ФЕДЕРАЛЬНОЕ ГОСУДАРСТВЕННОЕ БЮДЖЕТНОЕ   
ОБРАЗОВАТЕЛЬНОЕ УЧРЕЖДЕНИЕ ВЫСШЕГО ОБРАЗОВАНИЯ

«СИБИРСКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ ТЕЛЕКОММУНИКАЦИЙ И ИНФОРМАТИКИ»

КАФЕДРА ВС

ОТЧЁТ ПО ЛАБОРАТОРНОЙ РАБОТЕ №3  
«Оценка производительности подсистемы памяти.»

по дисциплине «Архитектура вычислительных систем»

Выполнил: студент гр. ИП-811  
Адов Артем Сергеевич

Проверил: ст. преп. Кафедры ВС   
Ткачёва Татьяна Алексеевна

Новосибирск 2020

Содержание:

Постановка задачи……………………………………………………………….…**3-5**

Выполнение работы…….……………………………………………….……………**6**

Результат работы……………………………………………………………...…..**7-11**

Приложение……………………………………………...................................…**12-19**

Вывод………………………………………………………………………………...**20**

# 

# Постановка задачи

**Задание:** разработать программу (benchmark) для оценки производительности подсистемы памяти.

1. Написать программу(функцию)на языке С/С++/C#для оценки производительности подсистемы памяти. На вход программы подать следующие аргументы.

1) Подсистема памяти. Предусмотреть возможность указать подсистему для проверки производительности: RAM (оперативная память), HDD/SSD и flash.

2) Размер блока данных в байтах, Кб или Мб. Если размерность не указана, то в байтах, если указана, то соответственно в Кбайтах или Мбайтах.

3) Число испытаний, т.е. число раз повторений измерений. Пример вызова программы: ./memory\_test –m RAM –b 1024|1Kb –l 10 или ./memory\_bandwidth ––memory-type RAM|HDD|SSD|flash ––block-size 1024|1Kb ––launch-count 10

В качестве блока данных использовать одномерный массив, в котором произведение числа элементов на их размерность равна требуемому размеру блока данных. Массив инициализировать случайными значениями. Для тестирования HDD/SSD и flash создать в программе файлы в соответствующих директориях. Измерение времени реализовать с помощью функции clock\_gettime() или аналогичной с точность до наносекунд. Измерять время исключительно на запись элемента в память или считывание из неё, без операций генерации или преобразования данных.

На выходе программы в одну строку CSV файла со следующей структурой: [MemoryType;BlockSize;ElementType;BufferSize;LaunchNum;Timer;WriteTime;AverageWriteTime;WriteBandwidth; AbsError(write);RelError(write);ReadTime;AverageReadTime;ReadBandwidthAbsError(read);RelError(read);], где

MemoryType –тип памяти (RAM|HDD|SSD|flash) или модель устройства, на котором проводятся испытания;

BlockSize – размер блока данных для записи и чтения на каждом испытании; ElementType – тип элементов используемых для заполнения массива данных; BufferSize – размер буфера, т.е. порции данных для выполнения одно операции записи или чтения;

LaunchNum – порядковый номер испытания;

Timer – название функции обращения к таймеру (для измерения времени); WriteTime – время выполнения отдельного испытания с номером LaunchNum [секунды];

AverageWriteTime – среднее время записи из LaunchNum испытаний [секунды]; WriteBandwidth – пропускная способность памяти (BLOCK\_SIZE/AverageWriteTime) \* 106 [Mb/s] AbsError(write) – абсолютная погрешность измерения времени записи или СКО [секунды];

RelError(write) – относительная погрешность измерения времени [%];

ReadTime – время выполнения отдельного испытания LaunchNum [секунды]; AverageReadTime – среднее время записи из LaunchNum испытаний [секунды]; ReadBandwidth – пропускная способность памяти (BLOCK\_SIZE/AverageReadTime) \* 106 [Mб/сек.] AbsError(read) – абсолютная погрешность измерения времени чтения или СКО [секунды];

RelError(read) – относительная погрешность измерения времени [%].

2. Написать программу(функцию) на языке С/С++/C# или скрипт (benchmark) реализующий серию испытаний программы(функции) из п.1. Оценить пропускную способность оперативной памяти при работе с блоками данных равными объёму кэш-линии, кэш-памяти L1, L2 и L3 уровня и превышающего его. Для HDD|SSD и flash провести серию из 20 испытаний с блоками данных начиная с 4 Мб с шагом 4Мб. Результаты всех испытаний сохранить в один CSV файл со структурой, описанной в п.1. \* Для HDD|SSD и flash оценить влияние размера буфера (BufferSize) на пропускную способность памяти.

