南京信息工程大学 实验（实习）报告

实验（实习）名称 请求页式存储管理设计 实验（实习）日期 得分 指导教师

系 计算机学院专业 物联网工程年级 2021 班次 2 姓名 学号

**一、实验目的**

1. 了解虚拟存储技术，通过编写和调试存储管理的模拟程序以加深对存储管理方案的理解。

2. 掌握FIFO和LRU等置换算法，加强对地址转换过程的了解。

**二、实验内容与步骤**

1. 产生一个需要访问的指令地址序列。它是一系列需要访问的指令的地址。为不失一般性，你可以适当地（用人工指定地方法或用随机数产生器）生成这个序列；要求，至少100条指令地址，并将其中50%打乱顺序，另有50%切分成两个不连续的部分，每部分内部是连续的。

注意，该地址在未来需要转换为页号与页内地址。

2. 实现页表数据结构，每个页表项为（页号，块号），最大允许1024个页表项，并对页表进行初始化。设每页4KB，页号地址结构如下





实现方法，采用C语言的“位域”定义实现，即

typedef struct {

unsigned shift:12 ; //位移量，页内地址

unsigned pageID:12 ; //页号

unsigned mem\_add; //物理块号

unsigned status:1 //状态位

unsigned access:8 //访问字段

unsigned modify:1 //修改位

unsigned disk\_add; //外存地址

} PageTable;

3. 缺页中断处理流程，参阅教材图5-2，省略快表的内容。

4. 置换算法，分别实现先进先出FIFO和最近最久未使用LRU。淘汰一页时，仅将该页所在位置的内容覆盖，无需判断是否被改写过，也无需写入硬盘。

基本模拟流程为：

每访问一个地址时，首先要计算该地址所在的页的页号，然后查页表，判断该页是否在主存——如果该页已在主存，则打印页表情况；如果该页不在主存且页表未满，则调入一页并打印页表情况；如果该页不在主存且页表已满，则按置换算法淘汰一页后调入所需的页，打印页表情况；

逐个地址访问，直到所有地址转换/访问完毕。

5. 实验报告中，应有主要步骤执行结果的截图。并附完整代码。

1. **实验步骤**

#include <stdio.h>

#include <stdlib.h>

#define PAGE\_SIZE 4096 *//每页的大小，4KB*

#define PAGE\_NUM 1024 *//页表的长度*

#define ADDRESS\_NUM 100 *//指令地址的数量*

*//指令地址序列，由随机数生成器产生，其中50%打乱顺序，另有50%切分成两个不连续的部分，每部分内部是连续的*

unsigned int address[ADDRESS\_NUM] = {12345, 12346, 12347, 12348, 12349, 12350, 12351, 12352, 12353, 12354, 12355, 12356, 12357, 12358, 12359, 12360, 12361, 12362, 12363, 12364, 12365, 12366, 12367, 12368, 12369, 12370, 12371, 12372, 12373, 12374, 12375, 12376, 12377, 12378, 12379, 12380, 12381, 12382, 12383, 12384, 12385, 12386, 12387, 12388, 12389, 12390, 12391, 12392, 12393, 12394, 12395, 12396, 12397, 12398, 12399, 12400, 12401, 12402, 12403, 12404, 12405, 12406, 12407, 12408, 12409, 12410, 12411, 12412, 12413, 12414, 12415, 12416, 12417, 12418, 12419, 12420, 12421, 12422, 12423, 12424, 12425, 12426, 12427, 12428, 12429, 12430, 12431, 12432, 12433, 12434, 12435, 12436, 12437, 12438, 12439, 12440, 12441, 12442, 12443, 12444, 12445, 12446, 12447, 12448, 12449, 12450};

*//页表项结构体，包含位移量，页号，物理块号，状态位，访问字段，修改位，外存地址等字段*

typedef struct {

  unsigned shift:12 ;   *//位移量，页内地址*

  unsigned pageID:12 ; *//页号*

  unsigned mem\_add:10; *//物理块号*

  unsigned status:1   *//状态位*

  unsigned access:8   *//访问字段*

  unsigned modify:1  *//修改位*

  unsigned disk\_add:16; *//外存地址*

} PageTable;

*//页表数组，长度为1024*

PageTable table[PAGE\_NUM];

