МИНИСТЕРСТВО ОБРАЗОВАНИЯ РЕСПУБЛИКИ БЕЛАРУСЬ
Учреждение образования
«Витебский государственный технологический университет»

# СИСТЕМНОЕ ПРОГРАММИРОВАНИЕ

Методические указания по выполнению
лабораторных работ для слушателей ФПКиПК специальностей
1-40 01 73 «Программное обеспечение информационных систем» и
9-09-0612-02 «Программное обеспечение информационных систем»

Витебск
2024

УДК 004.9
ББК 32.97
С40

Составители:

А. Н. Бизюк, А. С. Соколова

Одобрено кафедрой «Информационные системы и технологии»
УО «ВГТУ», протокол № 12 от 18.04.2024.

Рекомендовано к изданию редакционно-издательским
советом УО «ВГТУ», протокол № 9 от 31.05.2024.

**Системное программирование** : методические указания по выполнению лабораторных работ / сост.: А. Н. Бизюк, А. С. Соколова – Витебск: УО «ВГТУ», 2024. – 62 с.

В методических указания изложены краткие теоретические сведения и задания к лабораторным работам по дисциплине «Системное программирование». Издание предназначено для слушателей переподготовки специальностей 1-40 01 73 «Программное обеспечение информационных систем» и 9-09-0612-02 «Программное обеспечение информационных систем».

**УДК 004.9**
**ББК 32.97**

# СОДЕРЖАНИЕ

# ЛАБОРАТОРНАЯ РАБОТА 1.
# РАБОТА С ДИНАМИЧЕСКОЙ ПАМЯТЬЮ В WINDOWS

**Цель работы**: создание приложения с использованием WinAPI, демонстрирующего управление динамической памятью различными способами, а также реализация пользовательской кучи для динамического распределения памяти.

## *Основные теоретические сведения*

### 1.1 Управление динамической памятью

Управление памятью в Windows может быть выполнено с использованием различных функций и API операционной системы. Рассмотрим несколько примеров кода на языке C/C++ для выделения и освобождения памяти в Windows.

Выделение памяти с использованием `malloc` и `free` (C/C++):

```c
#include <stdio.h>
#include <stdlib.h>

int main() {
    // Выделение памяти под массив целых чисел
    int *arr = (int*)malloc(5 * sizeof(int));

    if (arr == NULL) {
        printf("Не удалось выделить память\n");
        return 1;
    }

    // Использование выделенной памяти
    for (int i = 0; i < 5; i++) {
        arr[i] = i * 10;
    }

    // Освобождение памяти после использования
    free(arr);

    return 0;
}
```

Выделение памяти с использованием функции VirtualAlloc (WinAPI):

```c
#include <Windows.h>
#include <stdio.h>

int main() {
```

```
    // Выделение 1 мегабайта (1048576 байт) виртуальной памяти
    LPVOID mem = VirtualAlloc(NULL, 1048576, MEM_COMMIT, PAGE_READWRITE);

    if (mem == NULL) {
        printf("Не удалось выделить виртуальную память\n");
        return 1;
    }

    // Использование выделенной виртуальной памяти

    // Освобождение виртуальной памяти
    VirtualFree(mem, 0, MEM_RELEASE);

    return 0;
}
```

Выделение и освобождение памяти с использованием C++ операторов new и delete:

```
#include <iostream>
#include <windows.h>

int main() {
    SetConsoleOutputCP(1251);
    // Выделение памяти под одно целое число
    int *num = new int;

    // Использование выделенной памяти
    *num = 42;
    std::cout << "Значение: " << *num << std::endl;

    // Освобождение памяти
    delete num;

    return 0;
}
```

## 1.2 Функции для работы с кучей

WinAPI предоставляет ряд функций для работы с кучей (памятью, выделяемой в куче). Рассмотрим эти функции более подробно:

- **HeapCreate** – создает новую кучу.

Синтаксис:

```
HANDLE HeapCreate(DWORD flOptions, SIZE_T dwInitialSize, SIZE_T dwMaximumSize);
```

Пример:

```
HANDLE hHeap = HeapCreate(0, 0, 0);
```

- **HeapAlloc** – выделяет блок памяти из кучи.

Синтаксис:

```
LPVOID HeapAlloc(HANDLE hHeap, DWORD dwFlags, SIZE_T dwBytes);
```

Пример:

```
int* pData = (int*)HeapAlloc(hHeap, 0, sizeof(int) * 10);
```

- **HeapFree** – освобождает блок памяти, выделенный ранее с помощью `HeapAlloc`.

Синтаксис:

```
BOOL HeapFree(HANDLE hHeap, DWORD dwFlags, LPVOID lpMem);
```

Пример:

```
HeapFree(hHeap, 0, pData);
```

- **HeapReAlloc** – изменяет размер выделенного блока памяти в куче.

Синтаксис:

```
LPVOID HeapReAlloc(HANDLE hHeap, DWORD dwFlags, LPVOID lpMem, SIZE_T dwBytes);
```

Пример:

```
pData = (int*)HeapReAlloc(hHeap, 0, pData, sizeof(int) * 20);
```

- **HeapDestroy** – уничтожает кучу и освобождает все связанные с ней ресурсы.

Синтаксис:

```
BOOL HeapDestroy(HANDLE hHeap);
```

Пример:

```
HeapDestroy(hHeap);
```

- **HeapSize** – возвращает размер выделенного блока памяти в куче.

Синтаксис:

```
SIZE_T HeapSize(HANDLE hHeap, DWORD dwFlags, LPCVOID lpMem);
```

Пример:

```
SIZE_T size = HeapSize(hHeap, 0, pData);
```

- **HeapValidate** – проверяет целостность кучи и выделенных блоков.

Синтаксис:

```
BOOL HeapValidate(HANDLE hHeap, DWORD dwFlags, LPCVOID lpMem);
```

Пример:

```
if (HeapValidate(hHeap, 0, pData)) {
    printf("Куча валидна.\n");
} else {
    printf("Куча повреждена.\n");
}
```

Пример 1. Создание кучи и выделение памяти:

```
#include <Windows.h>
#include <stdio.h>

int main() {
    SetConsoleOutputCP(1251);
    // Создание кучи
    HANDLE hHeap = HeapCreate(0, 0, 0);

    if (hHeap == NULL) {
        printf("Не удалось создать кучу\n");
        return 1;
    }

    // Выделение памяти из кучи
    int *data = (int*)HeapAlloc(hHeap, 0, sizeof(int) * 5);

    if (data == NULL) {
        printf("Не удалось выделить память из кучи\n");
        HeapDestroy(hHeap);
        return 1;
    }

    // Использование выделенной памяти
    for (int i = 0; i < 5; i++) {
        data[i] = i * 10;
    }

    // Освобождение памяти
    HeapFree(hHeap, 0, data);

    // Уничтожение кучи
    HeapDestroy(hHeap);

    return 0;
}
```

Пример 2. Выделение строки в куче:

```
#include <Windows.h>
#include <stdio.h>

int main() {
    SetConsoleOutputCP(1251);
    // Создание кучи
    HANDLE hHeap = HeapCreate(0, 0, 0);

    if (hHeap == NULL) {
        printf("Не удалось создать кучу\n");
        return 1;
    }

    // Выделение строки в куче
```

```
    char *str = (char*)HeapAlloc(hHeap, 0, 256);

    if (str == NULL) {
        printf("Не удалось выделить память для строки\n");
        HeapDestroy(hHeap);
        return 1;
    }

    // Копирование строки в выделенную память
    strcpy_s(str, 256, "Пример строки в куче");

    // Использование строки

    // Освобождение памяти
    HeapFree(hHeap, 0, str);

    // Уничтожение кучи
    HeapDestroy(hHeap);

    return 0;
}
```

### *Порядок выполнения работы*

**Задание 1.** Выполнить задание согласно варианту, используя различные способы выделения и освобождения динамической памяти:

- использование `malloc`, `realloc`, `free`;
- использование `new`, `delete` (только на C++);
- использование `VirtualAlloc`, `VirtualFree`;
- использование `HeapAlloc`, `HeapFree`.

*Варианты задания:*

1. Создать динамический массив. Размер вводится пользователем. Заполнить массив случайными числами. Вывести на экран содержимое массива и сумму чисел в массиве. Прибавить сумму чисел к каждому элементу массива, вывести массив на экран. Освободить память.
2. Создать динамический массив. Размер вводится пользователем. Заполнить массив случайными числами. Вывести на экран содержимое массива и среднее значение чисел в массиве. Прибавить среднее значение к каждому элементу массива, вывести массив на экран. Освободить память.
3. Создать динамический массив. Размер вводится пользователем. Заполнить массив случайными числами. Вывести на экран содержимое массива и максимальное число в массиве. Прибавить максимальное значение к каждому элементу массива, вывести массив на экран. Освободить память.
4. Создать динамический массив. Размер вводится пользователем. Заполнить массив случайными числами. Вывести на экран содержимое массива и минимальное число в массиве. Прибавить минимальное значение к каждому элементу массива, вывести массив на экран. Освободить память.

5. Создать динамический массив. Размер вводится пользователем. Заполнить массив случайными числами. Вывести на экран содержимое массива и количество четных чисел в массиве. Прибавить это количество к каждому элементу массива, вывести массив на экран. Освободить память.

6. Создать динамический массив. Размер вводится пользователем. Заполнить массив случайными числами. Вывести на экран содержимое массива и количество нечетных чисел в массиве. Прибавить это количество к каждому элементу массива, вывести массив на экран. Освободить память.