3. На основе CSV файла построить сводные таблицы и диаграммы отражающие: 1) Зависимость пропускной способности записи и чтения от размера блока данных (BlockSize) для разного типа памяти;

2) Зависимость погрешности измерения пропускной способности от размера блока данных для разного типа памяти;

3) Зависимость погрешности измерений от числа испытаний LaunchNum; 4) \* Зависимость пропускной способности памяти от размера буфера для HDD|SSD и flash памяти; 4. \*\* Оценить пропускную способность файла подкачки (windows) или раздела SWAP (linux). Сравнить с пропускной способностью RAM, HDD/SSD и flash.

# Выполнение работы

# Программа для оценки производительности подсистемы памяти написана на языке C. Испытаниям подвергаются следующие устройства памяти:

# • Жесткий диск TOSHIBA MK3263GSXN емкостью 320Гб;

# • Оперативная память поколения DDR2;

# • Flash-накопитель емкостью 32Гб.

# При запуске программы от пользователя требуется ввести тип памяти, над которой проводятся измерения, объем блока данных и количество испытаний. Для измерения времени используется функция clock() из библиотеки time.h. Время отсчитывается в тиках процессора . Оценка точности проводится по формуле дисперсии:

# Проведено тестирование RAM с блоками данных, равными объёму кэш памяти L1(64 Кб) и L2(1 Мб) уровня. Проведено тестирование HDD и flash с блоками данных 4Мб увеличивающимися с каждой итерацией теста на 4Мб.

# Результат работы

# 

# 

# 

# 

# 

Рисунок 1. Результат записи в файл table.csv

# 

# 

# 

# «RAM»

# 

# 

# 

# 

# Приложение

# *acs\_3.c*

#include <stdio.h>

#include <stdlib.h>

#include <time.h>

#include <math.h>

#define ERROR\_QUANTITY\_ARGUMENT\_IN\_MAIN 0b00000001

#define ERROR\_PARAMETR\_MEMORY\_TYPE 0b00000010

#define ERROR\_PARAMETR\_BLOCK\_SIZE 0b00000100

#define ERROR\_PARAMETR\_QUANTITY\_TEST 0b0001000

#define ERROR\_ALLOCATION\_MEMORY\_TEST\_ARRAY 0b00010000

#define ERROR\_OPEN\_FILE\_TEST 0b00100000

#define ERROR\_OPEN\_FILE\_CSV 0b01000000

#define ERROR\_ALLOCATION\_MEMORY\_ARRAY\_TIME 0b10000000

#define FIRST\_CODE\_SYMBOL\_BY\_DIGIT 0x30

#define LAST\_CODE\_SYMBOL\_BY\_DIGIT 0x39

#define CONVERT\_CODE\_SYMBOL\_BY\_DIGIT\_IN\_NUMBER 0x0F

#define CORRECT\_QUANTITY\_ARGUMENT\_IN\_MAIN 4

#define COEFFICIENT\_CONVERT\_IN\_BYTE 1024

#define SIZE\_TYPE\_DOUBLE 8

#define COEFFICIENT\_FOR\_RAND 10

#define QUANTITY\_OPERATION\_OF\_MEMORY 2

#define COEFFICIENT\_FOR\_CONVERT\_IN\_PERCENT 100

#define STEP\_SIZE\_MEMORY\_BY\_TEST 524288

#define INDEX\_MEMORY\_TYPE\_FROM\_ARGV 1

#define INDEX\_BLOCK\_SIZE\_FROM\_ARGV 2

#define INDEX\_QUANTITY\_TEST\_FROM\_ARGV 3

#define INDEX\_WRITE\_FOR\_ARRAY\_TIME 0

#define INDEX\_READ\_FOR\_ARRAY\_TIME 1

#define FILE\_NAME\_FOR\_HDD "/home/deathangel/Desktop/test\_file.txt"

#define FILE\_NAME\_FOR\_SSD "/home/deathangel/Desktop/file\_test.txt"

#define FILE\_NAME\_FOR\_FLASH "/media/deathangel/ANNA/file\_test.txt"

#define FILE\_NAME\_TABLE\_CVS "table.csv"

#define WRITE\_AND\_READ "w+"

#define ADDITIONAL\_RECORDING "a"

#define READ\_ONLY "r"

#define WRITE "write"

#define READ "read"

