Университет ИТМО
Факультет проргаммной инженерии и компьютерной техники

Лабораторная работа №1
по диспицлине
"Операционные системы"

Выполнила: **Завацкая Дарьвя Вадимовна**
Группа: **P3334**
Вариант: **io-lat-write, dedup, 1K**
Преподаватель: **Смирнов Виктор Игоревич**

г. Санкт-Петербург, 2025

1 Задание

1.1 Часть 1. Запуск программ

Необходимо реализовать собственную оболочку командной строки — shell. Выбор ОС для реализации производится на усмотрение студента. Shell должен предоставлять пользователю возможность запускать программы на компьютере с переданными аргументами командной строки и после завершения программы показывать реальное время её работы (подсчитать самостоятельно как «время завершения» – «время запуска»).

1.2 Часть 2. Мониторинг и профилирование

Разработать комплекс программ-нагрузчиков по варианту, заданному преподавателем. Каждый нагрузчик должен, как минимум, принимать параметр, который определяет количество повторений для алгоритма, указанного в задании. Программы должны нагружать вычислительную систему, дисковую подсистему или обе подсистемы сразу. Необходимо скомпилировать их без опций оптимизации компилятора.

Перед запуском нагрузчика попробуйте оценить время работы вашей программы или её результаты (если по варианту вам досталось измерение чего-либо). Постарайтесь обосновать свои предположения, основываясь на свой опыт, знания ОС и характеристики используемого аппаратного обеспечения.

1. Запустите программу-нагрузчик и зафиксируйте метрики её работы с помощью инструментов для профилирования. Сравните полученные результаты с ожидаемыми. Постарайтесь найти объяснение наблюдаемому.

2. Определите количество нагрузчиков, которое эффективно нагружает все ядра процессора на вашей системе. Как распределяются времена USER%, SYS%, WAIT%, а также реальное время выполнения нагрузчика? Какое количество переключений контекста (вынужденных и невынужденных) происходит при этом?

3. Увеличьте количество нагрузчиков вдвое, втрое, вчетверо. Как изменились времена, указанные на предыдущем шаге? Как ведёт себя ваша система?

4. Объедините программы-нагрузчики в одну, реализованную при помощи потоков выполнения, чтобы один нагрузчик эффективно нагружал все ядра вашей системы. Как изменились времена для того же объёма вычислений? Запустите одну, две, три таких программы.

5. Добавьте опции агрессивной оптимизации для компилятора. Как изменились времена? На сколько сократилось реальное время исполнения программы-нагрузчика?

1.3 Ограничения

Программа (комплекс программ) должна быть реализована на языке C, C++. Дочерние процессы должны быть созданы через заданные системные вызовы выбранной операционной системы с обеспечением корректного запуска и завершения процессов. Запрещено использовать высокоуровневые абстракции над системными вызовами. Необходимо использовать, в случае Unix, процедуры libc.

2 Листинг кода

2.1 Основные программы

Листинг 1: Main

```
#include "../include/shell.h"

int main() {
    ExecuteShell();
    return 0;
}
```

Листинг 2: Shell

```
#include "../include/shell.h"

// Функция для разбиения введенной строки на команду и аргументы
std::vector<std::string_view> splitCommand(const std::string& input) {
    std::vector<std::string_view> parts;
    size_t pos = 0, found;
    while ((found = input.find_first_of(' ', pos)) != std::string::npos) {
        if (found > pos) {
            parts.push_back(std::string_view(input.data() + pos, found - pos));
        }
        pos = found + 1;
    }
    if (pos < input.length()) parts.push_back(std::string_view(input.data() + pos));
    return parts;
}
// Функция для запуска процесса с заданной командой и аргументами
bool launchProcess(const std::string& command, const std::vector<std::string_view>& args) {
    std::string fullCommand = "\"" + command + "\"";
    for (const auto& arg : args) {
        fullCommand += " \"" + std::string(arg) + "\"";
    }

    HANDLE hToken;
    // Открываем текущий процесс для получения его токена доступа
```

```
25      if (!OpenProcessToken(
26              /* ProcessHandle = */ GetCurrentProcess(),
27              /* DesiredAccess = */ TOKEN_DUPLICATE |
                    TOKEN_ASSIGN_PRIMARY | TOKEN_QUERY,
28              /* TokenHandle = */ &hToken)) {
29          std::cerr << "Failed to open process token. Error code: "
                << GetLastError() << std::endl;
30          return false;
31      }
32
33      HANDLE hNewToken;
34      // Дублируем токен для нового процесса
35      if (!DuplicateTokenEx(
36              /* hExistingToken = */ hToken,
37              /* dwDesiredAccess = */ MAXIMUM_ALLOWED,
38              /* lpTokenAttributes = */ nullptr,
39              /* ImpersonationLevel = */ SecurityImpersonation,
40              /* TokenType = */TokenPrimary,
41              /* phNewToken = */ &hNewToken)) {
42          std::cerr << "Failed to duplicate token. Error code: " <<
                GetLastError() << std::endl;
43          CloseHandle(hToken);
44          return false;
45      }
46      CloseHandle(hToken);
47
48      STARTUPINFO si = {sizeof(STARTUPINFO)};
49      PROCESS_INFORMATION pi = {};
50      auto startTime = std::chrono::high_resolution_clock::now();
51
52      // Создаем новый процесс с использованием дублированного токена
53      if (!CreateProcessAsUserA(
54              /* hToken = */ hNewToken,
55              /* lpApplicationName = */ nullptr,
56              /* lpCommandLine = */ fullCommand.data(),
57              /* lpProcessAttributes = */ nullptr,
58              /* lpThreadAttributes = */ nullptr,
59              /* bInheritHandles = */ FALSE,
60              /* dwCreationFlags = */ 0,
61              /* lpEnvironment = */ nullptr,
62              /* lpCurrentDirectory = */ nullptr,
63              /* lpStartupInfo = */ &si,
64              /* lpProcessInformation = */ &pi)) {
65          std::cerr << "Failed to start process. Error code: " <<
                GetLastError() << std::endl;
66          CloseHandle(hNewToken);
67          return false;
68      }
69
70      // Завершения процесса
71      WaitForSingleObject(pi.hProcess, INFINITE);
72      auto endTime = std::chrono::high_resolution_clock::now();
73      auto executionTime = std::chrono::duration_cast<std::chrono::
            milliseconds>(endTime - startTime).count();
74      std::cout << "Execution time: " << executionTime << " ms" <<
            std::endl;
```

```
75
76      CloseHandle(pi.hProcess);
77      CloseHandle(pi.hThread);
78      CloseHandle(hNewToken);
79
80      return true;
81  }
82
83  // Основная функция оболочки
84  void ExecuteShell() {
85      std::string input;
86      std::cout << "Welcome to the new shell! Type 'exit' to exit the
            shell\n";
87      while (true) {
88          std::cout << "shell> ";
89          getline(std::cin, input);
90
91          auto parts = splitCommand(input);
92          if (parts.empty()) continue;
93
94          std::string_view command = parts[0];
95          std::vector<std::string_view> args(parts.begin() + 1, parts
                .end());
96
97          if (command == "exit") {
98              std::cout << "Exiting shell...\n";
99              break;
100         }
101         if (command == "info") {
102             std::cout << "dedup <inputFile> <outputFile>\nio-lat-
                    write <outputFile> <number of iterations>\n" << std
                    ::endl;
103             continue;
104         } else {
105             if (!launchProcess(std::string(command), args)) {
106                 std::cerr << "Error executing command: " << command
                        << std::endl;
107             }
108         }
109     }
110 }
```

Листинг 3: Dedup

```
1  #include <sstream>
2  #include "../include/dedup.h"
3
4
5  // Функция для чтения данных
6  std::vector<int> readDataFromFile(const std::string& filename) {
7      std::vector<int> data;
8      HANDLE file = CreateFileA(filename.c_str(),
9                                GENERIC_READ,
10                               FILE_SHARE_READ,
11                               NULL,
12                               OPEN_EXISTING,
```

```
                                FILE_ATTRIBUTE_NORMAL,
                                NULL);
    if (file == INVALID_HANDLE_VALUE) {
        std::cerr << "error opening file: " << filename << std::
            endl;
        return data;
    }

    char buffer[1024];
    DWORD bytesRead;
    std::string content;
    while (ReadFile(file,
                    buffer,
                    sizeof (buffer),
                    &bytesRead,
                    NULL)
           && bytesRead > 0) {
        content.append(buffer, bytesRead);
    }
    CloseHandle(file);

    std::size_t pos = 0;
    while (pos < content.size()) {
        try {
            std::size_t newPos;
            int value = stoi(content.substr(pos), &newPos);
            data.push_back(value);
            pos += newPos;
        } catch (const std::exception&) {
            break;
        }
    }

    return data;
}

// Функция для записи данных в файл
void writeDataToFile(const std::string& filename, const std::vector
    <int>& data) {
    HANDLE file = CreateFileA(filename.c_str(),
                                GENERIC_WRITE,
                                0,
                                NULL,
                                 CREATE_ALWAYS,
                                FILE_ATTRIBUTE_NORMAL,
                                NULL);
    if (file == INVALID_HANDLE_VALUE) {
        std::cerr << "error opening file: " << filename << std::
            endl;
        return;
    }

    std::string buffer;
    for (const int& value : data) buffer += std::to_string(value) +
        " ";