7. Создать динамический массив. Размер вводится пользователем. Заполнить массив случайными числами. Вывести на экран исходное содержимое массива и отсортированное по возрастанию содержимое. Освободить память.

8. Создать динамический массив. Размер вводится пользователем. Заполнить массив случайными числами. Вывести на экран исходное содержимое массива и отсортированное по убыванию содержимое. Освободить память.

9. Создать динамический массив. Размер вводится пользователем. Заполнить массив случайными числами. Вывести на экран исходное содержимое массива, переставить числа в массиве в обратном порядке и вывести на экран. Освободить память.

10. Создать динамический массив. Размер вводится пользователем. Заполнить массив элементами ряда Фибоначчи. Вывести на экран исходное содержимое массива, переставить числа в массиве в обратном порядке и вывести на экран. Освободить память.

**Задание 2.** Выполнить с использованием пользовательской кучи (функции `HeapCreate`, `HeapAlloc`, `HeapReAlloc`, `HeapFree`, `HeapDestroy`):

Создать пользовательскую кучу. Создать динамический массив из указателей. Размер вводится пользователем. Для каждого элемента массива выделить память по отдельности из пользовательской кучи. Указатели на созданные элементы сохранить в массиве. Выполнить задание с этим массивом согласно варианту из задания 1. В конце освободить память, уничтожив пользовательскую кучу.

### *Контрольные вопросы*

1. *Основные концепции:*

- Что такое динамическая память и как она отличается от статической памяти?
- Какие проблемы могут возникнуть при неправильном использовании динамической памяти?

2. *Функции управления памятью:*

- Какие функции Windows API используются для работы с динамической памятью?
- В чем различие между функциями `HeapAlloc` и `VirtualAlloc`?

- Как освободить память, выделенную функцией `HeapAlloc`? А функцией `VirtualAlloc`?

3. *Кучи (Heap):*

- Что такое куча (heap) и как она используется для управления памятью?
- Как создать и уничтожить кучу с помощью Windows API?
- Какие параметры можно задать при создании кучи?

4. *Виртуальная память:*

- Что такое виртуальная память и как она управляется в Windows?
- Какие преимущества и недостатки есть у использования виртуальной памяти по сравнению с кучей?

5. *Отладка и диагностика:*

- Каковы основные инструменты и методы отладки утечек памяти в Visual Studio?
- Как обнаружить и исправить утечки памяти в программах?

6. *Оптимизация и эффективность:*

- Какие методы можно использовать для минимизации фрагментации памяти?
- Каковы основные принципы написания эффективного кода, работающего с динамической памятью?

7. *Практическое применение:*

- Опишите процесс выделения и освобождения памяти с использованием функции `HeapAlloc`.
- Приведите пример кода, использующего функции `VirtualAlloc` и `VirtualFree`.
- Какие меры предосторожности необходимо соблюдать при работе с динамической памятью?

8. *Анализ и сравнение:*

- В каких случаях предпочтительно использовать кучу, а в каких – виртуальную память?
- Какое влияние на производительность оказывает использование кучи и виртуальной памяти?

# ЛАБОРАТОРНАЯ РАБОТА 2. ОПЕРАЦИИ С ФАЙЛАМИ В WINDOWS

**Цель работы**: изучение операций с файлами в операционной системе Windows с использованием системных вызовов и API. Приобретение навыков работы с файлами на низком уровне, включая создание, чтение, запись, удаление и управление файловыми атрибутами.

### *Основные теоретические сведения*

В Windows API (WinAPI) существует множество функций для работы с файлами и каталогами. Рассмотрим примеры работы с наиболее распространенными из них.

Пример кода на C/C++ для открытия файла, чтения из него и закрытия файла с использованием WinAPI:

```
#include <windows.h>
#include "iostream"

int main() {
    SetConsoleOutputCP(1251);
    // Открыть файл для чтения
    HANDLE hFile = CreateFile("example.txt", GENERIC_READ, 0, NULL,
OPEN_EXISTING, FILE_ATTRIBUTE_NORMAL, NULL);

    if (hFile != INVALID_HANDLE_VALUE) {
        // Чтение данных из файла
        char buffer[1024];
        DWORD bytesRead;
        if (ReadFile(hFile, buffer, sizeof(buffer), &bytesRead, NULL)) {
            // Обработка данных
        }

        // Закрыть файл
        CloseHandle(hFile);
    } else {
        // Обработка ошибки открытия файла
        std::cout << "Не удалось открыть файл" << std::endl;
    }

    return 0;
}
```

Создание файла в Windows с использованием WinAPI можно выполнить с помощью функции `CreateFile`. Она имеет множество параметров, которые позволяют настроить операцию создания или открытия файла подробно. Общий вид прототипа функции:

```
HANDLE CreateFile(
  LPCTSTR               lpFileName,
  DWORD                 dwDesiredAccess,
  DWORD                 dwShareMode,
  LPSECURITY_ATTRIBUTES lpSecurityAttributes,
  DWORD                 dwCreationDisposition,
  DWORD                 dwFlagsAndAttributes,
  HANDLE                hTemplateFile
);
```

- **lpFileName (LPCTSTR)**: представляет собой путь к файлу или устройству. Может быть строкой в формате многобайтовых символов (ANSI) или широких символов (Unicode), в зависимости от того, какой тип используется: LPCSTR для ANSI или LPCWSTR для Unicode.

Примеры:

ANSI: `"C:\\example.txt"`

Unicode: `L"C:\\example.txt"`

- **dwDesiredAccess (DWORD):** определяет режим доступа к файлу. Это флаги, которые указывают, какие операции можно выполнять с файлом. Например:
  - GENERIC_READ: разрешает чтение файла.
  - GENERIC_WRITE: разрешает запись в файл.

Можно комбинировать флаги с помощью операции побитового ИЛИ (|) для указания нескольких прав доступа.

- **dwShareMode (DWORD):** определяет режим совместного доступа к файлу. Это флаги, указывающие, какие типы доступа можно разрешить другим процессам. Например:
  - FILE_SHARE_READ: разрешает другим процессам читать файл.
  - FILE_SHARE_WRITE: разрешает другим процессам записывать в файл.
- **lpSecurityAttributes (LPSECURITY_ATTRIBUTES):** позволяет указать атрибуты безопасности для создаваемого файла. Если вы не хотите задавать специальные атрибуты безопасности, можете передать NULL.
- **dwCreationDisposition (DWORD):** определяет, что делать, если файл уже существует или не существует. Некоторые из возможных значений:
  - CREATE_NEW: создать новый файл (ошибка, если файл уже существует).
  - CREATE_ALWAYS: создать новый файл или перезаписать существующий.
  - OPEN_EXISTING: открыть только существующий файл.
  - OPEN_ALWAYS: открыть существующий файл или создать новый, если его нет.
- **dwFlagsAndAttributes (DWORD):** позволяет указать дополнительные атрибуты файла. Например:
  - FILE_ATTRIBUTE_NORMAL: обычный файл без специальных атрибутов.
  - FILE_ATTRIBUTE_READONLY: файл только для чтения.
- **hTemplateFile (HANDLE):** представляет собой дескриптор файла-шаблона, который используется для определения атрибутов и флагов создаваемого файла. Обычно передается как NULL.

После вызова функции `CreateFile`, она возвращает дескриптор файла (или специальное значение `INVALID_HANDLE_VALUE` в случае ошибки). Этот

дескриптор файла используется для дальнейших операций с файлом, таких как чтение, запись и закрытие.

Пример демонстрирующий, как создать новый файл:

```
#include <windows.h>
#include <iostream>

int main() {
    SetConsoleOutputCP(1251);
    LPCSTR fileName = "C:\\путь\\к\\вашему\\файлу\\example.txt"; // Замените
путь и имя файла по вашему усмотрению

    // Открыть или создать файл для записи
    HANDLE hFile = CreateFile(fileName, GENERIC_WRITE, 0, NULL, CREATE_NEW,
FILE_ATTRIBUTE_NORMAL, NULL);

    if (hFile != INVALID_HANDLE_VALUE) {
        // Файл успешно создан
        std::cout << "Файл '" << fileName << "' успешно создан." << std::endl;

        // Закрыть файл
        CloseHandle(hFile);
    } else {
        // Обработать ошибку
        DWORD error = GetLastError();
        if (error == ERROR_FILE_EXISTS) {
            std::cout << "Файл '" << fileName << "' уже существует." <<
std::endl;
        } else {
            std::cerr << "Не удалось создать файл '" << fileName << "'. Ошибка
" << error << std::endl;
        }
    }

    return 0;
}
```

Функция `ReadFile` в Windows API используется для чтения данных из файла. Её прототип:

```
BOOL ReadFile(
  HANDLE       hFile,
  LPVOID       lpBuffer,
  DWORD        nNumberOfBytesToRead,
  LPDWORD      lpNumberOfBytesRead,
  LPOVERLAPPED lpOverlapped
);
```

- **`hFile (HANDLE)`:** дескриптор файла, который был получен при открытии файла с помощью функции `CreateFile`. Указывает на файл, из которого будут читаться данные.

