Министерство науки и высшего образования РФ

Севастопольский государственный университет

Кафедра информатики и управления в технических системах

ОТЧЕТ

по лабораторной работе №5

ИССЛЕДОВАНИЕ АЛГОРИТМОВ ЗАМЕЩЕНИЯ СТРАНИЦ

по дисциплине «Операционные системы»

Выполнил:

Студент группы ИВТ/б 22-о

Черняев Н.Г.

Проверил:

Корепанова Н.Л.

г. Севастополь 2019

**Цель работы**

Исследовать стратегии и методы управления виртуальной памятью в современных ОС.

**Вариант №24(9)**

Задание:

Разработать программу, моделирующую работу двух заданных алгоритмов замещения страниц: LFU, LRU, реализации алгоритмов должны быть оформлены в виде подпрограмм. Считать, что виртуальная память имеет объем 1000 страниц. В оперативной памяти имеется 10 страниц, о которых известна вся необходимая для работы алгоритмов информация. Главная программа должна вводить исходные данные, вызывать подпрограммы, реализующие алгоритмы, и печатать результаты.

Исходные данные к программе:

– объем оперативной памяти в страницах (Vоп, к примеру, 4 страниц);

– состояние оперативной памяти (какие страницы в ней находятся)

– последовательность обращений к страницам REF(1), REF(2), ..., REF(N), (можно генерировать случайным образом, НО следует обратить внимание на то, что оба алгоритма должны получать одинаковые исходные данные).

Результаты:

– состояние оперативной памяти после выполнения обращений в моменты времени t1, ..., tN;

– число страничных прерываний из-за отсутствия необходимой страницы (Nстр).

**Текст программы**

#include "pch.h"

#include <Bits.h>

#include <vector>

#include <iostream>

#include <utility>

#include <unordered\_map>

using namespace std;

// Generic function to swap two pairs

void swap(pair<int, int>& a, pair<int, int>& b)

{

pair<int, int> temp = a;

a = b;

b = temp;

}

// Returns the index of the parent node

inline int parent(int i)

{

return (i - 1) / 2;

}

// Returns the index of the left child node

inline int left(int i)

{

return 2 \* i + 1;

}

// Returns the index of the right child node

inline int right(int i)

{

return 2 \* i + 2;

}

// Self made heap tp Rearranges

// the nodes in order to maintain the heap property

void heapify(vector<pair<int, int> >& v,

unordered\_map<int, int>& m, int i, int n)

{

int l = left(i), r = right(i), minim;

if (l < n)

minim = ((v[i].second < v[l].second) ? i : l);

else

minim = i;

if (r < n)

minim = ((v[minim].second < v[r].second) ? minim : r);

if (minim != i) {

m[v[minim].first] = i;

m[v[i].first] = minim;

swap(v[minim], v[i]);

heapify(v, m, minim, n);

}

}

// Function to Increment the frequency

// of a node and rearranges the heap

void increment(vector<pair<int, int> >& v,

unordered\_map<int, int>& m, int i, int n)

{

++v[i].second;

heapify(v, m, i, n);

}

// Function to Insert a new node in the heap

void insert(vector<pair<int, int> >& v,

unordered\_map<int, int>& m, int value, int& n)

{

if (n == v.size()) {

m.erase(v[0].first);

cout << "Cache block " << v[0].first

<< " removed.\n";

v[0] = v[--n];

heapify(v, m, 0, n);

}

v[n++] = make\_pair(value, 1);

m.insert(make\_pair(value, n - 1));

int i = n - 1;

// Insert a node in the heap by swapping elements

while (i && v[parent(i)].second > v[i].second) {

m[v[i].first] = parent(i);

m[v[parent(i)].first] = i;

swap(v[i], v[parent(i)]);

i = parent(i);

}

cout << "Cache block " << value << " inserted.\n";

}

// Function to refer to the block value in the cache

void refer(vector<pair<int, int> >& cache, unordered\_map<int,

int>& indices, int value, int& cache\_size)

{

if (indices.find(value) == indices.end())

insert(cache, indices, value, cache\_size);

else

increment(cache, indices, indices[value], cache\_size);

}

class LRUCache

{

// store keys of cache

list<int> dq;

// store references of key in cache

unordered\_map<int, list<int>::iterator> ma;

int csize; //maximum capacity of cache

public:

LRUCache(int);

void refer(int);

void display();

};

LRUCache::LRUCache(int n)

{

csize = n;

}

/\* Refers key x with in the LRU cache \*/

void LRUCache::refer(int x)

{

// not present in cache

if (ma.find(x) == ma.end())

{

// cache is full

if (dq.size() == csize)

{

//delete least recently used element

int last = dq.back();

dq.pop\_back();

ma.erase(last);

}

}

// present in cache

else

dq.erase(ma[x]);

// update reference

dq.push\_front(x);

ma[x] = dq.begin();

}

// display contents of cache

void LRUCache::display()

{

for (auto it = dq.begin(); it != dq.end();

it++)

cout << (\*it) << " ";

cout << endl;

}

// Driver Code

int main()

{

cout << "LFU:" << endl;

int cache\_max\_size = 4, cache\_size = 0;

vector<pair<int, int> > cache(cache\_max\_size);

unordered\_map<int, int> indices;

refer(cache, indices, 1, cache\_size);

refer(cache, indices, 2, cache\_size);

refer(cache, indices, 1, cache\_size);

refer(cache, indices, 3, cache\_size);

refer(cache, indices, 2, cache\_size);

refer(cache, indices, 4, cache\_size);

refer(cache, indices, 5, cache\_size);

cout << "\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*" << endl;

cout << "LRU:" << endl;

LRUCache ca(4);

ca.refer(1);

ca.refer(2);

ca.refer(1);

ca.refer(3);

ca.refer(2);

ca.refer(4);

ca.refer(5);

ca.display();

return 0;

}

**Тестовые примеры**

Для примера вводится максимальное количество страниц - 4. И вводятся в кэш следующий порядок страниц 1, 2, 1, 3, 2, 4, 5. Алгоритм LFU выталкивает реже всего используемую страницу, в данном примере – страница 3. Алгоритм LRU выталкивает дольше всего не использующуюся страницу, в данном примере – страница 1.

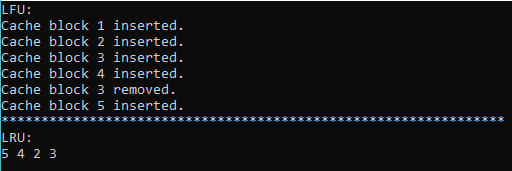


рис. 1 – результат работы программы

**Выводы**

В данной лабораторной работе были изучены стратегии и методы управления виртуальной памятью в современных ОС.