```

```
65      DWORD bytesWritten;
66      WriteFile(file,
67                buffer.c_str(),
68                buffer.size(),
69                &bytesWritten,
70                NULL);
71      if (!FlushFileBuffers(file)) {
72          std::cerr << "Error flushing buffers" << std::endl;
73      }
74      CloseHandle(file);
75  }
76
77  // Функция для удаления дубликатов из данных в файле
78  void dedup(const std::string& inputFile, const std::string&
        outputFile) {
79      std::vector<int> arr = readDataFromFile(inputFile);
80      if (arr.empty()) {
81          std::cerr << "no data to deduplicate" << std::endl;
82          return;
83      }
84      // Используем множество для удаления дубликатов
85      std::unordered_set<int> unique_elements(arr.begin(), arr.end())
            ;
86      std::vector<int> result(unique_elements.begin(),
            unique_elements.end());
87      writeDataToFile(outputFile, result);
88      std::cout << "deduplication complete. new file is: " <<
            outputFile << std::endl;
89  }
90
91  #ifndef TESTING
92  int main(int argc, char* argv[]) {
93      if (argc != 3) {
94          std::cerr << "Usage: dedup <inputFile> <outputFile>" << std
                ::endl;
95          return 1;
96      }
97      dedup(argv[1], argv[2]);
98      return 0;
99  }
100 #endif
```

Листинг 4: IO-lat-write

```
1   #include "../include/io_lat_write.h"
2
3   void IOLatWrite(int iterations, const std::string& filePath) {
4       const int block_size = 1024; // размер блока 1K
5       std::vector<char> data(block_size, 'A');
6
7       HANDLE file = CreateFileA(filePath.c_str(),
8                                 GENERIC_WRITE,
9                                 0,
10                                NULL,
11                                CREATE_ALWAYS,
12                                FILE_ATTRIBUTE_NORMAL,
```

```
13                                  NULL);
14      if (file == INVALID_HANDLE_VALUE) {
15          std::cerr << "error opening file: " << filePath<< std::endl
                ;
16          return;
17      }
18
19      // Генератор случайных чисел для выбора случайных смещений в файле
20      std::random_device rd;
21      std::mt19937 gen(rd());
22      // Распределение для выбора смещения от 0 до 1023
23      std::uniform_int_distribution<> dis(0, 1023);
24
25      auto start_time = std::chrono::high_resolution_clock::now();
26
27      // Цикл записи данных в файл заданное количество раз
28      for (int i = 0; i < iterations; ++i) {
29          int offset = dis(gen); // Генерируем случайное смещение
30          // Устанавливаем указатель записи в файл на сгенерированное смещение
31          if (SetFilePointer(file, offset, NULL, FILE_BEGIN) ==
                INVALID_SET_FILE_POINTER) {
32              std::cerr << "Error seeking in file " << std::endl;
33              CloseHandle(file);
34              return;
35          }
36
37          DWORD bytesWritten;
38          if (!WriteFile(file, data.data(), block_size, &bytesWritten
                , NULL) || bytesWritten != block_size) {
39              std::cerr << "Error writing to file" << std::endl;
40              return;
41          }
42          if (!FlushFileBuffers(file)) {
43              std::cerr << "Error flushing buffers" << std::endl;
44          }
45      }
46      auto end_time = std::chrono::high_resolution_clock::now();
47      auto elapsed_time = std::chrono::duration_cast<std::chrono::
            nanoseconds>(end_time - start_time).count();
48      std::cout << "average write time per iteration: " <<
            elapsed_time / iterations << " [ns]" << std::endl;
49
50      CloseHandle(file);
51  }
52  #ifndef TESTING
53  int main(int argc, char* argv[]) {
54      if (argc != 3) {
55          std::cerr << "Usage: io-lat-write <outputFile> <number of
                iterations>" << std::endl;
56          return 1;
57      }
58      int iterations = std::stoi(argv[2]);
59      IOLatWrite(iterations, argv[1]);
60      return 0;
61  }
62  #endif
```

2.2 Тесты

Листинг 5: Shell test

```
#include "../../include/shell.h"
#include <iostream>
#include <sstream>
#include <gtest/gtest.h>

// Функция для имитации работы shell
std::string simulateShellInput(const std::string& input) {
    std::istringstream inputStream(input);
    std::ostringstream outputStream;
    std::streambuf* cinBackup = std::cin.rdbuf();
    std::streambuf* coutBackup = std::cout.rdbuf();

    std::cin.rdbuf(inputStream.rdbuf());
    std::cout.rdbuf(outputStream.rdbuf());

    ExecuteShell();

    std::cin.rdbuf(cinBackup);
    std::cout.rdbuf(coutBackup);

    return outputStream.str();
}

// Тест команды exit
TEST(ShellTest, ExitCommand) {
    std::string output = simulateShellInput("exit\n");
    EXPECT_NE(output.find("Exiting shell..."), std::string::npos);
}

// Тест команды info
TEST(ShellTest, InfoCommand) {
    std::string output = simulateShellInput("info\nexit\n");
    EXPECT_NE(output.find("dedup <inputFile> <outputFile>"), std::
        string::npos);
    EXPECT_NE(output.find("io-lat-write <outputFile> <number of
        iterations>"), std::string::npos);
}

int main(int argc, char **argv) {
    ::testing::InitGoogleTest(&argc, argv);
    return RUN_ALL_TESTS();
}
```

Листинг 6: Dedup test

```
#include <gtest/gtest.h>
#include <fstream>
#include "../../include/dedup.h"


// Вспомогательная функция для чтения данных из файла
std::vector<int> readFileData(const std::string& filename) {
```

```
    std::ifstream file(filename);
    if (!file) {
        ADD_FAILURE() << "error opening file " << filename;
        return {};
    }
    std::vector<int> data;
    int value;
    while (file >> value) {
        data.push_back(value);
    }
    return data;
}

// Тест команды dedup
TEST(DedupTest, RemovesDuplicates) {
    std::string inputFile = "test_input.txt";
    std::string outputFile = "test_output.txt";

    // Заполняем файл с дублирующимися данными
    std::ofstream inFile(inputFile);
    if (!inFile) {
        FAIL() << "File creation error " << inputFile;
    }
    inFile << "1 2 2 3 4 4 5 5 6 6 6 7 8 8 9 9 10 10";
    inFile.close();

    std::ofstream outFile(outputFile);
    if (!outFile) {
        FAIL() << "File creation error " << outputFile;
    }
    outFile.close();

    // Запускаем dedup
    dedup(inputFile, outputFile);

    // Получаем данные из файла после выполнения команды
    std::vector<int> result = readFileData(outputFile);
    std::vector<int> expected = {10, 9, 8, 7, 6, 5, 4, 3, 2, 1};

    // Сравниваем полученный результат с ожидаемым
    EXPECT_EQ(result, expected);
}
```

Листинг 7: IO-lat-write test

```
#include <gtest/gtest.h>
#include <fstream>
#include "../../include/io_lat_write.h"

// Вспомогательная функция для проверки существования файла и его размера
bool isFileValid(const std::string& filename) {
    std::ifstream file(filename, std::ios::binary | std::ios::ate);
    if (!file.is_open()) return false;
    return file.tellg() > 0;
}

```

```
// Тест команды io-lat-write
TEST(IOLatWriteTest, CheckFile) {
    std::string outputFile = "test_io.txt";
    int iterations = 1000;

    // Запускаем io-lat-write
    IOLatWrite(iterations, outputFile);

    // Проверяем, что файл был создан и содержит данные
    EXPECT_TRUE(isFileValid(outputFile));
}
```