- **lpBuffer (LPVOID):** указатель на буфер, в который будут считываться данные из файла. Этот буфер должен быть предварительно выделен, и его размер должен быть не меньше, чем `nNumberOfBytesToRead`, чтобы вместить считанные данные.
- **nNumberOfBytesToRead (DWORD):** указывает количество байт, которые нужно прочитать из файла и поместить в буфер `lpBuffer`.
- **lpNumberOfBytesRead (LPDWORD):** указатель на переменную, в которую будет записано фактическое количество байт, которые были прочитаны из файла. Эта информация может быть полезной, чтобы узнать, сколько данных было считано.
- **lpOverlapped (LPOVERLAPPED):** используется для асинхронного чтения файлов и обычно оставляется равным `NULL` для синхронного чтения.

После вызова функции `ReadFile`, она возвращает `TRUE`, если чтение было успешным, и `FALSE`, если произошла ошибка. Если чтение было успешным, данные будут доступны в буфере `lpBuffer`, и количество считанных байт будет записано в переменную, на которую указывает `lpNumberOfBytesRead`.

Пример, демонстрирующий, как читать данные из файла:

```
#include <windows.h>
#include <iostream>

int main() {
    SetConsoleOutputCP(1251);
    LPCSTR fileName = "C:\\путь\\к\\вашему\\файлу\\example.txt"; // Замените
путь и имя файла по вашему усмотрению

    // Открыть файл для чтения
    HANDLE hFile = CreateFile(fileName, GENERIC_READ, FILE_SHARE_READ, NULL,
OPEN_EXISTING, FILE_ATTRIBUTE_NORMAL, NULL);

    if (hFile != INVALID_HANDLE_VALUE) {
        char buffer[1024];
        DWORD bytesRead;

        if (ReadFile(hFile, buffer, sizeof(buffer), &bytesRead, NULL)) {
            // Чтение прошло успешно
            if (bytesRead > 0) {
                // Вывести прочитанные данные на экран
                std::cout << "Прочитано " << bytesRead << " байт: " << buffer
<< std::endl;
            } else {
                std::cout << "Файл пуст." << std::endl;
            }
        } else {
            // Обработать ошибку чтения
            DWORD error = GetLastError();
            std::cerr << "Ошибка чтения файла. Код ошибки: " << error <<
std::endl;
```

```
        }

        // Закрыть файл
        CloseHandle(hFile);
    } else {
        // Обработать ошибку открытия файла
        DWORD error = GetLastError();
        std::cerr << "Не удалось открыть файл. Код ошибки: " << error <<
std::endl;
    }

    return 0;
}
```

Запись данных в файл в Windows с использованием WinAPI выполняется с помощью функции `WriteFile`. Эта функция позволяет записать данные из буфера в файл. Её прототип:

```
BOOL WriteFile(
  HANDLE       hFile,
  LPCVOID      lpBuffer,
  DWORD        nNumberOfBytesToWrite,
  LPDWORD      lpNumberOfBytesWritten,
  LPOVERLAPPED lpOverlapped
);
```

- **hFile (HANDLE):** дескриптор файла, указывающий на файл, в который будут записываться данные.
- **lpBuffer (LPCVOID):** указатель на буфер, из которого будут записываться данные в файл. Этот буфер должен содержать данные, которые вы хотите записать.
- **nNumberOfBytesToWrite (DWORD):** параметр указывает количество байт, которые нужно записать в файл из буфера `lpBuffer`.
- **lpNumberOfBytesWritten (LPDWORD):** указатель на переменную, в которую будет записано фактическое количество байт, которые были успешно записаны в файл.
- **lpOverlapped (LPOVERLAPPED):** параметр используется для асинхронной записи файлов и обычно оставляется равным `NULL` для синхронной записи.

После вызова функции `WriteFile`, она возвращает `TRUE`, если запись была успешной, и `FALSE`, если произошла ошибка. Если запись была успешной, количество фактически записанных байт будет записано в переменную, на которую указывает `lpNumberOfBytesWritten`.

Пример записи данных в файл:

```
#include <windows.h>
#include <iostream>
```

```
int main() {
    SetConsoleOutputCP(1251);
    LPCSTR fileName = "C:\\путь\\к\\вашему\\файлу\\example.txt"; // Замените
путь и имя файла по вашему усмотрению

    // Открыть файл для записи
    HANDLE hFile = CreateFile(fileName, GENERIC_WRITE, 0, NULL, CREATE_ALWAYS,
FILE_ATTRIBUTE_NORMAL, NULL);

    if (hFile != INVALID_HANDLE_VALUE) {
        const char *data = "Пример записи в файл.";
        DWORD bytesWritten;

        if (WriteFile(hFile, data, strlen(data), &bytesWritten, NULL)) {
            // Запись прошла успешно
            std::cout << "Записано " << bytesWritten << " байт." << std::endl;
        } else {
            // Обработать ошибку записи
            DWORD error = GetLastError();
            std::cerr << "Ошибка записи в файл. Код ошибки: " << error <<
std::endl;
        }

        // Закрыть файл
        CloseHandle(hFile);
    } else {
        // Обработать ошибку открытия файла
        DWORD error = GetLastError();
        std::cerr << "Не удалось открыть/создать файл. Код ошибки: " << error
<< std::endl;
    }

    return 0;
}
```

Для удаления файла в Windows с использованием WinAPI можно использовать функцию `DeleteFile`. Её прототип:

```
BOOL DeleteFile(
  LPCTSTR lpFileName
);
```

- **lpFileName (LPCTSTR):** представляет собой строку, содержащую путь к файлу, который вы хотите удалить. Обычно это Unicode-строка, представленная как LPCWSTR для широких символов или LPCTSTR для многобайтовых символов.

Пример использования функции `DeleteFile` для удаления файла:

```
#include <windows.h>
#include <iostream>

int main() {
```

```
    SetConsoleOutputCP(1251);
    LPCSTR fileName = "C:\\путь\\к\\вашему\\файлу\\example.txt"; // Замените
путь и имя файла по вашему усмотрению

    if (DeleteFile(fileName)) {
        std::cout << "Файл '" << fileName << "' успешно удалён." << std::endl;
    } else {
        DWORD error = GetLastError();
        std::cerr << "Не удалось удалить файл '" << fileName << "'. Код ошибки:
" << error << std::endl;
    }

    return 0;
}
```

Для переименования и перемещения файла в Windows с использованием WinAPI, можно воспользоваться функцией `MoveFile` или её вариациями. Например:

```
#include <windows.h>
#include <iostream>

int main() {
    SetConsoleOutputCP(1251);
    LPCSTR oldFileName = "C:\\путь\\к\\старому\\файлу\\old_file.txt"; //
Замените старый путь и имя файла
    LPCSTR newFileName = "C:\\путь\\к\\новому\\файлу\\new_file.txt"; //
Замените новый путь и имя файла

    if (MoveFile(oldFileName, newFileName)) {
        std::cout << "Файл успешно переименован в '" << newFileName << "'." <<
std::endl;
    } else {
        DWORD error = GetLastError();
        std::cerr << "Не удалось переименовать файл. Код ошибки: " << error <<
std::endl;
    }

    return 0;
}
```

Можно управлять атрибутами файла в Windows с использованием WinAPI с помощью функций `GetFileAttributes` и `SetFileAttributes`.

Пример получения атрибутов файла:

```
#include <windows.h>
#include <iostream>

int main() {
    SetConsoleOutputCP(1251);
    LPCSTR fileName = "C:\\путь\\к\\вашему\\файлу\\example.txt"; // Замените
путь и имя файла по вашему усмотрению
```

```
    DWORD fileAttributes = GetFileAttributes(fileName);

    if (fileAttributes != INVALID_FILE_ATTRIBUTES) {
        if (fileAttributes & FILE_ATTRIBUTE_DIRECTORY) {
            std::cout << "Файл '" << fileName << "' - это каталог." <<
std::endl;
        } else {
            std::cout << "Файл '" << fileName << "' - это обычный файл." <<
std::endl;
        }

        if (fileAttributes & FILE_ATTRIBUTE_READONLY) {
            std::cout << "Файл '" << fileName << "' доступен только для
чтения." << std::endl;
        }
    } else {
        DWORD error = GetLastError();
        std::cerr << "Не удалось получить атрибуты файла. Код ошибки: " <<
error << std::endl;
    }

    return 0;
}
```

Пример установки атрибутов файла:

```
#include <windows.h>
#include <iostream>

int main() {
    SetConsoleOutputCP(1251);
    LPCSTR fileName = "C:\\путь\\к\\вашему\\файлу\\example.txt"; // Замените
путь и имя файла по вашему усмотрению

    // Получить текущие атрибуты файла
    DWORD fileAttributes = GetFileAttributes(fileName);

    if (fileAttributes != INVALID_FILE_ATTRIBUTES) {
        // Установить новые атрибуты (например, сделать файл доступным только
для чтения)
        fileAttributes |= FILE_ATTRIBUTE_READONLY;

        if (SetFileAttributes(fileName, fileAttributes)) {
            std::cout << "Атрибуты файла '" << fileName << "' успешно
изменены." << std::endl;
        } else {
            DWORD error = GetLastError();
            std::cerr << "Не удалось изменить атрибуты файла. Код ошибки: " <<
error << std::endl;
        }
    } else {
        DWORD error = GetLastError();
```

```
        std::cerr << "Не удалось получить атрибуты файла. Код ошибки: " <<
error << std::endl;
    }

    return 0;
}
```

Для проверки существования файла в Windows с использованием WinAPI можно воспользоваться функцией `PathFileExists` из библиотеки `Shlwapi.h` или функцией `GetFileAttributes` и проверкой возвращаемого значения.

