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-813  
Бурдуковский Илья Александрович

Проверил: ст. преп. Кафедры ВС   
Токмашева Елизавета Ивановна

Новосибирск 2020

Содержание

[Выполнение работы 6](#_Toc55302170)

[Запуск программы 6](#_Toc55302171)

[Результат работы 7](#_Toc55302172)

[Приложение 11](#_Toc55302177)

**Постановка задачи**

*Разработать программу (benchmark) для оценки производительности подсистемы памяти.*

*1. Написать программум(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) 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) \* 10 6 [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) \* 10 6 [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;*

# Выполнение работы

Перед оценкой производительности подсистем памяти нужно определить размера кеш-линии:и. Например, с помощью команды lscpu



Запуск программы

Сборка программы:

$ gcc ./src/main.c -o main

Запуск программы:

$ ./main RAM 10 20

$ ./main HDD 10K 20 filepath

$ ./main SSD 10M 20 filepath

$ ./main flash 100M 20 filepath

Где параметр после названия исполняемого файла — это вид тестируемой памяти, допустимые значения RAM, HDD, SSD, flash.

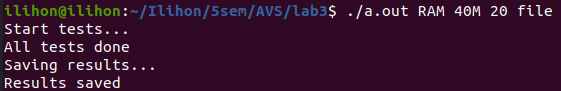
Для обозначения размера блока памяти доступны наименования K – килобайты, М – мегабайты, ничего - байты

После блока памяти указывается количество испытаний.

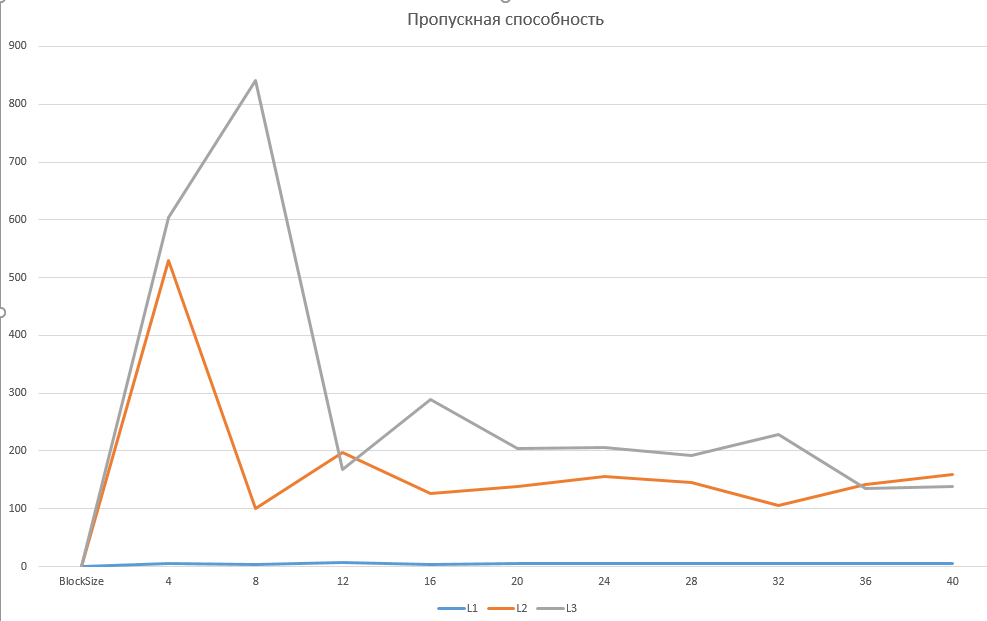
После количество испытаний указывается путь к файлу ля тестирования HDD, SSD и flash памяти.

