

Министерство образования Республики Беларусь
Учреждение образования «Белорусский государственный университет
информатики и радиоэлектроники»

Факультет компьютерных систем и сетей

Кафедра информатики

Дисциплина: Операционные среды и системное программирование

ОТЧЁТ
к лабораторной работе №5
на тему

УПРАВЛЕНИЕ ПОТОКАМИ, СРЕДСТВА СИНХРОНИЗАЦИИ

Выполнил: студент гр.253504
Фроленко К.Ю.
Проверил: ассистент кафедры информатики
Гриценко Н.Ю.

Минск 2025

СОДЕРЖАНИЕ

1	Формулировка задачи	3
2	Краткие теоретические сведения	4
3	Описание функций программы.....	6
3.1	Функция generate_test_file	6
3.2	Функция read_data	6
3.3	Функция write_array_to_file	6
3.4	Функция cmpfunc	6
3.5	Функция thread_sort	7
3.6	Функция merge	7
3.7	Функция main	7
3.8	Makefile	7
4	Пример выполнения программы	8
4.1	Запуск программы и процесс выполнения	8
	Вывод.....	9
	Список использованных источников	10
	Приложение А (обязательное)	11

1 ФОРМУЛИРОВКА ЗАДАЧИ

Целью данной лабораторной работы является изучение подсистемы потоков (*pthread*) и средств синхронизации в операционной системе *Unix*. В ходе работы осуществляется практическое освоение *API* библиотеки *pthread* для создания, управления и синхронизации потоков, что позволяет реализовывать параллельную обработку данных. Основное внимание уделяется разработке многопоточной программы сортировки, которая обеспечивает следующие функциональные возможности:

1 Параллельная сортировка массива: программа должна принимать большой массив числовых данных, разбивать его на равномерные сегменты и сортировать каждый сегмент в отдельном потоке с использованием функции *pthread_create*. Это позволяет значительно сократить время обработки по сравнению с однопоточной сортировкой.

2 Синхронизация потоков: для корректного объединения результатов работы потоков необходимо применять средства синхронизации, такие как мьютексы, барьеры или спин-блокировки. Это предотвращает возникновение гонок и обеспечивает безопасный доступ к общим ресурсам при объединении отсортированных сегментов.

3 Завершение и объединение потоков: после завершения сортировки отдельных сегментов программа должна корректно ожидать завершения всех потоков (с помощью *pthread_join*) и затем объединять результаты в единый отсортированный массив с использованием алгоритма слияния (*merge*).

4 Сравнение эффективности: в рамках работы планируется измерение времени выполнения многопоточной сортировки и его сравнение с однопоточной сортировкой, реализованной через стандартную функцию *qsort*. Это позволит оценить преимущества распараллеливания вычислений на многоядерных системах.

5 Модульность и автоматизация сборки: программа должна быть структурирована на отдельные модули, отвечающие за генерацию тестовых данных, многопоточную сортировку, объединение отсортированных фрагментов и вывод результатов. Кроме того, необходимо создать *makefile*, содержащий цели для сборки исполняемых файлов, очистки промежуточных файлов (*clean*) и, по возможности, тестирования работы приложения.

Результатом выполнения лабораторной работы станет работоспособное приложение, демонстрирующее практическое применение потоков и средств синхронизации в *Unix*-среде. Полученные знания и навыки по управлению потоками позволят глубже понять принципы параллельного программирования, оптимизации вычислений и эффективного распределения вычислительных ресурсов, что является важным аспектом при разработке высокопроизводительных и масштабируемых приложений.

2 КРАТКИЕ ТЕОРИТИЧЕСКИЕ СВЕДЕНИЯ

Параллельное программирование в *Unix*-среде с использованием потоков является одним из ключевых направлений для повышения производительности и эффективности приложений. Потоки представляют собой легковесные единицы выполнения, которые работают в рамках одного процесса, что позволяет им разделять общее адресное пространство и оперативную память. Такая архитектура существенно сокращает накладные расходы по сравнению с процессами, поскольку создание и переключение контекста между потоками происходит быстрее, а обмен данными между ними осуществляется непосредственно через общую память [1].

