面予以置换。所以LRU页面置换算法是根据页面调入内存后的使用情况进行决策的。由于无法预测各页面将来的使用情况，只能利用“最近的过去”作为“最近的将来”的近似， 因此，LRU 置换算法是选择最近最久未使用的页面予以淘汰。该算法赋予每个页面一个访问字段，用来记录一个页面自上次被访问以来所经历的时间 t，当需淘汰一个页面时，选择现有页面中其 t 值最大的，即最近最久未使用的页面予以淘汰。  **3、请求分页系统中的地址变换过程**      多级页表解决了当逻辑地址空间过大时，页表的长度会大大增加的问题。而采用多级页表时，一次访盘需要多次访问内存甚至磁盘，会大大增加一次访存的时间。  **（三）工作集与缺页率**  1）工作集  多数程序都显示出高度的局部性，也就是说，在一个时间段内，一组页面被反复引用。这组被反复引用的页面随着时间的推移，其成员也会发生变化。有时这种变化是剧烈的，有时这种变化则是渐进的。我们把这组页面的集合称为工作集。  2）缺页率  缺页率 = 缺页中断次数/总的页面访问次数。  四、实验内容  1．编写C程序，采用连续分配方式之动态分区分配存储管理，模拟实现首次、最佳、最坏适应算法的内存块分配与回收，要求每次分配与回收后显示出空闲分区和已分配分区的情况。假设在初始状态下，可用的内存空间为640KB。  (1) 设计一个作业申请队列以及作业完成后的释放顺序，实现主存的分配和回收。采用分区说明表进行。  (2) 或在程序运行过程，由用户指定申请与释放。  (3) 设计一个空闲区说明表，以保存某时刻主存空间占用情况。把空闲区说明表的变化情况以及各作业的申请、释放情况显示。  2．设计一个虚拟存储区和一个内存工作区，并使用下述常用页面置换算法计算访问命中率：  (1) 先进先出(first in first out，FIFO) 算法。  (2) 最近最久未使用(least recently used，LRU) 算法。  (3) 最优(optimal，OPT) 算法。  要求如下。  (1) 通过随机数产生一个指令序列，里面共320条指令。  (2) 将指令序列转换成页面序列。假设：①页面大小为1KB；②用户内存容量为4~32页；③用户虚存容量为32KB。在用户虚存中，按每页存放10条指令排列虚存地址，因此320条指令将存放在32个页面中。  (3) 计算并输出不同页面置换算法在不同内存容量下的访问命中率。访问命中率的计算公式为：  访问命中率=1-(页面失效次数/页面总数)  五、实验指导  **1、动态分区分配方式的模拟**  本实验的主要目的是模拟实现动态分区分配方式下内存的分配与回收，而设计的分配与回收算法涉及首次适应算法、最佳适应算法和最坏适应算法。根据动态分区分配的原理，主要需要建立两个数据结构——空闲分区表和已分配分区表，它们都需要包含分区的起始地址、长度等信息。所以，编写该程序首先要给定一个一定空间大小的内存，即申请空闲区空间最大值，并且要定义空间的各分区的作业标号、分区起始地址、分区长度，单位为字节、分区表的状态位、前向指针、后向指针、已分配分区表、空闲分区等。  通过定义空间分区后，还要定义空间分区链表并对其进行初始化，对空闲分区和已分配分区进行链表访问，对于空闲分区可以分配给新进来的进程使用，对于已分配的分区，则等进程执行结束后在回收空间，恢复空闲区。通过链表的访问实现整个空间分区的分配与回收。  当有新作业请求装入主存时，须查找空闲分区表，从中找出一个合适的空闲分区并将其分配给作业。然后按照作业需要的内存大小将其装人主存，剩下的部分仍为空闲分区，将其登记到空闲分区表中，作业占用的分区则登记到已分配分区表中。作业执行完毕后，应回收作业占用的分区，具体操作为：删除已分配分区表中的相关项，然后修改空闲分区表，并根据情况增加或合并空闲分区。  下面给出的示例代码实现了基于**首次适应算法**的内存分配与回收。需要注意的是，分配存储区时是从高地址开始的。输人“a”表示分配操作，分配时需要输人作业请求的内存大小；输入“r”表示回收操作，回收时需要输人回收分区的起始地址和大小。  #include <stdio.h>  #include <stdlib.h>  #include <string.h>  #include <ctype.h>  #define MAX 640  struct node //定义分区  {  int address,size;  struct node \*next;  };  typedef struct node RECT; /\*-----------函数定义------------\*/  RECT \*assignment(RECT \*head, int application) ; //分配分区  //针对首次适应算法回收分区  void firstfit(RECT \*head, RECT \*heada, RECT \*back1) ;  void bestfit(RECT \*head, RECT \*back1) ; //针对**最佳适应算法**回收分区，待扩充  int backcheck(RECT \*head, RECT \*back1) ; //合法性检查  void print(RECT \*head) ; //输出已分配分区 表或空闲分区  /\*----------变量定义-----------\*/  RECT \*head, \*heada, \*back, \*assign1, \*p;  int application1, maxblocknum;  char way; //用于定义分配方式：**首次适应、最佳适应、最坏适应**。目前未使用  int main()  {  char choose;  int check;  RECT \*allocated;  head=malloc(sizeof(RECT) ) ; //建立空闲分区表的初始状态  p=malloc(sizeof(RECT) ) ;  head->size=MAX;  head->address=0;  head->next=p;  maxblocknum=1;  p->size=MAX;  p->address=0;  p->next=NULL;  print(head) ; //输出空闲分区表的初始状态  //printf("Enter the allocation way (best or first (b/f))\n")；  //scanf("%c", &way) ;  way='f';  heada=malloc(sizeof(RECT) ) ; //建立已分配分区表的初始状态  heada->size=0;  heada->address=0;  heada->next=NULL;  //print(heada) ; //输出空闲分区表的初始状态  do  {  printf("Enter the allocate or reclaim(a/r) , or press other key to exit.\n") ;  scanf(" %c", &choose) ; //选择分配或回收  if (tolower(choose) =='a') //a为分配  {  printf("Input application：\n") ;  scanf(" %d", &application1) ; //输人申请的空间大小  assign1=assignment(head, application1) ; //调用分配函数以分配内存  if (assign1->address==-1) //分配不成功  printf("Too large application!Allocation fails!\n\n");  else //分配成功  printf("Allocation Success!ADDRESS=%5d\n", assign1->address) ;  printf("\n\*\*\*\*\*\*\*\*\*\*Unallocated Table\*\*\*\*\*\*\*\n");  print(head) ; //输出  printf("\n\*\*\*\*\*\*\*\*\*\*\*Allocated Table\*\*\*\*\*\*\*\*\*\*\*\*\*\n");  print(heada) ;  }  else if (tolower(choose) =='r') //回收内存  {  back=malloc(sizeof(RECT) ) ;  printf("Input address and Size：\n") ;  scanf("%d%d", &back->address, &back->size) ; //输入回收地址和大小  check=backcheck(head, back) ;  if (check==1)  {  if(tolower(way) =='f')  firstfit(head, heada, back) ; //首次适应算法回收  printf("\n\*\*\*\*\*\*\*\*\*Unallocated Table\*\*\*\*\*\*\n");  print(head) ;//输出  printf("\n\*\*w \*\*\*\*Allocated Table\*\*\*\*\*\*\*\*\*\n");  print(heada) ;  }  }  }while(tolower(choose) =='a'||tolower(choose) =='r') ;  exit(0) ;  } //main() end.  /\*-----------内存分配函数-----------\*/  RECT \*assignment(RECT \*head, int application)  {  RECT \*after, \*before, \*assign;  assign=malloc(sizeof(RECT) ) ; //申请分配空间  assign->size=application;  assign->next=NULL;  if(application>head->size ||application<0)  assign->address=-1; //申请无效  else  {  before=head;  after=head->next;  while(after->size<application) //遍历链表，查找合适的节点  {  before=before->next;  after=after->next;  }  if(after->size==application) //若节点大小等于申请大小，则完全分配  {  if(after->size==head->size)maxblocknum--;  before->next=after->next;  assign->address=after->address;  free(after) ;  }  else  {  if(after->size==head->size)maxblocknum--;  after->size=after->size-application; //大于申请空间时， 截取相应大小并分配  assign->address=after->address+after->size;  }  if (maxblocknum==0) //修改最大数和头节点  {  before=head;  head->size=0;  maxblocknum=1;  while(before!=NULL)  {  if(before->size>head->size)  {  head->size=before->size;  maxblocknum=1;  }  else if(before->size==head->size)  maxblocknum++;  before=before->next;  }  }  }  assign1=assign;  //修改已分配分区表，添加节点  after=heada;  while (after->next!=NULL)  after=after->next;  after->next=assign;  heada->size++;  return assign1; //返回分配给用户的地址  }  **/\*-------------首次适应算法------------\*/**  void firstfit(RECT \*head, RECT \*heada, RECT \*back1)  {  RECT \*before, \*after, \*back2;  int insert, del;  back2=malloc(sizeof(RECT) ) ;  back2->address=back1->address;  back2->size=back1->size;  back2->next=back1->next;  before=head;  after=head->next;  insert=0;  while(!insert) //将回收区插人空闲分区表  {  if((after==NULL)||((back1->address<=after->address) &&(back1->address>=before->address) ) )  {  before->next=back1;  back1->next=after;  insert=1;  }  else  {  before=before->next;  after=after->next;  }  }  if(back1->address==before->address+before->size) //与上一内存块合并  {  before->size=before->size+back1->size;  before->next=back1->next;  free(back1) ;  back1=before;  }  if((after!=NULL)&&(after->address==back1->address+back1->size) ) //与下一内存块合并  {  back1->size=back1->size+after->size;  back1->next=after->next;  free(after) ;  }  if(head->size<back1->size) //修改最大块值和最大块个数  {  head->size=back1->size;  maxblocknum=1;  }  else  if(head->size==back1->size)maxblocknum++; //修改已分配分区表，删除相应节点  before=heada;  after=heada->next;  del=0;  while(!del||after!=NULL) //将回收区从已分配分区表中删除  {  if((after->address==back2->address) &&(after->size==back2->size) )  {  before->next=after->next;  free(after) ;  del=1;  }  else  {  before=before->next;  after=after->next;  }  }  heada->size--;  }  /\*--------打印输出链表--------\*/  void print(RECT \*output)  {  RECT \*before;  int index;  before=output->next;  index=0;  if(output->next==NULL)  printf("NO part for print!\n") ;  else  {  printf("index\*\*\*\*address\*\*\*\*end\*\*\*\*\*size\*\*\*\*\n") ;  while(before!=NULL)  {  printf("----------------------\n");  printf("%-9d%-9d%-9d%-9d\n", index, before->address, before->  address+before->size-1, before->size) ;  printf("----------------------\n");  index++;;  before=before->next;  }  }  }  /\*检查回收块的合法性，back1为要回收节点的地址\*/  int backcheck(RECT \*head, RECT \*back1)  {  RECT \*before;  int check=1;  if(back1->address<0||back1->size<0)check=0; //地址和大小不能为负数  before=head->next;  while((before!=NULL) &&check) //地址不能和空闲分区表中的节点重叠  if(((back1->address<before->address) && (back1->address+back1->size>before -> address))||((back1->address>=before->address)&&(back1->address<before ->address+before->size) ) )  check=0;  else  before=before->next;  if(check==0) printf("Error input!\n") ;  return check;  }  **2、页面置换算法的模拟**  针对实验内容，需要注意以下几点。  (1) 首先使用随机函数srand()和rand()随机产生指令序列，然后将指令序列转换成相应的页面序列。  (2)设计页面类型、页面控制结构等数据结构。  (3)计算使用指定页面置换算法时的访问命中率。  随机生成页面访问序列时，也可以使用符合局部访问特性的随机生成算法：  (1) 确定虚拟内存的尺寸N，工作集的起始位置p，工作集中包含的页数e，工作集移动率m（每处理m个页面访问则将起始位置p +1），以及一个范围在0和1之间的值t；  (2) 生成m个取值范围在p和p + e间的随机数，并记录到页面访问序列串中；  (3) 生成一个随机数r，0 ≤ r ≤ 1；  (4) 如果r < t，则为p生成一个新值，否则p = (p + 1) mod N；  (5) 如果想继续加大页面访问序列串的长度，请返回第2步，否则结束。  下面给出的示例代码实现了计算使用FIFO算法时的访问命中率，但对于LRU和OPT算法未实现。在以下示例代码中，随机数的取值比较复杂，指令地址是按如下原则产生的：  ①50%的指令是顺序执行的；  ②25%的指令均匀地分布在前地址部分；  ③25%的指令均匀地分布在后地址部分。  #include <stdio.h>  #include <stdlib.h>  #include <string.h>  #include <unistd.h>  #define TRUE 1  #define FALSE 0  #define INVALID -1  #define total\_instruction 320 //模拟的指令数  #define total\_vp 32 //模拟的虚拟页面数  typedef struct //页面结构  {  int pn; //页号  int pfn; //内存块号  int counter; //一个周期内访问页面的次数  int time; //访问时间  }pl\_type;  pl\_type pl[total\_vp];  typedef struct pfc\_struct //页面控制结构  {  int pn; //页号  int pfn; //内存块号  struct pfc\_struct\*next;  } pfc\_type;  pfc\_type pfc[total\_vp]; //用户进程虚页控制结构  pfc\_type \*freepf\_head; //空内存页头指针  pfc\_type \*busypf\_head; //忙内存页头指针  pfc\_type \*busypf\_tail; //忙内存页尾指针  int disaffect; //页面失效次数  int a[total\_instruction]; //指令流数据组  int page[total\_instruction]; //每条指令所属页号  int offset[total\_instruction]; //每页装人10条指令后取得的页号偏移值  void initialize(); //初始化数据  void FIFO(); //计算使用FIFO算法时的访问命中率  