Пример использования `PathFileExists`:

Для компиляции примера в Code::Blocks необходимо подключить библиотеку в настройках компилятора:

1. Project → Build options → Linker settings → Add
2. Добавить «shlwapi».

```
#include <windows.h>
#include <Shlwapi.h> // Для PathFileExists

int main() {
    SetConsoleOutputCP(1251);
    PCSTR fileName = "C:\\путь\\к\\вашему\\файлу\\example.txt"; // Замените
путь и имя файла по вашему усмотрению

    if (PathFileExists(fileName)) {
        // Файл существует
        printf("Файл '%s' существует.\n", fileName);
    } else {
        // Файл не существует
        printf("Файл '%s' не существует.\n", fileName);
    }

    return 0;
}
```

Пример использования `GetFileAttributes`:

```
#include <windows.h>
#include <iostream>

int main() {
    SetConsoleOutputCP(1251);
    LPCSTR fileName = "C:\\путь\\к\\вашему\\файлу\\example.txt"; // Замените
путь и имя файла по вашему усмотрению

    DWORD fileAttributes = GetFileAttributes(fileName);

    if (fileAttributes != INVALID_FILE_ATTRIBUTES && !(fileAttributes &
FILE_ATTRIBUTE_DIRECTORY)) {
        // Файл существует
        std::cout << "Файл '" << fileName << "' существует." << std::endl;
    } else {
```

```
        // Файл не существует или это каталог
        std::cout << "Файл '" << fileName << "' не существует или это каталог."
<< std::endl;
    }

    return 0;
}
```

Для поиска файлов в Windows с использованием WinAPI, можно воспользоваться функцией `FindFirstFile` и её итеративной версией `FindNextFile`. Эти функции позволяют искать файлы по определенным критериям в указанном каталоге. Пример поиска файлов:

```
#include <windows.h>
#include <iostream>

int main() {
    SetConsoleOutputCP(1251);
    LPCSTR searchPath = "C:\\путь\\к\\каталогу\\*.*"; // Замените путь к
каталогу по вашему усмотрению

    WIN32_FIND_DATA findFileData;
    HANDLE hFind = FindFirstFile(searchPath, &findFileData);

    if (hFind != INVALID_HANDLE_VALUE) {
        do {
            if (!(findFileData.dwFileAttributes & FILE_ATTRIBUTE_DIRECTORY)) {
                // Элемент найден, и это не каталог
                std::cout << "Имя файла: " << findFileData.cFileName <<
std::endl;
            }
        } while (FindNextFile(hFind, &findFileData) != 0);

        FindClose(hFind); // Закрыть дескриптор поиска
    } else {
        DWORD error = GetLastError();
        std::cerr << "Не удалось выполнить поиск файлов. Код ошибки: " << error
<< std::endl;
    }

    return 0;
}
```

Для работы с каталогами в Windows с использованием WinAPI вы можете использовать функции, такие как `CreateDirectory`, `RemoveDirectory`, `SetCurrentDirectory`, и `GetCurrentDirectory`.

Пример создания каталога:

```
#include <windows.h>
#include <iostream>

int main() {
```

```
    SetConsoleOutputCP(1251);
    LPCSTR directoryName = "C:\\путь\\к\\новому\\каталогу"; // Замените путь и
имя каталога по вашему усмотрению

    if (CreateDirectory(directoryName, NULL)) {
        std::cout << "Каталог '" << directoryName << "' успешно создан." <<
std::endl;
    } else {
        DWORD error = GetLastError();
        if (error == ERROR_ALREADY_EXISTS) {
            std::cerr << "Каталог '" << directoryName << "' уже существует." <<
std::endl;
        } else {
            std::cerr << "Не удалось создать каталог. Код ошибки: " << error <<
std::endl;
        }
    }

    return 0;
}
```

Пример удаления каталога:

```
#include <windows.h>
#include <iostream>

int main() {
    SetConsoleOutputCP(1251);
    LPCSTR directoryName = "C:\\путь\\к\\каталогу\\для_удаления"; // Замените
путь и имя каталога по вашему усмотрению

    if (RemoveDirectory(directoryName)) {
        std::cout << "Каталог '" << directoryName << "' успешно удалён." <<
std::endl;
    } else {
        DWORD error = GetLastError();
        std::cerr << "Не удалось удалить каталог. Код ошибки: " << error <<
std::endl;
    }

    return 0;
}
```

Установка текущего каталога:

```
#include <windows.h>
#include <iostream>

int main() {
    SetConsoleOutputCP(1251);
    LPCSTR newDirectory = "C:\\путь\\к\\новому\\каталогу"; // Замените путь и имя
каталога по вашему усмотрению

    if (SetCurrentDirectory(newDirectory)) {
        std::cout << "Текущий каталог установлен в '" << newDirectory << "'." <<
std::endl;
```

```
        } else {
            DWORD error = GetLastError();
            std::cerr << "Не удалось установить текущий каталог. Код ошибки: " <<
    error << std::endl;
        }

        return 0;
    }
```

Пример получения текущего каталога:

```
#include <windows.h>
#include <iostream>

int main() {
    SetConsoleOutputCP(1251);
    CHAR currentDirectory[MAX_PATH];

    if (GetCurrentDirectory(MAX_PATH, currentDirectory) != 0) {
        std::cout << "Текущий каталог: " << currentDirectory << std::endl;
    } else {
        DWORD error = GetLastError();
        std::cerr << "Не удалось получить текущий каталог. Код ошибки: " <<
error << std::endl;
    }

    return 0;
}
```

Функции CreateDirectory и RemoveDirectory могут возвращать ошибку ERROR_ALREADY_EXISTS, если каталог уже существует.

Функция `GetFileInformationByHandle` предоставляет информацию о файле на основе его дескриптора. Эта функция предоставляет подробные сведения о файле, такие как размер, атрибуты, время создания и др. Пример использования функции `GetFileInformationByHandle`:

```
#include <windows.h>
#include <iostream>
#include <ctime>

int main() {
    SetConsoleOutputCP(1251);
    LPCSTR fileName = "C:\\путь\\к\\вашему\\файлу\\example.txt"; // Замените
путь и имя файла по вашему усмотрению

    // Открыть файл для получения дескриптора
    HANDLE hFile = CreateFile(fileName, GENERIC_READ, FILE_SHARE_READ, NULL,
OPEN_EXISTING, 0, NULL);

    if (hFile != INVALID_HANDLE_VALUE) {
        BY_HANDLE_FILE_INFORMATION fileInfo;
        if (GetFileInformationByHandle(hFile, &fileInfo)) {
            std::cout << "Имя файла: " << fileName << std::endl;
```

```
            std::cout << "Размер файла: " << fileInfo.nFileSizeLow << " байт"
<< std::endl;
            std::cout << "Атрибуты файла: " << fileInfo.dwFileAttributes <<
std::endl;

            SYSTEMTIME sysTime;
            FileTimeToSystemTime(&fileInfo.ftCreationTime, &sysTime);
            std::cout << "Дата создания: " << sysTime.wDay << "." <<
sysTime.wMonth << "." << sysTime.wYear
                << " " << sysTime.wHour << ":" << sysTime.wMinute << ":" <<
sysTime.wSecond << std::endl;

            FileTimeToSystemTime(&fileInfo.ftLastAccessTime, &sysTime);
            std::cout << "Последний доступ: " << sysTime.wDay << "." <<
sysTime.wMonth << "." << sysTime.wYear
                << " " << sysTime.wHour << ":" << sysTime.wMinute << ":" <<
sysTime.wSecond << std::endl;

            FileTimeToSystemTime(&fileInfo.ftLastWriteTime, &sysTime);
            std::cout << "Последнее изменение: " << sysTime.wDay << "." <<
sysTime.wMonth << "."
                << sysTime.wYear << " " << sysTime.wHour << ":" <<
sysTime.wMinute << ":" << sysTime.wSecond << std::endl;
        } else {
            DWORD error = GetLastError();
            std::cerr << "Не удалось получить информацию о файле. Код ошибки: "
<< error << std::endl;
        }

        // Закрыть дескриптор файла
        CloseHandle(hFile);
    } else {
        DWORD error = GetLastError();
        std::cerr << "Не удалось открыть файл. Код ошибки: " << error <<
std::endl;
    }

    return 0;
}
```

Для закрытия файла в Windows с использованием WinAPI, можно использовать функцию `CloseHandle`, передав в неё дескриптор файла, который был получен при его открытии с помощью функции `CreateFile` или других функций, возвращающих дескриптор файла. Пример закрытия файла:

```
#include <windows.h>
#include <iostream>

int main() {
    SetConsoleOutputCP(1251);
    LPCSTR fileName = "C:\\путь\\к\\вашему\\файлу\\example.txt"; // Замените
путь и имя файла по вашему усмотрению
```

```
    // Открыть файл для чтения
    HANDLE hFile = CreateFile(fileName, GENERIC_READ, FILE_SHARE_READ, NULL,
OPEN_EXISTING, 0, NULL);

    if (hFile != INVALID_HANDLE_VALUE) {
        // Выполняйте операции с файлом

        // Закрыть дескриптор файла
        CloseHandle(hFile);
        std::cout << "Файл закрыт." << std::endl;
    } else {
        DWORD error = GetLastError();
        std::cerr << "Не удалось открыть файл. Код ошибки: " << error <<
std::endl;
    }

    return 0;
}
```

### *Порядок выполнения работы*

**Задание 1.** Написать программу на языке C или C++ которая:

- Создает каталог с названием группы.
- Создает в этом каталоге файл с именем, соответствующем вашей фамилии.
- Записывает в этот файл текущую дату.
- Считывает содержимое файла и выводит на экран.
- Переименовывает файл, добавляя к названию инициалы.
- Определяет размер файла и выводит на экран.
- Удаляет ранее созданные файл и каталог.
- Выводит на экран список файлов и каталогов на диске C.

