Министерство науки и высшего образования Российской Федерации
Федеральное государственное бюджетное образовательное учреждение высшего образования
«Московский государственный технический университет имени Н.Э. Баумана
(национальный исследовательский университет)»
(МГТУ им. Н.Э. Баумана)

ФАКУЛЬТЕТ «ИНФОРМАТИКА И СИСТЕМЫ УПРАВЛЕНИЯ»

КАФЕДРА «ПРОГРАММНОЕ ОБЕСПЕЧЕНИЕ ЭВМ И ИНФОРМАЦИОННЫЕ ТЕХНОЛОГИИ»

ОТЧЕТ ПО УЧЕБНОЙ ПРАКТИКЕ

Студент **Шавиш Тарек**

Группа **ИУ7И-21Б**

Тип практики **Проектно-технологическая практика**

Название предприятия **НУК ИУ МГТУ им. Н. Э. Баумана**

Студент _____ **Шавиш Тарек**

Руководитель практики _____ **Ломовской И. В.**

Руководитель практики _____ **Кострицкий А. С.**

Оценка _____

2023 г.

ВВЕДЕНИЕ

Цель даннои практики состаит в овладении методами последовательного получении исполняемого файла, изучении хранения различных структур в памяти, проведении трассировки приложения, исследовании характеристик программного обеспечения и оформлении результатов работы над задачей.

Задачи:

1. Обеспечить автоматизацию функционального тестирования.

a) Реализовать скрипты отладочной и релизной сборок.

b) Реализовать скрипты отладочной сборки с санитайзерами.

c) Реализовать скрипт очистки побочных файлов.

d) Реализовать компаратор для сравнения содержимого двух текстовых файлов, располагающегося после первого вхождения подстроки «Result:␣».

e) Реализовать скрипт pos_case.sh для проверки позитивного тестового случая по определённым далее правилам.

f) Реализовать скрипт neg_case.sh для проверки негативного тестового случая по определённым далее правилам.

2. Изучить процесс получения исполняемого файла и организации объектныхи исполняемых файлов.

a) Изучите этапы получения исполняемого файла на примере программы.
В отчёте приведите команду для выполнения каждого этапа, «выжимку» результатов выполнения этапа, краткое описание того,что происходит на этапе.

b) Выполните дизассемблирование полученного объектного файла.
Чем отличается результат дизассемблирования от полученной программы на языке ассемблера?

c) Добавьте отладочную информацию к объектному файлу. Что \
изменилось в объектном файле по сравнению с предыдущим пунктом?

3. Изучить процесс трассировки приложения.
a) Покажите, как в памяти представлены переменные типов char, int,unsigned, long long. Рассмотрите как положительные значения этих переменных, так и отрицательные.
b) Покажите, как в памяти представлен массив целых чисел.

Продемонстрируйте особенности выполнения операции сложения указателя с целым числом на примере массива.

c) Опишите указатели для работы с компонентами трехмерного массива.
Чему равен размер элемента соответствующего компонента?
Проведите теоретический расчёт. Результата проверьте с помощью gdb.

d) Рассчитайте суммарный размер памяти, который занимает каждаясимвольная структура данных. Каков размер «полезных» и «вспомогательных» данных?

e) Покажите дамп памяти, который содержит структуру. На дампе покажите расположение каждого поля структуры.

f) Упакуйте структуру. На дампе покажите расположение каждого
поля структуры.

g) Переставляя поля Вашей структуры, добейтесь, чтобы занимаемое структурой место стало минимальным. Упаковка для выполнения этого задания не используется.

4. Постановка замерного эксперимента.

a) Используя функцию nanosleep и каждый из четырёх известных Вам способов замеров времени (gettimeofday, clock_gettime, clock,__rdtsc), исследовать среднее значение времени выполнения вызовов функции nanosleep для задержки в 1с, 100мс, 50мс, 10мс.

b) На основе задачи №4 ЛР№2 (сортировка) по курсу «Программирование на Си» проведите сравнение производительности работы программы по трём плоскостям: разные способы работы с элементами одномерного массива, разные уровни оптимизации, разные исходные массивы.

c) Проведите сравнение производительности работы программы для умножения квадратных матриц по двум плоскостям: разные способы умножения матриц, разные уровни оптимизации.