#define NAME\_TYPE\_DOUBLE "double"

#define NAME\_FUNCTION\_CALL\_FOR\_TIME "clock()"

int Check\_Argc(int argc)

{

if(argc == CORRECT\_QUANTITY\_ARGUMENT\_IN\_MAIN)

{

return 0;

}

return ERROR\_QUANTITY\_ARGUMENT\_IN\_MAIN;

}

int Check\_Memory\_Type(char argv[])

{

if((argv[0] == 'R') && (argv[1] == 'A') && (argv[2] == 'M') && (argv[3] == '\0'))

{

return 0;

}

if((argv[0] == 'H') && (argv[1] == 'D') && (argv[2] == 'D') && (argv[3] == '\0'))

{

return 0;

}

if((argv[0] == 'S') && (argv[1] == 'S') && (argv[2] == 'D') && (argv[3] == '\0'))

{

return 0;

}

if((argv[0] == 'f') && (argv[1] == 'l') && (argv[2] == 'a') && (argv[3] == 's') && (argv[4] == 'h') && (argv[5] == '\0'))

{

return 0;

}

return ERROR\_PARAMETR\_MEMORY\_TYPE;

}

int Convert\_Block\_Size(unsigned long int\* quantity\_element\_in\_array, char argv[])

{

for(short int i = 0; argv[i] != '\0'; i++)

{

if((argv[i] >= FIRST\_CODE\_SYMBOL\_BY\_DIGIT) && (argv[i] <= LAST\_CODE\_SYMBOL\_BY\_DIGIT))

{

(\*quantity\_element\_in\_array) = ((\*quantity\_element\_in\_array) + (int)(argv[i] & CONVERT\_CODE\_SYMBOL\_BY\_DIGIT\_IN\_NUMBER));

(\*quantity\_element\_in\_array) = ((\*quantity\_element\_in\_array) \* 10);

if(argv[i+1] == '\0')

{

(\*quantity\_element\_in\_array) = (((\*quantity\_element\_in\_array) / 10) / SIZE\_TYPE\_DOUBLE);

}

}

else

{

(\*quantity\_element\_in\_array) = ((\*quantity\_element\_in\_array) / 10);

if((argv[i] == 'K') && (argv[i+1] == 'b') && (argv[i+2] == '\0'))

{

(\*quantity\_element\_in\_array) = ((\*quantity\_element\_in\_array) \* (COEFFICIENT\_CONVERT\_IN\_BYTE / SIZE\_TYPE\_DOUBLE));

return 0;

}

else

{

if((argv[i] == 'M') && (argv[i+1] == 'b') && (argv[i+2] == '\0'))

{

(\*quantity\_element\_in\_array) = ((\*quantity\_element\_in\_array) \* (COEFFICIENT\_CONVERT\_IN\_BYTE / SIZE\_TYPE\_DOUBLE) \* COEFFICIENT\_CONVERT\_IN\_BYTE);

return 0;

}

else

{

return ERROR\_PARAMETR\_BLOCK\_SIZE;

}

}

}

}

if((\*quantity\_element\_in\_array) > 0)

{

return 0;

}

return ERROR\_PARAMETR\_BLOCK\_SIZE;

}

int Convert\_Quantity\_Test(int\* quantity\_test, char argv[])

{

for(short int i = 0; argv[i] != '\0'; i++)

{

if((argv[i] >= FIRST\_CODE\_SYMBOL\_BY\_DIGIT) && (argv[i] <= LAST\_CODE\_SYMBOL\_BY\_DIGIT))

{

(\*quantity\_test) = ((\*quantity\_test) + (int)(argv[i] & CONVERT\_CODE\_SYMBOL\_BY\_DIGIT\_IN\_NUMBER));

(\*quantity\_test) = ((\*quantity\_test) \* 10);

}

else

{

return ERROR\_PARAMETR\_QUANTITY\_TEST;

}

}

(\*quantity\_test) = ((\*quantity\_test) / 10);

if((\*quantity\_test) > 0)

{

return 0;

}

return ERROR\_PARAMETR\_QUANTITY\_TEST;

}

int Fill\_Rand\_Test\_Array(double\*\* test\_array, int quantity\_element\_in\_array)

{

double temp = 0.0;

(\*test\_array) = (double\*)calloc(quantity\_element\_in\_array, sizeof(double));

if((\*test\_array) == NULL)

{

return ERROR\_ALLOCATION\_MEMORY\_TEST\_ARRAY;