Основным инструментом для работы с потоками в *Unix* является библиотека *POSIX Threads* (*pthread*). Функция *pthread_create* используется для порождения нового потока, которому передаётся указатель на функцию, определяющую его задачу, а также набор параметров, необходимых для выполнения. Потоки могут быть запущены в двух режимах: *joinable*, при котором основной поток может дожидаться завершения дочернего потока с помощью *pthread_join*, и *detached*, когда поток самостоятельно завершает работу и освобождает все связанные ресурсы без необходимости ожидания. Такой выбор позволяет гибко управлять жизненным циклом потоков в зависимости от специфики решаемой задачи.

Синхронизация является критически важным аспектом при работе с потоками, так как одновременный доступ к общим ресурсам может привести к состояниям гонки и некорректной работе приложения. Для защиты критических секций используются мьютексы (*pthread_mutex*). Мьютексы обеспечивают эксклюзивный доступ к данным, что позволяет избежать конфликтов при их изменении. Помимо мьютексов, в многопоточных приложениях применяются барьеры (*pthread_barrier*), которые синхронизируют выполнение группы потоков, заставляя их ожидать достижения определённой точки выполнения, а также спин-блокировки (*pthread_spinlock*), эффективные в условиях высокой конкуренции, когда время ожидания минимально и накладные расходы на блокировку должны быть сведены к минимуму. Современные реализации могут также использовать «быстрые мьютексы» (*futex*), что позволяет добиться ещё более низких задержек при синхронизации [2].

Кроме того, важную роль играют условные переменные (*pthread_cond*), которые позволяют потокам ждать наступления определённого события. Использование условных переменных в сочетании с мьютексами обеспечивает возможность реализации более сложных схем синхронизации, например, при условном ожидании данных или сигналов от других потоков. Это особенно важно при реализации алгоритмов, где требуется точное согласование действий нескольких потоков.

Завершение потоков может осуществляться как естественным путём посредством вызова *pthread_exit*, так и принудительно через *pthread_cancel*. Принудительное завершение требует особого внимания к «точкам отмены»

(*cancellation points*), где поток проверяет, не поступил ли запрос на его отмену, что позволяет безопасно завершить работу без нарушения целостности данных. Такой механизм гарантирует, что ресурсы, занятые потоком, будут корректно освобождены даже в случае аварийного завершения.

Управление потоками и синхронизация играют ключевую роль в реализации параллельных алгоритмов, таких как многопоточная сортировка. При распределении задачи сортировки большого массива данных, массив делится на несколько фрагментов, каждый из которых сортируется в отдельном потоке. Затем, используя механизмы синхронизации, результаты работы потоков объединяются в единый отсортированный массив. Такой подход позволяет существенно снизить время выполнения задачи за счёт использования многоядерных процессоров, при этом корректное управление потоками предотвращает возникновение конфликтов и обеспечивает надежность работы приложения.

Автоматизация сборки проектов с использованием системы *make* является неотъемлемой частью современного процесса разработки. *Makefile* позволяет задать зависимости между исходными файлами, автоматически компилировать проект, а также проводить очистку промежуточных артефактов. Такой подход не только ускоряет процесс разработки, но и способствует поддержанию единообразного стиля кода, что особенно важно при работе в команде и дальнейшем сопровождении программного продукта [3].

Таким образом, глубокое понимание принципов управления потоками и использования средств синхронизации является фундаментальным для разработки высокопроизводительных, масштабируемых и отказоустойчивых приложений в *Unix*-среде. Эти знания позволяют разработчикам эффективно использовать ресурсы системы, оптимизировать распределение вычислительной нагрузки и создавать надежные программные решения, способные справиться с высокими требованиями современных вычислительных задач.

3 ОПИСАНИЕ ФУНКЦИЙ ПРОГРАММЫ

В рамках данной лабораторной работы разработана программа для многопоточной сортировки, которая состоит из двух основных модулей: генератора тестовых данных и самой программы сортировки. Реализованы следующие функции: *generate_test_file*, *read_data*, *write_array_to_file*, *cmpfunc*, *thread_sort*, *merge*, *main*, *makefile* для автоматизации сборки.

3.1 Функция *generate_test_file*

Функция *generate_test_file*, реализованная в модуле *generate.c*, отвечает за создание входного файла с тестовыми данными. Она открывает файл для записи, записывает сначала количество элементов, а затем генерирует случайные числа в заданном диапазоне. При возникновении ошибки (например, невозможности открыть файл) функция выводит сообщение об ошибке и завершает работу. Данная функция позволяет подготовить исходный набор данных, который будет использоваться для сортировки.