void LRU(); //计算使用LRU算法时的访问命中率，未实现  void OPT(); //计算使用OPT算法时的访问命中率，未实现  int main()  {  int s, i, j;  srand (10\*getpid() );  s=(float)319\*rand()/32767/32767/2+1;  for(i=0 ; i<total\_instruction ; i+=4) //通过随机函数随机生成320条指令  {  if (s<0|| s>319)  {  printf("When i==%d， Error， s==%d\n", i, s);  exit(0);  }  a[i]=s;  a[i+1]=a[i]+1;  a[i+2] =(float) a[i] \*rand() /32767/32767/2;  a[i+3]=a[i+2]+1;  s=(float) (318-a[i+2] ) \*rand() /32767/32767/2+a[i+2] +2;  if ((a[i+2]>318)||(s>319))  printf("a[%d+2]，a number which is：%d and s==%d\n", i, a[i+2], s);  }  //将指令序列转换为页面地址流  for(i=0; i<total\_instruction; i++)  {  page[i] =a[i] /10;  offset[i] =a[i] %10;  }  //用户工作区从4个页面变换到32个页面  for(i=4; i<=32; i++)  {  printf("%2d page frames", i) ;  FIFO(i) ;  printf("\n") ;  }  }  void initialize(int total\_pf)  {  int i, diseffect;  diseffect = 0;  for(i=0; i<total\_vp; i++)  {  pl[i].pn=i;  pl[i].pfn=INVALID;  pl[i].counter=0;  pl[i].time=-1;  }  for(i=0; i<total\_pf-1; i++)  {  pfc[i].next=&pfc[i+1];  pfc[i].pfn=i;  }  pfc[total\_pf-1].next=NULL;  pfc[total\_pf-1].pfn=total\_pf-1;  freepf\_head=&pfc[0] ;  }  void FIFO(int total\_pf)  {  int i,j, diseffect ;  pfc\_type \*p ;  initialize(total\_pf) ;  busypf\_head=busypf\_tail=NULL;  for(i=0; i<total\_instruction; i++)  {  if (pl[page[i] ].pfn==INVALID) //页面失效  {  diseffect+=1; //页面失效次数  if (freepf\_head==NULL) //无空闲页面  {  p=busypf\_head->next;  pl[busypf\_head->pn].pfn=INVALID;  freepf\_head=busypf\_head; //释放忙页面的第一个页面  freepf\_head->next=NULL;  busypf\_head=p;  }  p=freepf\_head->next; //按FIFO方式将新页面调人内存页面  freepf\_head->next=NULL;  freepf\_head->pn=page[i];  pl[page[i] ] .pfn=freepf\_head->pfn;  if (busypf\_tail==NULL)  busypf\_head=busypf\_tail=freepf\_head;  else  {  busypf\_tail->next=freepf\_head; //减少一个空闲页面  busypf\_tail=freepf\_head;  }  freepf\_head=p;  }  }  printf(" FIFO: %6.4f", 1-(float) diseffect/320);  }  六、实验过程（80分）  本实验要求学生按照以下操作步骤（含代码、数据、图表等）完成实验，并提交实验结果截图。  （一）**内存的分配与回收**（40分）  **1、最先适应分配算法(FF)**  解题思路：  这段代码实现了一个简单的内存分配和释放过程，并使用了中文作为输出信息的语言。它通过使用链表数据结构来管理内存分区，通过菜单选项让用户选择分配内存、释放内存以及显示当前内存分区的状态。分配内存使用的是最先适应算法。  实验代码：  #include <stdio.h>  #include <stdlib.h>  #include <string.h>  #include <ctype.h>  #define MAX 640  struct node //定义分区  {  int address,size;  struct node \*next;  };  typedef struct node RECT; /\*-----------函数定义------------\*/  RECT \*assignment(RECT \*head, int application) ; //分配分区  //针对首次适应算法回收分区  void firstfit(RECT \*head, RECT \*heada, RECT \*back1) ;  void bestfit(RECT \*head, RECT \*back1) ; //针对**最佳适应算法**回收分区，待扩充  int backcheck(RECT \*head, RECT \*back1) ; //合法性检查  void print(RECT \*head) ; //输出已分配分区表或空闲分区  /\*----------变量定义-----------\*/  RECT \*head, \*heada, \*back, \*assign1, \*p;  int application1, maxblocknum;  char way; //用于定义分配方式：**首次适应、最佳适应、最坏适应**。目前未使用  int main()  {  char choose;  int check;  RECT \*allocated;  head=malloc(sizeof(RECT) ) ; //建立空闲分区表的初始状态  p=malloc(sizeof(RECT) ) ;  head->size=MAX;  head->address=0;  head->next=p;  maxblocknum=1;  p->size=MAX;  p->address=0;  p->next=NULL;  print(head) ; //输出空闲分区表的初始状态  //printf("Enter the allocation way (best or first (b/f))\n")；  //scanf("%c", &way) ;  way='f';  heada=malloc(sizeof(RECT) ) ; //建立已分配分区表的初始状态  heada->size=0;  heada->address=0;  heada->next=NULL;  //print(heada) ; //输出空闲分区表的初始状态  do  {  printf("Enter the allocate or reclaim(a/r) , or press other key to exit.\n") ;  scanf(" %c", &choose) ; //选择分配或回收  if (tolower(choose) =='a') //a为分配  {  printf("Input application：\n") ;  scanf(" %d", &application1) ; //输人申请的空间大小  assign1=assignment(head, application1) ; //调用分配函数以分配内存  if (assign1->address==-1) //分配不成功  printf("Too large application!Allocation fails!\n\n");  else //分配成功  printf("Allocation Success!ADDRESS=%5d\n", assign1->address) ;  printf("\n\*\*\*\*\*\*\*\*\*\*Unallocated Table\*\*\*\*\*\*\*\n");  print(head) ; //输出  printf("\n\*\*\*\*\*\*\*\*\*\*\*Allocated Table\*\*\*\*\*\*\*\*\*\*\*\*\*\n");  print(heada) ;  }  else if (tolower(choose) =='r') //回收内存  {  back=malloc(sizeof(RECT) ) ;  printf("Input address and Size：\n") ;  scanf("%d%d", &back->address, &back->size) ; //输入回收地址和大小  check=backcheck(head, back) ;  if (check==1)  {  if(tolower(way) =='f')  firstfit(head, heada, back) ; //首次适应算法回收  printf("\n\*\*\*\*\*\*\*\*\*Unallocated Table\*\*\*\*\*\*\n");  print(head) ;//输出  printf("\n\*\*w \*\*\*\*Allocated Table\*\*\*\*\*\*\*\*\*\n");  print(heada) ;  }  }  }while(tolower(choose) =='a'||tolower(choose) =='r') ;  exit(0) ;  } //main() end.  /\*-----------内存分配函数-----------\*/  RECT \*assignment(RECT \*head, int application)  {  RECT \*after, \*before, \*assign;  assign=malloc(sizeof(RECT) ) ; //申请分配空间  assign->size=application;  assign->next=NULL;  if(application>head->size ||application<0)  assign->address=-1; //申请无效  else  {  before=head;  after=head->next;  while(after->size<application) //遍历链表，查找合适的节点  {  before=before->next;  after=after->next;  }  if(after->size==application) //若节点大小等于申请大小，则完全分配  {  if(after->size==head->size)maxblocknum--;  before->next=after->next;  assign->address=after->address;  free(after) ;  }  else  {  if(after->size==head->size)maxblocknum--;  after->size=after->size-application; //大于申请空间时， 截取相应大小并分配  assign->address=after->address+after->size;  }  if (maxblocknum==0) //修改最大数和头节点  {  before=head;  head->size=0;  maxblocknum=1;  while(before!=NULL)  {  if(before->size>head->size)  {  head->size=before->size;  maxblocknum=1;  }  else if(before->size==head->size)  maxblocknum++;  before=before->next;  }  }  }  assign1=assign;  //修改已分配分区表，添加节点  after=heada;  while (after->next!=NULL)  after=after->next;  after->next=assign;  heada->size++;  return assign1; //返回分配给用户的地址  }  **/\*-------------首次适应算法------------\*/**  void firstfit(RECT \*head, RECT \*heada, RECT \*back1)  {  RECT \*before, \*after, \*back2;  int insert, del;  back2=malloc(sizeof(RECT) ) ;  back2->address=back1->address;  back2->size=back1->size;  back2->next=back1->next;  before=head;  after=head->next;  insert=0;  while(!insert) //将回收区插人空闲分区表  {  if((after==NULL)||((back1->address<=after->address) &&(back1->address>=before->address) ) )  {  before->next=back1;  back1->next=after;  insert=1;  }  else  {  before=before->next;  after=after->next;  }  }  if(back1->address==before->address+before->size) //与上一内存块合并  {  before->size=before->size+back1->size;  before->next=back1->next;  free(back1) ;  back1=before;  }  if((after!=NULL)&&(after->address==back1->address+back1->size) ) //与下一内存块合并  {  back1->size=back1->size+after->size;  back1->next=after->next;  free(after) ;  }  if(head->size<back1->size) //修改最大块值和最大块个数  {  head->size=back1->size;  maxblocknum=1;  }  else  if(head->size==back1->size)maxblocknum++; //修改已分配分区表，删除相应节点  before=heada;  after=heada->next;  del=0;  while(!del||after!=NULL) //将回收区从已分配分区表中删除  {  if((after->address==back2->address) &&(after->size==back2->size) )  {  before->next=after->next;  free(after) ;  del=1;  }  else  {  before=before->next;  after=after->next;  }  }  heada->size--;  }  /\*--------打印输出链表--------\*/  void print(RECT \*output)  {  RECT \*before;  int index;  before=output->next;  index=0;  if(output->next==NULL)  printf("NO part for print!\n") ;  else  {  printf("index\*\*\*\*address\*\*\*\*end\*\*\*\*\*size\*\*\*\*\n") ;  while(before!=NULL)  {  printf("----------------------\n");  printf("%-9d%-9d%-9d%-9d\n", index, before->address, before->  address+before->size-1, before->size) ;  printf("----------------------\n");  index++;;  before=before->next;  }  }  }  /\*检查回收块的合法性，back1为要回收节点的地址\*/  int backcheck(RECT \*head, RECT \*back1)  {  RECT \*before;  int check=1;  if(back1->address<0||back1->size<0)check=0; //地址和大小不能为负数  before=head->next;  while((before!=NULL) &&check) //地址不能和空闲分区表中的节点重叠  if(((back1->address<before->address) && (back1->address+back1->size>before -> address))||((back1->address>=before->address)&&(back1->address<before ->address+before->size) ) )  check=0;  else  before=before->next;  if(check==0) printf("Error input!\n") ;  return check;  }  实验结果：    性能分析：  时间复杂度分析：  分配内存操作的时间复杂度为O(n)，其中n为分区数量。在最坏情况下，需要遍历所有分区才能找到合适的分区进行内存分配。  释放内存操作的时间复杂度为O(n)，其中n为分区数量。需要遍历分区链表以找到要释放的分区。  显示内存状态的时间复杂度为O(n)，其中n为分区数量。需要遍历分区链表并打印每个分区的信息。  空间复杂度分析：  内存分配和释放过程中不需要额外的空间，仅使用了一个链表来存储分区信息。因此，空间复杂度为O(n)，其中n为分区数量。  死锁分析：  该程序并没有引入并发或多线程的概念，因此不存在死锁的问题。  算法性能分析：  该程序使用了最先适应算法来分配内存。在分配内存时，它会遍历分区链表以找到第一个大小足够的分区进行分配。这种算法相对简单，但可能会导致产生碎片化的问题，即剩余空间被拆分为多个小的不可用分区，影响内存利用率。  对于释放内存的操作，该程序通过遍历分区链表来找到要释放的分区并进行释放。这个过程相对简单且效率较高。  **2、最优适应分配算法(BF)**  解题思路：  该代码实现了最优适应分配算法，通过遍历分区链表找到最适合的分区来进行分配。在每次分配时，遍历分区链表，找到大小大于等于申请大小且碎片化更小的分区。然后，将该分区进行划分，分配给申请的应用程序，并将剩余部分作为一个新的分区插入链表中。