}

for(unsigned long int i = 0; i < quantity\_element\_in\_array; i++)

{

temp = (double)(rand()%COEFFICIENT\_FOR\_RAND) + 0.5;

(\*test\_array)[i] = ((double)(rand()%COEFFICIENT\_FOR\_RAND) / temp);

}

return 0;

}

int Write\_In\_Csv\_File(char\* argv[], double\* array\_time[], double absolute\_oversight[], double relativity\_oversight[], unsigned long int quantity\_element\_in\_array, int number\_test)

{

FILE\* file = NULL;

double\* average\_time = NULL;

average\_time = (double\*)calloc(QUANTITY\_OPERATION\_OF\_MEMORY,sizeof(double));

if(average\_time == NULL)

{

return ERROR\_ALLOCATION\_MEMORY\_ARRAY\_TIME;

}

file = fopen(FILE\_NAME\_TABLE\_CVS,ADDITIONAL\_RECORDING);

if(file == NULL)

{

return ERROR\_OPEN\_FILE\_CSV;

}

for(short int i = 0; i < QUANTITY\_OPERATION\_OF\_MEMORY; i++)

{

for(short int j = 0; j < number\_test; j++)

{

average\_time[i] = average\_time[i] + array\_time[i][j];

}

average\_time[i] = (average\_time[i] / number\_test);

}

fprintf(file,"%s, %lu, %s, %lu, %d, %s, %f, %f, %Lf, %f, %f, %f, %f, %Lf, %f, %f\n",

argv[INDEX\_MEMORY\_TYPE\_FROM\_ARGV],

(quantity\_element\_in\_array \* SIZE\_TYPE\_DOUBLE),

NAME\_TYPE\_DOUBLE,

quantity\_element\_in\_array,

number\_test,

NAME\_FUNCTION\_CALL\_FOR\_TIME,

array\_time[INDEX\_WRITE\_FOR\_ARRAY\_TIME][number\_test - 1],

average\_time[INDEX\_WRITE\_FOR\_ARRAY\_TIME],

((long double)(quantity\_element\_in\_array \* SIZE\_TYPE\_DOUBLE) / (long double)average\_time[INDEX\_WRITE\_FOR\_ARRAY\_TIME]),

absolute\_oversight[INDEX\_WRITE\_FOR\_ARRAY\_TIME],

relativity\_oversight[INDEX\_WRITE\_FOR\_ARRAY\_TIME],

array\_time[INDEX\_READ\_FOR\_ARRAY\_TIME][number\_test - 1],

average\_time[INDEX\_READ\_FOR\_ARRAY\_TIME],

((long double)(quantity\_element\_in\_array \* SIZE\_TYPE\_DOUBLE) / (long double)average\_time[INDEX\_READ\_FOR\_ARRAY\_TIME]),

absolute\_oversight[INDEX\_READ\_FOR\_ARRAY\_TIME],

relativity\_oversight[INDEX\_READ\_FOR\_ARRAY\_TIME]);

fclose(file);

return 0;

}

int Find\_Oversight(double\* array\_time[], double\* absolute\_oversight[], double\* relativity\_oversight[], int size\_array)

{

double dispersion\_value = 0.0;

double mean = 0.0;

//write

(\*absolute\_oversight) = (double\*)calloc(QUANTITY\_OPERATION\_OF\_MEMORY,sizeof(double));

if(absolute\_oversight == NULL)

{

return ERROR\_ALLOCATION\_MEMORY\_ARRAY\_TIME;

}

(\*relativity\_oversight) = (double\*)calloc(QUANTITY\_OPERATION\_OF\_MEMORY,sizeof(double));

if(relativity\_oversight == NULL)

{

return ERROR\_ALLOCATION\_MEMORY\_ARRAY\_TIME;

}

for(short int i = 0; i < size\_array; i++)

{

mean = mean + array\_time[INDEX\_WRITE\_FOR\_ARRAY\_TIME][i];

}

mean = (mean / (double)size\_array);

for(short int i = 0; i < size\_array; i++)

{

dispersion\_value = (dispersion\_value + pow((array\_time[INDEX\_WRITE\_FOR\_ARRAY\_TIME][i] - mean), 2));

}

dispersion\_value = (dispersion\_value / (double)size\_array);

(\*absolute\_oversight)[INDEX\_WRITE\_FOR\_ARRAY\_TIME] = sqrt(dispersion\_value);