3.2 Функция *read_data*

Реализованная в модуле *sort.c*, функция *read_data* выполняет чтение входного файла, в котором сначала указывается количество элементов, а затем – сами числовые данные. Функция выделяет необходимую память для хранения массива и проверяет корректность считанных данных. При возникновении ошибок (например, если файл отсутствует или данные повреждены) функция выводит соответствующее сообщение и завершает выполнение программы. Считанный массив используется далее для сортировки.

3.3 Функция *write_array_to_file*

Эта функция предназначена для записи отсортированного массива в выходной файл. Она открывает файл для записи, выводит сначала количество элементов, а затем записывает сами значения массива. Функция обеспечивает сохранение результатов сортировки для последующей проверки корректности работы программы.

3.4 Функция *cmpfunc*

Функция *cmpfunc* служит для сравнения двух целых чисел и используется стандартной функцией *qsort*. Она возвращает разность между сравниваемыми элементами, что позволяет упорядочить их в возрастающем порядке. Корректность работы этой функции является критически важной для достижения правильного результата сортировки.

3.5 Функция *thread_sort*

Функция *thread_sort* отвечает за сортировку отдельного сегмента массива в отдельном потоке. Каждый поток получает структуру с указателем на общий массив и границами своего сегмента. Внутри функции вызывается *qsort* для сортировки именно этого фрагмента. После завершения сортировки поток корректно завершается через *pthread_exit*, что позволяет основному процессу дождаться всех потоков с помощью *pthread_join*.

3.6 Функция *merge*

Функция *merge* реализует алгоритм слияния двух отсортированных сегментов массива в один упорядоченный фрагмент. Она последовательно сравнивает элементы из двух частей, копирует наименьшие значения во вспомогательный массив, а затем обновляет исходный массив. Этот процесс повторяется до тех пор, пока все сегменты не объединятся в окончательно отсортированный массив.

3.7 Функция *main*

Основная функция *main* объединяет все этапы работы программы сортировки. Сначала она считывает данные из входного файла с помощью *read_data*, затем создает две копии исходного массива: одна для многопоточной сортировки, другая – для однопоточной сортировки с использованием *qsort*. После этого происходит запуск нескольких потоков для сортировки отдельных сегментов массива, что реализуется через *thread_sort*. После завершения работы потоков основной процесс объединяет отсортированные сегменты с помощью функции *merge*. Время выполнения многопоточной сортировки измеряется с помощью *gettimeofday*, а затем сравнивается с временем однопоточной сортировки. Результаты сортировки записываются в соответствующие файлы, а итоговая информация выводится на экран.

3.8 Makefile

Для автоматизации сборки проекта используется *makefile*, который определяет цели для компиляции как генератора тестовых данных, так и программы сортировки. *Makefile* задает переменные компиляции (например, *CFLAGS* с оптимизацией, предупреждениями и поддержкой *pthread*) и включает цель *clean* для удаления временных файлов и артефактов сборки. Это упрощает процесс тестирования и дальнейшего сопровождения проекта.

4 ПРИМЕР ВЫПОЛНЕНИЯ ПРОГРАММЫ

4.1 Запуск программы и процесс выполнения

При запуске программы сначала выполняется генерация тестовых данных. Программа *generate* принимает на вход количество элементов и создаёт файл с указанным числом случайных чисел, выводя сообщение об успешном создании файла. Затем запускается программа *sort*, которая считывает данные из файла, выводит в консоль сообщение о прочтении элементов и отображает результаты измерения времени работы алгоритмов.