ЗАДАЧИ

Задание а:

Описание алгоритма решения:

Мы начинаем процесс, компилируя исходный код с использованием компилятора gcc и опций, указанных ниже:

"-std=c99" - устанавливает стандарт языка C версии iso9899:1999
"-Wall" - включает все предупреждения компилятора
"-Werror" - преобразует все предупреждения в ошибки компиляции
"-Wpedantic" - обеспечивает строгое соблюдение стандарта языка
"-Wextra" - включает дополнительные предупреждения компилятора
"-Wfloat-equal" - предупреждает, если сравниваются числа с плавающей точкой
"-Wfloat-conversion" - предупреждает, если происходит потеря точности при преобразовании чисел с плавающей точкой
"-Wvla" - предупреждает, если используются массивы переменной длины
"-O3" (для сборки выпуска) - устанавливает максимальный уровень оптимизации
"-O0" (для сборки отладочной версии) - отключает оптимизацию
"-c" - компилирует в объектный файл
"-g3" (для сборки отладочной версии) - устанавливает максимальное количество отладочной информации

Затем мы выполняем линковку объектного файла с математической библиотекой и создаем исполняемый файл "app.exe".

Результаты работы программы:

main.o - Скомпилированный программный объект C
app.exe - Исполняемый файл

Code:

build_debug.sh

```
#!/bin/bash

gcc -c main.c -std=c99 -Wall -Werror -Wpedantic -Wextra -Wfloat-equal -Wfloat-conversion -Wvla -O0 -g3 -fprofile-arcs -ftest-coverage -lm
gcc main.o -o app.exe -lm -fprofile-arcs -ftest-coverage
```

build_release.sh

```
#!/bin/bash

gcc -c main.c -std=c99 -Wall -Werror -Wpedantic -Wextra -Wfloat-equal -Wfloat-conversion -Wvla -O3
gcc main.o -o app.exe -lm
```

Задание 2:

Описание алгоритма решения:

Для начала компилируем исходный код с использованием компилятора gcc и следующих опций:

"-std=c99" - используется стандарт iso9899:1999
"-Wall" - выводит все предупреждения
"-Werror" - рассматривает все предупреждения как ошибки
"-Wpedantic" - строгое соблюдение стандарта
"-Wextra" - дополнительные предупреждения
"-Wfloat-equal" - предупреждает при сравнении чисел с плавающей точкой
"-Wfloat-conversion" - предупреждает о потере точности чисел с плавающей точкой при преобразовании типов
"-Wvla" - предупреждает при использовании массивов переменной длины
"-O0" - отсутствие оптимизации для сборки отладочной версии
"-g3" - максимальное количество отладочной информации для сборки отладочной версии
"-fprofile-arcs" - включает поддержку сбора данных о покрытии кода для анализа выполнения тестов
"-ftest-coverage" - включает генерацию отчета о покрытии кода
"-fsanitize=address" - включает анализ адресной санитарии для обнаружения ошибок использования памяти

Затем мы линкуем объектный файл с математической библиотекой, указываем "app.exe" как выходной файл, и включаем флаги для поддержки сбора данных о покрытии кода и анализа адресной санитарии.

Результаты работы программы:

app.exe - Исполняемый файл

Code:

build_debug_asan.sh

```
#!/bin/bash

gcc -c main.c -std=c99 -Wall -Werror -Wpedantic -Wextra -Wfloat-equal -Wfloat-conversion -Wvla -O0 -g3 -fprofile-arcs -ftest-coverage -fsanitize=address
gcc main.o -o app.exe -lm -fprofile-arcs -ftest-coverage -fsanitize=address
```

Компилируем исходный код компилятором clang с ключами:

build_debug_msan.sh

```
#!/bin/bash

clang -c main.c -std=c99 -Wall -Werror -Wpedantic -Wextra -Wfloat-equal -Wfloat-conversion -Wvla -O0 -g3 -fsanitize=address,undefined -fprofile-arcs -ftest-coverage
clang main.o -o app.exe -lm -fsanitize=address,undefined -fprofile-arcs -ftest-coverage
```

Задание 3:

Описание алгоритма решения:

Данная команда используется для удаления файлов с расширениями .o, .exe, .gcno, .gcda и .gcov из текущей директории. Флаг -r используется для рекурсивного удаления всех поддиректорий и их содержимого, а флаг -f используется для безусловного удаления без запроса подтверждения. В данном случае команда выполняет удаление объектных файлов, исполняемых файлов и файлов, связанных с инструментами для измерения покрытия кода тестами.

Результаты работы программы:

Все сгенерированные файлы удалены

clean.sh

```
#!/bin/sh

rm -rf ./*.o ./*.exe ./*.gcno ./*.gcda ./*.gcov
```

Задание 4:

Описание алгоритма решения:
Этот скрипт на языке bash используется для сравнения содержимого двух файлов с различными параметрами. Скрипт принимает три аргумента: имя первого файла, имя второго файла и атрибут сравнения (тип данных).

