# 实验三：分页存储管理的模拟实现

|  |  |
| --- | --- |
| 实验名称 | 实验三：分页存储管理的模拟实现 |
| 姓名学号 | 高丽 2017302207 |
| 1. **实验目的** 2. 用C++模拟实现分页存储管理，以更好掌握页式存储管理的实现原理。 3. 进一步深入了解内存管理机制、位示图、页表等知识以及内存分配回收操作。 4. 同时提高使用编程解决实际问题的能力。 | |
| 1. **实验内容** 2. 使用语言：C++ 3. 数据结构：   我的设计主要包含三个结构体：进程结构体、页表链表、内存结构体（包含位示图），以及还包括一个全局的进程的容器，该容器容纳所有正在处理或未处理的进程。定义了两个全局变量：内存memory和进程容器processes。   1. 页表链表   页表用链表的方式表示，根据进程需要的内存块数，动态分配页表项（页节点），页节点的页号和对应内存块号均从0起始。页表项结构体如下：  struct Page {  int page\_no; //页号  int block\_no; //对应的内存块号  struct Page\* next;//指向下一页  };   1. 进程结构体（Process）   typedef struct Process {  int no; //进程号  int arrive\_time; //到达时间  int exec\_time; //需要的执行时间  int N; //作业需要的内存块数  //根据输入的需要的内存/内存块大小，计算结果向上取整，从而计算进程需要的内存块数量。  int isalloc; //标志位，代表该进程是否已经被分配内存：  0表示未分配内存；1表示已分配内存  Page\* page\_table; //页表指针，页表带一个不存储任何信息的头节点  }PCB;     1. 内存结构体（Memory）   typedef struct Memory {  int block\_num; //内存块数  int block\_size; //内存块大小，即页面大小  int freeblock\_num; //空闲块数  char bitmap[bitmap\_maxsize];//位示图  }Mem;   1. 位示图（bitmap[bitmap\_maxsize]）  * 用一个char数组表示：char bitmap[bitmap\_maxsize]，每一个char元素为一个字节（即8位），因为位示图中每一比特位表示一块内存的状态，故每一个char元素可表示8块内存的状态。 * 定义宏bitmap\_maxsize 500表示内存块最多为500\*8 = 4000个块；定义宏memfree为1表示该块空闲，memalloc为0表示该块已被分配。 * char[i]由低到高位的数值表示块号从小到大的状态，例如char[0]的第0位表示第0块，第7位表示第7块。 * 表示第j块内存的数组下标为 j / 8；所在的位数为i % 8 。  1. 全局的进程的容器（processes）  * 使用vector容器：vector <Process>processes，方便操作。 * 起初全部进程装入容器，当某进程完成并释放内存后从该容器中删去。  1. 流程： 2. 总体思路：（流程图如下）      1. 具体实现：  * 由以下几个函数构成：   void pagestr();//外层循环函数  void setbit\_blocki(int i, int v);//第i块状态设置为v i在[0,block\_num-1] 范围中  int getbit\_blocki(int i);//获得第i块状态  void init\_bitmap();//初始化位示图，设置接近1/3的块被占用  void print\_bitmap();//打印位示图  void init\_Mem();//初始化内存  bool creat\_PCB(Process &p);//输入信息，创建进程  bool comp(const Process& a, const Process& b);//用于进程序列排序  void print\_pagetable(Process p)//打印页表，当页表过长时会询问是否继续打印  bool alloc\_mem(Process& p);//给p分配内存块  bool free\_mem(Process& p);//释放进程p占用的内存   * 两个位示图操作函数说明：   setbit\_blocki()：  设置第i块状态，计算chari = i/8表示i块所在char在数组中的下标，从而获取字符c = memory.bitmap[chari]，计算int biti = i%8表示第i块所在的比特位。如果设置为memfree即1，则用位运算c |= (1 << biti)，否则c &= ~(1 << biti)，最后把c赋给数组中的char变量。  getbit\_blocki()：  获得第i块状态，同样的方式计算chari和c、biti，返回结果(c >> biti) & 1代表第i块的状态。   * 详细说明： * 第一步：调用init\_Mem()函数初始化内存   (i)输入系统内存大小以及内存块大小以（KB为单位），初始化内存结构体计算得到内存块数（结果向下取整，多余碎片内存舍弃）。  (ii)调用init\_bitmap()函数初始化位示图：先将所有内存块全部设置memfree，再根据系统时间srand()生成随机种子（每次相同条件下位示图每次初始化不同），rand()函数生成随机数即内存块号，调用setbit\_blocki()设置该内存块为memalloc，设置分配的块占总块数的1/3左右。  (iii)考虑到随机块号会有重叠，故遍历位示图，计算空闲块个数并赋值给memory.freeblock\_num，  (iv)调用print\_bitmap()函数：询问是否打印位示图，若打印，调用getbit\_blocki()函数，获得每块状态，按照3位/行打印。   * 第二步： 创建进程   (i)输入进程个数，调用creat\_PCB()函数：分别输入进程号、需要内存空间、到达时间、执行时间，计算需要内存块数N（向上取整），判断N是否大于初始空闲块数，若是，则创建失败，返回false，否则，动态分配页表头节点，并将下一个指针设为NULL，isalloc设置为0，返回true。  (ii)如果创建成功，加入进程容器。   * 第三步：调用pagestr()进行操作，其for循环为：   (i) 初始time= 0，调用sort()和comp()函数按照到达时间对进程排序。  (ii) 判断容器是否为空，空则结束。  (iii) 遍历容器，判断到达时间加执行时间是否等于当前time，如果是，则已完成，调用free\_mem()释放内存：循环删除页表节点，并调用setbit\_blocki()更改位示图。将该进程从容器中删去。  (iv) 再遍历容器，到达时间等于time且isalloc==1的进程分配内存，调用alloc\_mem()函数：首先判断N是否大于空闲块数，是则失败，到达时间++，否则，根据位示图查找空闲块，生成N个页节点，依次插入在页表链表尾部，同时调用setbit\_blocki()更改位示图，isalloc设为1。调用print\_pagetable()询问是否打印页表。 | |
| 1. **实验结果**（红色箭头为输入）   系统内存大小和内存块大小可自由设定，只要不超过4000块即可，这里为了方便显示页表，设定系统有内存64KB，每块1KB。  ① 输入内存大小为64KB，内存块大小为1KB，输出得到块数。  ② 输入y查看位示图，输出得到初始化后位示图和空闲块数。    ③ 输入6个进程信息。    ④开始模拟，当页表长度大于10时询问是否打印页表，这里全部打印。如下图。                  ⑤其他测试：  当同样的条件下，输入的进程内存大小远超初始空闲块数，（该次系统初始化内存空闲块数为45个，输入进程需内存块50个）结果如下图： | |
| 1. **结果分析**   每次输出都会得到第i秒的系统情况。  初始化时，输入设定系统有内存64KB，每块1KB，随机初始化位示图，并打印，剩余空闲块数45块。  共有6个进程，信息如下：  进程[0]，需要33KB，0s到达，执行时间为3s；  进程[1]，需要5KB，2s到达，执行时间为5s；  进程[4]，需要21KB，1s到达，执行时间为4s；  进程[2]，需要10KB，4s到达，执行时间为1s；  进程[5]，需要30KB，5s到达，执行时间为2s；  进程[3]，需要32KB，8s到达，执行时间为2s。  模拟过程：  第0秒。进程[0]到达，分配33块，并打印页表信息，还剩12块。  第1秒。进程[0]占用内存，进程[4]到达，需要21块，等待，还剩12块。  第2秒。进程[0]完成，释放33块，进程[4]分配21块，打印页表信息，进程[1]分配5块，还剩19块。  第3秒。进程[1][4]占用内存，还剩19块。  第4秒。进程[1][4]仍在占用内存，进程[2]到达，分配10块。还剩9块。  第5秒。进程[4]完成，释放21块，进程[2]完成，释放10块，进程[1]占用内存。进程[5]到达，分配30块。剩余10块。  第6秒。进程[1]完成，释放内存块5个，进程[5]占用内存中，进程[3]到达，需要32块，不够，等待，剩余15块。  第7秒。进程[5]，完成，释放30块，进程[3]到达，分配32块，剩余13块。  第8秒。进程[3]占用内存中，仍然剩余13块。  第9秒。进程[3]完成，释放32块。剩余45块。  第10秒。全部完成，结束。  以上为对实验结果全部流程的分析。  同时，由于模拟的是中间过程，0时刻就被占用了一部分内存（考虑到可能中间过程之前有进程占用且未释放、或者是系统本身占用一定内存），如果输入的进程需要内存的大小超过系统初始化时空闲块的总和，则进程无法加入容器，显示创建失败。除非考虑后续系统还会某一时刻释放初始化前的内存，否则这类进程不参与模拟。  综上，本次模拟过程完全符合算法要求。 | |