2.3 Доп. программы для 2-ой части

Листинг 8: Многопоточная программа Dedup

```
#include <iostream>
#include <vector>
#include <unordered_set>
#include <fstream>
#include <sstream>
#include <thread>
#include <mutex>
#include <Windows.h>

// Функция для чтения данных
std::vector<int> readDataFromFile(const std::string& filename) {
    std::vector<int> data;
    HANDLE file = CreateFileA(filename.c_str(),
                              GENERIC_READ,
                              FILE_SHARE_READ,
                              NULL,
                              OPEN_EXISTING,
                              FILE_ATTRIBUTE_NORMAL,
                              NULL);
    if (file == INVALID_HANDLE_VALUE) {
        std::cerr << "Error opening file: " << filename << std::
            endl;
        return data;
    }

    char buffer[1024];
    DWORD bytesRead;
    std::string content;
    while (ReadFile(file, buffer, sizeof(buffer), &bytesRead, NULL)
        && bytesRead > 0) {
        content.append(buffer, bytesRead);
    }
    CloseHandle(file);

    std::size_t pos = 0;
    while (pos < content.size()) {
        try {
            std::size_t newPos;
            int value = stoi(content.substr(pos), &newPos);
            data.push_back(value);
```

```
            pos += newPos;
        } catch (const std::exception&) {
            break;
        }
    }

    return data;
}

// Функция для записи данных в файл
void writeDataToFile(const std::string& filename, const std::vector<int>& data) {
    HANDLE file = CreateFileA(filename.c_str(),
                              GENERIC_WRITE,
                              0,
                              NULL,
                              CREATE_ALWAYS,
                              FILE_ATTRIBUTE_NORMAL,
                              NULL);
    if (file == INVALID_HANDLE_VALUE) {
        std::cerr << "Error opening file: " << filename << std::endl;
        return;
    }

    std::string buffer;
    for (const int& value : data) {
        buffer += std::to_string(value) + " ";
    }

    DWORD bytesWritten;
    WriteFile(file, buffer.c_str(), buffer.size(), &bytesWritten, NULL);
    CloseHandle(file);
}

// Функция для удаления дубликатов в отдельной части массива
void dedupWorker(const std::vector<int>& arr, std::unordered_set<int>& unique_elements, std::mutex& mtx) {
    std::unordered_set<int> localSet(arr.begin(), arr.end());

    std::lock_guard<std::mutex> lock(mtx);
    unique_elements.insert(localSet.begin(), localSet.end());
}

void dedup(const std::string& inputFile, const std::string& outputFile, int threadCount) {
    std::vector<int> arr = readDataFromFile(inputFile);
    if (arr.empty()) {
        std::cerr << "No data to deduplicate" << std::endl;
        return;
    }

    size_t chunkSize = arr.size() / threadCount;
    std::vector<std::thread> threads;
    std::unordered_set<int> unique_elements;
```

```
90      std::mutex mtx;
91
92      for (int i = 0; i < threadCount; ++i) {
93          size_t startIdx = i * chunkSize;
94          size_t endIdx = (i == threadCount - 1) ? arr.size() :
               startIdx + chunkSize;
95          threads.emplace_back(dedupWorker, std::vector<int>(arr.
               begin() + startIdx, arr.begin() + endIdx), std::ref(
               unique_elements), std::ref(mtx));
96      }
97
98      for (auto& t : threads) {
99          t.join();
100     }
101
102     std::vector<int> result(unique_elements.begin(),
           unique_elements.end());
103     writeDataToFile(outputFile, result);
104     std::cout << "Deduplication complete. New file is: " <<
           outputFile << std::endl;
105 }
106
107 #ifndef TESTING
108 int main(int argc, char* argv[]) {
109     if (argc != 4) {
110         std::cerr << "Usage: dedup <inputFile> <outputFile> <
               threads>" << std::endl;
111         return 1;
112     }
113
114     int threadCount = std::stoi(argv[3]);
115     dedup(argv[1], argv[2], threadCount);
116
117     return 0;
118 }
119 #endif
```

3 Предположения о свойствах программ-нагрузчиков

dedup

- **Нагрузка на процессор (USER%):** Средняя. Основная часть вычислений приходится на разбор входного файла (`stoi()`) и удаление дубликатов (`unordered_set`).
- **Нагрузка на диск (SYS%):** Средняя. Чтение и запись данных через `CreateFileA`, `ReadFile`, `WriteFile` создают нагрузку на файловую систему. Однако запись происходит одним большим буфером, а не мелкими операциями, что снижает влияние на SYS%.
- **Ожидание операций ввода-вывода (WAIT%):** Низкая — Средняя. Если файл большой, возможны задержки на чтение, но большая часть работы — это вычисления в оперативной памяти.

Оценка времени выполнения: Время выполнения будет зависеть от скорости чтения файла, удаления дубликатов и записи обратно. Для файла размером 727 КБ, скорее всего, это займет 3000–5000 мс (3-5 секунд).

io-lat-write

- **Нагрузка на процессор (USER%):** Низкая. CPU практически не загружен, так как основная работа — это системные вызовы записи.
- **Нагрузка на диск (SYS%):** Высокая. Частые `WriteFile` с `FlushFileBuffers` вызывают принудительную запись на диск без кеширования.
- **Ожидание операций ввода-вывода (WAIT%):** Высокая. Так как запись идет в случайные места файла.

Оценка времени выполнения: Каждая итерация записи будет занимать некоторое время из-за случайных смещений в файле и операций ввода-вывода. Время выполнения для 1000 итераций может составить от 2000 мс до 10000 мс (2-10 секунд), в зависимости от скорости диска и эффективности операционной системы.

4 Результаты измерений и метрик

4.1 Dedup

Запуск 1 программы-нагрузчика

```
shell> dedup "D:\dasha\UNI\OS\in1.txt" "D:\dasha\UNI\OS\out.txt"
deduplication complete. new file is: D:\dasha\UNI\OS\out.txt
Execution time: 6905 ms
```

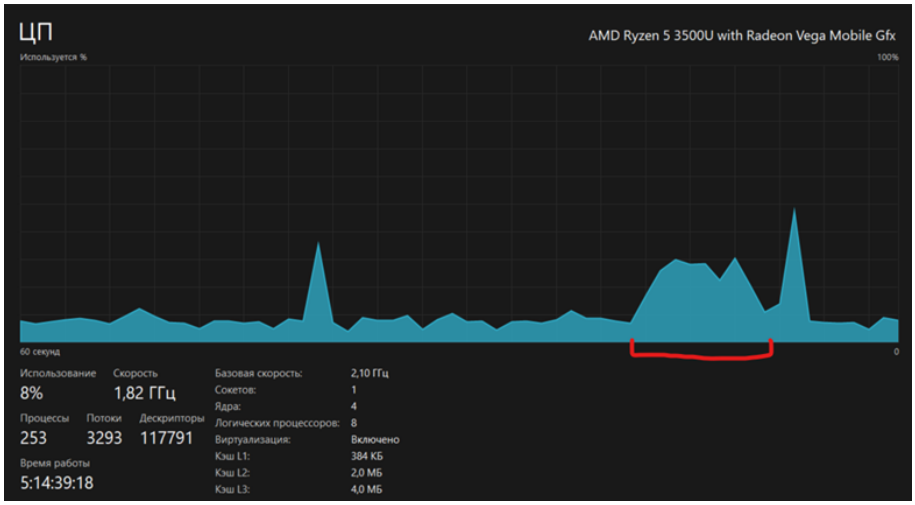


Рис. 1: Нагрузка ЦП

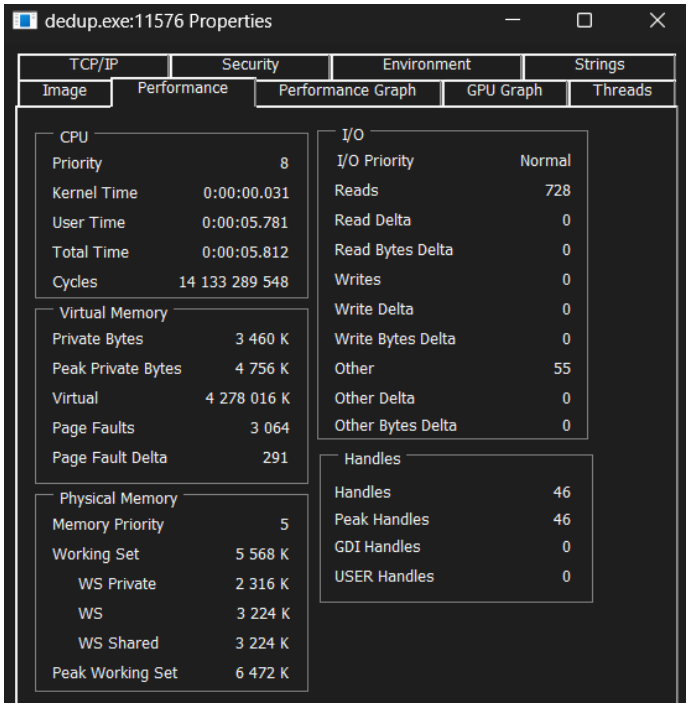


Рис. 2: Dedup properties (Performance)