**Задание 2.** Написать программу на языке C или C++ согласно варианту. Все действия с файлами и каталогами осуществлять с помощью функций WinAPI.

*Варианты задания:*

1. Создать программу для чтения содержимого текстового файла и подсчета количества слов в нем с использованием WinAPI.
2. Написать утилиту, которая будет сравнивать два текстовых файла и определять, совпадают ли они.
3. Написать программу для поиска и замены заданной строки в текстовом файле с использованием функций WinAPI.
4. Создать программу, которая будет выводить список всех файлов и поддиректорий в заданной директории с указанием их размеров.
5. Написать утилиту для создания списка файлов в заданной директории и сохранения его в текстовом файле.

6. Разработать приложение для сортировки содержимого текстового файла в алфавитном порядке и сохранения результата в новом файле с использованием WinAPI.

7. Создать утилиту для поиска дубликатов файлов в заданной директории и ее поддиректориях.

8. Разработать программу для сравнения содержимого двух текстовых файлов и вывода различий на экран.

9. Написать приложение для поиска всех файлов в заданной директории и ее поддиректориях. Результат поиска должен выводиться на экран.

10. Напишите утилиту для создания списка файлов в заданной директории и сохранения его в текстовом файле.

### *Контрольные вопросы*

1. *Теоретические основы:*

- Что такое файловая система и какие виды файловых систем поддерживаются в Windows?
- Какие основные операции с файлами поддерживаются в Windows API?
- В чем различие между системными вызовами и API-функциями?

2. *Основные API-функции:*

- Какие функции Windows API используются для создания и открытия файла? Приведите примеры.
- Как с помощью Windows API можно прочитать данные из файла и записать данные в файл? Опишите процесс.
- Какие функции используются для удаления файла в Windows?

3. *Работа с атрибутами файлов:*

- Какие атрибуты файлов поддерживаются в Windows?
- Как изменить атрибуты файла с помощью Windows API?
- Какой системный вызов позволяет проверить существование файла и его атрибуты?

4. *Обработка ошибок:*

- Какие наиболее распространенные ошибки могут возникать при работе с файлами в Windows и как их обрабатывать?
- Как в программе на C/C++ отловить и корректно обработать ошибку «файл не найден»?
- Что такое дескриптор файла и как правильно с ним работать, чтобы избежать утечек ресурсов?

5. *Практическая реализация:*

- Какой порядок выполнения операций с файлами в типичной программе на C/C++ с использованием Windows API?
- Приведите пример кода на C/C++, который создает файл, записывает в него данные и затем закрывает его.
- Какие меры предосторожности нужно соблюдать при работе с файлами, чтобы избежать повреждения данных?

6. *Анализ и отладка:*

- Как можно отладить программу, работающую с файлами, для выявления и устранения ошибок?
- Какие инструменты Windows помогают мониторить операции с файлами (например, Process Monitor)?
- Какие существуют методы проверки целостности файлов после выполнения операций записи?

# ЛАБОРАТОРНАЯ РАБОТА 3. УПРАВЛЕНИЕ ПРОЦЕССАМИ В WINDOWS

**Цель работы**: Изучение механизмов управления процессами в операционной системе Windows, приобретение навыков программирования для создания, завершения, приостановки и возобновления процессов, а также для получения информации о процессах с использованием системных вызовов Windows API.

## *Основные теоретические сведения*

Управление процессами в Windows API (WinAPI) включает в себя создание, управление и взаимодействие с процессами и потоками в операционной системе Windows. Ниже приведены основные функции и концепции, используемые для работы с процессами в WinAPI.

Создание процессов в операционной системе Windows с использованием Windows API (WinAPI) осуществляется с помощью функции `CreateProcess`. Эта функция позволяет запустить новый процесс и настроить различные атрибуты его выполнения. Пример использования функции `CreateProcess` на языке C++:

```
#include <windows.h>
#include <tchar.h>

int main() {
    SetConsoleOutputCP(1251);
    // Имя исполняемого файла, который нужно запустить
    LPCTSTR applicationName = _T("C:\\Path\\To\\Your\\Program.exe");

    // Командная строка для передачи процессу
    LPTSTR commandLine = NULL;

    // Защитные атрибуты процесса и потока (обычно NULL)
    LPSECURITY_ATTRIBUTES processAttributes = NULL;
    LPSECURITY_ATTRIBUTES threadAttributes = NULL;

    // Флаги создания процесса
    BOOL inheritHandles = FALSE;
```

```
    DWORD creationFlags = 0;
    LPVOID environment = NULL;
    LPCTSTR currentDirectory = NULL;
    STARTUPINFO startupInfo;
    PROCESS_INFORMATION processInfo;

    // Заполнение структуры STARTUPINFO
    ZeroMemory(&startupInfo, sizeof(STARTUPINFO));
    startupInfo.cb = sizeof(STARTUPINFO);

    // Создание нового процесса
    BOOL result = CreateProcess(
        applicationName,     // Имя исполняемого файла
        commandLine,         // Командная строка
        processAttributes,   // Атрибуты процесса
        threadAttributes,    // Атрибуты потока
        inheritHandles,      // Флаг наследования дескрипторов
        creationFlags,       // Флаги создания процесса
        environment,         // Переменные окружения (обычно NULL)
        currentDirectory,    // Текущий рабочий каталог (обычно NULL)
        &startupInfo,        // Структура STARTUPINFO
        &processInfo         // Структура PROCESS_INFORMATION
    );

    if (result) {
        // Процесс успешно создан
        // Вы можете получить информацию о процессе, используя processInfo

        // Закрыть дескрипторы, чтобы избежать утечек ресурсов
        CloseHandle(processInfo.hProcess);
        CloseHandle(processInfo.hThread);
    } else {
        // Произошла ошибка при создании процесса
        DWORD error = GetLastError();
        // Обработка ошибки
    }

    return 0;
}
```

Функция `CreateProcess` возвращает `TRUE`, если процесс успешно создан, и `FALSE`, если произошла ошибка. В случае ошибки, можно использовать `GetLastError` для получения кода ошибки и дальнейшей обработки.

Функция `CreateProcess` имеет множество параметров для настройки создания нового процесса. Основные из них:

- **`lpApplicationName (LPCTSTR)`**: имя исполняемого файла, который нужно запустить. Этот параметр может быть `NULL`, если имя исполняемого файла включено в строку командной строки.
- **`lpCommandLine (LPTSTR)`**: командная строка, передаваемая запускаемому процессу. Этот параметр может содержать имя исполняемого

файла и его аргументы. Если `lpApplicationName` не является `NULL`, то `lpCommandLine` будет использоваться для передачи аргументов командной строки. В противном случае, `lpCommandLine` должен содержать полную команду.

- **`lpProcessAttributes (LPSECURITY_ATTRIBUTES)`**: атрибуты безопасности процесса. Обычно устанавливаются в `NULL` для использования атрибутов по умолчанию.
- **`lpThreadAttributes (LPSECURITY_ATTRIBUTES)`**: атрибуты безопасности потока. Обычно устанавливаются в `NULL` для использования атрибутов по умолчанию.
- **`bInheritHandles (BOOL)`**: флаг, указывающий, должны ли дескрипторы открытых файлов и другие ресурсы наследоваться процессом, созданным функцией `CreateProcess`.
- **`dwCreationFlags (DWORD)`**: флаги создания процесса, определяющие различные параметры и поведение процесса. Например, вы можете использовать `CREATE_NEW_CONSOLE`, чтобы создать новое окно консоли для процесса.
- **`lpEnvironment (LPVOID)`**: указатель на блок переменных окружения, которые будут использоваться в новом процессе. Обычно устанавливается в `NULL`, чтобы процесс унаследовал текущее окружение.
- **`lpCurrentDirectory (LPCTSTR)`**: текущий рабочий каталог для нового процесса. Обычно устанавливается в `NULL`, чтобы процесс использовал текущий рабочий каталог родительского процесса.
- **`lpStartupInfo (LPSTARTUPINFO)`**: указатель на структуру `STARTUPINFO`, которая содержит информацию о создаваемом процессе, такую как дескрипторы для ввода, вывода и ошибок.
- **`lpProcessInformation (LPPROCESS_INFORMATION)`**: указатель на структуру `PROCESS_INFORMATION`, в которой будут возвращены дескрипторы процесса и потока после успешного создания процесса.

