**实验四 存储管理**

# 姓 名 吴梓弘 学 号 21013054 成绩

实验时间 11.23 指导教师(签名)

**（诚信声明：本实验报告内容，均由本人亲自上机完成。 签名：\_\_\_\_\_\_\_\_\_\_\_\_\_）**

一．实验目的

1.通过请求页式存储管理中页面置换算法模拟设计，了解虚拟存储技术的特点

2.掌握请求页式存储管理的页面置换算法。

二．实验工具与设备

装有 Linux 操作系统的计算机。

三．实验内容

计算并输出下面各种算法在不同内存容量下的命中率：

1. FIFO（先进先出算法）
2. LRU （最近最少使用算法）
3. OPT （最优算法）
4. LFU （最少使用页面算法）
5. CLOCK （时钟算法）

代码实现：

#include <stdio.h>

#include <stdlib.h>

#include <stdbool.h>

#define INSTRUCTION\_COUNT 320

#define PAGE\_SIZE 10

#define VIRTUAL\_PAGES 32

bool isPageInMemory(int cache[], int capacity, int page) {

for (int i = 0; i < capacity; ++i) {

if (cache[i] == page) {

return true;

}

}

return false;

}

int findPageIndex(int cache[], int capacity, int page) {

for (int i = 0; i < capacity; ++i) {

if (cache[i] == page) {

return i;

}

}

return -1;

}

void printCache(int cache[], int capacity) {

for (int i = 0; i < capacity; ++i) {

printf("%d ", cache[i]);

}

printf("\n");

}

float fifo(int instructionSequence[], int n, int capacity) {

int cache[capacity];

int page\_faults = 0;

int front = 0, rear = 0;

for (int i = 0; i < n; ++i) {

int pageIndex = instructionSequence[i] / PAGE\_SIZE;

if (!isPageInMemory(cache, capacity, pageIndex)) {

if (rear < capacity) {

cache[rear++] = pageIndex;

} else {

cache[front] = pageIndex;

front = (front + 1) % capacity;

}

page\_faults++;

}

}

// 修改计算命中率的方式

return (float)(n - page\_faults) / n;

}

float lru(int instructionSequence[], int n, int capacity) {

int cache[capacity];

int page\_faults = 0;

for (int i = 0; i < n; ++i) {

int pageIndex = instructionSequence[i] / PAGE\_SIZE;

if (!isPageInMemory(cache, capacity, pageIndex)) {

if (page\_faults < capacity) {

cache[page\_faults++] = pageIndex;

} else {

int index = findPageIndex(cache, capacity, cache[0]);

for (int j = 0; j < index; ++j) {

cache[j] = cache[j + 1];

}

cache[index] = pageIndex;

}

page\_faults++;

}

}

// 修改计算命中率的方式

return (float)(n - page\_faults) / n;

}

float opt(int instructionSequence[], int n, int capacity) {

int cache[capacity];

int page\_faults = 0;

for (int i = 0; i < n; ++i) {

int pageIndex = instructionSequence[i] / PAGE\_SIZE;

if (!isPageInMemory(cache, capacity, pageIndex)) {

if (page\_faults < capacity) {

cache[page\_faults++] = pageIndex;

} else {

int index = -1;

int farthest = 0;

for (int j = 0; j < capacity; ++j) {

int nextIndex = i + 1;

while (nextIndex < n && instructionSequence[nextIndex] / PAGE\_SIZE != cache[j]) {

nextIndex++;

}

if (nextIndex == n) {

index = j;

break;

} else if (nextIndex > farthest) {

farthest = nextIndex;

index = j;

}

}

cache[index] = pageIndex;

}

page\_faults++;

}

}

// 修改计算命中率的方式

return (float)(n - page\_faults) / n;

}

float lfu(int instructionSequence[], int n, int capacity) {

int cache[capacity];

int usage[capacity];

int page\_faults = 0;

for (int i = 0; i < n; ++i) {

int pageIndex = instructionSequence[i] / PAGE\_SIZE;

if (!isPageInMemory(cache, capacity, pageIndex)) {

if (page\_faults < capacity) {

cache[page\_faults] = pageIndex;

usage[page\_faults++] = 1;

} else {

int index = 0;

int minUsage = usage[0];

for (int j = 1; j < capacity; ++j) {

if (usage[j] < minUsage) {

minUsage = usage[j];

index = j;

}

}

cache[index] = pageIndex;

usage[index] = 1;

}

page\_faults++;

} else {

int index = findPageIndex(cache, capacity, pageIndex);

usage[index]++;