# Результат работы

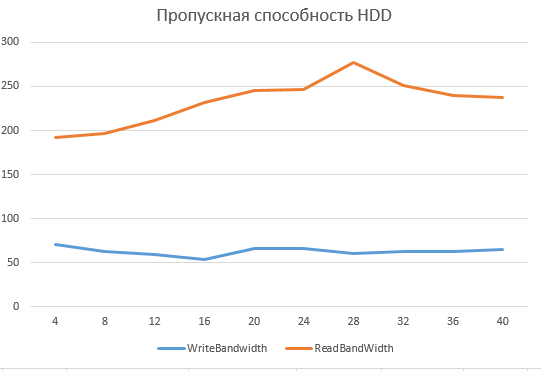
**Запуск программы**



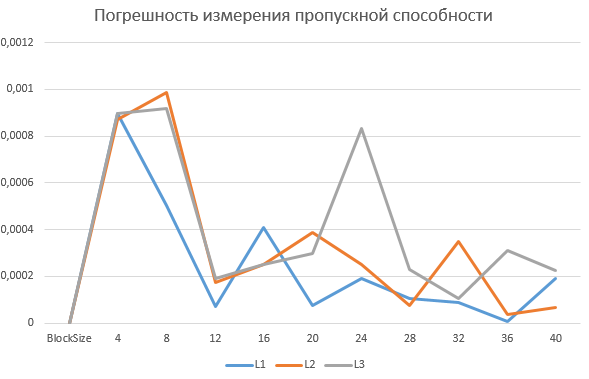
**График зависимости пропускной способности от размера блока данных (МБ) для RAM**



## **График зависимости пропускной способности от размера блока данных (МБ) для записи и считывания HDD**



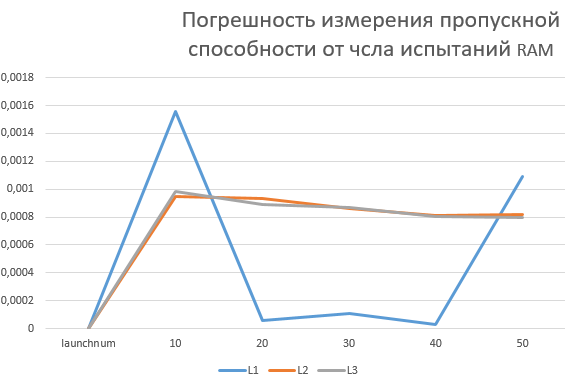
**График зависимости погрешностей измерений пропускной способности от размера блока данных (МБ) для RAM**



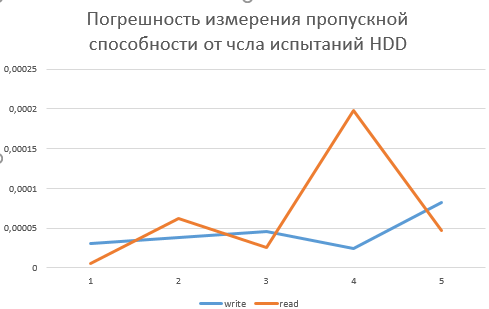
## **График зависимости пропускной способности от размера блока данных (МБ) для записи и считывания HDD**

## 

## **График зависимости пропускной способности от числа испытаний RAM**



**График зависимости пропускной способности от числа испытаний HDD**



Можно отметить следующие интересные особенности:

* Бенчмарк показывает довольно нестабильные результаты на виртуальной машине при работе с HDD памятью.

# Приложение

#include <iostream>

#include <time.h>

#include <fstream>

#include <stdlib.h>

#include <string>

#include <string.h>

#include <chrono>

#include <math.h>

using namespace std;

struct CmdLine{

string MemType;

int block\_size{};

int alloc\_size{};

int launch\_amount{};

string filepath{};

};

void parse\_arguments(int argc, char \*argv[], CmdLine \*arguments);

void Clock\_ram(int block\_size, int alloc\_size, int repeats, long long \*results, double &absolute, double &relative, double &average);

void Clock\_hdd\_ssd\_flash\_write(int block\_size, int alloc\_size, int repeats, string filename, long long \*results, double &absolute, double &relative, double &average);

void Clock\_hdd\_ssd\_flash\_read(int block\_size, int alloc\_size, int repeats, string filename, long long \*results, double &absolute, double &relative, double &average);