Если атрибут сравнения установлен как "int", скрипт сравнивает только целочисленные значения, извлекая их из файлов с помощью команды grep и сравнивая с помощью команды diff. Если содержимое файлов идентично, скрипт завершается с кодом возврата 0, иначе - с кодом возврата 1.

Если атрибут сравнения не указан или не равен "int", скрипт сравнивает всё содержимое обоих файлов с помощью команды diff. Если содержимое файлов идентично, скрипт завершается с кодом возврата 0, иначе - с кодом возврата 1.

Code:

```
#!/bin/bash

file1="$1"
file2="$2"
attribute="$3"

if [ "$attribute" = "int" ]; then
  # Compare only integers
  if diff <(grep -Eo "[0-9]+" "$file1") <(grep -Eo "[0-9]+" "$file2") >/dev/null; then
    (exit 0)
  else
    (exit 1)
  fi
else
  # Compare entire file
  if diff "$file1" "$file2" >/dev/null; then
    (exit 0)
  else
    (exit 1)
  fi
fi
```

Задание 5:

Описание алгоритма решения:

1 - Проверяем количество аргументов. Если аргументов недостаточно, выводим ошибку и завершаем скрипт с кодом 2.

2 - Проверяем существование файлов, переданных в аргументах. Если хотя бы один файл не существует, выводим ошибку и завершаем скрипт с кодом 3.

3 - С помощью команды grep и регулярного выражения находим все числа в каждом файле. Чтобы избежать проблем с числами, разделенными пробелами, заменяем все пробелы на двойные пробелы с помощью команды sed.

4 - Удаляем пробелы в начале и конце каждой найденной строки с помощью двух команд sed.

5 - Сравниваем содержимое двух файлов с помощью команды cmp. Если файлы идентичны, выводим сообщение "Файлы идентичны" и завершаем скрипт с кодом 0. Если файлы различаются, выводим сообщение "Файлы отличаются" и завершаем скрипт с кодом

Code:

```
#!/bin/bash
if [ "$#" -lt 2 ]; then
  echo "Error: Insufficient arguments provided. Usage: $0 <file1> <file2>"
  exit 2
fi

if [ ! -f "$1" ] || [ ! -f "$2" ]; then
  echo "Error: File(s) not found"
  exit 3
fi

file1_nums=$(grep -Eo '[0-9]+' "$1" | sed 's/ /\ \ /g' | sed 's/^[ \t]*//;s/[ \t]*$//')
file2_nums=$(grep -Eo '[0-9]+' "$2" | sed 's/ /\ \ /g' | sed 's/^[ \t]*//;s/[ \t]*$//')
if [ "$3" = "int" ]; then
  if cmp <(echo "$file1_nums") <(echo "$file2_nums") >/dev/null; then
    echo "Files are identical"
    exit 0
  else
    echo "Files are different"
    exit 1
  fi
else
  # Compare entire files
  if cmp "$1" "$2" >/dev/null; then
    echo "Files are identical"
    exit 0
  else
    echo "Files are different"
    exit 1
  fi
fi
```

Задание 6:

Описание алгоритма решения:

Данный скрипт предназначен для проверки результатов выполнения программы. Вначале он проверяет наличие двух файлов - входного и выходного, переданных в качестве аргументов командной строки, а также проверяет наличие исполняемого файла "app.exe". Затем скрипт передает первый файл в качестве аргумента исполняемому файлу "main" и записывает вывод программы в файл "out.txt". После этого скрипт использует скрипт "comparator.sh" для сравнения содержимого файла "out.txt" с эталонным результатом, указанным в выходном файле для данного теста.

Если программа успешно завершается (код возврата равен 0) и содержимое файлов совпадает, скрипт выводит сообщение "PASSED" и завершается с кодом 0. В случае несовпадения содержимого файлов, скрипт выводит сообщение "FAILED", а код ошибки возвращается скриптом "comparator.sh".

Если один из аргументов не задан, скрипт завершается с кодом 2. Если файл(ы) не существует, скрипт завершается с кодом 3 или 4, в зависимости от наличия входного или выходного файла.

```
#!/bin/bash
../../main < "../data/pos_$1_in.txt" > "out.txt"
rc=$?

if ./comparator.sh out.txt ../data/pos_"$1"_out.txt; then
    echo -e POS_"$1": "\e[32mPASSED\e[0m"
else
    echo -e POS_"$1": "\e[31mFAILED\e[0m"
    ./comparator.sh out.txt ../data/pos_"$1"_out.txt
fi

#rm -f "out.txt"
```

Задание 7:

Описание алгоритма решения:

Также в этом случае скрипт используется для проверки отрицательных тестов. Он проверяет наличие двух файлов, переданных в качестве аргументов командной строки, а также наличие исполняемого файла "app.exe".