Для получения информации о процессе в операционной системе Windows с помощью Windows API (WinAPI) можно использовать несколько функций и структур данных. Вот основные средства для получения информации о процессе:

- **`OpenProcess`**: позволяет открыть существующий процесс и получить дескриптор процесса (`HANDLE`), который можно использовать для выполнения различных операций над процессом. Эта функция принимает в качестве параметров идентификатор процесса (`PID`) и права доступа.

```
HANDLE hProcess = OpenProcess(PROCESS_QUERY_INFORMATION | PROCESS_VM_READ,
FALSE, processId);
```

- **`GetProcessId`**: используется для получения идентификатора процесса (PID) по его дескриптору.

```
DWORD processId = GetProcessId(hProcess);
```

- **`GetProcessTimes`**: позволяет получить информацию о времени выполнения процесса, включая время начала выполнения и использования ЦП.

```
FILETIME creationTime, exitTime, kernelTime, userTime;
if (GetProcessTimes(hProcess, &creationTime, &exitTime, &kernelTime,
&userTime)) {
    // Обработка информации о времени выполнения процесса
}
```

- **`GetProcessMemoryInfo`**: позволяет получить информацию о потреблении памяти процессом.

```
PROCESS_MEMORY_COUNTERS memInfo;
if (GetProcessMemoryInfo(hProcess, &memInfo, sizeof(memInfo))) {
    // Обработка информации о потреблении памяти процессом
}
```

- **`QueryFullProcessImageName`**: используется для получения полного пути к исполняемому файлу процесса.

```
TCHAR processPath[MAX_PATH];
DWORD pathSize = sizeof(processPath) / sizeof(TCHAR);
if (QueryFullProcessImageName(hProcess, 0, processPath, &pathSize)) {
    // Обработка пути к исполняемому файлу процесса
}
```

- **`CloseHandle`**: используется для освобождения ресурсов, связанных с дескриптором процесса.

```
CloseHandle(hProcess);
```

В приведенных примерах `hProcess` – это дескриптор открытого процесса, а `processId` – идентификатор процесса.

Управление приоритетом процессов в операционной системе Windows можно выполнять с помощью Windows API (WinAPI). Приоритет процесса определяет, как операционная система распределяет процессорное время между процессами. Основные функции и концепции, связанные с управлением приоритетом процессов:

- **Установка приоритета процесса**. Для установки приоритета процесса используется функция `SetPriorityClass`. Эта функция изменяет приоритет выполнения всего процесса.

```
BOOL success = SetPriorityClass(GetCurrentProcess(), priority);
```

Здесь priority может принимать одно из следующих значений:

  - `HIGH_PRIORITY_CLASS`: высокий приоритет.
  - `NORMAL_PRIORITY_CLASS`: нормальный приоритет (по умолчанию).
  - `IDLE_PRIORITY_CLASS`: низкий приоритет.
  - `REALTIME_PRIORITY_CLASS`: реальное время (осторожно при использовании, так как это может привести к зависанию системы).

- **Получение текущего приоритета процесса**. Для получения текущего приоритета процесса используется функция `GetPriorityClass`.

```
DWORD priority = GetPriorityClass(GetCurrentProcess());
```

Значение `priority` будет одним из перечисленных выше констант.

- **Установка приоритета потока**. Внутри процесса можно устанавливать приоритет для отдельных потоков с помощью функции `SetThreadPriority`. Это позволяет управлять приоритетами выполнения различных задач внутри одного процесса.

```
BOOL success = SetThreadPriority(hThread, priority);
```

Здесь `hThread` – дескриптор потока, а `priority` – желаемый приоритет потока.

- **Получение текущего приоритета потока**. Для получения текущего приоритета потока используется функция `GetThreadPriority`.

```
int priority = GetThreadPriority(hThread);
```

Значение `priority` будет числовым представлением приоритета потока.

Управление потоками внутри процесса в операционной системе Windows можно выполнять с использованием Windows API (WinAPI). Вот основные функции и концепции, связанные с управлением потоками:

- **Создание потока.** Для создания нового потока внутри процесса используется функция `CreateThread`. Эта функция позволяет запустить новый поток и выполнить в нем определенную функцию.

```
HANDLE hThread = CreateThread(
    NULL,                   // Атрибуты безопасности потока (обычно NULL)
    0,                      // Размер стека (0 = размер стека по умолчанию)
    ThreadFunction,         // Функция, которая будет выполнена в потоке
    lpParam,                // Дополнительные параметры для функции
    0,                      // Флаги создания потока (0 = запуск сразу после
создания)
    &dwThreadId             // Идентификатор потока
);
```

Здесь `ThreadFunction` – это указатель на функцию, которая будет выполняться в потоке, и `lpParam` – дополнительные параметры, которые могут быть переданы в функцию.

- **Завершение потока.** Для завершения выполнения потока используется функция ExitThread. Вызов этой функции приведет к завершению текущего потока.

```
ExitThread(0);
```

- **Ожидание завершения потока.** Для ожидания завершения выполнения потока используется функци `WaitForSingleObject` или `WaitForMultipleObjects`, в зависимости от количества потоков, которые нужно ожидать.

```
DWORD dwExitCode;
DWORD dwWaitResult = WaitForSingleObject(hThread, INFINITE);

if (dwWaitResult == WAIT_OBJECT_0) {
```

```
    // Поток завершил выполнение
    GetExitCodeThread(hThread, &dwExitCode);
    // Обработка результата выполнения потока (dwExitCode)
} else {
    // Обработка ошибки ожидания
}
```

- **Установка приоритета потока.** Установка приоритета выполнения потока внутри процесса выполняется с помощью функции `SetThreadPriority`.

```
BOOL success = SetThreadPriority(hThread, priority);
```

Здесь `hThread` – дескриптор потока, а `priority` – желаемый приоритет потока.

- **Получение текущего приоритета потока.** Для получения текущего приоритета потока используется функция `GetThreadPriority`.

```
int priority = GetThreadPriority(hThread);
```

- **Закрытие дескриптора потока.** После завершения работы с потоком, его дескриптор должен быть закрыт с помощью функции `CloseHandle`.

```
CloseHandle(hThread);
```

Основные механизмы синхронизации, доступные в Windows API:

- **Критические секции (Critical Sections):** предоставляют простой и легковесный способ синхронизации между потоками внутри одного процесса. Функции для работы с критическими секциями включают `InitializeCriticalSection`, `EnterCriticalSection`, `LeaveCriticalSection`, и `DeleteCriticalSection`.
- **Мьютексы (Mutexes)**: могут использоваться для синхронизации между разными процессами, а не только потоками внутри одного процесса. Функции для работы с мьютексами включают `CreateMutex`, `WaitForSingleObject`, и `ReleaseMutex`.
- **Семафоры (Semaphores)**: позволяют контролировать доступ к ресурсам, когда требуется счетчик. Функции для работы с семафорами включают `CreateSemaphore`, `WaitForSingleObject`, и `ReleaseSemaphore`.
- **События (Events)**: События используются для уведомления одного или нескольких потоков или процессов о возникновении события. Функции для работы с событиями включают `CreateEvent`, `SetEvent`, `WaitForSingleObject`, и другие.
- **Критические ресурсы и мьютексы файла (File Mapping and File Mutexes)**: позволяют синхронизировать доступ к разделяемым данным, которые находятся в памяти или на диске. Самые распространенные функции включают `CreateFileMapping`, `MapViewOfFile`, и `WaitForSingleObject`.
- **Readers-Writers Locks**: позволяют определять правила доступа для читающих и записывающих потоков к общим данным. Функции для работы с

Reader-Writer Locks включают `InitializeSRWLock`, `AcquireSRWLockExclusive`, и `AcquireSRWLockShared`.

- **События завершения (Completion Events)**: используются для уведомления о завершении выполнения асинхронных операций ввода – вывода. Функции для работы с событиями завершения включают `CreateIoCompletionPort`, `GetQueuedCompletionStatus`, и другие.

Выбор подходящего механизма синхронизации зависит от конкретных требований приложения и структуры данных. Важно правильно использовать синхронизацию, чтобы избежать проблем с гонками данных, блокировками и ожиданиями.

В Windows API (WinAPI) есть несколько способов завершения процессов, включая нормальное завершение и принудительное завершение. Вот основные функции и методы для завершения процессов:

- **ExitProcess** – используется для нормального завершения текущего процесса. Это завершает выполнение текущей программы и завершает процесс.

```
ExitProcess(exitCode);
```

Здесь `exitCode` – код завершения процесса, который будет возвращен операционной системой.

- **TerminateProcess** – используется для принудительного завершения другого процесса. Это позволяет завершить процесс, даже если он не отвечает или заблокирован.

```
BOOL success = TerminateProcess(hProcess, exitCode);
```

Здесь `hProcess` – дескриптор процесса, который нужно завершить, и `exitCode` – код завершения процесса.

- **WM_CLOSE и PostMessage**: если необходимо завершить приложение с графическим интерфейсом, вы можете отправить сообщение `WM_CLOSE` главному окну приложения с помощью функции `PostMessage`. Это позволит приложению выполнить закрытие как обычно.

```
PostMessage(hWnd, WM_CLOSE, 0, 0);
```

Здесь `hWnd` – дескриптор главного окна приложения.

Обратите внимание, что при использовании функции `TerminateProcess` процесс завершается немедленно и не выполняет никаких завершающих операций, что может привести к утечкам ресурсов и несохранению данных. По возможности рекомендуется использовать нормальное завершение процессов с помощью `ExitProcess` или закрытие окон с использованием `WM_CLOSE` и `PostMessage`, чтобы приложение могло выполнить необходимые действия перед завершением.