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

{

cout.precision(15);

if( argc < 4)

{

printf("Invalid inputs\n ");

return -1;

}

auto arguments = CmdLine();

parse\_arguments(argc,argv,&arguments);

//cout << arguments.MemType << endl << arguments.block\_size << endl << arguments.alloc\_size

// << endl << arguments.launch\_amount << endl << arguments.filepath;

double absolute = 0;

double relative = 0;

double average = 0;

double absolute\_read = 0;

double relative\_read = 0;

double average\_read = 0;

cout << "Start tests..." << endl;

long long result[arguments.launch\_amount];

long long result\_read[arguments.launch\_amount];

if (arguments.MemType == "RAM"){

Clock\_ram(arguments.block\_size, arguments.alloc\_size, arguments.launch\_amount, result, absolute, relative, average);

/\*for(int i = 0; i<arguments.launch\_amount;i++){

cout << result[i]<< endl;

}\*/

}

else{

Clock\_hdd\_ssd\_flash\_write(arguments.block\_size, arguments.alloc\_size, arguments.launch\_amount,

arguments.filepath, result, absolute, relative, average);

Clock\_hdd\_ssd\_flash\_read(arguments.block\_size, arguments.alloc\_size, arguments.launch\_amount,

arguments.filepath, result\_read, absolute\_read, relative\_read, average\_read);

/\*for(int i = 0; i<arguments.launch\_amount;i++){

cout << result[i]<< endl;

}\*/

}

//cout << absolute << " - "<< relative << " - " << average <<endl;

//cout << absolute\_read << " - "<< relative\_read << " - " << average\_read <<endl;

cout << "All tests done" << endl;

cout << "Saving results..." << endl;

ofstream benchmark\_output;

benchmark\_output.open("bench result.csv", ios\_base::app);

benchmark\_output << "MemoryType;BlockSize;ElementType;BufferSize;LaunchNum;Timer;"

"WriteTime;AverageWriteTime;WriteBandwidth;AbsErr(write);RelErr(write);ReadTime;"

"AverageReadTime;ReadBandWidth;AbsErr(read);RelErr(read)" << endl;

//cout << arguments.MemType << endl << arguments.block\_size << endl << arguments.alloc\_size

// << endl << arguments.launch\_amount << endl << arguments.filepath;

for(int i=0; i < arguments.launch\_amount;i++)

{

benchmark\_output << arguments.MemType << ";"<< arguments.block\_size << ";byte;" <<

arguments.alloc\_size << ";" << i+1 << ";chrono;" << result[i] << ";" << average << ";" <<

arguments.block\_size / average << ";" << absolute << ";" << relative << ";";

if(arguments.MemType != "RAM"){

benchmark\_output << result\_read[i] << ";" << average\_read << ";" <<

arguments.block\_size / average\_read << ";" << absolute\_read << ";" << relative\_read << ";";

}

else{

benchmark\_output << result[i] << ";" << average << ";" <<

arguments.block\_size / average << ";" << absolute << ";" << relative << ";";

}

benchmark\_output << endl;

}

benchmark\_output.close();

cout << "Results saved" << endl;

return 0;

}

string parse\_memory\_type(char\* str) {

if (strcmp(str,"RAM")==0)

return "RAM";

else if (strcmp(str,"HDD")==0)

return "HDD";

else if (strcmp(str,"SSD")==0)

return "SSD";

else if (strcmp(str,"flash")==0)

return "flash";

else

throw std::runtime\_error("Incorrect memory type");

}

int parse\_block\_size(char\* str) {

int value = 0;

char type[10]="\0";

if(sscanf(str,"%d %s",&value,type)<1)

{

printf("Incorrect block size\n");

return -1;

}

if (strcmp(type,"\0")==0)

return value;

else if (strcmp(type,"K")==0)

return value \* (1 << 10);

else if (strcmp(type,"M")==0)

return value \* (1 << 20);

else

throw runtime\_error("Incorrect block size type");