Сначала скрипт проверяет наличие двух файлов, переданных в качестве аргументов командной строки, и также проверяет наличие исполняемого файла "app.exe". Далее, первый файл передается в качестве аргумента исполняемому файлу, и вывод программы записывается в файл "out.txt". Затем скрипт сравнивает содержимое "out.txt" с эталонным результатом из отрицательного теста, используя команду diff, и получает код возврата.

Если код возврата diff равен 0, а код возврата выполнения программы не равен 0, то скрипт завершается с кодом 0 (тест считается пройденным). В противном случае скрипт завершается с кодом 1 (тест считается не пройденным).

Если один из аргументов не задан, скрипт завершается с кодом 2. Если файл(ы) не существует, скрипт завершается с кодом 3 или 4. После выполнения теста временный файл "out.txt" удаляется.

```
#!/bin/bash
../../main < "../data/neg_$1_in.txt" > "out.txt"
rc=$?
{
diff out.txt ../data/neg_"$1"_out.txt
rc_s=$?
} &> /dev/null

if [ $rc_s = 0 ] && [ $rc != 0 ]
then
    echo -e NEG_"$1": "\e[32mPASSED\e[0m"
else
    diff out.txt ../data/neg_"$1"_out.txt
   echo -e NEG_"$1": "\e[31mFAILED\e[0m"
fi
rm -f "out.txt"
```

Задание 8:

Описание алгоритма решения:

Этот скрипт автоматизирует процесс тестирования программы на основе набора тестовых данных, хранящихся в файлах. Вначале скрипт проверяет наличие 100 пар файлов для тестирования. Эти файлы должны соответствовать формату "pos_{номер}in.txt" и "pos_{номер}out.txt", где {номер} - числовой идентификатор тестового набора. Если такие файлы существуют, скрипт запускает скрипт pos_case.sh для каждого тестового набора, передавая ему идентификатор теста.

Аналогично, скрипт проверяет наличие 100 пар файлов "neg_{номер}in.txt" и "neg_{номер}out.txt", где {номер} - числовой идентификатор тестового набора. Если эти файлы существуют, скрипт запускает скрипт neg_case.sh для каждого набора тестов.

Таким образом, скрипт выполняет запуск скриптов pos_case.sh и neg_case.sh для соответствующих тестовых наборов, предоставляя им необходимые данные для тестирования программы.

```
#!/bin/bash
for ((i=1;i<100;i++))
do
    num=$(printf "%02d" $i)
    if [ -e "../data/pos_${num}_in.txt" ] && [ -e "../data/pos_${num}_out.txt" ]
    then
        ./pos_case.sh $num
    else
        break
    fi
done

echo

for ((i=1;i<100;i++))
do
    num=$(printf "%02d" $i)
```

```
    if [ -e "../data/neg_${num}_in.txt" ] && [ -e "../data/neg_${num}_out.txt" ]
    then
        ./neg_case.sh $num
    else
        break
    fi
done
```

Задача 2

Задание а

1. Препроцессорная обработка:

В этом этапе происходят следующие операции:

- Удаление комментариев

- Вставка содержимого из других файлов

- Выполнение макроподстановок

- Условная компиляция

Comand:

cpp main.c > app.i

Результатом этого шага является файл промежуточного кода (единицы трансляции) под названием main.i.

2. Трансляция кода на языке C в язык ассемблера:

Шаг трансляции кода на языке C в язык ассемблера включает следующие действия:

- Компиляция кода на языке C с использованием опций `-S`, `-fverbose-asm` и `-masm=intel`.

- Команда:

gcc -std=c99 -S -fverbose-asm -masm=intel -o app.s app.i

- Результатом выполнения этого шага является файл с исходным кодом на языке ассемблера, который называется app.s.

3. Ассемблирование файла app.s в объектный файл:

Для ассемблирования файла app.s в объектный файл выполняется следующая команда:

as -o app.o app.s

4. Компоновка объектного файла (app.o) в исполняемый файл:

На этом этапе выполняются следующие действия:

- Связывание переменных и функций, которые отсутствуют в объектном файле.

- Вставка специального кода, который создает окружение для вызова функции main и выполняет необходимые операции после её завершения.

Comands:

ld --entry main --dynamic-linker /lib64/ld-linux-x86-64.so.2 -o app.exe app.o -lc -lm

gcc -o app.exe app.o

Задание b:

objdump -S --disassemble app.exe > app.dump

Дисассемблированный файл отличается от исходного файла тем, что он не содержит имен переменных и строк.