}

}

// 修改计算命中率的方式

return (float)(n - page\_faults) / n;

}

float clock(int instructionSequence[], int n, int capacity) {

typedef struct {

int data;

int referenced;

} Page;

Page cache[capacity];

int hand = 0;

int page\_faults = 0;

for (int i = 0; i < n; ++i) {

int pageIndex = instructionSequence[i] / PAGE\_SIZE;

if (!isPageInMemory((int \*) cache, capacity, pageIndex)) {

while (cache[hand].referenced) {

cache[hand].referenced = 0;

hand = (hand + 1) % capacity;

}

cache[hand].data = pageIndex;

cache[hand].referenced = 1;

hand = (hand + 1) % capacity;

page\_faults++;

}

}

// 修改计算命中率的方式

return (float)(n - page\_faults) / n;

}

int main() {

int instructionSequence[INSTRUCTION\_COUNT];

srand(123);

int start = rand() % (INSTRUCTION\_COUNT - 1);

int m = start;

for (int i = 0; i < INSTRUCTION\_COUNT; ++i) {

instructionSequence[i] = m;

m = (m + 1) % INSTRUCTION\_COUNT;

if (i % 4 == 1 || i % 4 == 2) {

m = rand() % (start + 1);

} else if (i % 4 == 3) {

m = (rand() % (INSTRUCTION\_COUNT - 1 - (start + 2))) + (start + 2);

}

}

int pageAddressStream[INSTRUCTION\_COUNT / PAGE\_SIZE];

for (int i = 0; i < INSTRUCTION\_COUNT / PAGE\_SIZE; ++i) {

pageAddressStream[i] = instructionSequence[i \* PAGE\_SIZE] / PAGE\_SIZE;

}

int min\_capacity = 4;

int max\_capacity = 32;

for (int capacity = min\_capacity; capacity <= max\_capacity; capacity ++) {

// 以下四行为修复部分

float fifo\_rate = fifo(instructionSequence, INSTRUCTION\_COUNT, capacity);

float lru\_rate = lru(instructionSequence, INSTRUCTION\_COUNT, capacity);

float opt\_rate = opt(instructionSequence, INSTRUCTION\_COUNT, capacity);

float lfu\_rate = lfu(instructionSequence, INSTRUCTION\_COUNT, capacity);

float clock\_rate = clock(instructionSequence, INSTRUCTION\_COUNT, capacity);

printf("\n%d\n", capacity);

printf("FIFO: %.4f\n", fifo\_rate);

printf("LRU: %.4f\n", lru\_rate);

printf("OPT: %.4f\n", opt\_rate);

printf("LFU: %.4f\n", lfu\_rate);

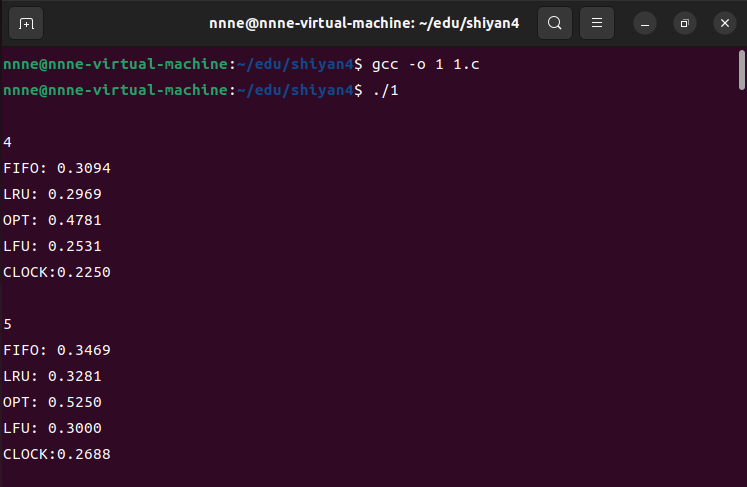
printf("CLOCK:%.4f\n", clock\_rate);