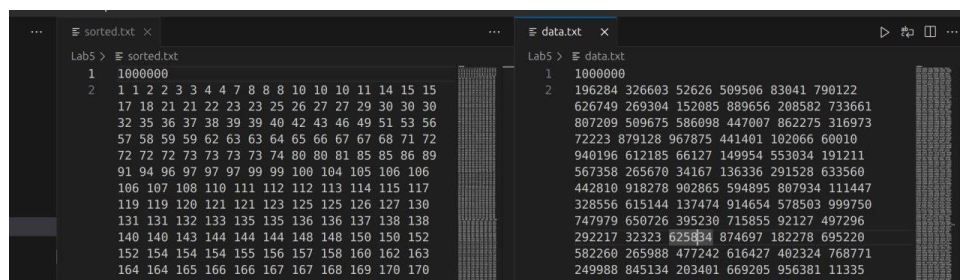
На рисунке 4.1 приведён пример консольного вывода, где видно сообщение о создании файла, количество прочитанных элементов, а также время работы многопоточной и однопоточной сортировок.

```
janklumin@janklumin-ASUS-Zenbook-14-UX3405MA-UX3405MA:~/Workspace/BSUIR-Labs/OSISP/Lab5$ ./generate 1000000
Файл 'data.txt' с 1000000 элементами успешно создан.
janklumin@janklumin-ASUS-Zenbook-14-UX3405MA-UX3405MA:~/Workspace/BSUIR-Labs/OSISP/Lab5$ ./sort
Прочитано 1000000 элементов из файла data.txt
Время многопоточной сортировки: 43.585 мс
Время однопоточной сортировки: 151.441 мс
Отсортированные массивы записаны в файлы 'sorted2.txt' (многопоточная) и 'sorted.txt' (однопоточная).
```

Рисунок 4.1 – Генерация данных и результаты сортировки

После вывода в консоль программа записывает отсортированные массивы в файлы *sorted2* и *sorted.txt*, что дополнительно подтверждает корректное выполнение алгоритма.

Для наглядной демонстрации работы алгоритма сортировки показываются содержимое входного файла и результирующего файла. Исходный файл содержит несортированные данные, которые после обработки многопоточной сортировкой преобразуются в упорядоченный массив. На рисунке 4.2 наглядно видна разница между исходным неупорядоченным набором чисел и отсортированным результатом.



```
sorted.txt
1 1000000
2 1 1 2 2 3 3 4 4 7 8 8 8 10 10 10 11 14 15 15
17 18 21 21 22 23 23 25 26 27 27 29 30 30 30
32 35 36 37 38 39 39 40 42 43 46 49 51 53 56
57 58 59 59 62 63 64 65 66 67 67 68 71 72
72 72 72 73 73 73 74 80 80 81 85 85 86 89
91 94 96 97 97 97 99 99 100 104 105 106 106
106 107 108 110 111 112 112 113 114 115 117
119 119 120 121 121 123 125 125 126 127 130
131 131 132 133 135 135 136 136 137 138 138
140 140 143 144 144 144 148 148 150 150 152
152 154 154 154 155 156 157 158 160 162 163
164 164 165 166 166 167 167 168 169 170 170
170 171 171 172 172 172 174 176 177 177 178

data.txt
1 1000000
2 196284 326603 52626 509506 83041 790122
626749 260304 152885 889656 208582 733661
807209 509675 586098 447007 862275 316973
72223 879128 967875 441401 182666 60010
940196 612185 66127 149954 553034 191211
567358 265670 34167 136336 291528 633560
442810 918278 902865 594895 807934 111447
328556 615144 137474 914654 578503 999750
747979 650726 395230 715855 92127 497296
292217 32323 625834 874697 182278 695220
582260 265988 477242 616427 402324 768771
249988 845134 203481 669205 956381 11335
202084 284028 143831 424478 716044 721235
```

Рисунок 4.2 – Содержимое файлов до сортировки и после сортировки

Таким образом, результаты демонстрируют, что программа успешно выполняет параллельную сортировку, сокращая время обработки за счёт использования потоков, и корректно преобразует исходный неупорядоченный массив в отсортированный набор данных.

ВЫВОД

В ходе выполнения лабораторной работы была успешно разработана программа для многопоточной сортировки, демонстрирующая практическое применение потоков и средств синхронизации в *Unix*-среде. Реализация проекта позволила применить и закрепить знания о создании и управлении потоками с использованием библиотеки *pthread*, а также о методах синхронизации и объединения результатов параллельной обработки данных.

Программа была организована так, что исходный массив данных разбивается на равномерные сегменты, каждый из которых сортируется в отдельном потоке. После завершения сортировки отдельных сегментов осуществляется их объединение с помощью алгоритма слияния, что позволяет получить окончательно отсортированный массив. Измерение времени выполнения многопоточной сортировки в сравнении с однопоточным методом показало существенное сокращение времени обработки, что особенно заметно на многоядерных системах. Это подтверждает эффективность распараллеливания вычислений для обработки больших объемов данных.