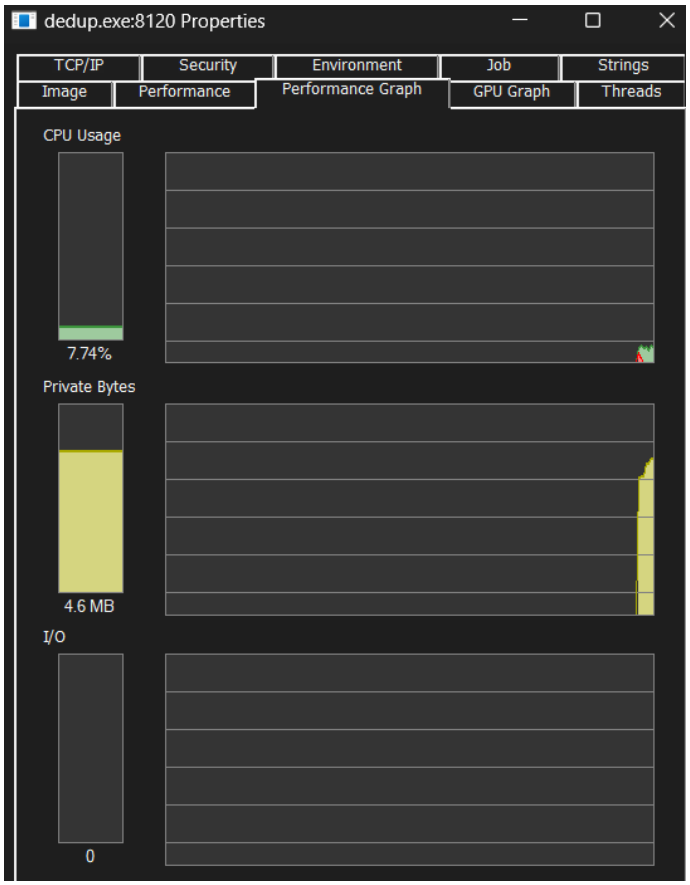


Рис. 3: Dedup properties (Performance Graph)

Полученные результаты:

- **Время выполнения:** Время выполнения оказалось на 1-2 секунды дольше, чем ожидалось. Это может быть связано с дополнительными операциями ввода-вывода, даже несмотря на использование одного большого буфера.
- **ЦП:** Программа использует более высокий процент CPU, чем ожидалось. Это логично, так как значительная часть работы связана с обработкой данных в оперативной памяти и удалением дубликатов.
- **IO Performance:** Несмотря на то, что программа выполняет операции чтения и записи в файл, показатели read bytes и write bytes равны 0. Это может быть связано с кэшированием операций ввода-вывода на уровне операционной системы, что приводит к отсутствию явных записей о физических операциях с диском.

4.2 IO-lat-write

Запуск 1 программы-нагрузчика

```
shell>io-lat-write "D:\dasha\UNI\OS\io.txt" 10000
average write time per iteration: 1306813 [ns]
Execution time: 13097 ms
```

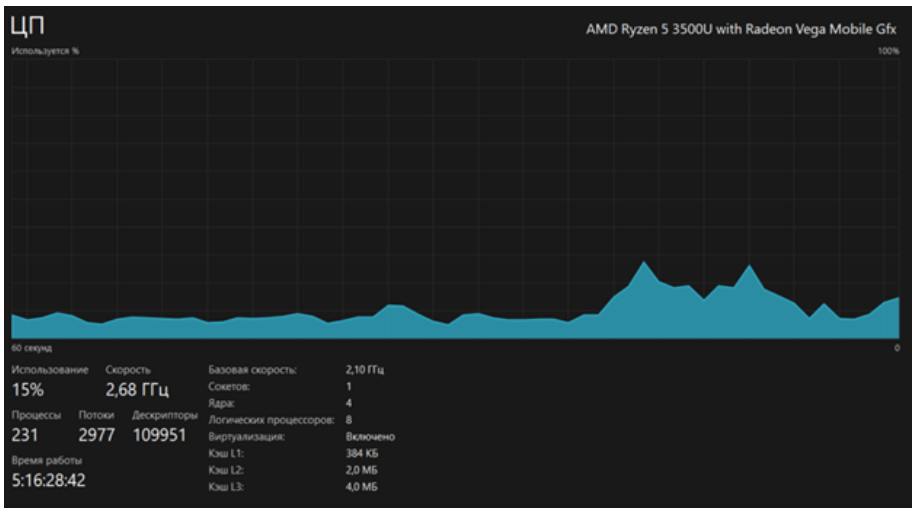


Рис. 4: Нагрузка ЦП

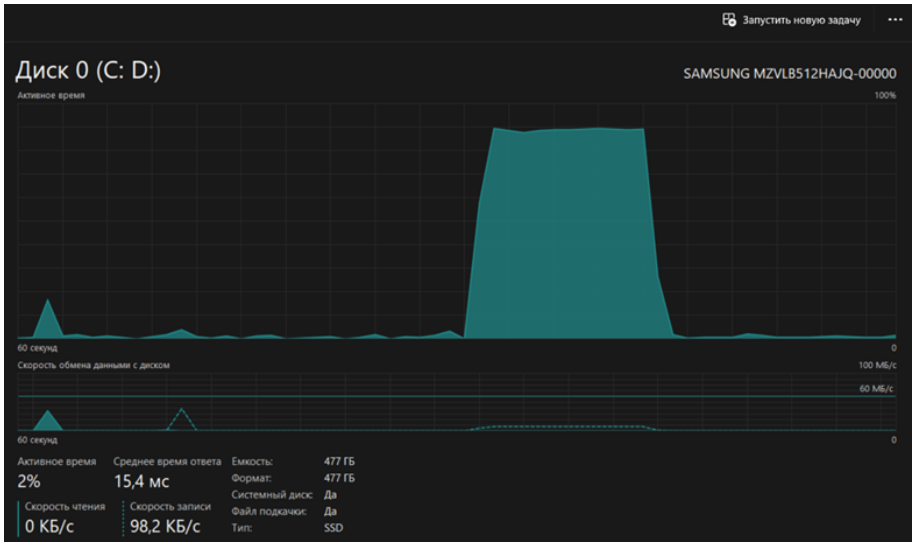


Рис. 5: Нагрузка диск

Рис. 6: IO-lat-write properties (Performance)

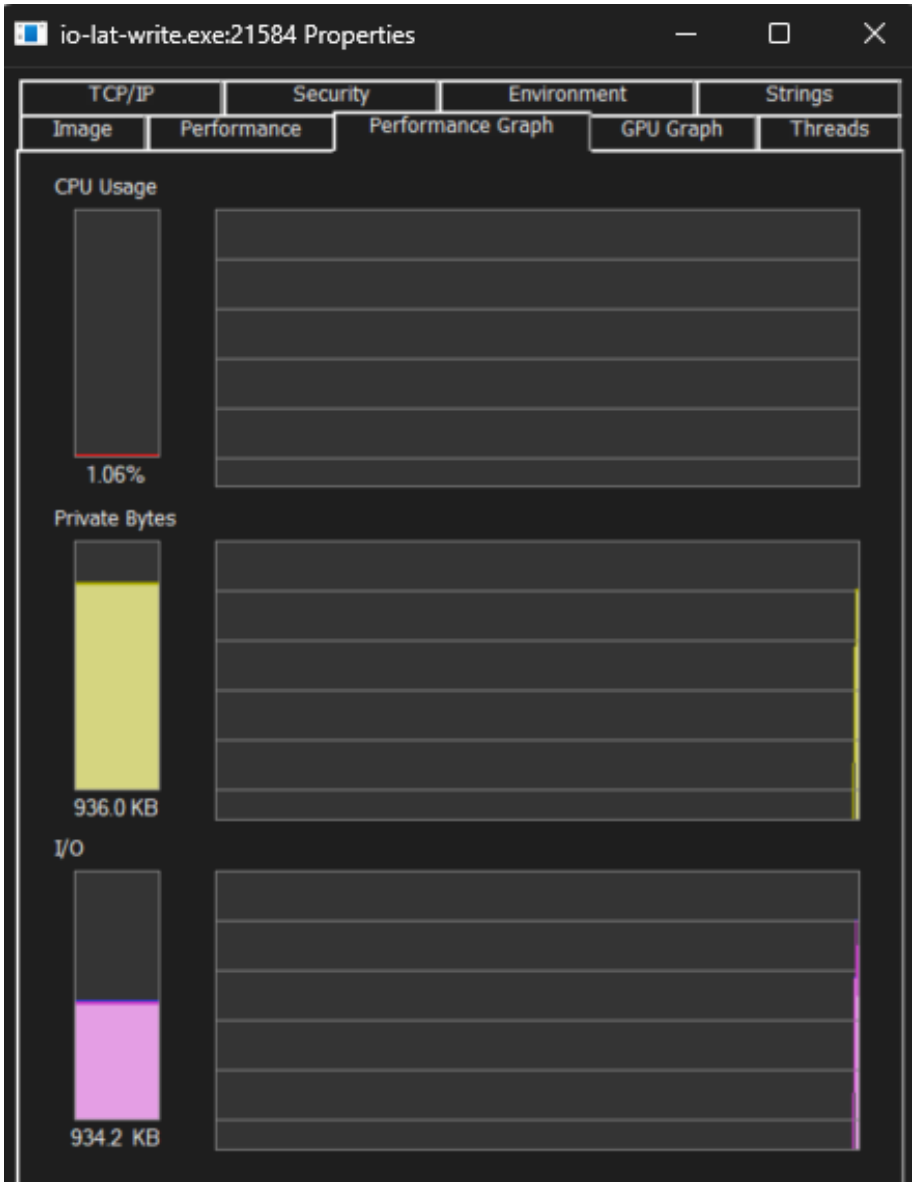


Рис. 7: IO-lat-write properties (Performance Graph)