Задание c:

После добавления отладочной информации к объектному файлу произошли изменения в его размере. В результате добавления были включены пути заголовочных файлов и макросы.

Задача 3

Задание a:

Массив:
int a[5] = {123, 100, 324, 5, 4};
Представление массива в памяти:
0000007b 00000064 00000144 00000005 00000004
Элементы массива хранятся последовательно.

Задание b:

Доступ к элементу i матрицы j строки k столбца:
a[i][j][k] === *((*(a + i) + j) + k)
Размер компонента равен размеру типа компонента.
Размер arr[0][0][0] равен размеру int (4 байта на моей машине).
Размер arr[0][0] равен sizeof(int) * 4 = 4 * 4 = 16 байт (на моей машине).
Размер arr[0] равен 16 * 3 = 48 байт (на моей машине).
Размер arr равен 48 * 2 = 96 байт (на моей машине).
Вычисления с помощью gdb:
(gdb) print(sizeof(arr))
$8 = 96
(gdb) print(sizeof(arr[0]))
$9 = 48
(gdb) print(sizeof(arr[0][0]))
$10 = 16 (gdb) print(sizeof(arr[0][0][0]))
$11 = 4

Задание c:

Различные примеры массивов и их размеры в байтах:

1. Массив символов:

char str_arr[] = "Hello, World!";

Размер в байтах равен длине строки. В данном случае размер равен 14.
sizeof(str_arr) // 14

Различные примеры массивов и их размеры в байтах:

2. Указатель на строковый литерал:

char *str_ptr = "Hello";
Размер в байтах равен размеру указателя. На моей машине размер указателя составляет 8 байт.

sizeof(str_ptr) // 8

3. Двумерный массив строк:

char arr_1[][9] = {"January", "February", "March"};
Размер в байтах определяется путем умножения размера строки на количество строк. В данном случае размер равен 9 * 3 = 27.

sizeof(arr_1) // 27

4. Массив указателей на строки:

char *arr_2[] = {"January", "February", "March"};

Размер в байтах определяется умножением размера указателя на количество строк. В данном случае размер равен 8 * 3 = 24.

sizeof(arr_2) // 24

В дополнение к этому, размер "дополнительных" данных, таких как нулевой символ ('\0') в конце каждой строки, равен 1 байту.

Задание d:

struct User
{
int age;
char something;
long long height;
} test;
(gdb) print sizeof(test)
$1 = 16
(gdb) print &test
$2 = (struct User *) 0x7fffffffded0
(gdb) x /16bx 0x7fffffffded0
0x7fffffffded0: 0x01 0x00 0x00 0x00 0x02 0x00 0x00 0x00
0x7fffffffded8: 0x03 0x00 0x00 0x00 0x00 0x00 0x00 0x00

Название поля	Тип поля	Размер типа	Размер в структуре	Адрес	Значение адреса
age	int	4	4	0x7fffffffded0	0x01 0x00 0x00 0x00
something	char	1	4	0x7fffffffded4	0x02 0x00 0x00 0x00
height	Long long	8	8	0x7fffffffded8	0x03 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x0

Поле something типа char было выровнено, и теперь размер этого типа в структуре составляет 4 байта, в то время как размер char по-прежнему равен 1 байту.

Задание e:

Упакованная структура:
(gdb) print sizeof(test)
$3 = 13
(gdb) print & test
$4 = (struct User *) 0x7fffffffded3
(gdb) x /13bx 0x7fffffffded3
0x7fffffffded3: 0x01 0x00 0x00 0x00 0x02
0x7fffffffdedb: 0x00 0x00 0x00 0x00 0x00
0x03
0x00
0x00

В этом случае, размер структуры соответствует сумме размеров типов полей структуры.
Задание g
struct User
{
char something;
int age;
char something3;
int marks_avg;
char something2;
} test;
(gdb) print sizeof(test)
$1 = 20
struct User2
{
int age;
int marks_avg;
char something2;
char something3;
char something;
} test;
(gdb) print sizeof(test)
$1 = 12
Структура имеет выравнивание по границе. В оптимальной версии структуры "user" выравнивание составляет 1 байт.