Особое внимание было уделено обработке ошибок, таких как неудачные вызовы функций для выделения памяти, открытия файлов или создания потоков. Программа корректно обрабатывает подобные ситуации, что повышает её надежность и устойчивость к возможным сбоям в процессе выполнения. Дополнительно, модульная структура кода и автоматизация сборки с помощью *makefile* значительно упростили тестирование и дальнейшее сопровождение проекта, обеспечивая единообразие в организации кода и быстроту внесения изменений.

Полученные знания и практические навыки в области управления потоками и синхронизации могут быть успешно применены при разработке более сложных параллельных алгоритмов и оптимизации вычислительных процессов в современных системах. Реализованный подход к распараллеливанию позволяет не только сократить время выполнения вычислительно затратных задач, но и обеспечить эффективное использование ресурсов многоядерных процессоров, что является важным аспектом для создания высокопроизводительных приложений.

Таким образом, поставленные задачи лабораторной работы были успешно решены, а результаты экспериментов подтвердили, что многопоточная сортировка значительно превосходит по эффективности однопоточную обработку данных. Дальнейшее развитие проекта может включать оптимизацию алгоритма слияния, увеличение числа потоков в зависимости от аппаратных возможностей, а также исследование альтернативных методов синхронизации для повышения производительности и масштабируемости системы.

СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ

[1] GCC Online Documentation [Электронный ресурс]. – Режим доступа: <https://gcc.gnu.org/onlinedocs/>. – Дата доступа: 02.03.2025.

[2] man7.org. «pthread_create(3)» [Электронный ресурс]. – Режим доступа: https://man7.org/linux/man-pages/man3/pthread_create.3.html. – Дата доступа: 03.03.2025.

[3] GNU Make Manual [Электронный ресурс]. – Режим доступа: <https://www.gnu.org/software/make/manual/>. – Дата доступа: 03.03.2025.

ПРИЛОЖЕНИЕ А

(обязательное)

Исходный код программы

```
#include <pthread.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <sys/time.h>

#define NUM_THREADS 20

int cmpfunc(const void *a, const void *b) {
    int int_a = *(int *)a;
    int int_b = *(int *)b;
    return int_a - int_b;
}

int *read_data(const char *filename, int *n_out) {
    FILE *fp = fopen(filename, "r");
    if (!fp) {
        perror("Ошибка открытия входного файла");
        exit(EXIT_FAILURE);
    }
    int n;
    if (fscanf(fp, "%d", &n) != 1) {
        fprintf(stderr, "Ошибка чтения количества элементов\n");
        exit(EXIT_FAILURE);
    }
    int *arr = malloc(n * sizeof(int));
    if (!arr) {
        perror("Ошибка выделения памяти");
        exit(EXIT_FAILURE);
    }
    for (int i = 0; i < n; i++) {
        if (fscanf(fp, "%d", &arr[i]) != 1) {
            fprintf(stderr, "Ошибка чтения элемента %d\n", i);
            exit(EXIT_FAILURE);
        }
    }
    fclose(fp);
    *n_out = n;
    return arr;
}

void write_array_to_file(const char *filename, int *arr, int n) {
    FILE *fp = fopen(filename, "w");
    if (!fp) {
        perror("Ошибка открытия выходного файла");
        exit(EXIT_FAILURE);
    }
    fprintf(fp, "%d\n", n);
    for (int i = 0; i < n; i++) {
        fprintf(fp, "%d ", arr[i]);
    }
    fclose(fp);
}

typedef struct {
    int *array;
    int left;
    int right;
}
```

```

} ThreadData;

void *thread_sort(void *arg) {
    ThreadData *data = (ThreadData *)arg;
    qsort(data->array + data->left, data->right - data->left, sizeof(int),
        cmpfunc);
    pthread_exit(NULL);
}

void merge(int *arr, int left, int mid, int right, int *aux) {
    int i = left, j = mid, k = left;
    while (i < mid && j < right) {
        if (arr[i] <= arr[j])
            aux[k++] = arr[i++];
        else
            aux[k++] = arr[j++];
    }
    while (i < mid)
        aux[k++] = arr[i++];
    while (j < right)
        aux[k++] = arr[j++];
    for (i = left; i < right; i++)
        arr[i] = aux[i];
}

typedef struct {
    int left;
    int right;
} Segment;

int main() {
    const char *input_file = "data.txt";
    int n;
    int *orig_array = read_data(input_file, &n);
    printf("Прочитано %d элементов из файла %s\n", n, input_file);

    int *array_multithread = malloc(n * sizeof(int));
    int *array_singlethread = malloc(n * sizeof(int));
    if (!array_multithread || !array_singlethread) {
        perror("Ошибка выделения памяти");
        exit(EXIT_FAILURE);
    }
    memcpy(array_multithread, orig_array, n * sizeof(int));
    memcpy(array_singlethread, orig_array, n * sizeof(int));
    free(orig_array);

    struct timeval start, end;
    double elapsed_multithread, elapsed_singlethread;

    pthread_t threads[NUM_THREADS];
    ThreadData threadData[NUM_THREADS];
    int seg_size = n / NUM_THREADS;
    for (int i = 0; i < NUM_THREADS; i++) {
        threadData[i].array = array_multithread;
        threadData[i].left = i * seg_size;
        threadData[i].right = (i == NUM_THREADS - 1) ? n : (i + 1) * seg_size;
    }
    gettimeofday(&start, NULL);
    for (int i = 0; i < NUM_THREADS; i++) {
        if (pthread_create(&threads[i], NULL, thread_sort, &threadData[i]) != 0) {
            perror("Ошибка создания потока");
            exit(EXIT_FAILURE);
        }
    }
}