(\*relativity\_oversight)[INDEX\_WRITE\_FOR\_ARRAY\_TIME] = ((\*absolute\_oversight)[INDEX\_WRITE\_FOR\_ARRAY\_TIME] / mean \* COEFFICIENT\_FOR\_CONVERT\_IN\_PERCENT);

//read

dispersion\_value = 0.0;

mean = 0.0;

for(short int i = 0; i < size\_array; i++)

{

mean = mean + array\_time[INDEX\_READ\_FOR\_ARRAY\_TIME][i];

}

mean = (mean / (double)size\_array);

for(short int i = 0; i < size\_array; i++)

{

dispersion\_value = (dispersion\_value + pow((array\_time[INDEX\_READ\_FOR\_ARRAY\_TIME][i] - mean), 2));

}

dispersion\_value = (dispersion\_value / (double)size\_array);

(\*absolute\_oversight)[INDEX\_READ\_FOR\_ARRAY\_TIME] = sqrt(dispersion\_value);

(\*relativity\_oversight)[INDEX\_READ\_FOR\_ARRAY\_TIME] = ((\*absolute\_oversight)[INDEX\_READ\_FOR\_ARRAY\_TIME] / mean \* COEFFICIENT\_FOR\_CONVERT\_IN\_PERCENT);

return 0;

}

int Benchmark\_For\_Memory(unsigned long int quantity\_element\_in\_array, int quantity\_test, char\* argv[])

{

double\*\* array\_time = NULL;

double start\_time = 0, final\_time = 0;

FILE\* file = NULL;

double\* test\_array = NULL;

double\* absolute\_oversight = NULL;

double\* relativity\_oversight = NULL;

double value = 0.0;

array\_time = (double\*\*)calloc(QUANTITY\_OPERATION\_OF\_MEMORY,sizeof(double\*));

for(short int i = 0; i < QUANTITY\_OPERATION\_OF\_MEMORY; i++)

{

array\_time[i] = (double\*)calloc(quantity\_test,sizeof(double));

}

if(array\_time == NULL)

{

return ERROR\_ALLOCATION\_MEMORY\_ARRAY\_TIME;

}

if(argv[INDEX\_MEMORY\_TYPE\_FROM\_ARGV][0] == 'R')

{

for(short int i = 0; i < quantity\_test; i++)

{

//write

if(Fill\_Rand\_Test\_Array(&test\_array, quantity\_element\_in\_array) != 0)

{

return ERROR\_ALLOCATION\_MEMORY\_TEST\_ARRAY;

}

for(unsigned long int j = 0; j < quantity\_element\_in\_array; j++)

{

start\_time = clock();

value = test\_array[j];

final\_time = (clock() - start\_time) / CLOCKS\_PER\_SEC;

array\_time[INDEX\_WRITE\_FOR\_ARRAY\_TIME][i] = (array\_time[INDEX\_WRITE\_FOR\_ARRAY\_TIME][i] + final\_time);

}

//read

for(unsigned long int j = 0; j < quantity\_element\_in\_array; j++)

{

value = ((double)(rand()%COEFFICIENT\_FOR\_RAND) / (double)(rand()%COEFFICIENT\_FOR\_RAND));

start\_time = clock();

test\_array[j] = value;

final\_time = (clock() - start\_time) / CLOCKS\_PER\_SEC;

array\_time[INDEX\_READ\_FOR\_ARRAY\_TIME][i] = (array\_time[INDEX\_READ\_FOR\_ARRAY\_TIME][i] + final\_time);

}

Find\_Oversight(array\_time, &absolute\_oversight, &relativity\_oversight, i + 1);

if(Write\_In\_Csv\_File(argv, array\_time, absolute\_oversight, relativity\_oversight, quantity\_element\_in\_array, (i+1)) != 0)

{

return ERROR\_OPEN\_FILE\_CSV;

}

}

return 0;

}

if(argv[INDEX\_MEMORY\_TYPE\_FROM\_ARGV][0] == 'H')

{

file = fopen(FILE\_NAME\_FOR\_HDD,WRITE\_AND\_READ);

}

if(argv[INDEX\_MEMORY\_TYPE\_FROM\_ARGV][0] == 'S')

{

file = fopen(FILE\_NAME\_FOR\_SSD,WRITE\_AND\_READ);

}

if(argv[INDEX\_MEMORY\_TYPE\_FROM\_ARGV][0] == 'f')

{

file = fopen(FILE\_NAME\_FOR\_FLASH,WRITE\_AND\_READ);

}

if(file == NULL)

{

return ERROR\_OPEN\_FILE\_TEST;

}

for(short int i = 0; i < quantity\_test; i++)