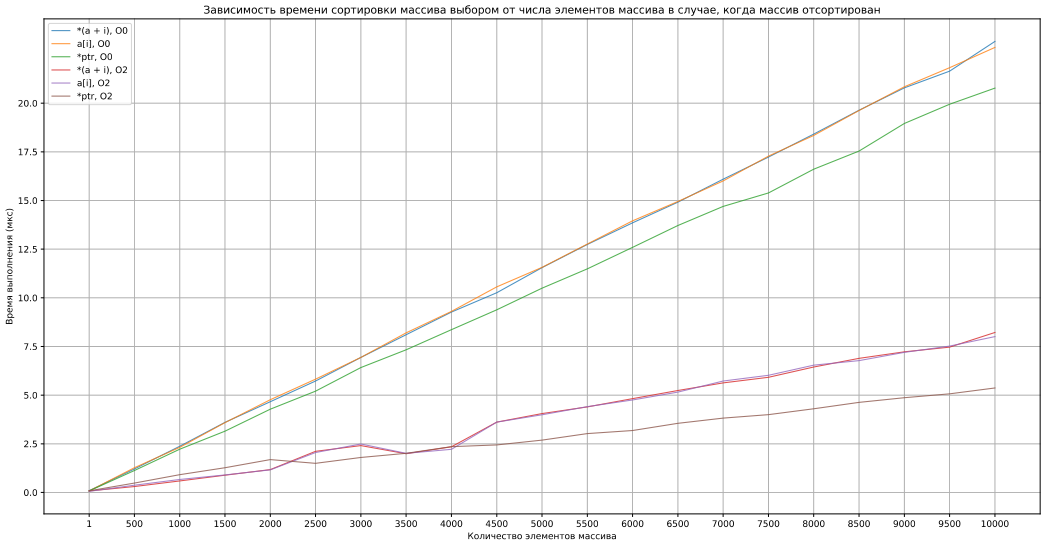
Задача 4

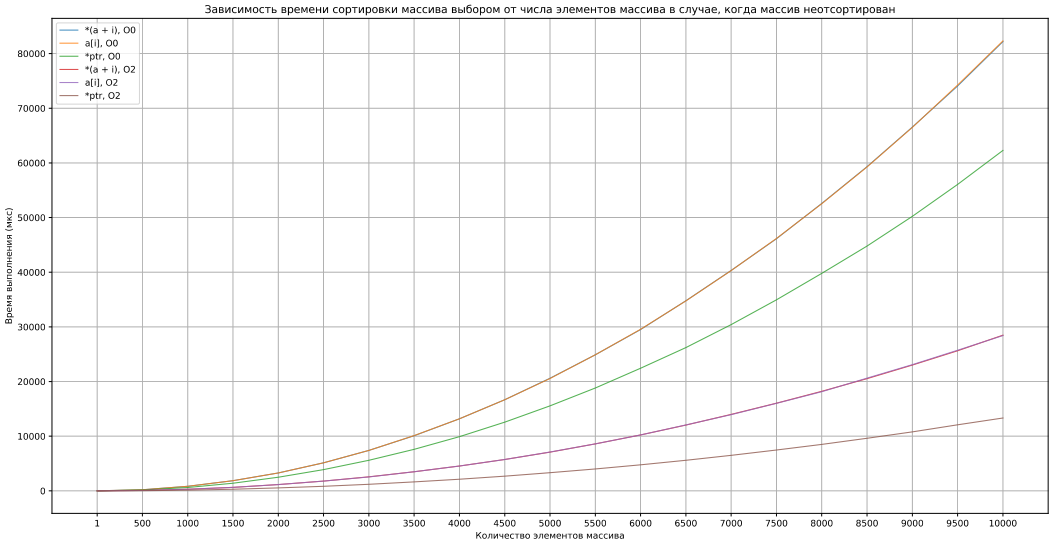
Задание a:

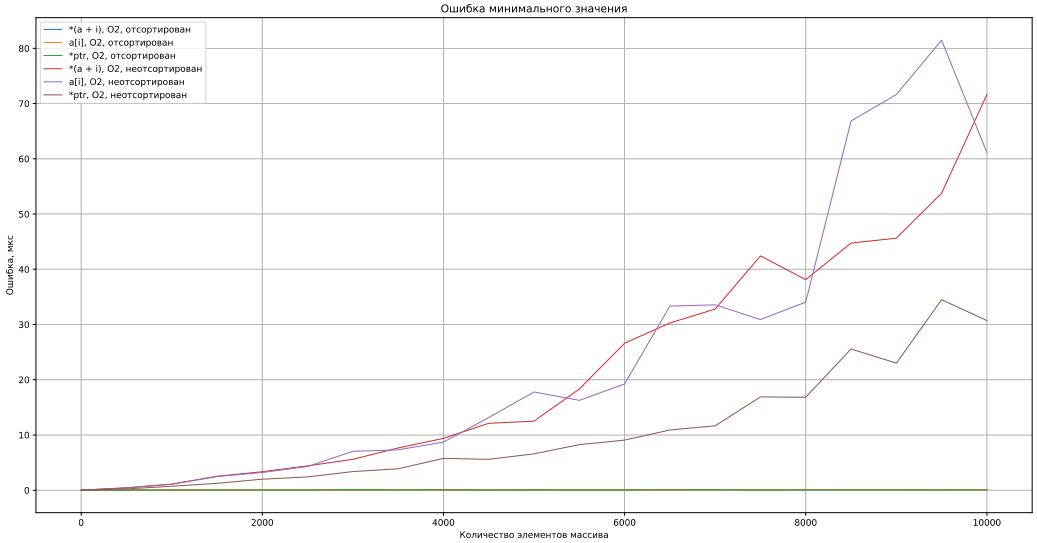
	Среднее время, мс			
nanosleep, мс	gettimeofday	clock_gettime	clock	__rdtsc
1	1	1	3	2
10	10.00	10.00	4.80	21.00
50	50.20	50.20	8.60	105.80
100	100.06	100.11	6.29	211.39