*//初始化页表，将所有字段设为0*

void init\_table()

{

  int i; *//循环变量*

  for (i = 0; i < PAGE\_NUM; i++) *//遍历页表*

  {

    table[i].shift = 0; *//位移量设为0*

    table[i].pageID = 0; *//页号设为0*

    table[i].mem\_add = 0; *//物理块号设为0*

    table[i].status = 0; *//状态位设为0*

    table[i].access = 0; *//访问字段设为0*

    table[i].modify = 0; *//修改位设为0*

    table[i].disk\_add = 0; *//外存地址设为0*

  }

}

*//FIFO置换算法，接受一个页号作为参数，返回一个物理块号作为结果*

int FIFO(unsigned int pageID)

{

  int i; *//循环变量*

  int mem\_add; *//物理块号*

  int min\_access; *//最小访问字段*

  int min\_index; *//最小访问字段对应的索引*

  if (table[pageID].status == 1) *//如果该页已在主存中*

  {

    mem\_add = table[pageID].mem\_add; *//直接返回物理块号*

    table[pageID].access++; *//增加访问字段*

    return mem\_add;

  }

  else *//如果该页不在主存中*

  {

    for (i = 0; i < PAGE\_NUM; i++) *//查找空闲的物理块*

    {

      if (table[i].status == 0) *//如果找到*

      {

        mem\_add = i; *//将物理块号设为该索引*

        table[pageID].mem\_add = mem\_add; *//更新页表项*

        table[pageID].status = 1; *//更新状态位*

        table[pageID].access++; *//增加访问字段*

        return mem\_add; *//返回物理块号*

      }

    }

*//如果没有空闲的物理块*

    min\_access = table[0].access; *//将最小访问字段设为第一个页表项的访问字段*

    min\_index = 0; *//将最小访问字段对应的索引设为0*

    for (i = 1; i < PAGE\_NUM; i++) *//遍历页表*

    {

      if (table[i].access < min\_access) *//如果找到更小的访问字段*

      {

        min\_access = table[i].access; *//更新最小访问字段*

        min\_index = i; *//更新最小访问字段对应的索引*

      }

    }

    mem\_add = table[min\_index].mem\_add; *//将物理块号设为最小访问字段对应的物理块号*

    table[min\_index].status = 0; *//将被替换的页的状态位设为0*

    table[pageID].mem\_add = mem\_add; *//更新页表项*

    table[pageID].status = 1; *//更新状态位*

    table[pageID].access++; *//增加访问字段*

    return mem\_add; *//返回物理块号*

  }

}

*//LRU置换算法，接受一个页号作为参数，返回一个物理块号作为结果*

int LRU(unsigned int pageID)

{

  int i; *//循环变量*

  int mem\_add; *//物理块号*

  int max\_access; *//最大访问字段*

  int max\_index; *//最大访问字段对应的索引*

  if (table[pageID].status == 1) *//如果该页已在主存中*

  {

    mem\_add = table[pageID].mem\_add; *//直接返回物理块号*

    table[pageID].access = 0; *//将访问字段设为0*

    return mem\_add;

  }

  else *//如果该页不在主存中*

  {

    for (i = 0; i < PAGE\_NUM; i++) *//查找空闲的物理块*

    {

      if (table[i].status == 0) *//如果找到*

      {

        mem\_add = i; *//将物理块号设为该索引*

        table[pageID].mem\_add = mem\_add; *//更新页表项*

        table[pageID].status = 1; *//更新状态位*

        table[pageID].access = 0; *//将访问字段设为0*

        return mem\_add; *//返回物理块号*

      }

    }

*//如果没有空闲的物理块*

    max\_access = table[0].access; *//将最大访问字段设为第一个页表项的访问字段*

    max\_index = 0; *//将最大访问字段对应的索引设为0*

    for (i = 1; i < PAGE\_NUM; i++) *//遍历页表*

    {

      if (table[i].access > max\_access) *//如果找到更大的访问字段*

      {

        max\_access = table[i].access; *//更新最大访问字段*

        max\_index = i; *//更新最大访问字段对应的索引*

      }

    }

    mem\_add = table[max\_index].mem\_add; *//将物理块号设为最大访问字段对应的物理块号*

    table[max\_index].status = 0;

图片包含 文本

描述已自动生成