Полученные результаты:

- **Время выполнения:** Программа выполняется в пределах предполагаемого диапазона (2000–10000 мс), но с небольшим превышением.
- **ЦП:** Процессор использовался в небольшом объеме (15-20%), что подтверждает низкую нагрузку на вычисления, поскольку работа программы в основном направлена на записи на диск.
- **IO Performance:** Задержки и высокая нагрузка на диск (90%) объясняются частыми операциями записи на диск в случайные места файла. Ожидания ввода-вывода высоки из-за случайных смещений, что замедляет процесс записи.

4.3 Количество нагрузчиков эффективно нагружающие ядра процессора

Характеристики: 4 ядра, 8 логических процессоров.
Полная нагрузка всех 8 логических процессоров происходит при 8 одновременно запущенных программ-нагрузчиков.

Рис. 8: Нагрузка ЦП

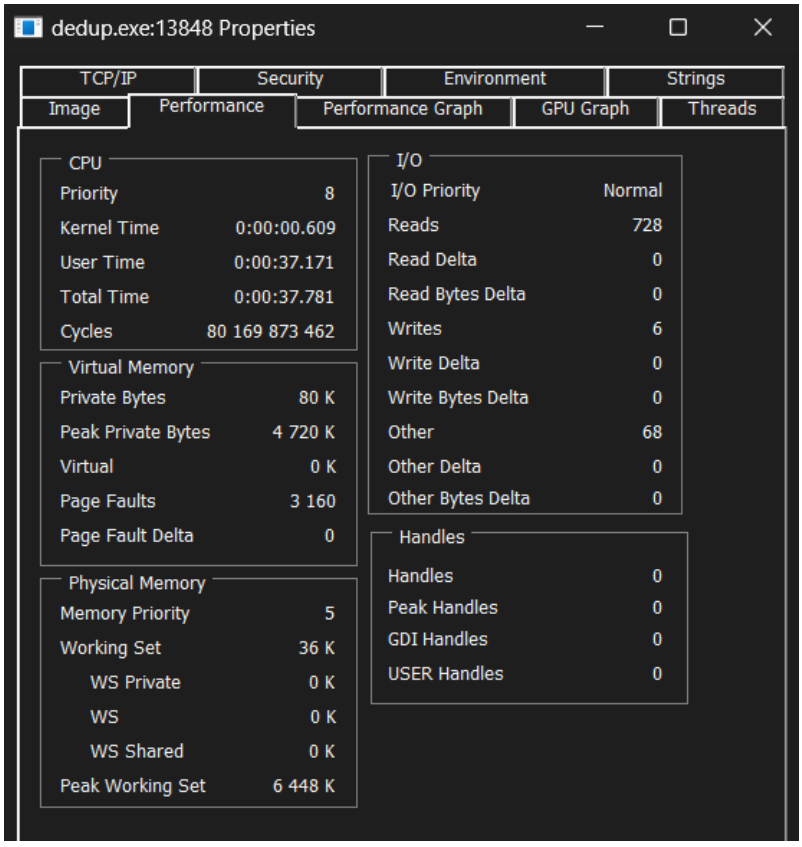


Рис. 9: Dedup properties

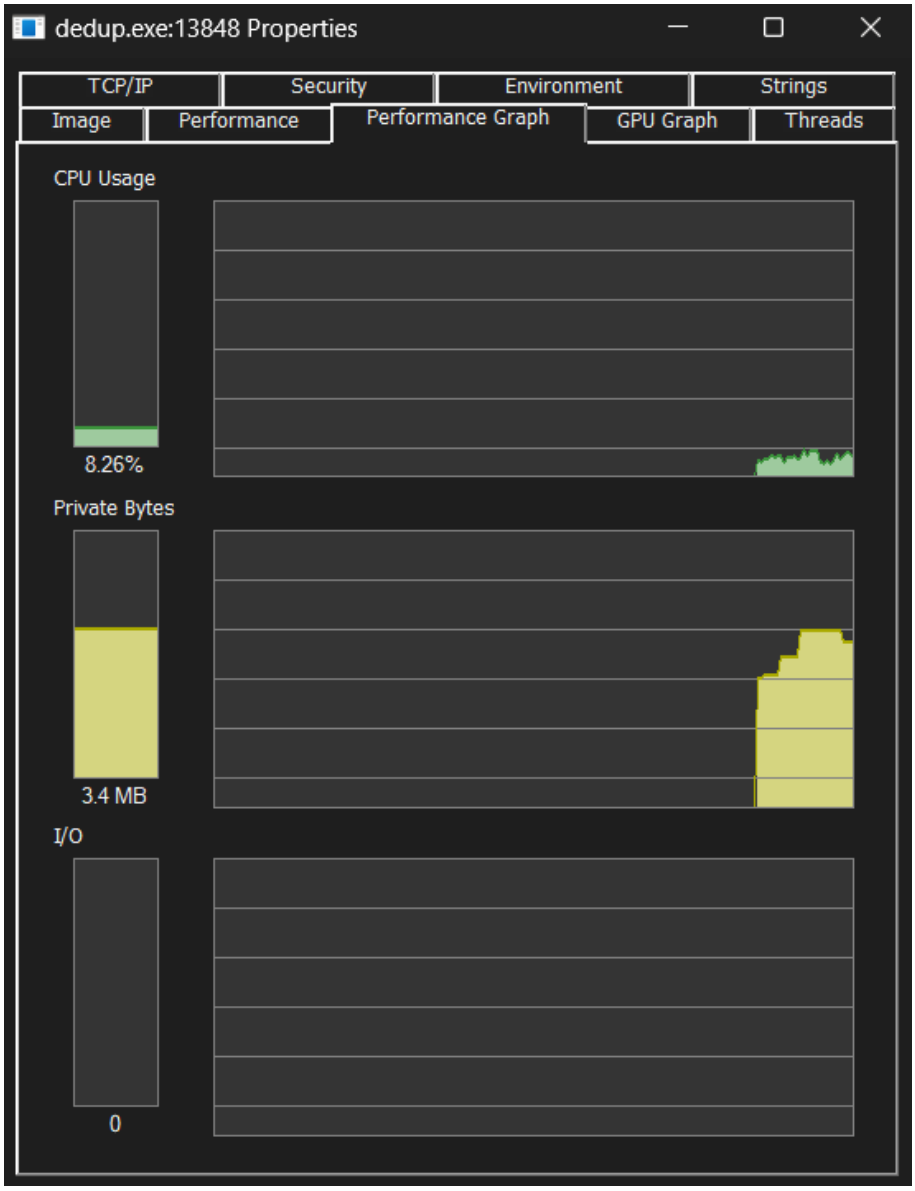


Рис. 10: Dedup properties (Performance Graph)

Полученные результаты:

- **USER%:** Время выполнения в пользовательском режиме составило 37.171 секунды, что является основным временем работы программы. Это ожидаемо, поскольку программа занимается вычислениями (удаление дубликатов), а значительная часть работы выполняется на уровне обработки данных.
- **SYS%:** Время в режиме ядра составило 0.609 секунды. Это небольшое значение указывает на то, что программа не выполняет много системных вызовов, а в основном работает в пользовательском режиме. Время в привилегированном режиме может быть связано с операциями с файлами (например, чтение и запись), но это значение остается незначительным.

