四川大学计算机学院、软件学院

实验报告

学号：2022141460155 姓名：林诺晗 专业：计算金融 班级：计金班 第13周

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 课程名称 | 操作系统实验 | | 实验课时 | 2 |
| 实验项目 | 页面置换算法的模拟 | | 实验时间 | 2024.05.23 |
| 实验目的 | （1）理解虚拟内存管理的原理和技术  （2）掌握请求分页存储管理的常用理论——页面置换算法  （3）理解请求分页中的按需调页机制 | | | |
| 实验环境 | VMware Workstation 17 Pro  Ubuntu 22.04.3 LTS | | | |
| 实验内容（算法、程序、步骤和方法） | 设计一个虚拟存储区和内存工作区，并使用下述常用页面置换算法计算访问命中率。  （1）先进先出算法（FIFO）  （2）最近最少使用算法（LRU）  **实验要求**  （1）通过随机数产生一个指令序列，共320条指令。  （2）设计页面类型、页面控制结构等数据结构。  （3）将指令序列转换成页面序列。  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。  ④在用户虚存中，按每页存放10条指令排列虚存地址，即320条指令存在32个页面中。  （4）计算并输出不同置换算法在不同内存容量下的命中率。命中率计算公式为：  命中率 = 1 – 页面失效次数 / 页面总数  **算法一：**先进先出算法（FIFO）  **源代码：**  #include <stdio.h>  #include <stdlib.h>  #include <time.h>  #define MAX\_PAGES 32  #define INSTRUCTIONS 320  #define PAGE\_SIZE 10  #define MAX\_FRAMES 32  // 定义一个页面结构  typedef struct {  int number;  int inMemory; // 使用int来表示是否在内存中，1表示true，0表示false  } Page;  // 定义页面置换算法的结果结构  typedef struct {  int pageFaults;  double hitRate;  } Result;  // FIFO页面置换算法实现  Result fifoAlgorithm(int\* instructionSequence, int memorySize) {  Page frames[MAX\_FRAMES]; // 内存帧  int pageFaults = 0;  int totalPages = 0;    // 初始化内存帧  for (int i = 0; i < memorySize; i++) {  frames[i].number = -1;  frames[i].inMemory = 0;  }    // 模拟指令流  for (int i = 0; i < INSTRUCTIONS; i++) {  int page = instructionSequence[i];  int found = 0;    // 检查页面是否已加载  for (int j = 0; j < memorySize; j++) {  if (frames[j].number == page) {  found = 1;  break;  }  }    // 如果页面未加载，则发生页面失效  if (!found) {  pageFaults++;  // 替换最老的页面  for (int j = 0; j < memorySize - 1; j++) {  frames[j] = frames[j + 1];  }  frames[memorySize - 1].number = page;  frames[memorySize - 1].inMemory = 1;  }    totalPages++;  }    // 计算命中率  double hitRate = (totalPages > 0) ? (1.0 - (double)pageFaults / totalPages) : 0;    return (Result){pageFaults, hitRate};  }  int main() {  srand(time(NULL)); // 初始化随机数生成器  int instructionSequence[INSTRUCTIONS];  // 随机生成指令序列  for (int i = 0; i < INSTRUCTIONS; i++) {  instructionSequence[i] = rand() % MAX\_PAGES; // 指令指向页面号  }  // 测试不同的内存容量  for (int memorySize = 4; memorySize <= MAX\_FRAMES; memorySize++) {  Result result = fifoAlgorithm(instructionSequence, memorySize);  printf("%2d page frames FIFO: %.4f\n", memorySize, result.hitRate);  }  return 0;  }  IMG_256  IMG_256  ****编译程序****：使用编译器（如gcc）将.c文件编译成可执行文件。  ****运行程序****：编译成功后，会生成一个可执行文件，程序运行如下：  IMG_256  结果显示：当内存页面比较少的时候，访问命中率不高，随着内存页面的增多，访问命中率开始提高。  **算法二**：最近最少使用算法（LRU）  **源代码：**  #include <stdio.h>  #include <stdlib.h>  #include <time.h>  #define INSTRUCTIONS 320  #define PAGE\_SIZE 10  #define MAX\_PAGES 32  #define MIN\_PAGES 4  typedef struct {  int number;  int is\_loaded;  } Page;  typedef struct {  Page \*frames[MAX\_PAGES]; // 内存帧数组  int size; // 内存大小  int page\_faults; // 页面失效次数  } Memory;  typedef struct node {  int page\_number;  struct node \*prev;  struct node \*next;  } Node;  typedef struct {  Node \*head;  Node \*tail;  } LRUQueue;  // 初始化内存  void init\_memory(Memory \*mem, int size);  // 检查页面是否在内存中  int is\_page\_in\_memory(Memory \*mem, int page\_number);  // 初始化LRU队列  void init\_lru(LRUQueue \*lru);  // 将页面添加到LRU队列的末尾  void lru\_add(LRUQueue \*lru, int page\_number);  // 从LRU队列中移除最老的页面  void lru\_remove\_oldest(LRUQueue \*lru, Memory \*mem);  // 加载页面到内存，使用LRU替换策略  void lru\_load\_page(Memory \*mem, LRUQueue \*lru, int page\_number);  // 计算命中率  double calculate\_hit\_rate(Memory \*mem, LRUQueue \*lru, int \*instructions, int