最后，打印出分区的起始地址和大小  实验代码：  #include <stdio.h>  #include <stdlib.h>  #include<limits.h>  // 定义分区结构  struct node {  int address;  int size;  struct node\* next;  };  typedef struct node RECT;  // 最优适应分配算法  RECT\* bestFitAllocation(RECT\* head, int application) {  RECT\* current = head;  RECT\* bestFit = NULL;  int minFragmentation = INT\_MAX; // 最小碎片化大小  // 找到最适合的分区  while (current != NULL) {  // 分区大小大于等于申请大小并且碎片化更小  if (current->size >= application && current->size - application < minFragmentation) {  bestFit = current;  minFragmentation = current->size - application;  }  current = current->next;  }  // 如果找到了最适合的分区  if (bestFit != NULL) {  // 分配分区  RECT\* newPartition = (RECT\*)malloc(sizeof(RECT));  newPartition->address = bestFit->address + application;  newPartition->size = bestFit->size - application;  newPartition->next = bestFit->next;  bestFit->size = application;  bestFit->next = newPartition;  }  return head;  }  int main() {  // 初始化分区链表  RECT\* head = (RECT\*)malloc(sizeof(RECT));  head->address = 0;  head->size = 100;  head->next = NULL;  // 执行最优适应分配算法  head = bestFitAllocation(head, 50);  head = bestFitAllocation(head, 30);  head = bestFitAllocation(head, 20);  // 打印分区信息  RECT\* current = head;  while (current != NULL) {  printf("分区起始地址：%d，分区大小：%d\n", current->address, current->size);  current = current->next;  }  return 0;  }  实验结果：    性能分析：  最优适应分配算法在性能方面具有一定的优势，尤其在处理动态内存分配时表现良好。下面是对其性能进行分析的几个方面：  分区利用率：最优适应分配算法能够选择大小最适合的分区进行分配，从而减少了内存碎片的产生。它会尽量选择与申请大小最接近的空闲分区，因此可以更充分地利用可用内存空间，提高分区利用率。  分区查找效率：在每次分配时，最优适应算法需要遍历整个分区链表以找到最适合的分区。虽然需要遍历的时间复杂度为O(n)，其中n为分区的数量，但由于分区链表通常不会很长，所以查找效率较高。  分区合并效率：最优适应分配算法在释放分区时，可能需要合并相邻的空闲分区。由于分配的分区大小是相对较小的，因此在合并时可能会产生较多的空闲分区。这可能会导致后续分配操作的开销增加，因为需要遍历更多的分区来找到合适的分区。  算法复杂度：最优适应算法的时间复杂度与分区数量成正比，即O(n)，其中n为分区的数量。因此，当分区数量较大时，算法的性能可能会受到影响。但在实际应用中，分区数量通常不会很大，因此算法的性能仍然是可接受的。  **3、最坏适应分配算法(WF)**  解题思路：  该算法通过遍历所有分区，找到最大的满足分配要求的分区来进行分配。如果找到合适的分区，则更新该分区的大小和起始地址，并在必要时移除大小为0的分区。最后，打印分配后的分区情况。  实验代码：  #include <stdio.h>  #include <stdlib.h>  struct node {  int address;  int size;  struct node\* next;  };  typedef struct node RECT;  RECT\* worst\_fit\_allocation(RECT\* head, int application);  int main() {  RECT\* head = NULL;  // 创建初始分区  RECT\* partition1 = (RECT\*)malloc(sizeof(RECT));  partition1->address = 0;  partition1->size = 100;  partition1->next = NULL;  head = partition1;  RECT\* partition2 = (RECT\*)malloc(sizeof(RECT));  partition2->address = 200;  partition2->size = 150;  partition2->next = NULL;  partition1->next = partition2;  RECT\* partition3 = (RECT\*)malloc(sizeof(RECT));  partition3->address = 400;  partition3->size = 300;  partition3->next = NULL;  partition2->next = partition3;  // 调用最坏适应分配算法  int application = 250;  head = worst\_fit\_allocation(head, application);  // 打印分配后的分区情况  RECT\* current = head;  printf("最坏适应分配后的分区情况：\n");  while (current != NULL) {  printf("分区起始地址：%d，分区大小：%d\n", current->address, current->size);  current = current->next;  }  // 释放分区内存  current = head;  while (current != NULL) {  RECT\* temp = current;  current = current->next;  free(temp);  }  return 0;  }  RECT\* worst\_fit\_allocation(RECT\* head, int application) {  RECT\* current = head;  RECT\* selected\_partition = NULL;  int max\_size = -1;  // 找到最大的满足分配要求的分区  while (current != NULL) {  if (current->size >= application && current->size > max\_size) {  max\_size = current->size;  selected\_partition = current;  }  current = current->next;  }  // 如果找到合适的分区  if (selected\_partition != NULL) {  // 更新分区信息  selected\_partition->size -= application;  selected\_partition->address += application;  // 如果分区大小变为0，则移除该分区  if (selected\_partition->size == 0) {  RECT\* temp = head;  if (temp == selected\_partition) {  head = head->next;  } else {  while (temp->next != selected\_partition) {  temp = temp->next;  }  temp->next = selected\_partition->next;  }  free(selected\_partition);  }  }  return head;  }  实验结果：    性能分析：  最坏适应分配算法的性能可以从以下几个方面进行分析：  时间复杂度：最坏适应分配算法需要遍历所有分区来找到最大的满足分配要求的分区。因此，时间复杂度取决于分区的数量和分区链表的长度。在最坏情况下，需要遍历所有分区，因此时间复杂度为O(n)，其中n是分区的数量。  分区利用率：最坏适应分配算法的一个缺点是可能导致分区利用率较低。因为它倾向于选择最大的可用分区，这可能会导致较小的分区无法被充分利用。当有多个小的可用分区时，最坏适应分配算法可能会选择其中最大的分区，导致较小的分区无法容纳更小的应用。  分区碎片化：最坏适应分配算法可能会导致分区碎片化。当较大的分区被分配给较小的应用时，会产生分区内部的碎片。这些碎片无法再被其他应用利用，导致分区的整体利用率降低。  算法的优点：最坏适应分配算法的优点是可以避免出现较小的分区无法容纳大的应用的情况。它倾向于选择最大的可用分区，因此适用于需要较大连续内存空间的应用场景  （二）**请求分页系统中的置换算法**（40分）  1．最佳置换算法（Optimal）  解题思路：  该程序实现了最佳置换算法（OPT）来计算页面的访问命中率。程序中使用了三个全局变量disaffect表示页面失效次数，a表示指令流数据组，page表示每条指令所属的页号。  在OPT()函数中，首先初始化内存块数组mem，然后遍历指令流数据组，判断当前页面是否已经在内存中。