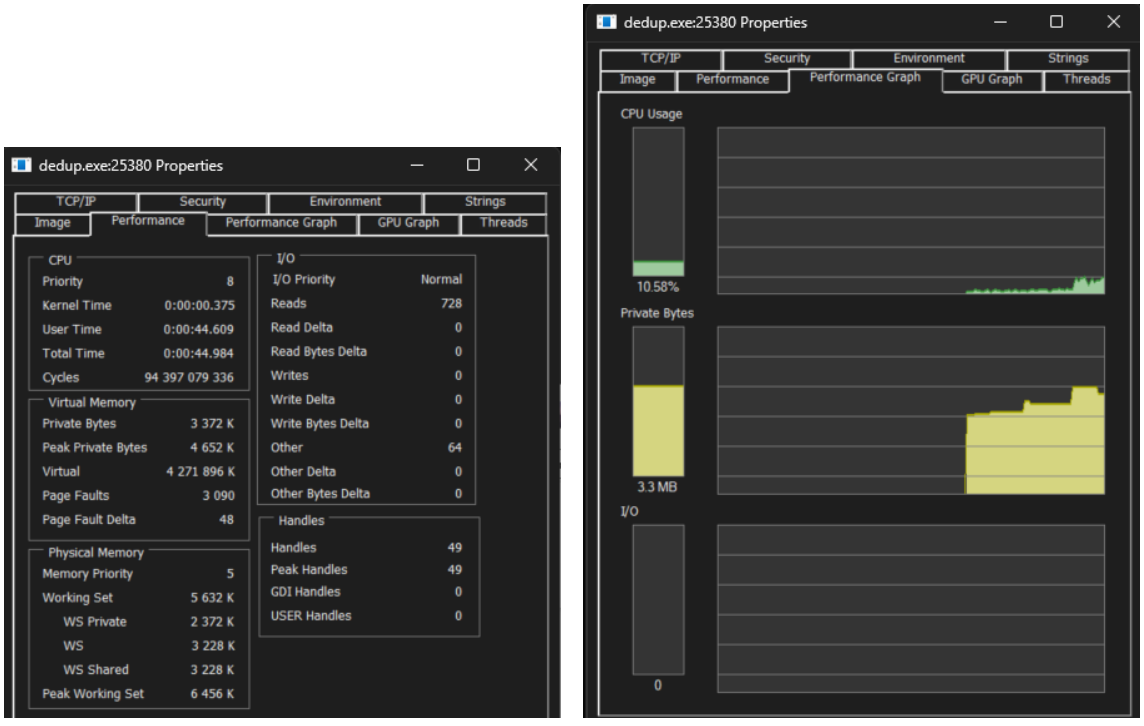
- **WAIT%:** Поскольку в данных указано IO = 0, это говорит о том, что программа не ожидает завершения операций ввода-вывода, и, следовательно, WAIT% близок к нулю. Это также подтверждается низким значением операций чтения и записи (728 чтений и 68 других операций), что указывает на минимальную нагрузку на диск в ходе работы программы

- **Реальное время выполнения нагрузчика:** Реальное время выполнения программы составило 37.781 секунды. Это значение включает все операции, такие как чтение данных, обработка (дедупликация) и запись в файл. Это общее время работы программы, которое включает как вычисления, так и операции с файлами.

- **Переключения контекста:** Количество переключений контекста для каждого из 8 процессов находится в диапазоне от 38 000 до 47 000. Это свидетельствует о высоком уровне многозадачности, так как операционная система активно переключает контексты между потоками и процессами.

4.4 Увеличенное количество нагрузчиков

16 нагрузчика:

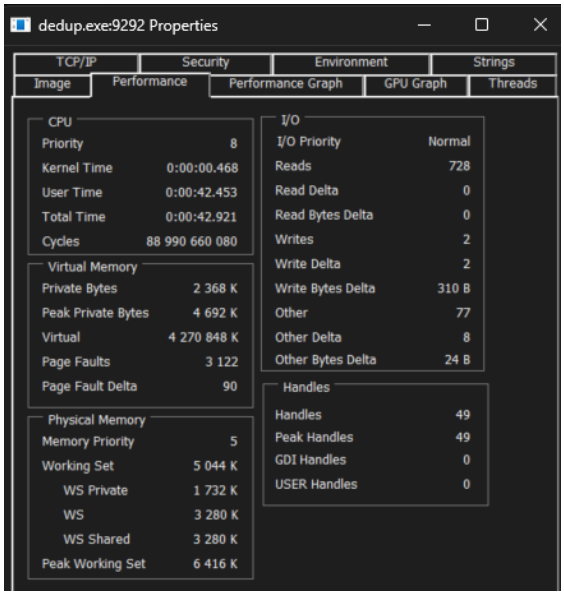


(a) Dedup properties (Performance)

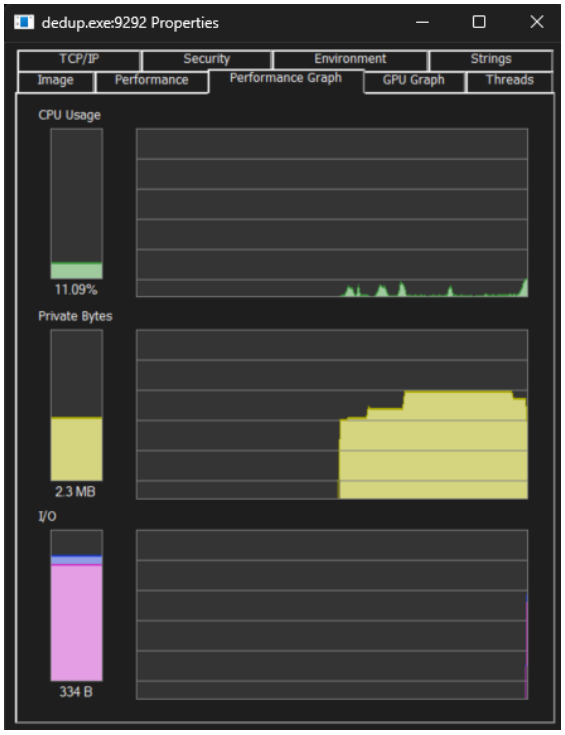
(b) Dedup properties (Performance Graph)

Рис. 11: 16 нагрузчика

24 нагрузчика:



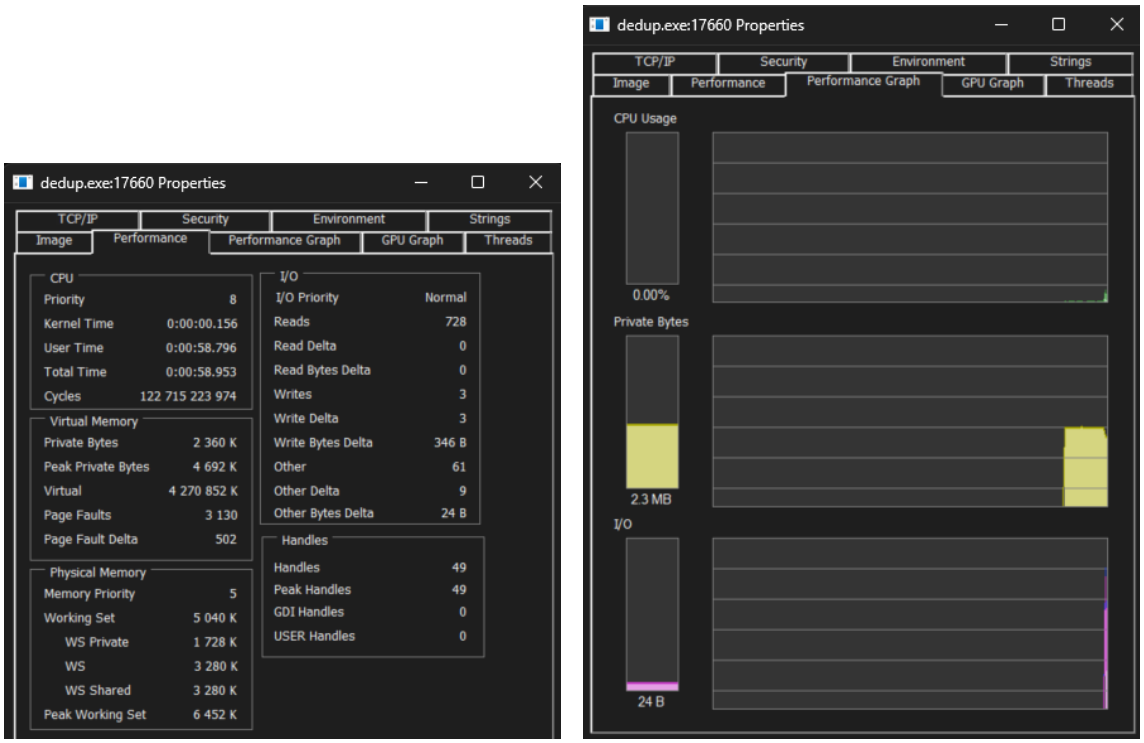
(a) Dedup properties (Performance)



(b) Dedup properties (Performance Graph)

Рис. 12: 24 нагрузчика

32 нагрузчика:



(a) Dedup properties (Performance) (b) Dedup properties (Performance Graph)

Рис. 13: 32 нагрузчика

Полученные результаты:

- **USER%:** Увеличивается с увеличением числа нагрузчиков (от 37.171 до 58.796 секунд). Это логично, так как больше процессов выполняет вычисления одновременно. Однако рост не является линейным, что указывает на появление накладных расходов из-за управления потоками.
- **SYS%:** Время CPU Kernel Time (работа в привилегированном режиме) остается небольшим ($0.375 \rightarrow 0.468 \rightarrow 0.156$). Это означает, что нагрузка на ядро системы не увеличивается пропорционально числу процессов. Это объясняется тем, что основная нагрузка идет на обработку данных в памяти, а не на системные вызовы
- **WAIT%:** На ранних этапах число операций I/O практически не меняется. Однако при 24 и 36 нагрузчиках появляются небольшие операции записи, хотя общее число операций ввода-вывода остается низким. Это может указывать на конкуренцию за ресурсы (файлы, диск, память), что приводит к незначительному увеличению времени ожидания.
- **Реальное время выполнения нагрузчика:** Время выполнения нагрузки растет с 37.781 секунд до 58.953 секунд при увеличении числа нагрузчиков. Однако производительность растет не линейно, а с замедлением. о

Первые увеличения (с 8 до 16 и 24 нагрузчиков) почти равномерно распределяют работу между всеми логическими процессорами. о Однако начиная с 36 нагрузчиков видно, что CPU usage становится очень маленьким (0-8%). Это явный признак того, что система начинает страдать от чрезмерного количества переключений контекста и потерь производительности.

- **Переключения контекста:**
  - 16: 40 000 - 46 000 (небольшой рост, но в пределах нормы)
  - 24: 29 000 - 60 000 (рост разброса, система начинает испытывать трудности с управлением потоками)
  - 32: 9 000 - 17 000 (резкий спад, процессоры тратят больше времени в ожидании, а не в вычислениях)

До 16 нагрузчиков система справляется хорошо. При 24 нагрузчиках переключения растут, но система еще работает. При 36 нагрузчиках происходит перегрузка, и процессоры не справляются с управлением процессами, теряя эффективность.
Чем больше нагрузчиков, тем сильнее падает производительность на один процесс, так как процессоры тратят больше времени на переключение контекста, чем на реальные вычисления. Оптимальное количество нагрузчиков для данной системы: 16-24, дальше производительность не увеличивается, а падает.

4.5 Многопоточная программа

Запуск программы-нагрузчика Dedup с 8 потоками

```
shell> dedup_v2 in_d_threads.txt out.txt 8
```

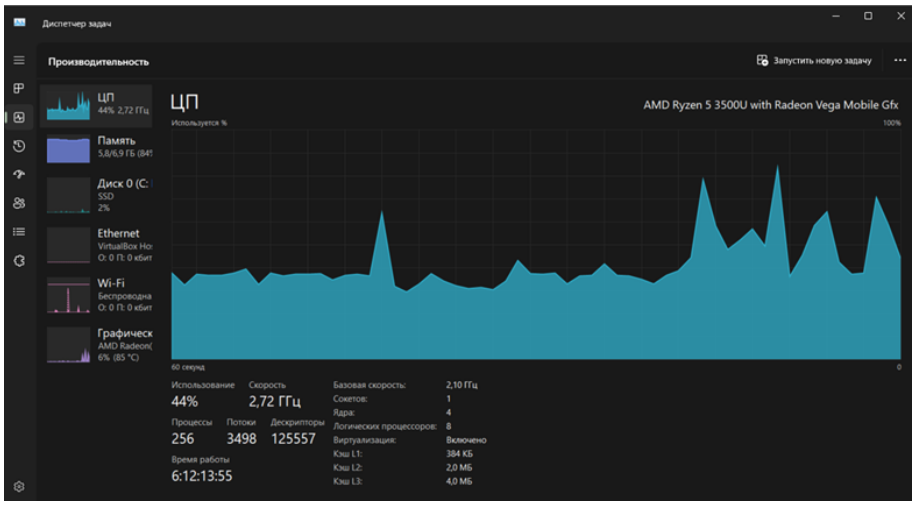


Рис. 14: CPU

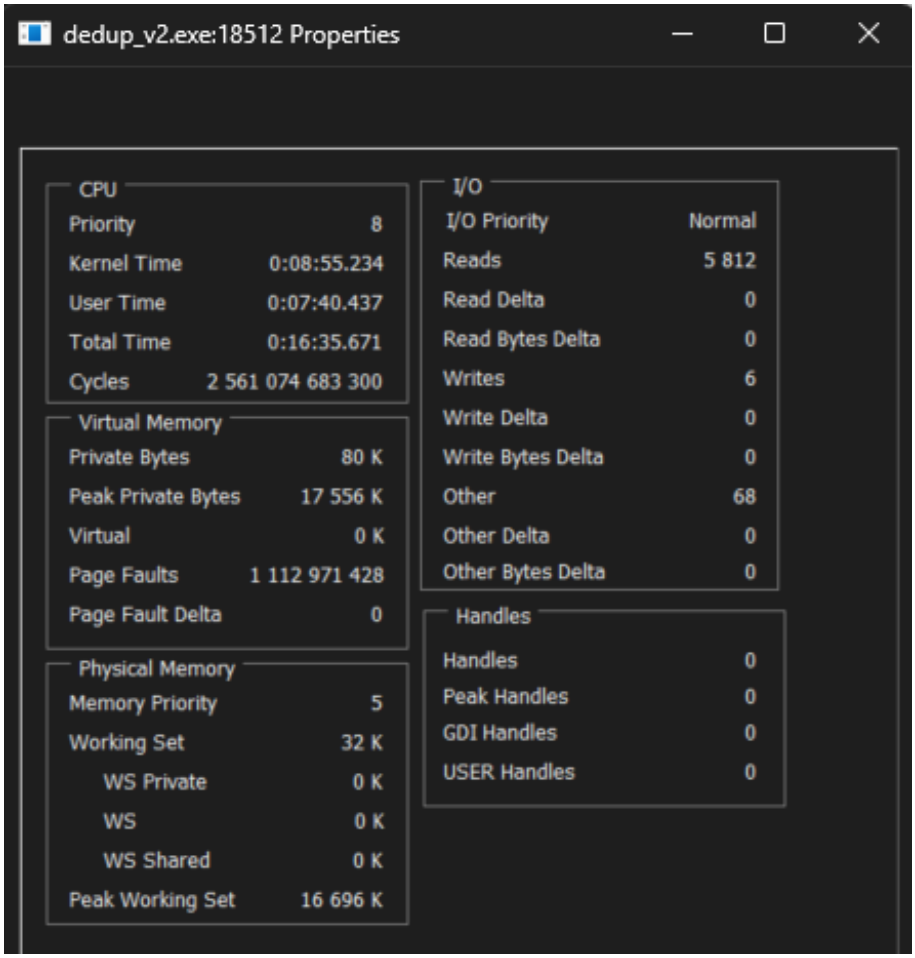


Рис. 15: Dedup properties

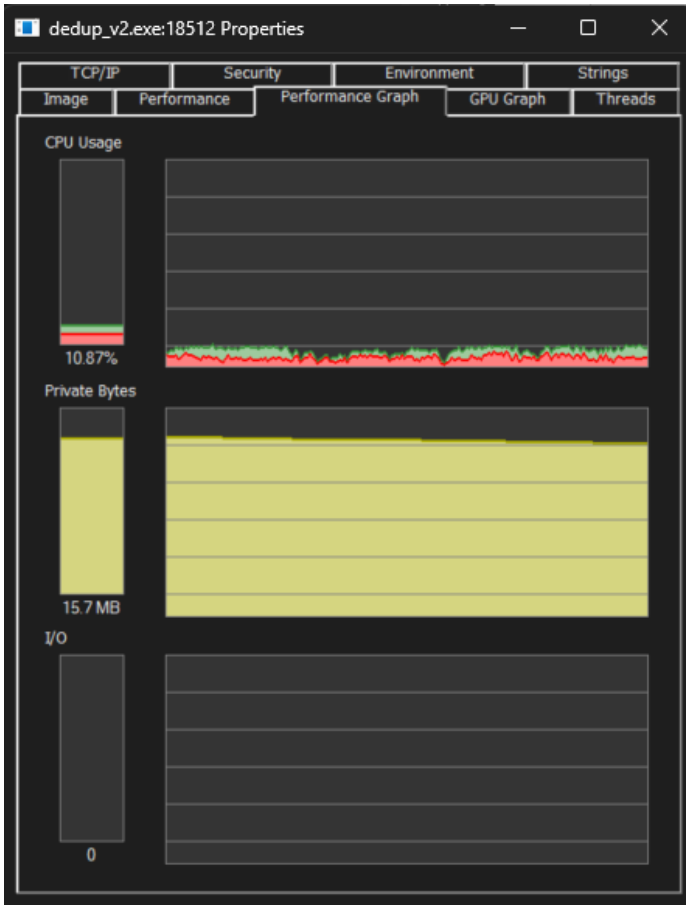


Рис. 16: Dedup properties (Performance)

Полученные результаты:

Параметр	**8 отдельных программ**	**1 программа с 8 потоками**
Общая загрузка CPU	100%	40%
CPU Kernel Time	~4.87 сек	535 сек
CPU User Time	~297.36 сек	460 сек
Total Time	~302.24 сек	995 сек
CPU Usage (на процесс)	8-11%	10.87%
I/O Reads	5824	5812
Other I/O	544	44
Переключение контекста	300 000+	180 000

Таблица 1: Сравнение работы 8 отдельных программ и 1 программы с 8 потоками

- Общее время выполнения увеличилось (995 сек против 302 сек).