}

void parse\_arguments(int argc, char \*argv[], CmdLine \*arguments) {

if (argc > 3) {

arguments->MemType = parse\_memory\_type(argv[1]);

arguments->block\_size = parse\_block\_size(argv[2]);

arguments->launch\_amount = atoi(argv[3]);

if (arguments->block\_size > (4 \* (1 << 20)))

arguments->alloc\_size = 4 \* (1 << 20);

else

arguments->alloc\_size = arguments->block\_size;

if(arguments->MemType == "HDD" || arguments->MemType == "SSD" || arguments->MemType == "flash" ){

if (argc > 4) {

arguments->filepath = argv[4];

}

else{

arguments->filepath = "BenchmarkWritingTest.txt";

}

}

} else {

throw runtime\_error("You must provide memory type, block size and launch amount");

}

}

void Clock\_ram(int block\_size, int alloc\_size, int repeats, long long \*results, double &absolute, double &relative, double &average) {

long long microseconds;

for (unsigned int i = 0; i < repeats; ++i) {

uint8\_t \*arr;

int counter = 0;

auto start\_time = chrono::high\_resolution\_clock::now();

while (counter < block\_size) {

arr = new uint8\_t[alloc\_size];

counter += alloc\_size;

}

auto stop\_time = chrono::high\_resolution\_clock::now();

delete[] arr;

auto result = stop\_time - start\_time;

microseconds = chrono::duration\_cast<chrono::nanoseconds>(result).count();

results[i] = microseconds;

average += microseconds;

}

average = average / repeats;

absolute = fabs(average - (double)(microseconds));

relative = absolute / average \* 100;

return;

}

void Clock\_hdd\_ssd\_flash\_write(int block\_size, int alloc\_size, int repeats, string filename, long long \*results, double &absolute, double &relative, double &average) {

long long microseconds;

uint8\_t arr[alloc\_size];

for (unsigned int i = 0; i < repeats; ++i) {

for (int j = 0; j < alloc\_size; ++j) {

arr[j] = rand() % (1 << 8);

}

int counter = 0;

auto file = ofstream(filename, ios::binary | ios::trunc);

auto start\_time = chrono::high\_resolution\_clock::now();

while (counter < block\_size) {

file.write(reinterpret\_cast<const char \*>(arr), sizeof(arr));

counter += alloc\_size;

}

auto stop\_time = chrono::high\_resolution\_clock::now();

file.close();

auto result = stop\_time - start\_time;

microseconds = chrono::duration\_cast<chrono::nanoseconds>(result).count();

results[i] = microseconds;

average += microseconds;

}

average = average / repeats;

absolute = fabs(average - (double)(microseconds));

relative = absolute / average \* 100;

return;

}

void Clock\_hdd\_ssd\_flash\_read(int block\_size, int alloc\_size, int repeats, string filename, long long \*results, double &absolute, double &relative, double &average) {

long long microseconds;

uint8\_t arr[alloc\_size];

for (unsigned int i = 0; i < repeats; ++i) {

for (int j = 0; j < alloc\_size; ++j) {

arr[j] = rand() % (1 << 8);

}

int counter = 0;

auto file = fstream(filename, ios::binary | ios::trunc);

while (counter < block\_size) {

file.write(reinterpret\_cast<const char \*>(arr), sizeof(arr));

counter += alloc\_size;

}

file.close();

file = fstream(filename, ios::in | ios::binary);

auto start\_time = chrono::high\_resolution\_clock::now();

while (!file.eof()) {

file.read(reinterpret\_cast<char \*>(arr), sizeof(arr));

}

auto stop\_time = chrono::high\_resolution\_clock::now();

file.close();

auto result = stop\_time - start\_time;

microseconds = chrono::duration\_cast<chrono::nanoseconds>(result).count();

results[i] = microseconds;

average += microseconds;

}

average = average / repeats;

absolute = fabs(average - (double)(microseconds));

relative = absolute / average \* 100;

return;

}