{

if(Fill\_Rand\_Test\_Array(&test\_array, quantity\_element\_in\_array) != 0)

{

return ERROR\_ALLOCATION\_MEMORY\_TEST\_ARRAY;

}

for(unsigned long int j = 0; j < quantity\_element\_in\_array; j++)

{

start\_time = clock();

fprintf(file,"%lf",test\_array[j]);

final\_time = (clock() - start\_time) / CLOCKS\_PER\_SEC;

array\_time[INDEX\_WRITE\_FOR\_ARRAY\_TIME][i] = (array\_time[INDEX\_WRITE\_FOR\_ARRAY\_TIME][i] + final\_time);

}

//read

for(unsigned long int j = 0; j < quantity\_element\_in\_array; j++)

{

value = ((double)(rand()%COEFFICIENT\_FOR\_RAND) / (double)(rand()%COEFFICIENT\_FOR\_RAND));

start\_time = clock();

fscanf(file,"%lf",&test\_array[j]);

final\_time = (clock() - start\_time) / CLOCKS\_PER\_SEC;

array\_time[INDEX\_READ\_FOR\_ARRAY\_TIME][i] = (array\_time[INDEX\_READ\_FOR\_ARRAY\_TIME][i] + final\_time);

}

Find\_Oversight(array\_time, &absolute\_oversight, &relativity\_oversight, i + 1);

if(Write\_In\_Csv\_File(argv, array\_time, absolute\_oversight, relativity\_oversight, quantity\_element\_in\_array, (i+1)) != 0)

{

return ERROR\_OPEN\_FILE\_CSV;

}

quantity\_element\_in\_array = quantity\_element\_in\_array + STEP\_SIZE\_MEMORY\_BY\_TEST;//

}

fclose(file);

return 0;

}

int Create\_Table\_Csv()

{

FILE\* file = NULL;

file = fopen(FILE\_NAME\_TABLE\_CVS,READ\_ONLY);

{

if(file != NULL)

{

fclose(file);

file = NULL;

return 0;

}

}

file = fopen(FILE\_NAME\_TABLE\_CVS,WRITE\_AND\_READ);

if(file == NULL)

{

return ERROR\_OPEN\_FILE\_CSV;

}

fprintf(file, "MemoryType, BlockSize, ElementType, BufferSize, LaunchNum, Timer, WriteTime, AverageWriteTime, WriteBandwidth, AbsError(write), RelError(write), ReadTime, AverageReadTime, ReadBandwidth, AbsError(read), RelError(read)\n");

fclose(file);

return 0;

}

int main(int argc, char\* argv[])

{

printf("%d",argc);

int error\_flag = 0b0;

unsigned long int quantity\_element\_in\_array = 0;

int quantity\_test = 0;

srand(time(NULL));

error\_flag = (error\_flag | Check\_Argc(argc));

// printf("ERR:%d\n",error\_flag);

error\_flag = (error\_flag | Create\_Table\_Csv());

error\_flag = (error\_flag | Check\_Memory\_Type(argv[INDEX\_MEMORY\_TYPE\_FROM\_ARGV]));

// printf("ERR:%d\n",error\_flag);

error\_flag = (error\_flag | Convert\_Block\_Size(&quantity\_element\_in\_array, argv[INDEX\_BLOCK\_SIZE\_FROM\_ARGV]));

// printf("ERR:%d\n",error\_flag);

error\_flag = (error\_flag | Convert\_Quantity\_Test(&quantity\_test, argv[INDEX\_QUANTITY\_TEST\_FROM\_ARGV]));

// printf("ERR:%d\n",error\_flag);

error\_flag = (error\_flag | Benchmark\_For\_Memory(quantity\_element\_in\_array, quantity\_test, argv));

printf("ERR:%d\n",error\_flag);

return error\_flag;

}

# *acs\_3.sh*

#!/bin/bash

gcc -O2 -Wall -o key acs\_3.c -lm;

./key RAM 64Kb 20;

./key RAM 1Mb 20;

./key HDD 4Mb 20;

./key flash 4Mb 20;

Вывод:

В результате выполнения лабораторной работы написана программа для оценки производительности подсистемы памяти (RAM|HDD|flash) на языке C. В результате оценки производительности памяти и анализа построенных диаграммам можно сделать некоторые выводы о том, что:

1) Оперативная память производительнее чем HDD|flash подсистемы памяти

2) Чем больше размер тестируемого блока данных и больше количество проведенных испытаний, тем меньше погрешность