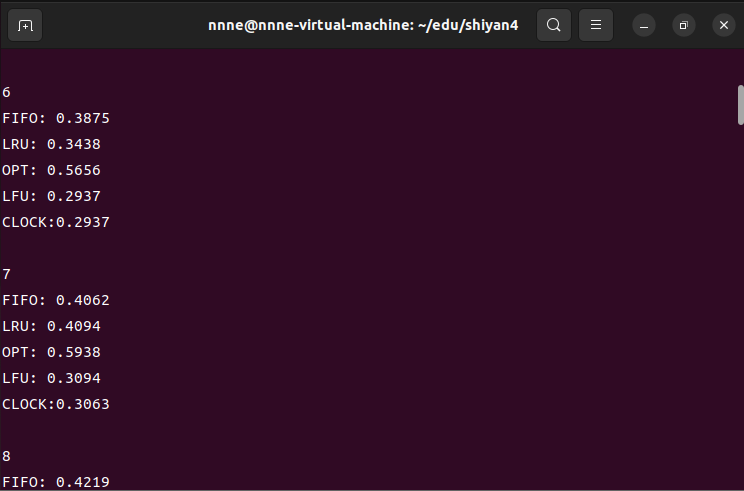
}

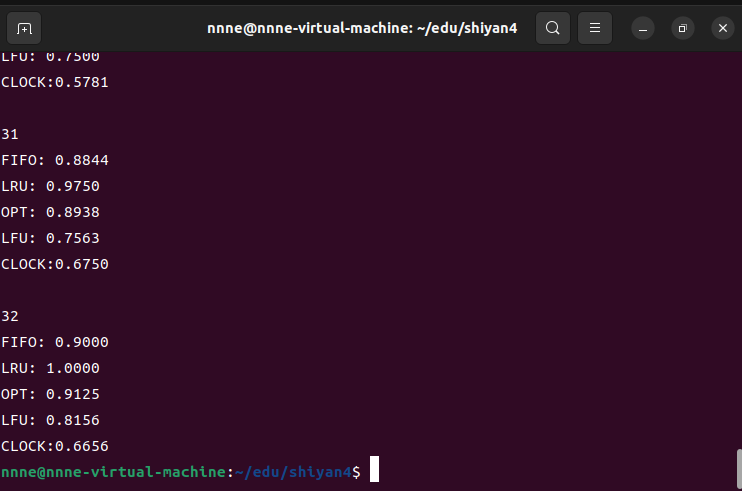
return 0;

}

运行结果：







# 四．思考题

1. 为什么要进行内存管理，虚拟存储器的特点是什么？

内存管理

多任务运行： 当计算机同时执行多个任务时，每个任务都需要内存。内存管理确保这些任务能够共享计算机内存而不发生干扰。

有限内存资源： 计算机内存是有限的，而程序的大小和数量可能超过可用内存总和。因此，内存管理确保有效利用有限内存资源。

动态分配： 程序可能在运行时需要灵活地分配内存。内存管理确保对动态分配的内存进行合理管理和释放。

防止内存泄漏： 内存泄漏是指程序分配了内存但未释放，导致系统内存耗尽。内存管理系统能够追踪分配的内存块，并在不再需要时释放，防止内存泄漏

虚拟存储器的特点

虚拟地址空间： 虚拟存储器为每个程序提供一个虚拟地址空间，使得程序认为它能够独占整个计算机内存。然而，实际上这些程序可能不会同时存在于物理内存中。

分页和分段： 虚拟存储器利用分页和分段技术，将程序分割成固定大小的页或可变大小的段。这样做能更有效地利用内存，并允许将程序的不同部分分配到不同的物理存储位置。

页面置换： 当物理内存不足时，虚拟存储器使用页面置换算法将不再活跃的页面移出内存，为新的页面腾出空间。这使得程序可以继续执行，尽管并非所有页面都在物理内存中。

页面错误处理： 如果程序访问的页面不在物理内存中，将会发生页面错误。虚拟存储器通过将相应的页面调入内存来解决页面错误，以满足程序的访问请求。

1. 几种内存管理算法有何区别与联系？

1、先进先出算法：

区别： 使用队列数据结构，最先进入队列的页面最先被替换，不考虑页面的访问频率和优先级

联系： 与其他算法相比，FIFO简单且易于实现

2、最近最少使用：

区别： 替换最近最长时间未被使用的页面。

联系： 对FIFO的优化，不按照“先进”，而是按照最后一次被调用

3、最少使用页面：

区别： 替换使用频率最低的页面。

联系： 与LRU不同，LFU关注的是页面被访问的次数，而不是时间

4、最优页面置换：

区别： 替换在未来最长时间内不再被访问的页面，理论上的最佳算法，根据结果进行页面置换

联系：用以衡量其他置换算法的效率

5、时钟算法：

区别： 基于时钟的算法，使用类似于物理时钟的数据结构。

联系： 时钟算法是对FIFO的一种改进，它考虑页面是否被访问过，如果被访问则不替换，否则替换。时钟算法的实现相对简单，介于FIFO和LRU之间。