```

```

for (int i = 0; i < NUM_THREADS; i++) {
    pthread_join(threads[i], NULL);
}
int *aux = malloc(n * sizeof(int));
if (!aux) {
    perror("Ошибка выделения памяти для вспомогательного массива");
    exit(EXIT_FAILURE);
}
Segment segments[NUM_THREADS];
for (int i = 0; i < NUM_THREADS; i++) {
    segments[i].left = threadData[i].left;
    segments[i].right = threadData[i].right;
}
int numSegments = NUM_THREADS;
while (numSegments > 1) {
    int newCount = (numSegments + 1) / 2;
    for (int i = 0; i < numSegments / 2; i++) {
        int left = segments[2 * i].left;
        int mid = segments[2 * i].right;
        int right = segments[2 * i + 1].right;
        merge(array_multithread, left, mid, right, aux);
        segments[i].left = left;
        segments[i].right = right;
    }
    if (numSegments % 2 == 1) {
        segments[newCount - 1] = segments[numSegments - 1];
    }
    numSegments = newCount;
}
gettimeofday(&end, NULL);
elapsed_multithread = (end.tv_sec - start.tv_sec) * 1000.0 +
    (end.tv_usec - start.tv_usec) / 1000.0;
printf("Время многопоточной сортировки: %.3f мс\n", elapsed_multithread);

gettimeofday(&start, NULL);
qsort(array_singlethread, n, sizeof(int), cmpfunc);
gettimeofday(&end, NULL);
elapsed_singlethread = (end.tv_sec - start.tv_sec) * 1000.0 +
    (end.tv_usec - start.tv_usec) / 1000.0;
printf("Время однопоточной сортировки: %.3f мс\n", elapsed_singlethread);

write_array_to_file("sorted2.txt", array_multithread, n);
write_array_to_file("sorted.txt", array_singlethread, n);
printf("Отсортированные массивы записаны в файлы 'sorted2.txt' "
    "(многопоточная) и 'sorted.txt' (однопоточная).\n");

free(array_multithread);
free(array_singlethread);
free(aux);
return 0;
}
#include <stdio.h>
#include <stdlib.h>
#include <time.h>

void generate_test_file(const char *filename, int n) {
    FILE *f = fopen(filename, "w");
    if (!f) {
        perror("Ошибка открытия файла для записи");
        exit(EXIT_FAILURE);
    }
    fprintf(f, "%d\n", n);
    for (int i = 0; i < n; i++) {
        int num = rand() % 1000000;

```

```

        fprintf(f, "%d ", num);
    }
    fclose(f);
    printf("Файл '%s' с %d элементами успешно создан.\n", filename, n);
}

int main(int argc, char *argv[]) {
    if (argc != 2) {
        fprintf(stderr, "Использование: %s <количество элементов>\n", argv[0]);
        return 1;
    }
    const char *filename = "data.txt";
    int n = atoi(argv[1]);
    srand(time(NULL));
    generate_test_file(filename, n);
    return 0;
}

```