instructions\_count);  // 清理内存  void free\_memory(Memory \*mem);  // 清理LRU队列  void free\_lru(LRUQueue \*lru);  int main() {  srand((unsigned int)time(NULL)); // 设置随机数种子  int instructions[INSTRUCTIONS]; // 指令序列  for (int i = 0; i < INSTRUCTIONS; i++) {  instructions[i] = i / PAGE\_SIZE; // 指令到页面的映射  }  LRUQueue lru;  Memory mem;  for (int memory\_size = MIN\_PAGES; memory\_size <= MAX\_PAGES; memory\_size++) {  init\_memory(&mem, memory\_size);  init\_lru(&lru);  double hit\_rate = calculate\_hit\_rate(&mem, &lru, instructions, INSTRUCTIONS);  printf("Memory size: %2d pages, LRU Hit Rate: %.4f\n", memory\_size, hit\_rate );  free\_memory(&mem);  free\_lru(&lru);  }  return 0;  }  void init\_memory(Memory \*mem, int size) {  mem->size = size;  mem->page\_faults = 0;  for (int i = 0; i < size; i++) {  mem->frames[i] = NULL;  }  }  int is\_page\_in\_memory(Memory \*mem, int page\_number) {  for (int i = 0; i < mem->size; i++) {  if (mem->frames[i] != NULL && mem->frames[i]->number == page\_number) {  return 1;  }  }  return 0;  }  void init\_lru(LRUQueue \*lru) {  lru->head = NULL;  lru->tail = NULL;  }  void lru\_add(LRUQueue \*lru, int page\_number) {  Node \*new\_node = (Node \*)malloc(sizeof(Node));  new\_node->page\_number = page\_number;  new\_node->prev = NULL;  new\_node->next = NULL;  if (lru->tail != NULL) {  lru->tail->next = new\_node;  new\_node->prev = lru->tail;  } else {  lru->head = new\_node;  }  lru->tail = new\_node;  }  void lru\_remove\_oldest(LRUQueue \*lru, Memory \*mem) {  if (lru->head == NULL) return;  Node \*oldest = lru->head;  if (lru->head->next != NULL) {  lru->head = oldest->next;  lru->head->prev = NULL;  } else {  lru->head = NULL;  lru->tail = NULL;  }  free(mem->frames[oldest->page\_number]);  free(oldest);  }  void lru\_load\_page(Memory \*mem, LRUQueue \*lru, int page\_number) {  if (is\_page\_in\_memory(mem, page\_number)) {  return; // 页面已加载，无需替换  }  // 检查是否有空闲帧  int empty\_frame\_index = -1;  for (int i = 0; i < mem->size; i++) {  if (mem->frames[i] == NULL) {  empty\_frame\_index = i;  break;  }  }  // 替换最老的页面  if (empty\_frame\_index == -1) {  lru\_remove\_oldest(lru, mem);  }  // 加载新页面  Page \*new\_page = (Page \*)malloc(sizeof(Page));  new\_page->number = page\_number;  new\_page->is\_loaded = 1;  mem->frames[empty\_frame\_index] = new\_page;  // 添加新页面到LRU队列  lru\_add(lru, page\_number);  mem->page\_faults++;  }  double calculate\_hit\_rate(Memory \*mem, LRUQueue \*lru, int \*instructions, int instructions\_count) {  int hits = 0;  for (int i = 0; i < instructions\_count; i++) {  if (!is\_page\_in\_memory(mem, instructions[i])) {  lru\_load\_page(mem, lru, instructions[i]);  } else {  hits++;  }  }  double hit\_rate = (double)hits / instructions\_count;  return hit\_rate;  }  void free\_memory(Memory \*mem) {  for (int i = 0; i < mem->size; i++) {  if (mem->frames[i] != NULL) {  free(mem->frames[i]);  mem->frames[i] = NULL;  }  }  }  void free\_lru(LRUQueue \*lru) {  Node \*current = lru->head;  while (current != NULL) {  Node \*next = current->next;  free(current);  current = next;  }  lru->head = lru->tail = NULL;  }  IMG_256  IMG_256  IMG_256  IMG_256  ****编译程序****：使用编译器（如gcc）将.c文件编译成可执行文件。  ****运行程序****：编译成功后，会生成一个可执行文件，程序运行如下：  IMG_256  结果显示：当内存页面比较少的时候，访问命中率不高，随着内存页面的增多，访问命中率开始提高。 | | | |
| 结论  （结果） | 实验成功，按照相关要求完成程序编写及实现，并熟练掌握了相关命令的使用。  设计了一个虚拟存储区和内存工作区，并使用常用页面置换算法FIFO和LRU计算访问命中率。 | | | |
| 小结 | （1）理解了虚拟内存管理的原理和技术  （2）掌握了请求分页存储管理的常用理论——页面置换算法  （3）理解了请求分页中的按需调页机制 | | | |
| 指导老师  评议 |  | | | |
| 成绩评定： | 指导教师签名： | | |