|  |
| --- |
| 北 京 邮 电 大 学  实 验 报 告  课程名称： 操作系统原理  院系：计算机学院（国家示范性软件工程学院）  班级： 2021211318  姓名： 万志恒  学号： 2021212413  教师： 赵方  成绩：  2023年 春季学期 |
| 实验目的 1. 了解虚拟存储技术的特点；  2. 掌握请求页式管理的页面置换算法。 实验环境 硬件：PC微型计算机、8核、16.0G内存、467G硬盘  软件：Windows操作系统、Clion、MinGw   1. **实验任务及内容**   项目内容：  1. 通过随机数产生一个指令序列，共 320 条指令。其地址按下述原则生成：  （1）50%的指令是顺序执行的；  （2）50%的指令是均匀分布在前地址部分；  （3）50%的指令是均匀分布在后地址部分；  具体的实施方法是：  A. 在[0，319]的指令地址之间随机选取一起点 M；  B. 顺序执行一条指令，即执行地址为 M+1 的指令；  C. 在前地址[0，M+1]中随机选取一条指令并执行，该指令的地址为 M’；  D. 顺序执行一条指令，其地址为 M’+1；  E. 在后地址[M’+2，319]中随机选取一条指令并执行；  F. 重复 A—E，直到执行 320 次指令。  2. 指令序列变换成页地址流  设：  （1）页面大小为 1K；  （2）用户内存容量为 4 页到 32 页；  （3）用户虚存容量为 32K。  在用户虚存中，按每 1K 存放 10 条指令排列虚存地址，即 320 条指令在虚存中的存放方式为：  第 0 条—第 9 条指令为第 0 页（对应虚存地址为[0，9]）；  第 10 条—第 19 条指令为第 1 页（对应虚存地址为[10，19]）；  ……………………  第 310 条—第 319 条指令为第 31 页（对应虚存地址为[310，319]）；  按以上方式，用户指令可组成 32 页。  3. 计算并输出下述各种算法在不同内存容量下的命中率。  A. 先进先出（FIFO）页面置换算法  B. 最近最久未使用（LRU）页面置换算法--最近最少使用算法  C. 最少使用（LFR）页面置换算法  D. 最佳（Optimal）页面置换算法  源代码：  #include <iostream> #include <cstdlib> #include <ctime> #include <queue> #define ins\_num 320 #define mem\_size 32 //内存大小（4~32页）  using namespace std;  class ins{ public:  ins() : num(-1), pg\_num(-1) {};  int num;  int pg\_num; }ins\_list[ins\_num];  class page{ public:  page() : num(-1), wt(0) {};  explicit page(int p\_num, int p\_wt) : num(p\_num), wt(p\_wt){};  int num;  int wt;//权值 };  //class ptr\_epg{ //public: // ext\_page \*p; // // bool operator<(const ptr\_epg &b) const{ // return this->p->wt > b.p->wt; // return this->p->wt == b.p->wt ? this->p->wt > b.p->wt : this->p->wt < b.p->wt; // }//用于LFU算法中对调用次数的比较 //}epg\_ptr[32];  //int ins\_list[ins\_num];//指令序列 //int in\_mem[ins\_num]={0}; //标识指令在内存中  int LFR() {  int pg\_ft=0;  queue <page> pg\_tb;  for(int i=0; i<ins\_num; i++){  bool pg\_in\_mem = false;  int j; // printf("页%3d进入时内存中页号：", ins\_list[i].pg\_num);  for(j = (int)pg\_tb.size(); j>0; j--){ // printf("%2d:%d ",pg\_tb.front().num, pg\_tb.front().wt);  if (pg\_tb.front().num == ins\_list[i].pg\_num) {  pg\_in\_mem = true;  pg\_tb.front().wt++;  }  pg\_tb.push(pg\_tb.front());  pg\_tb.pop();  } // cout<<endl;  if(!pg\_in\_mem){//插入新页  pg\_ft++;  if(pg\_tb.size()==mem\_size){ // 找到最小权值的页（最少访问）  int j;  page lfr\_pg = pg\_tb.front();  for(j=(int)pg\_tb.size(); j>0; j--){  if(lfr\_pg.wt>pg\_tb.front().wt){  lfr\_pg=pg\_tb.front();  }  pg\_tb.push(pg\_tb.front());  pg\_tb.pop();  } // pop lfr\_page  