**НЕГОСУДАРСТВЕННОЕ ОБРАЗОВАТЕЛЬНОЕ ЧАСТНОЕ** **УЧРЕЖДЕНИЕ ВЫСШЕГО ОБРАЗОВАНИЯ** **«МОСКОВСКИЙ ФИНАНСОВО-ПРОМЫШЛЕННЫЙ УНИВЕРСИТЕТ** **“СИНЕРГИЯ”»**

|  |  |  |
| --- | --- | --- |
| **Факультет/Институт** |  | Информационные техеологии |
|  |  | (наименование факультета/ Института) |
| **Направление/специальность** |  | **Информационные системы и программирование** |
| **подготовки:** |  | (код и наименование направления /специальности подготовки) |
| **Форма обучения:** |  | Очная |
|  |  | (очная, очно-заочная, заочная) |
|  |  |  |

**Отчет по лабораторной работе №2**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **на тему** |  | «Переменные окружения». | | |
|  |  | (наименование темы) | | |
|  |  |  | | |
| **по дисциплине** | | |  | **Системное программирование | ПМ.01 | Разработка модулей программного обеспечения для компьютерных систем** |
|  | | |  | (наименование дисциплины) |

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Обучающийся** |  | Старчик Артемий Филиппович |  |  |
|  |  | (ФИО) |  | (подпись) |
| **Группа** |  | ДКИП-206Прог |  |  |
|  |  |  |  |  |

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Преподаватель** |  | Сибирев Иван Валерьевич |  |  |
|  |  | (ФИО) |  | (подпись) |

**Москва 2025г.**

Лабораторная работа № 1

Файловые операции средствами

системных вызовов

1. Цель работы: получить навыки разработки приложений, реализующих операции с файлами средствами системных вызовов Linux API

на языке C в операционных системах семейства Linux.

7 Удалить из текста все пробелы   
1. Имя входного файла

2. Количество замен

#include <iostream>

#include <fstream>

int main(int argc, char\*\* argv) {

if (argc != 2) {

std::cerr << "Usage: " << argv[0] << " input\_file" << std::endl;

return 1;

}

std::ifstream input(argv[1]);

if (!input) {

std::cerr << "Error opening input file: " << argv[1] << std::endl;

return 1;

}

char ch;

int spaceCount = 0;

std::string outputText;

while (input.get(ch)) {

if (ch == ' ') {

spaceCount++;

} else {

outputText += ch; // Сохраняем символы, кроме пробелов

}

}

input.close();

// Вывод текста без пробелов

std::cout << outputText << std::endl;

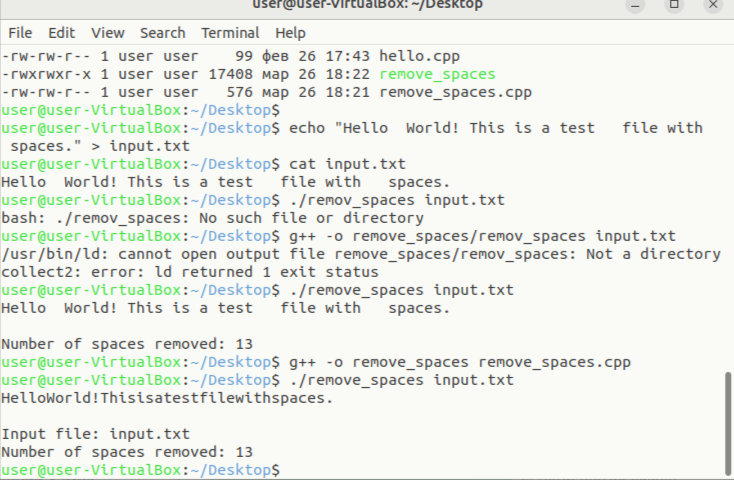
// Вывод имени входного файла и количества замен

std::cout << "Input file: " << argv[1] << std::endl;

std::cout << "Number of spaces removed: " << spaceCount << std::endl;

return 0;

}



**6. Содержание отчета**

**6.1. Цель работы**

Разработать программу на C++, которая удаляет все пробелы из текстового файла и подсчитывает их количество. Проверить работу программы на тестовых данных и проанализировать особенности работы с файлами в Linux.

**7. Контрольные вопросы**

**1. Стандартные дескрипторы файлов**

Стандартные дескрипторы файлов:

* **0** (STDIN\_FILENO) — стандартный ввод (например, клавиатура).
* **1** (STDOUT\_FILENO) — стандартный вывод (например, терминал).
* **2** (STDERR\_FILENO) — стандартный поток ошибок.

Отличие от обычных дескрипторов :  
Они автоматически открываются системой при запуске процесса и не требуют явного вызова **open()**. Они уже привязаны к устройствам ввода/вывода.

**2. Универсальность модели ввода-вывода UNIX**

В UNIX все устройства, файлы, сокеты и т.д. представлены как файлы. Это позволяет использовать единый набор системных вызовов (**open**, **read**, **write**, **close**) для работы с любыми ресурсами, что обеспечивает унифицированный интерфейс.