如果页面未在内存中，则发生页面失效，计算未来最长时间内不会被访问的页面，并进行置换  实验代码：  #include <stdio.h>  #include <stdlib.h>  #include <string.h>  #include <unistd.h>  #define TRUE 1  #define FALSE 0  #define INVALID -1  #define total\_instruction 320 // 模拟的指令数  #define total\_vp 32 // 模拟的虚拟页面数  typedef struct // 页面结构  {  int pn; // 页号  int pfn; // 内存块号  int counter; // 一个周期内访问页面的次数  int time; // 访问时间  } pl\_type;  pl\_type pl[total\_vp];  int disaffect; // 页面失效次数  int a[total\_instruction]; // 指令流数据组  int page[total\_instruction]; // 每条指令所属页号  int offset[total\_instruction]; // 每页装载10条指令后取得的页号偏移值  void initialize(); // 初始化数据  void FIFO(); // 计算使用FIFO算法时的访问命中率  void LRU(); // 计算使用LRU算法时的访问命中率，未实现  void OPT(); // 计算使用OPT算法时的访问命中率  void initialize() {  // 初始化数据  disaffect = 0;  // 生成指令流数据组  for (int i = 0; i < total\_instruction; i++) {  a[i] = rand() % 256;  }  // 计算每条指令所属页号和每页装载10条指令后的页号偏移值  for (int i = 0; i < total\_instruction; i++) {  page[i] = a[i] / 10;  offset[i] = a[i] % 10;  }  }  void FIFO() {  int mem[total\_vp]; // 内存块数组  int p = 0; // 内存块指针  int hit = 0; // 命中次数  // 初始化内存块数组  for (int i = 0; i < total\_vp; i++) {  mem[i] = INVALID;  }  for (int i = 0; i < total\_instruction; i++) {  int current\_page = page[i];  int found = FALSE;  // 判断当前页面是否已在内存中  for (int j = 0; j < total\_vp; j++) {  if (mem[j] == current\_page) {  found = TRUE;  hit++;  break;  }  }  if (!found) {  // 页面失效，进行置换  disaffect++;  mem[p] = current\_page;  p = (p + 1) % total\_vp;  }  }  // 计算命中率  double hit\_rate = (double)hit / total\_instruction \* 100;  double miss\_rate = 100 - hit\_rate;  // 打印结果  printf("使用FIFO算法的访问命中率：\n");  printf("命中次数：%d\n", hit);  printf("命中率：%.2lf%%\n", hit\_rate);  printf("失效次数：%d\n", disaffect);  printf("失效率：%.2lf%%\n", miss\_rate);  }  void LRU() {  // TODO: 实现LRU算法的访问命中率计算  }  void OPT() {  int mem[total\_vp]; // 内存块数组  int hit = 0; // 命中次数  // 初始化内存块数组  for (int i = 0; i < total\_vp; i++) {  mem[i] = INVALID;  }  for (int i = 0; i < total\_instruction; i++) {  int current\_page = page[i];  int found = FALSE;  // 判断当前页面是否已在内存中  for (int j = 0; j < total\_vp; j++) {  if (mem[j] == current\_page) {  found = TRUE;  hit++;  break;  }  }  if (!found) {  // 页面失效，进行置换  disaffect++;  int max\_distance = 0;  int replace\_index = -1;  // 寻找未来最长时间内不会被访问的页面  for (int j = 0; j < total\_vp; j++) {  int next\_distance = 0;  int next\_page = mem[j];  // 在未来的指令流中寻找下一次访问页面的位置  for (int k = i + 1; k < total\_instruction; k++) {  if (page[k] == next\_page) {  break;  }  next\_distance++;  }  if (next\_distance > max\_distance) {  max\_distance = next\_distance;  replace\_index = j;  }  }  if (replace\_index != -1) {  mem[replace\_index] = current\_page;  }  }  }  // 计算命中率  double hit\_rate = (double)hit / total\_instruction \* 100;  double miss\_rate = 100 - hit\_rate;  // 打印结果  printf("使用OPT算法的访问命中率：\n");  printf("命中次数：%d\n", hit);  printf("命中率：%.2lf%%\n", hit\_rate);  printf("失效次数：%d\n", disaffect);  printf("失效率：%.2lf%%\n", miss\_rate);  }  int main() {  initialize();  FIFO();  // LRU();  // OPT();  return 0;  }  实验结果：    性能分析：  对于最佳置换算法（OPT），它是一种理论上的最佳算法，即在未来的指令流中选择最长时间内不会被访问的页面进行置换。由于它可以预测未来的访问模式，所以在理论上可以达到最佳的命中率。  性能分析如下：  时间复杂度：在每个指令的执行过程中，需要遍历内存中的所有页面，然后再遍历未来的指令流来寻找最长时间内不会被访问的页面。因此，时间复杂度为O(n^2)，其中n是虚拟页面的数量。  空间复杂度：需要使用一个大小为total\_vp的内存块数组来存储页面，因此空间复杂度为O(n)。  优点：  理论上能够达到最佳的命中率，尽可能减少页面失效的次数。  不受访问模式的影响，可以适用于任何指令流。  缺点：  实际上，由于无法准确预测未来的访问模式，实现OPT算法需要对未来指令流进行预测。这在实际中是不可行的，因为无法事先获得完整的未来指令流信息。  需要额外的内存空间来维护内存块数组。  算法复杂度高，执行时间较长。  2．先进先出法（Fisrt In First Out）  算法分析：  使用先进先出（FIFO）页面置换算法计算访问命中率。在初始化函数initialize()中，首先初始化各数据结构和参数。然后，通过循环遍历指令流，在每个指令处进行页面访问判断。如果当前指令的页面号在内存中不存在，即缺页，会触发页面置换。使用FIFO算法，选择下一个牺牲页面进行替换，并更新相应的数据结构。最后，根据页面失效次数计算访问命中率，并打印结果  实验代码：  实验#include <stdio.h>  #include <stdlib.h>  #include <string.h>  #include <unistd.h>  #define TRUE 1  #define FALSE 0  #define INVALID -1  #define total\_instruction 320 // 模拟的指令数  #define total\_vp 32 // 模拟的虚拟页面数  typedef struct {  int pn; // 页号  int pfn; // 内存块号  int counter; // 一个周期内访问页面的次数  int time; // 访问时间  } pl\_type;  pl\_type pl[total\_vp];  typedef struct pfc\_struct {  int pn; // 页号  int pfn; // 内存块号  struct pfc\_struct\* next;  } pfc\_type;  pfc\_type pfc[total\_vp]; // 用户进程虚页控制结构  pfc\_type\* freepf\_head; // 空内存页头指针  pfc\_type\* busypf\_head; // 忙内存页头指针  pfc\_type\* busypf\_tail; // 忙内存页尾指针  int disaffect; // 页面失效次数  int a[total\_instruction]; // 指令流数据组  int page[total\_instruction]; // 每条指令所属页号  int offset[total\_instruction]; // 每页装入10条指令后取得的页号偏移值  void initialize(); // 初始化数据  void FIFO(); // 计算使用FIFO算法时的访问命中率  void LRU(); // 计算使用LRU算法时的访问命中率，未实现  void OPT(); // 计算使用OPT算法时的访问命中率，未实现  int main() {  initialize();  FIFO();  return 0;  }  void initialize() {  // 初始化数据的代码  // ...  }  void FIFO() {  int mem\_block\_count = 4; // 内存块的数量  int hit\_count = 0; // 命中次数  int i, j, k;  int pf\_num = 0; // 实际分配给用户的物理页面数  int current\_page; // 当前访问的页面  // 初始化页面控制结构  for (i = 0; i < total\_vp; i++) {  pfc[i].pn = INVALID;  pfc[i].pfn = INVALID;  pfc[i].next = NULL;  }  // 初始化空内存页链表  freepf\_head = &pfc[0];  for (i = 0; i < mem\_block\_count - 1; i++) {  pfc[i].next = &pfc[i + 1];  }  pfc[mem\_block\_count - 1].next = NULL;  // 遍历指令流  for (i = 0; i < total\_instruction; i++) {  current\_page = page[i];  // 检查页面是否在内存中  pfc\_type\* temp = busypf\_head;  while (temp != NULL) {  if (temp->pn == current\_page) {  hit\_count++; // 命中  break;  }  temp = temp->next;  }  // 页面不在内存中，进行页面调度  if (temp == NULL) {  // 获取一个空闲的内存块  pfc\_type\* free\_block = freepf\_head;  freepf\_head = freepf\_head->next;  // 将页面分配给内存块  free\_block->pn = current\_page;  free\_block->pfn = pf\_num++;  free\_block->next = NULL;  // 将内存块加入到忙内存页链表的尾部  if (busypf\_head == NULL) {  busypf\_head = free\_block;  busypf\_tail = free\_block;  } else {  busypf\_tail->next = free\_block;  busypf\_tail = free\_block;  }  // 如果内存块数超过了限制，执行FIFO页面置换  if (pf\_num > mem\_block\_count) {  pfc\_type\* replace\_block = busypf\_head;  // 更新忙内存页链表的头指针  busypf\_head = busypf\_head->next;  // 将被替换的页面移动到空闲内存页链表的尾部  replace\_block->next = NULL;  freepf\_head = replace\_block;  // 更新页面失效次数  disaffect++;  }  }  }  // 计算访问命中率  float hit\_rate = (float)hit\_count / total\_instruction \* 100;  printf("FIFO算法的访问命中率：%.2f%%\n", hit\_rate);  }  void LRU() {  // LRU算法的实现代码  // ...  }  void OPT() {  // OPT算法的实现代码  // ...  }  结果：    性能分析：  下面对其性能进行简要分析：  页面置换算法选择：该代码中使用了FIFO算法作为页面置换策略。FIFO算法的原理是按照页面进入内存的顺序进行置换，即最早进入内存的页面被置换出去。FIFO算法简单且易于实现，但它没有考虑页面的访问频率和重要性，可能导致低效的页面置换决策。  访问命中率：代码通过统计命中次数来计算访问命中率。命中次数表示在指令流中访问到了已经在内存中的页面的次数。访问命中率是衡量页面置换算法效果的重要指标，高的命中率意味着较好的性能。  页面分配和置换：代码中通过维护空闲内存页链表和忙碌内存页链表来管理页面的分配和置换。当需要分配新的页面时，从空闲链表中获取一个空闲内存块。当内存块数量超过限制时，执行FIFO页面置换，将最早进入内存的页面替换出去。  性能分析：FIFO算法的性能受到多种因素的影响，包括内存块数量、指令流的特征以及页面访问模式等。下面是一些可能影响性能的因素：  内存块数量：内存块数量会影响页面的分配和置换频率。较少的内存块可能导致更频繁的页面置换，增加了页面失效的可能性，从而降低了访问命中率。较多的内存块可以减少页面置换的频率，提高了访问命中率。  指令流特征：指令流的访问模式会影响页面命中率。如果指令流中存在局部性原理，即对同一页面的连续访问较为频繁，那么命中率可能较高。相反，如果指令流的访问分布较为均匀或具有较大的跳跃性，那么命中率可能较低。  页面置换策略：FIFO算法在某些情况下可能导致较低的命中率。因为它只考虑了页面的进入顺序，而没有考虑页面的访问频率和重要性。当存在访问频繁的页面时，FIFO算法可能会频繁地将这些页面置换出去，从而降低命中率。  页面大小和页面失效次数：页面大小与页面失效次数的关系也会影响性能。较小的页面大小可能导致更频繁的页面失效，从而降低命中率。增加页面大小可以减少页面失效次数，提高命中率。  3．最近最久未使用（Least Recently Used）  解题思路：  代码实现了LRU（最近最久未使用）算法的页面置换过程，并计算了访问命中率。程序通过初始化页面结构数组、生成模拟指令流数据、计算页号和偏移值等步骤，模拟了指令的访问过程。在每个指令访问时，根据当前页号判断是否命中，如果命中则更新页面的访问时间；如果未命中，则根据最久未使用的页面进行置换操作，并更新内存块中的页面信息。最后，计算命中率并打印结果  实验代码：  #include <stdio.h>  #include <stdlib.h>  #include <string.h>  #include <unistd.h>  #define TRUE 1  #define FALSE 0  #define INVALID -1  #define total\_instruction 320 //模拟的指令数  #define total\_vp 32 //模拟的虚拟页面数  typedef struct //页面结构  {  int pn; //页号  int pfn; //内存块号  int counter; //一个周期内访问页面的次数  int time; //访问时间  } pl\_type;  pl\_type pl[total\_vp];  int a[total\_instruction]; //指令流数据组  int page[total\_instruction]; //每条指令所属页号  int offset[total\_instruction]; //每页装入10条指令后取得的页号偏移值  void initialize(); //初始化数据  void FIFO(); //计算使用FIFO算法时的访问命中率  void LRU(); //计算使用LRU算法时的访问命中率  void OPT(); //计算使用OPT算法时的访问命中率，未实现  int main()  {  initialize();  LRU();  return 0;  }  void initialize()  {  // 初始化页面结构数组pl  for (int i = 0; i < total\_vp; i++) {  pl[i].pn = i;  pl[i].pfn = INVALID;  pl[i].counter = 0;  pl[i].time = INVALID;  }  // 生成模拟指令流数据  for (int i = 0; i < total\_instruction; i++) {  a[i] = rand() % total\_vp; // 随机生成指令所属页号  }  // 计算每条指令的页号和偏移值  for (int i = 0; i < total\_instruction; i++) {  page[i] = a[i] / 10;  offset[i] = a[i] % 10;  }  }  void LRU()  {  int mem[3] = {-1, -1, -1}; // 内存块，初始时没有页面驻留  int hit\_count = 0; // 命中次数  for (int i = 0; i < total\_instruction; i++) {  int cur\_page = page[i]; // 当前指令所属页号  // 查找页面是否在内存中  int found = FALSE;  for (int j = 0; j < 3; j++) {  if (mem[j] == cur\_page) {  found = TRUE;  break;  }  }  if (found) {  // 命中，更新页面的访问时间  for (int j = 0; j < total\_vp; j++) {  if (pl[j].pn == cur\_page) {  pl[j].time = i;  break;  }  }  hit\_count++;  } else {  // 未命中，需要进行页面置换  int min\_time = total\_instruction + 1;  int min\_index = -1;  // 查找最久未使用的页面  for (int j = 0; j < total\_vp; j++) {  if (pl[j].pfn != INVALID) {  if (pl[j].time < min\_time) {  min\_time = pl[j].time;  min\_index = j;  }  }  }  if (min\_index != -1) {  // 找到最久未使用的页面，进行置换  int pfn = pl[min\_index].pfn; // 被置换出的页面号  // 更新页面结构  pl[min\_index].pfn = INVALID;  pl[min\_index].counter = 0;  pl[min\_index].time = INVALID;  // 更新内存块  for (int j = 0; j < 3; j++) {  if (mem[j] == pfn) {  mem[j] = cur\_page;  break;  }  }  }  }  }  // 计算命中率并打印结果  float hit\_rate = (float)hit\_count / total\_instruction \* 100;  printf("LRU算法的访问命中率：%.2f%%\n", hit\_rate);  }  void FIFO()  {  // TODO: 实现FIFO算法的访问命中率计算  }  void OPT()  {    实验结果：    性能分析：  下面对代码的性能进行简要分析：  时间复杂度：  初始化数据的时间复杂度为O(total\_vp + total\_instruction)，其中total\_vp为虚拟页面数，total\_instruction为指令数。  