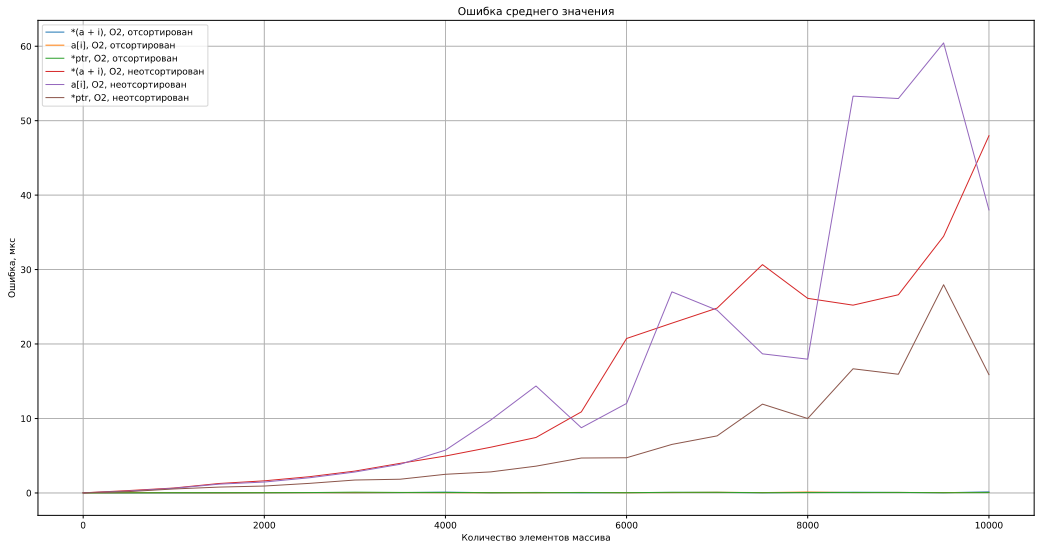
Задание b:

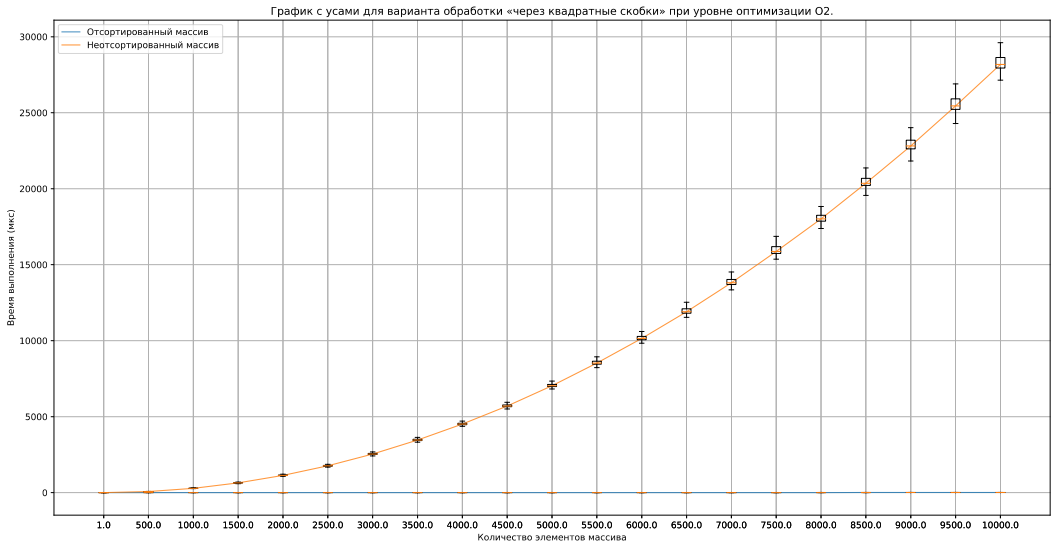
Зависимость размера случайно сгенерированного массива и времени сортировки для различных методов доступа к элементам массива представлена в графике.