- Многопоточная программа использует CPU менее эффективно, нагрузка всего 40% вместо 100%.
- Многопоточная программа делает меньше переключений контекста, что говорит о сниженной накладной стоимости.

Изменения при запуске 2, 3 многопоточных программ:

- Загрузка CPU увеличится ближе к 100%.
- Общее время выполнения сократится, так как процессор будет использовать больше потоков.
- Увеличатся переключения контекста, так как теперь будет несколько процессов, каждый со своими потоками.

4.6 Программы с оптимизацией

Запуск программы Dedup

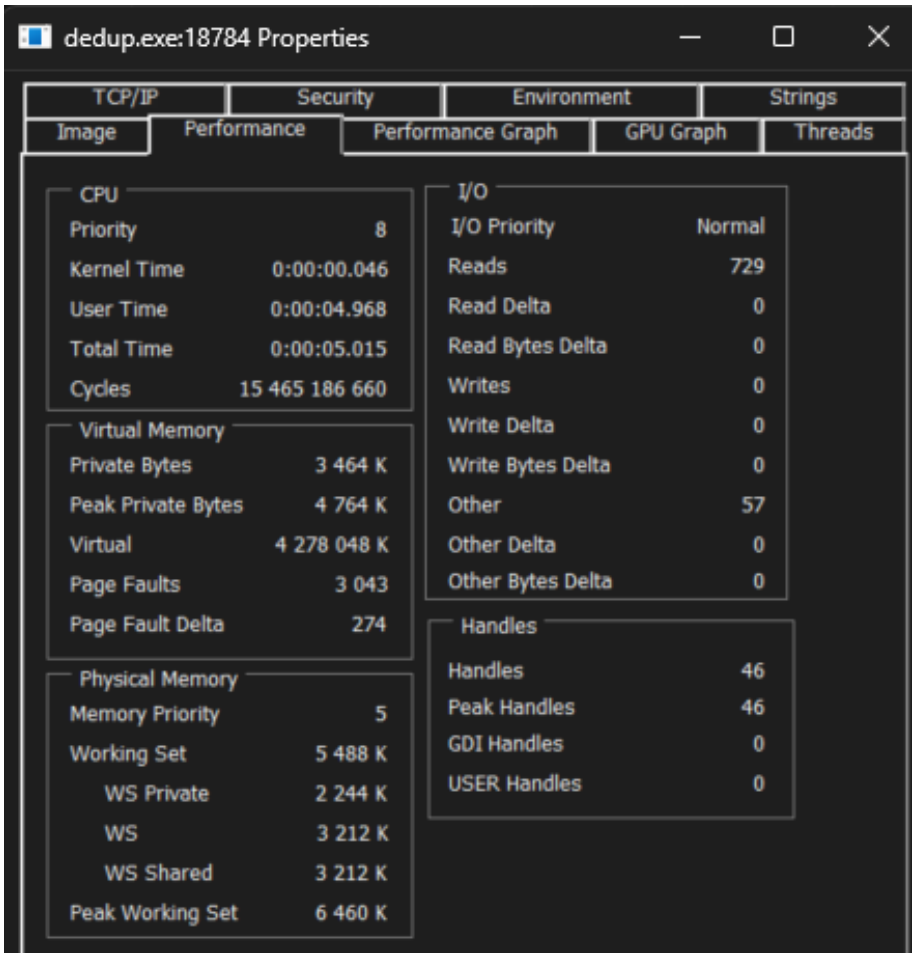


Рис. 17: Dedup properties

Полученные результаты:

После добавления агрессивной оптимизации (-O3) общее время выполнения программы уменьшилось с 5.812 сек до 5.015 сек, что составляет примерно 13.7% ускорения.

Основные изменения:

- user time уменьшилось с 5.781 сек до 4.968 сек (компилятор смог оптимизировать вычисления, убрав лишние инструкции).
- kernel time немного увеличилось (с 0.031 сек до 0.046 сек), но это несущественное изменение, возможно, связано с изменением порядка выполнения системных вызовов.
- total time уменьшилось, что означает реальный прирост скорости выполнения.

Запуск программы IO-lat-write

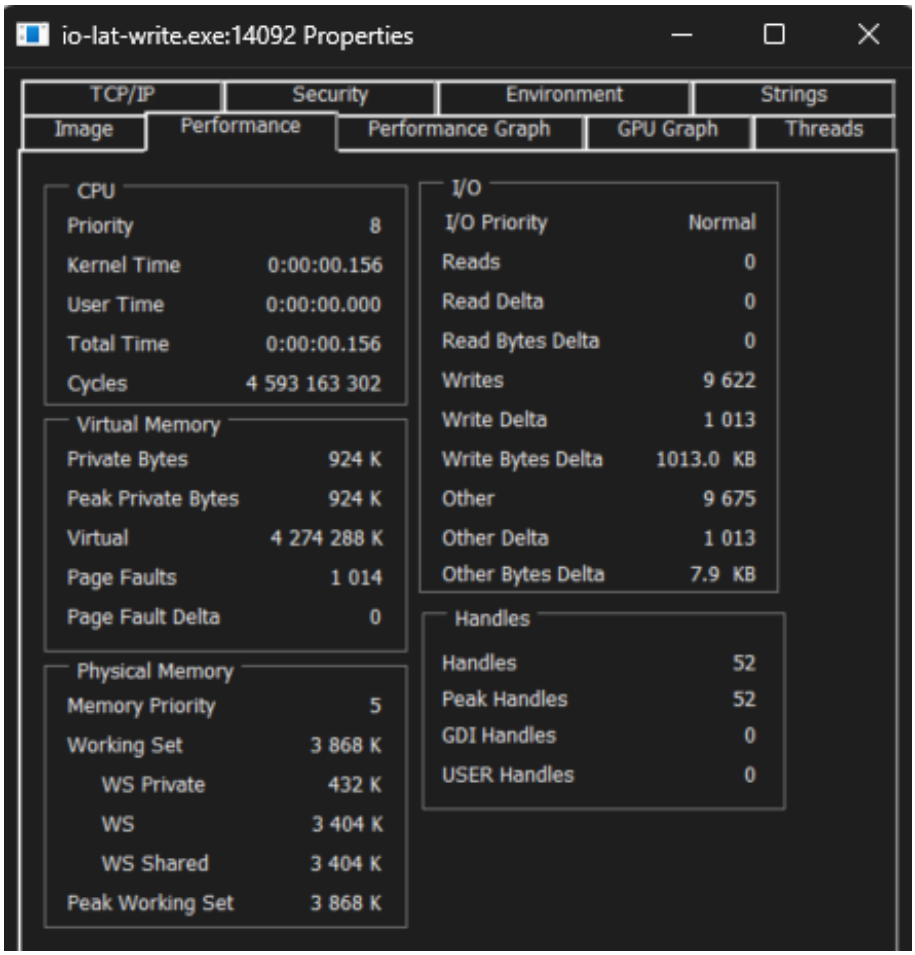


Рис. 18: IO-lat-write properties

После добавления агрессивной оптимизации (-O3) общее время выполнения программы уменьшилось с 1.078 сек до 0.156 сек.
Основные изменения:

- user time осталось 0.000, так как программа выполняет в основном системные операции.
- kernel time сократилось с 1.078 сек до 0.156 сек, что указывает на оптимизацию работы с системными вызовами.
- total time уменьшилось с 1.078 сек до 0.156 сек (7 раз быстрее).

Оптимизация -O3 значительно сократила время выполнения io-lat-write, так как компилятор, вероятно, уменьшил накладные расходы при работе с системными вызовами ввода-вывода.

5 Вывод

Во время выполнения лабораторной работы была разработана оболочка командной строки shell для запуска программ с аргументами и вычисления времени их выполнения. Реализованы программы-нагрузчики для тестирования вычислительной и дисковой подсистем. Проведено профилирование с анализом CPU, IO и задержек. Выявлены закономерности нагрузки на систему, влияние многопоточности и оптимизации компилятора. Получены навыки работы с процессами, потоками и инструментами мониторинга.