Для получения информации о процессах в операционной системе Windows можно использовать Windows API. Существует несколько функций и структур данных, которые позволяют получить информацию о текущих работающих процессах. Пример получения списка процессов и их атрибутов:

```
#include <windows.h>
#include <stdio.h>
#include <tlhelp32.h>
#include <tchar.h>

int main() {
    SetConsoleOutputCP(1251);
    // Создаем объект, представляющий снимок всех процессов
    HANDLE hProcessSnap = CreateToolhelp32Snapshot(TH32CS_SNAPPROCESS, 0);

    if (hProcessSnap == INVALID_HANDLE_VALUE) {
        // Обработка ошибки
        return 1;
    }

    // Структура, в которую будет сохранен атрибут процесса
    PROCESSENTRY32 pe32;
    pe32.dwSize = sizeof(PROCESSENTRY32);

    // Получаем информацию о первом процессе в снимке
    if (!Process32First(hProcessSnap, &pe32)) {
        CloseHandle(hProcessSnap);
        // Обработка ошибки
        return 1;
    }

    // Перебираем все процессы в снимке
    do {
        _tprintf(_T("Процесс ID: %d, Имя: %s\n"), pe32.th32ProcessID,
pe32.szExeFile);
        // Здесь можно получать и обрабатывать другие атрибуты процесса

    } while (Process32Next(hProcessSnap, &pe32));

    CloseHandle(hProcessSnap);

    return 0;
}
```

Для успешной компиляции и выполнения этого кода необходимо включить библиотеку `kernel32.lib` и указать верное символическое имя или путь к исполняемому файлу.

Для получения информации о модулях (библиотеках и исполняемых файлах) внутри процесса в операционной системе Windows, вы можете использовать Windows API. Основным инструментом для этой задачи является функция `EnumProcessModules`. Пример использования:

```
#include <windows.h>
#include <psapi.h>
#include <tchar.h>
#include <stdio.h>

int main() {
    SetConsoleOutputCP(1251);
```

```
    DWORD processId = GetCurrentProcessId(); // Идентификатор текущего процесса
    HANDLE hProcess = OpenProcess(PROCESS_QUERY_INFORMATION | PROCESS_VM_READ,
FALSE, processId);

    if (hProcess == NULL) {
        // Обработка ошибки открытия процесса
        return 1;
    }

    HMODULE hModules[1024];
    DWORD cbNeeded;

    // Получаем список модулей внутри процесса
    if (EnumProcessModules(hProcess, hModules, sizeof(hModules), &cbNeeded)) {
        for (DWORD i = 0; i < (cbNeeded / sizeof(HMODULE)); i++) {
            TCHAR szModName[MAX_PATH];

            // Получаем имя модуля
            if (GetModuleFileNameEx(hProcess, hModules[i], szModName,
sizeof(szModName) / sizeof(TCHAR))) {
                _tprintf(_T("Модуль #%u: %s\n"), i, szModName);
            }
        }
    }

    CloseHandle(hProcess);
    return 0;
}
```

Не забудьте включить библиотеку `psapi.lib` при компиляции и убедитесь, что код выполняется с правами, позволяющими открывать процессы и читать их модули.

Функции `OpenProcessToken` и `GetTokenInformation` в Windows API используются для получения информации о безопасности и разрешениях, связанных с процессом. Рассмотрим их использование подробнее:

- **OpenProcessToken**: используется для открытия дескриптора безопасности (токена) процесса. Этот токен содержит информацию о безопасности, такую как SID (идентификатор безопасности) пользователя и группы, разрешения и другие атрибуты безопасности процесса. Вот как можно использовать `OpenProcessToken`:

```
#include <windows.h>
#include <tchar.h>

int main() {
    SetConsoleOutputCP(1251);
    DWORD processId = ...; // Идентификатор процесса, для которого нужно
получить токен
    HANDLE hProcess = OpenProcess(PROCESS_QUERY_INFORMATION, FALSE, processId);

    if (hProcess == NULL) {
        // Обработка ошибки
        return 1;
```

```
    }

    HANDLE hToken;
    if (OpenProcessToken(hProcess, TOKEN_QUERY, &hToken)) {
        // Теперь у вас есть дескриптор токена (hToken) для процесса,
        // который вы можете использовать для получения информации о
безопасности.

        // Закрываем дескриптор токена после использования.
        CloseHandle(hToken);
    }

    CloseHandle(hProcess);
    return 0;
}
```

- **GetTokenInformation**: После открытия дескриптора токена с помощью `OpenProcessToken`, можно использовать функцию `GetTokenInformation`, чтобы получить различные атрибуты и информацию о безопасности. Необходимо предоставить буфер для хранения информации. Пример получения информации о SID (идентификаторе безопасности) пользователя из токена:

```
HANDLE hToken = ...; // Дескриптор токена
DWORD dwSize = 0;

// Получаем размер буфера, необходимый для информации о SID.
GetTokenInformation(hToken, TokenUser, NULL, 0, &dwSize);

// Выделяем буфер и получаем информацию о SID.
PTOKEN_USER pTokenUser = (PTOKEN_USER)malloc(dwSize);
if (GetTokenInformation(hToken, TokenUser, pTokenUser, dwSize, &dwSize)) {
    // Теперь у вас есть информация о SID пользователя.

    // Освобождаем выделенный буфер после использования.
    free(pTokenUser);
}

// Закрываем дескриптор токена.
CloseHandle(hToken);
```

**Критические секции** (Critical Sections) – это механизм синхронизации в Windows API, который позволяет защитить доступ к общим данным от одновременного доступа нескольких потоков внутри одного процесса. Они обычно используются для предотвращения гонок данных и обеспечения корректного доступа к разделяемым ресурсам. Пример использования критических секций:

```
#include <windows.h>

// Объявляем глобальную критическую секцию
```

```
CRITICAL_SECTION g_criticalSection;

// Функция, выполняемая в потоках
DWORD WINAPI ThreadFunction(LPVOID lpParam) {
    // Входим в критическую секцию
    EnterCriticalSection(&g_criticalSection);

    // Здесь можно выполнять операции с общими данными

    // Выходим из критической секции
    LeaveCriticalSection(&g_criticalSection);

    return 0;
}

int main() {
    SetConsoleOutputCP(1251);
    // Инициализируем критическую секцию
    InitializeCriticalSection(&g_criticalSection);

    // Создаем потоки, которые будут использовать критическую секцию
    HANDLE hThread1 = CreateThread(NULL, 0, ThreadFunction, NULL, 0, NULL);
    HANDLE hThread2 = CreateThread(NULL, 0, ThreadFunction, NULL, 0, NULL);

    // Ожидаем завершения потоков
    WaitForSingleObject(hThread1, INFINITE);
    WaitForSingleObject(hThread2, INFINITE);

    // Закрываем дескрипторы потоков
    CloseHandle(hThread1);
    CloseHandle(hThread2);

    // Уничтожаем критическую секцию
    DeleteCriticalSection(&g_criticalSection);

    return 0;
}
```

Использование критических секций обычно предпочтительнее внутри одного процесса, так как они более эффективны и проще в использовании по сравнению с другими механизмами синхронизации, такими как мьютексы и семафоры. Критические секции обеспечивают внутреннюю синхронизацию в пределах одного процесса и не подходят для синхронизации между разными процессами.

**Мьютексы** (Mutexes) – это механизм синхронизации в Windows API, который используется для управления доступом к разделяемым ресурсам, чтобы предотвратить гонки данных между несколькими потоками или процессами. Мьютексы обычно используются для синхронизации между потоками в разных процессах. Пример использования мьютексов для синхронизации между двумя потоками:

```c
#include <windows.h>
#include <stdio.h>

// Объявляем глобальный мьютекс
HANDLE g_mutex;

// Функция, выполняемая в потоках
DWORD WINAPI ThreadFunction(LPVOID lpParam) {
    // Попытка захвата мьютекса
    DWORD dwWaitResult = WaitForSingleObject(g_mutex, INFINITE);

    if (dwWaitResult == WAIT_OBJECT_0) {
        // Мьютекс успешно захвачен

        // Здесь можно выполнять операции с разделяемыми ресурсами

        // Освобождение мьютекса
        ReleaseMutex(g_mutex);
    } else {
        // Обработка ошибки
    }

    return 0;
}

int main() {
    SetConsoleOutputCP(1251);
    // Создаем мьютекс
    g_mutex = CreateMutex(NULL, FALSE, NULL);

    if (g_mutex == NULL) {
        // Обработка ошибки создания мьютекса
        return 1;
    }

    // Создаем два потока
    HANDLE hThread1 = CreateThread(NULL, 0, ThreadFunction, NULL, 0, NULL);
    HANDLE hThread2 = CreateThread(NULL, 0, ThreadFunction, NULL, 0, NULL);

    // Ожидаем завершения потоков
    WaitForSingleObject(hThread1, INFINITE);
    WaitForSingleObject(hThread2, INFINITE);

    // Закрываем дескрипторы потоков
    CloseHandle(hThread1);
    CloseHandle(hThread2);

    // Закрываем дескриптор мьютекса
    CloseHandle(g_mutex);

    return 0;
}
```

Это простой пример использования мьютексов для синхронизации между двумя потоками. Мьютексы также могут использоваться для синхронизации между разными процессами, если они используют один и тот же мьютекс с именем, доступным для обоих процессов.