Зависимость времени сортировки массива выбором от числа элементов массива в случае, когда массив неотсортирован
*(a + i), O0
a[i], O0
*ptr, O0
*(a + i), O2
a[i], O2
*ptr, O2
Время выполнения (мкс)
Количество элементов массива


Ошибка минимального значения
*(a + i), O2, отсортирован
a[i], O2, отсортирован
*ptr, O2, отсортирован
*(a + i), O2, неотсортирован
a[i], O2, неотсортирован
*ptr, O2, неотсортирован
Ошибка, мкс
Количество элементов массива





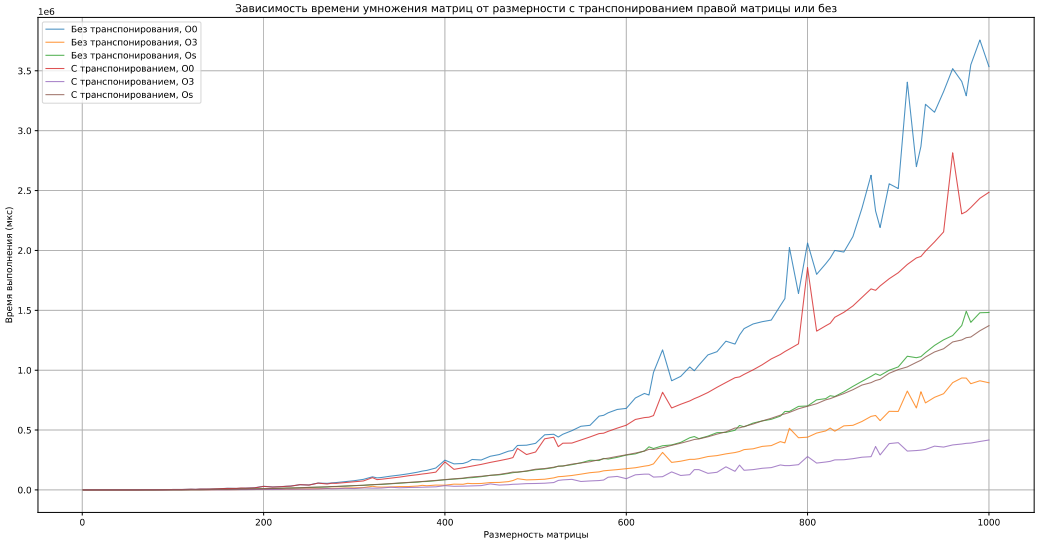
Вывод:

Методы обработки элементов массива, основанные на использовании указателей, имеют более высокую эффективность при работе с произвольными данными. Это связано с тем, что операции с указателями позволяют перемещаться по данным в памяти с меньшими затратами времени, поскольку требуется лишь передвинуть указатель на следующий элемент, а не копировать его значение.

Однако, при обработке упорядоченных элементов предпочтительнее использовать метод обращения к элементам по индексам. В таком случае доступ к значению переменной осуществляется непосредственно по ее индексу, что может быть более эффективным в сравнении с поиском адреса переменной по указателю и последующим разыменованием. Операции сравнения значений переменных при этом могут требовать больше времени, но доступ к элементам массива по индексу компенсирует этот недостаток, упрощая процесс обработки упорядоченных данных.

Задание c:

Зависимость размера случайно сгенерированных матриц и времени умножения матриц для различных методов умножения представлена на графике.



Результат:

При умножении матриц с использованием операции транспонирования, достигается повышение эффективности работы. Это объясняется тем, что при доступе к элементам массива, процессор загружает в кэш также соседние элементы, что способствует ускоренному доступу к ним. Такое ускорение компенсирует затраты, связанные с выполнением операции транспонирования матрицы.

ЗАКЛЮЧЕНИЕ

В процессе проведения моих исследований я приобрел набор ценных навыков, которые включают последовательное создание исполняемых файлов, дизассемблирование исполняемых файлов, работу с отладчиком gdb, анализ объектов в памяти и интерпретацию полученных данных, измерение скорости выполнения программ, обработку результатов, построение графиков с использованием gnuplot, а также представление практических результатов в удобном формате. В результате успешно достигнута поставленная цель работы, и все задачи выполнены в полном объеме.