for(j=(int)pg\_tb.size(); j>0; j--){  if(lfr\_pg.num==pg\_tb.front().num){  pg\_tb.pop();  }else{  pg\_tb.push(pg\_tb.front());  pg\_tb.pop();  }  }   } // 新载入页的初始化  page cur\_pg(ins\_list[i].pg\_num, 1);  pg\_tb.push(cur\_pg);  }  }  return pg\_ft;  { // priority\_queue不能实时更新，并不好用，不如直接遍历找最小权值页 // for(int i=0; i<32; i++){ // e\_pg[i].num=i; // epg\_ptr[i].p=&e\_pg[i]; // }  // priority\_queue<ptr\_epg > pg\_tb;//存储当前位于内存的页号 // priority\_queue<ptr\_epg > cpy\_tb;//副本用于遍历 // for(int i=0; i<ins\_num; i++){ // bool pg\_in\_mem=false;//标识位是否在内存中 // printf("指令%3d进入时内存中页号及其权值：", ins\_list[i].num); // cpy\_tb=pg\_tb; // while(cpy\_tb.size()>0){ // printf("%2d:%d ", cpy\_tb.top().p->num, cpy\_tb.top().p->wt); // if(cpy\_tb.top().p->num==ins\_list[i].pg\_num){ // pg\_in\_mem= true; // } // cpy\_tb.pop(); // } // if(pg\_in\_mem== false){ // if(pg\_tb.size()==mem\_size){ // pg\_tb.top().p->wt=0;//移出内存的页重置wt权值 // pg\_tb.pop(); // } // pg\_tb.push(epg\_ptr[ins\_list[i].pg\_num]); // pg\_ft++; // } // e\_pg[ins\_list[i].pg\_num].wt++; //Priority\_Queue不能实时更新！ // cpy\_tb=pg\_tb; // pg\_tb=cpy\_tb; // cout<<endl; // }  } }  int OPT() {  int pg\_ft=0;  queue <page> pg\_tb;  for(int i=0; i<ins\_num; i++){  bool pg\_in\_mem = false;  int j;  // 调试  { // j = (int)pg\_tb.size(); // //计算权值别忘记考虑对以后再也不来的页的wt更新！！！ // //计算各页权值：wt=ins\_num-（future-current） // j=(int)pg\_tb.size(); // for(j; j>0; j--){ // bool will\_come= false; // for(int k = i+1; k<ins\_num; k++){ // if(pg\_tb.front().num==ins\_list[k].pg\_num){ // will\_come= true; // pg\_tb.front().wt=ins\_num-(k-i); // break; // } // if(will\_come== false) pg\_tb.front().wt=0; // } // pg\_tb.push(pg\_tb.front()); // pg\_tb.pop(); // }  } // printf("页%3d进入时内存中页号：", ins\_list[i].pg\_num);   for(j=(int)pg\_tb.size(); j>0; j--){ // printf("%2d:%d ",pg\_tb.front().num, pg\_tb.front().wt);  if (pg\_tb.front().num == ins\_list[i].pg\_num) {  pg\_in\_mem = true;  }  pg\_tb.push(pg\_tb.front());  pg\_tb.pop();  } // cout<<endl; // 更新权值： // 计算各页权值：wt=ins\_num-（future-current）,对于再也不来的页，权值为0 // 计算权值别忘记考虑对以后再也不来的页的wt更新为0！！！ // 权值计算的优化：对于0<wt<319的页，可以对其wt直接++，而不用再遍历一次, // 但这样的话就得每次更新而非缺页时更新，但这种不用遍历，还是快一些，所以还是每次更新叭  for(j=(int)pg\_tb.size(); j>0; j--){  if(pg\_tb.front().wt>0&&pg\_tb.front().wt<319){  pg\_tb.front().wt++;  }else if(pg\_tb.front().wt==319)  {  bool will\_come= false;  for(int k = i+1; k<ins\_num; k++){  if(pg\_tb.front().num==ins\_list[k].pg\_num){  will\_come= true;  pg\_tb.front().wt=ins\_num-(k-i);  break;  }  }  if(!will\_come) pg\_tb.front().wt=0;  }  pg\_tb.push(pg\_tb.front());  pg\_tb.pop();  }  if(!pg\_in\_mem){//插入新页  pg\_ft++;  if(pg\_tb.size()==mem\_size){ // 找到将来最久不被访问的页： // 找到最小权值的页（将来最晚访问）  page lfr\_pg = pg\_tb.front();  for(j=(int)pg\_tb.size(); j>0; j--){  if(lfr\_pg.wt>pg\_tb.front().wt){  lfr\_pg=pg\_tb.front();  }  pg\_tb.push(pg\_tb.front());  pg\_tb.pop();  }  // pop future\_lru\_page  for(j=(int)pg\_tb.size(); j>0; j--){  if(lfr\_pg.num==pg\_tb.front().num){  pg\_tb.pop();  }else{  pg\_tb.push(pg\_tb.front());  pg\_tb.pop();  }  }  } // 新载入页的初始化  page cur\_pg(ins\_list[i].pg\_num, 0);  for(int k = i+1; k<ins\_num; k++){  if(cur\_pg.num==ins\_list[k].pg\_num){  cur\_pg.wt=ins\_num-(k-i);  break;  }  }  pg\_tb.push(cur\_pg);  }  { // //计算各页权值：wt=ins\_num-（future-current） // j=(int)pg\_tb.size(); // for(j; j>0; j--){ // bool will\_come= false; // for(int k = i+1; k<ins\_num; k++){ // if(pg\_tb.front().num==ins\_list[k].pg\_num){ // will\_come= true; // pg\_tb.front().wt=ins\_num-(k-i); // break; // } // if(will\_come== false) pg\_tb.front().wt=0; // } // pg\_tb.push(pg\_tb.front()); // pg\_tb.pop(); // }   // j=pg\_tb.size(); // printf("页%3d进入后内存中页号：", ins\_list[i].pg\_num); // for(j; j>0; j--){ // printf("%2d:%d ",pg\_tb.front().num, pg\_tb.front().wt); // pg\_tb.push(pg\_tb.front()); // pg\_tb.pop(); // } // cout<<endl;  }  }  return pg\_ft; }  结果：     |  |  |  |  |  |  |  |  |  |  |  | | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | | 平均缺页次数 | 4 | | 8 | | 16 | | 24 | | 32 | | | FIFO PF | 147.8 | | 129.4 | | 92.6 | | 53.2 | | 31.6 | | | LRU PF | 147 | | 131 | | 93.4 | | 52.2 | | 31.6 | | | LFR PF | 150.4 | | 131.2 | | 93 | | 50 | | 31.6 | | | OPT PF | 114.8 | | 84.2 | | 54.4 | | 34.8 | | 31.6 | | | 平均命中率 | 4 | 8 | | 16 | | 24 | | 32 | | | FIFO HR | 0.538 | 0.596 | | 0.711 | | 0.834 | | 0.901 | | | LRU HR | 0.541 | 0.591 | | 0.708 | | 0.837 | | 0.901 | | | LFR HR | 0.530 | 0.590 | | 0.709 | | 0.844 | | 0.901 | | | OPT HR | 0.641 | 0.737 | | 0.830 | | 0.891 | | 0.901 | |     实验结论：根据上述结果可以看出，在实验模拟的实际内存与虚拟内存大小下，FIFO、LRU、LFR算法的效率差别很小。不同算法的缺页次数随内存的增大而减小，且增大内存对减少缺页的效果在虚拟内存大小的1/8~3/4段最为显著，当内存大于虚拟内存的3/4时增大内存的效果减弱。在确定的虚拟内存大小下，以OPT算法为参照，FIFO、LRU、LFR算法的效率随着实际内存的增大先降低，实际内存大小超过虚拟内存的1/2后又开始逐渐升高。 实验心得及体会 本次实验我通过C++模拟了两种较为复杂的页面调度算法，这大大锻炼了我的编程能力，同时让我对这两种算法的特点与实现有了更深的印象。通过对实验结果进行分析，我对不同页面调度算法的特点有了更清晰的印象。 |