LRU算法的页面置换过程需要遍历内存块和页面结构数组，时间复杂度为O(total\_instruction)。  计算命中率的时间复杂度为O(1)。  总体而言，代码的时间复杂度较低，适用于模拟较小规模的页面置换场景。  空间复杂度：  页面结构数组pl和页面控制结构pfc的空间复杂度均为O(total\_vp)，其中total\_vp为虚拟页面数。  内存块mem和其他辅助变量所占用的空间复杂度为O(1)。  代码的空间复杂度与虚拟页面数相关，但不会随指令数的增加而线性增长。  访问命中率：  代码根据指令流数据模拟了页面的访问过程，并计算了LRU算法的访问命中率。  LRU算法通过维护页面的访问时间来决定置换页面，具有较好的页面命中性能。  通过计算命中率，可以评估页面置换算法的性能。  六．实验结果分析（20分）  1．实验思考  （1）修改上述程序，使分配内存时从低地址开始。  #include <stdio.h>  #include <stdlib.h>  #include <string.h>  #include <unistd.h>  #define MEMORY\_SIZE 1024 // 内存大小  #define TRUE 1  #define FALSE 0  #define INVALID -1  typedef struct {  int address;  int size;  struct node\* next;  } RECT;  RECT\* assignment(RECT\* head, int application);  void printMemoryMap(RECT\* head);  int main() {  RECT\* memory\_map = (RECT\*)malloc(sizeof(RECT)); // 内存块链表头节点  memory\_map->address = 0;  memory\_map->size = MEMORY\_SIZE;  memory\_map->next = NULL;  // 分配内存  memory\_map = assignment(memory\_map, 256);  printMemoryMap(memory\_map);  // 分配内存  memory\_map = assignment(memory\_map, 512);  printMemoryMap(memory\_map);  // 分配内存  memory\_map = assignment(memory\_map, 128);  printMemoryMap(memory\_map);  free(memory\_map);  return 0;  }  RECT\* assignment(RECT\* head, int application) {  RECT\* p = head;  RECT\* pre = NULL;  while (p != NULL) {  if (p->size >= application) {  // 找到足够大的空闲内存块  if (p->size == application) {  // 内存块大小正好等于申请大小，直接分配  if (pre == NULL) {  // 分配的是第一个内存块  head = p->next;  } else {  pre->next = p->next;  }  free(p);  } else {  // 内存块大小大于申请大小，拆分内存块  RECT\* new\_node = (RECT\*)malloc(sizeof(RECT));  new\_node->address = p->address;  new\_node->size = application;  new\_node->next = NULL;  p->address += application;  p->size -= application;  if (pre == NULL) {  // 分配的是第一个内存块  head = new\_node;  } else {  pre->next = new\_node;  }  }  break;  }  pre = p;  p = p->next;  }  return head;  }  void printMemoryMap(RECT\* head) {  RECT\* p = head;  printf("Memory Map:\n");  while (p != NULL) {  printf("Address: %d, Size: %d\n", p->address, p->size);  p = p->next;  }  printf("\n");  }  （2）实现计算使用LRU和OPT算法时的访问命中率。  （3）修改指令序列的产生方法(如简单生成320条指令，指令地址无具体要求)，并与以上示例代码的结果进行比较，说明随机指令序列的产生对程序运行结果有何影响。  修改指令序列的产生方法(如简单生成320条指令，指令地址无具体要求)，并与以上示例代码的结果进行比较，说明随机指令序列的产生对程序运行结果有何影响。  访问模式的随机性：随机生成的指令序列可能导致不同的访问模式，包括顺序访问、随机访问、局部性访问等。不同的访问模式可能会对程序的缓存命中率和内存访问模式产生影响。  内存局部性的变化：随机指令序列可能导致程序在执行过程中出现不连续的内存访问。这可能会增加缓存失效的次数，对程序性能产生一定的影响。  指令地址的分布：随机指令序列的生成会导致指令地址的分布具有一定的随机性。这可能会影响分页算法或页面置换算法的性能，如对页面的选择和置换策略产生影响。  （4）分析比较各种页面置换算法之间的差异。  FIFO (First-In-First-Out) 算法：  简单易实现：使用队列数据结构来维护页面访问顺序。  缺乏局部性：无法考虑页面的访问频率和重要性，容易产生Belady现象，即页面置换次数随内存增加而增加。  不公平：可能导致长时间驻留在内存中但很少被访问的页面，影响了缓存命中率。  LRU (Least Recently Used) 算法：  基于局部性原理：将最近最少使用的页面置换出去。  较好的性能：相对于FIFO算法，LRU算法在一定程度上解决了Belady现象，通常能够获得更好的缓存命中率。  实现较复杂：需要记录页面的访问时间或使用计数，实现起来相对复杂。  OPT (Optimal) 算法：  理想的页面置换算法：在某个时刻，选择将未来最长时间内不再被访问的页面进行置换。  需要未来信息：OPT算法需要事先获得对未来指令访问情况的预测，这是不可实现的。  作为比较基准：OPT算法通常被用作其他算法性能评估的基准，用来评估其他算法的近似效果。  2．分析、总结  分配算法的分析和总结：  首次适应算法 (First Fit)：分配时从空闲分区链表中找到第一个满足大小要求的分区。适用于动态分区管理，简单高效，但可能导致碎片化问题。  最佳适应算法 (Best Fit)：分配时从空闲分区链表中找到最小的满足大小要求的分区。能够更好地利用内存空间，但分配效率较低。  最坏适应算法 (Worst Fit)：分配时从空闲分区链表中找到最大的满足大小要求的分区。容易产生碎片，但在某些场景下能够提供更好的性能。  总结：这些分配算法主要解决了动态分区分配的问题。首次适应算法简单高效，适用于常见的内存管理需求。最佳适应算法能够更好地利用内存空间，但分配效率较低。最坏适应算法则在某些场景下能够提供更好的性能，但容易产生碎片。  置换算法的分析和总结：  FIFO (First-In-First-Out) 算法：简单易实现，但缺乏局部性，容易产生Belady现象，性能较差。  LRU (Least Recently Used) 算法：基于局部性原理，能够更好地利用局部访问模式，性能相对较好，但实现较复杂。  OPT (Optimal) 算法：理想的算法，选择未来最长时间内不再被访问的页面置换，但无法实现。  总结：这些置换算法主要解决了内存页置换的问题。FIFO算法简单易实现，但性能较差；LRU算法相对较好，能够获得较好的缓存命中率，但实现复杂；OPT算法是理想的算法，但无法实现。在实际应用中，常用的页面置换算法是LRU算法，它在综合性能和实现难度之间取得了一定的平衡。  3．体会  通过分析和讨论分配算法和置换算法，我对内存管理中的关键问题和不同的解决方案有了更深入的理解。以下是我对这些算法的体会：  分配算法的选择：不同的分配算法适用于不同的场景。首次适应算法简单高效，适用于常见的内存管理需求；最佳适应算法能够更好地利用内存空间，但分配效率较低；最坏适应算法在某些情况下能够提供更好的性能，但容易产生碎片。在实际应用中，需要根据具体的内存管理需求和性能要求选择合适的算法。  置换算法的权衡：不同的置换算法在性能和实现复杂性之间存在权衡。FIFO算法简单易实现，但性能较差；LRU算法相对较好，能够获得较好的缓存命中率，但实现较复杂；OPT算法是理想的算法，但无法实现。在实际应用中，常用的页面置换算法是LRU算法，它在综合性能和实现难度之间取得了一定的平衡。  综合考虑和实践：在选择和使用这些算法时，需要综合考虑多个因素，包括应用需求、内存资源、性能要求和实现复杂性等。理论分析只是一方面，实际应用中的测试和评估是非常重要的。通过实践和实验，可以更好地了解算法在特定场景下的表现，并根据实际情况进行调整和优化。  总体而言，了解和理解不同的分配算法和置换算法对于有效管理和利用内存资源至关重要。在实际应用中，需要根据具体情况进行选择和优化，以达到最佳的内存管理和性能表现。 | | | |
|  | | | |
|  | | | |