**3. Различия вызова open()**

* Создание нового файла :  
  Флаг **O\_CREAT** обязателен, и нужно указать права доступа (например, **0644**).  
  Пример: **open("file.txt", O\_CREAT | O\_WRONLY, 0644)**.
* Открытие существующего файла :  
  Флаг **O\_CREAT** не требуется. Если файл не существует, вызов **open()** завершится ошибкой.

**4. Форматы третьего аргумента open()**

Третий аргумент (права доступа) задается в формате 8-битного числа, например:

* **0644** — чтение/запись для владельца, только чтение для группы и других.
* **0755** — выполнение для владельца, чтение/выполнение для группы и других.

**5. Дополнительные флаги open()**

* **O\_TRUNC**: Обрезает файл до длины 0 при открытии (если файл существует).
* **O\_APPEND**: Запись в конец файла.
* **O\_EXCL**: Совместно с **O\_CREAT** — создает файл только если он не существует.
* **O\_NONBLOCK**: Открывает файл в неблокирующем режиме.

**6. Основные ошибки при открытии файла**

* **EACCES** — недостаточно прав для доступа.
* **ENOENT** — файл не существует.
* **ENOSPC** — нет свободного места на диске.
* **EEXIST** — файл уже существует (если используется **O\_CREAT | O\_EXCL**).

**7. Особенности функции read()**

* Возвращает количество считанных байт (может быть меньше, чем запрошенное).
* Возвращает **0** при достижении конца файла.
* Может вернуть **-1** при ошибке.

**8. Особенности функции write()**

* Возвращает количество записанных байт (может быть меньше, чем запрошено).
* Может вернуть **-1** при ошибке (например, полный диск).

**9. Причина вызова close()**

Функция **close()** освобождает ресурсы, связанные с файлом (например, дескриптор), и сохраняет изменения. Если не вызвать **close()**, данные могут потеряться или не быть записаны в файл.

**10. Функция lseek()**

Изменяет позицию чтения/записи в файле. Например, перемещает указатель на начало файла, конец или текущую позицию.

**11. Допустимые аргументы lseek()**

* **whence** может быть:
  + **SEEK\_SET** — смещение от начала файла.
  + **SEEK\_CUR** — смещение от текущей позиции.
  + **SEEK\_END** — смещение от конца файла.

**12. Файлы, где нельзя использовать lseek()**

* Трубопроводы (pipes).
* Сокеты (sockets).
* Другие не seekable устройства (например, терминал).

**13. Виды блокировок файла в Linux**

* Совместимые блокировки (Shared locks) : Разрешают чтение, но запрещают запись.
* Эксклюзивные блокировки (Exclusive locks) : Запрещают чтение и запись для других процессов.

**14. Структура блокировки flock\_t**

struct flock {

short l\_type; // F\_RDLCK (чтение), F\_WRLCK (запись), F\_UNLCK (снятие)

short l\_whence; // SEEK\_SET, SEEK\_CUR, SEEK\_END

off\_t l\_start; // Начальная позиция

off\_t l\_len; // Длина блокируемой области (0 — до конца файла)

pid\_t l\_pid; // PID процесса, установившего блокировку

};

**5. Блокировка с помощью fcntl()**

1. Создать структуру **flock**.
2. Вызвать **fcntl(fd, F\_SETLK, &lock)** для установки/снятия блокировки.  
   Пример:

struct flock lock;

lock.l\_type = F\_WRLCK; // Эксклюзивная блокировка

lock.l\_whence = SEEK\_SET;

lock.l\_start = 0;

lock.l\_len = 0;

fcntl(fd, F\_SETLK, &lock);

**16. Блокировка с помощью lockf()**

Пример:

lockf(fd, F\_LOCK, 0); // Эксклюзивная блокировка всего файла

lockf(fd, F\_ULOCK, 0); // Снятие блокировки

**17. Различия fcntl() и lockf()**

* **fcntl()** более гибкая (можно блокировать отдельные области файла).
* **lockf()** проще в использовании, но ограничена блокировкой всего файла.
* **fcntl()** использует структуру **flock**, а **lockf()** — простые аргументы.

**18. Снятие блокировки**

Блокировка снимается автоматически при:

* Закрытии файла (**close()**).
* Завершении процесса.

**19. Совместимость с stdio.h**

Нельзя использовать системные функции блокировки (например, **fcntl()**) с файлами, открытыми через **fopen()** из **<stdio.h>**, так как библиотека **stdio** управляет своими дескрипторами и может конфликтовать с низкоуровневыми вызовами.

**20. Особенности блокировок в stdio.h**

Библиотека **<stdio.h>** не предоставляет нативных функций блокировки файлов. Можно использовать **flockfile()** и **funlockfile()**, но они блокируют поток, а не сам файл на уровне ОС. Это ограничение делает их непригодными для межпроцессных блокировок.