**Семафоры** (Semaphores) – это ещё один механизм синхронизации в Windows API, который используется для контроля доступа к разделяемым ресурсам между несколькими потоками или процессами. Семафоры могут позволить нескольким потокам одновременно получить доступ к общему ресурсу в ограниченном количестве. Пример использования семафора для синхронизации между несколькими потоками:

```
#include <windows.h>
#include <stdio.h>

// Объявляем глобальный семафор и устанавливаем начальное значение
HANDLE g_semaphore;

// Функция, выполняемая в потоках
DWORD WINAPI ThreadFunction(LPVOID lpParam) {
    // Попытка уменьшить счетчик семафора
    DWORD dwWaitResult = WaitForSingleObject(g_semaphore, INFINITE);

    if (dwWaitResult == WAIT_OBJECT_0) {
        // Семафор успешно уменьшен

        // Здесь можно выполнять операции с разделяемыми ресурсами

        // Увеличение счетчика семафора
        ReleaseSemaphore(g_semaphore, 1, NULL);
    } else {
        // Обработка ошибки
    }

    return 0;
}

int main() {
    SetConsoleOutputCP(1251);
    // Создаем семафор с начальным счетчиком
    g_semaphore = CreateSemaphore(NULL, 2, 2, NULL); // В данном примере,
начальный счетчик равен 2

    if (g_semaphore == NULL) {
        // Обработка ошибки создания семафора
        return 1;
    }

    // Создаем два потока
    HANDLE hThread1 = CreateThread(NULL, 0, ThreadFunction, NULL, 0, NULL);
    HANDLE hThread2 = CreateThread(NULL, 0, ThreadFunction, NULL, 0, NULL);

    // Ожидаем завершения потоков
```

```
    WaitForSingleObject(hThread1, INFINITE);
    WaitForSingleObject(hThread2, INFINITE);

    // Закрываем дескрипторы потоков
    CloseHandle(hThread1);
    CloseHandle(hThread2);

    // Закрываем дескриптор семафора
    CloseHandle(g_semaphore);

    return 0;
}
```

Семафоры предоставляют мощный механизм для управления доступом к разделяемым ресурсам в многопоточных и многопроцессных приложениях.

**События** (Events) – это механизм синхронизации в Windows API, который используется для уведомления одного или нескольких потоков о наступлении определенного события. События могут быть использованы для синхронизации между потоками или процессами, когда один поток ждет, пока другой поток или процесс оповестит его о наступлении события. Пример использования событий:

```
#include <windows.h>
#include <stdio.h>

// Объявляем глобальное событие
HANDLE g_event;

// Функция, выполняемая в потоках
DWORD WINAPI ThreadFunction(LPVOID lpParam) {
    // Ожидание события
    DWORD dwWaitResult = WaitForSingleObject(g_event, INFINITE);

    if (dwWaitResult == WAIT_OBJECT_0) {
        // Событие успешно сработало

        // Здесь можно выполнять действия, связанные с событием

        printf("Событие сработало в потоке.\n");
    } else {
        // Обработка ошибки
    }

    return 0;
}

int main() {
    SetConsoleOutputCP(1251);
    // Создаем событие
    g_event = CreateEvent(NULL, FALSE, FALSE, NULL);

    if (g_event == NULL) {
        // Обработка ошибки создания события
```

```
        return 1;
    }

    // Создаем поток
    HANDLE hThread = CreateThread(NULL, 0, ThreadFunction, NULL, 0, NULL);

    // Задержка для демонстрации события
    Sleep(2000);

    // Устанавливаем событие
    SetEvent(g_event);

    // Ожидаем завершения потока
    WaitForSingleObject(hThread, INFINITE);

    // Закрываем дескриптор потока
    CloseHandle(hThread);

    // Закрываем дескриптор события
    CloseHandle(g_event);

    return 0;
}
```

События могут быть использованы для синхронизации и сигнализации между потоками или процессами, и они часто используются в многозадачных и многопроцессных приложениях для организации совместной работы потоков.

### *Порядок выполнения работы*

1. Написать программу, которая создает новый процесс с помощью функции `CreateProcess`. Новый процесс должен выполнять простую консольную программу, которую также необходимо написать.
2. Расширить предыдущую программу, добавив возможность передачи аргументов командной строки новому процессу.
3. Написать программу, которая получает информацию о текущем процессе с помощью функции `GetCurrentProcess`.
4. Написать программу, которая получает идентификатор текущего процесса с помощью функции `GetCurrentProcessId` и выводит его на экран.
5. Написать программу, которая получает дескриптор текущего процесса и использует его для изменения приоритета процесса с помощью функции `SetPriorityClass`.
6. Написать программу, которая завершает работу другого процесса с помощью функции `TerminateProcess`. Для этого необходимо получить дескриптор процесса с помощью функции `OpenProcess`.
7. Написать программу, которая создает новый поток в текущем процессе с помощью функции `CreateThread`. Новый поток должен выполнять простую функцию, которую также необходимо написать.

8. Написать программу, которая синхронизирует работу нескольких потоков с помощью событий (функции `CreateEvent`, `SetEvent`, `WaitForSingleObject` или `WaitForMultipleObjects`).

9. Написать программу, которая использует функции `GetExitCodeProcess` и `GetExitCodeThread` для получения кодов завершения процесса и потока.

### *Контрольные вопросы*

1. *Основные понятия:*
   - Что такое процесс в операционной системе Windows?
   - Какие атрибуты характеризуют процесс?
2. *Создание процессов:*
   - Какую функцию Windows API используют для создания нового процесса?
   - Какие параметры необходимо передать функции `CreateProcess` для успешного создания процесса?
   - Чем отличаются синхронное и асинхронное создание процесса?
3. *Получение информации о процессах:*
   - Какие функции Windows API используются для получения идентификатора текущего процесса и информации о других процессах?
   - Как получить информацию о времени работы процесса (начало, пользовательское время, системное время)?
4. *Завершение процессов:*
   - Какая функция используется для завершения процесса?
   - Какие параметры необходимы для завершения процесса с использованием функции `TerminateProcess`?
   - Как корректно завершить процесс, чтобы избежать утечек ресурсов?
5. *Приостановка и возобновление процессов:*
   - Какие функции Windows API используются для приостановки и возобновления процессов?
   - В каких случаях может потребоваться приостановка процесса?
   - Какой эффект может иметь приостановка основного потока процесса на его выполнение?
6. *Мониторинг процессов:*
   - Какие утилиты и инструменты в Windows можно использовать для мониторинга состояния процессов?
   - Как можно программно реализовать мониторинг активных процессов в системе?
   - Какие параметры процессов являются ключевыми для мониторинга?
7. *Безопасность и защита процессов:*
   - Какие методы защиты процессов от несанкционированного доступа существуют в Windows?
   - Как ограничить права доступа к процессу?

- Какие функции и параметры Windows API используются для реализации безопасности процессов?

*8. Практические аспекты:*

- Как определить, что процесс успешно завершился?
- Какие возможные ошибки могут возникнуть при управлении процессами, и как их диагностировать?
- Как проверить, что процесс находится в ожидающем (приостановленном) состоянии?

*9. Отладка процессов:*

- Какие средства и функции Windows API могут использоваться для отладки процессов?
- Как программно осуществить пошаговое выполнение процесса?
- Какие методы можно использовать для анализа причин сбоя процесса?

# ЛАБОРАТОРНАЯ РАБОТА 4.
# ЯЗЫК КОМАНДНОГО ИНТЕРПРИТАТОРА SHELL

**Цель работы**: Освоение основ работы с языком командного интерпретатора Shell, изучение основных команд и возможностей Shell, а также развитие навыков написания простых скриптов для автоматизации задач в операционной системе Unix/Linux.

## *Основные теоретические сведения*

Рассмотрим структуру скрипта Shell:

```
#!/bin/bash

# Комментарий: Начало скрипта

# Объявление переменных
VAR1="Hello"
VAR2="World"

# Основная часть скрипта
echo "$VAR1 $VAR2"   # Выводит "Hello World"

# Завершение скрипта
exit 0
```

- **Шебанг (Shebang)**: `#!/bin/bash` – это строка в начале скрипта, указывающая на то, какой интерпретатор использовать. В данном случае используется интерпретатор Bash.

- **Комментарии**: Комментарии помогают описать, что делает скрипт. Они начинаются с символа `#` и игнорируются интерпретатором.
- **Объявление переменных**: Здесь объявляются переменные `VAR1` и `VAR2`, которые содержат строки «Hello» и «World» соответственно.
- **Основная часть скрипта**: Это место, где выполняются основные действия. В данном случае, скрипт просто выводит содержимое переменных с помощью команды `echo`.
- **Завершение скрипта**: Команда `exit 0` указывает на успешное завершение скрипта. Код `0` обычно означает успешное выполнение, а другие значения могут указывать на различные ошибки или условия завершения.

Переменные в shell играют ключевую роль, позволяя хранить и использовать данные в скриптах. Переменные объявляются без использования ключевых слов, просто путем присваивания значения. Например:

```
VAR="value"
```

Для использования переменной в тексте скрипта нужно указать её имя, предваряя его символом доллара `$`. Например:

```
echo $VAR
```

В shell есть множество специальных переменных, таких как `$0`, `$1`, `$2`, и т. д., которые хранят аргументы командной строки. Например:

```
echo "Имя этого скрипта: $0"
```

Можно использовать команду `read`, чтобы прочитать значение из ввода и присвоить его переменной. Например:

```
read NAME
echo "Привет, $NAME!"
```

Для удаления переменной используется команда `unset`. Например:

```
unset VAR
```

Переменные могут быть интерполированы в строковых значениях. Например:

```
GREETING="Hello"
echo "$GREETING, world!"
```

Все переменные, которые вы экспортируете из shell, становятся переменными окружения для любых запущенных подкоманд или программ. Например:

```
export MY_VAR="value"
```

В bash и других расширениях shell можно использовать ассоциативные массивы для хранения данных. Например:

```
declare -A fruits
fruits[apple]="red"
fruits[banana]="yellow"
echo "Apple is ${fruits[apple]}"
```

Специальные переменные в shell представляют собой переменные, которые предопределены и предназначены для выполнения определенных задач.

- **$0**: имя исполняемого файла скрипта (имя команды).
- **$1, $2, ..., $N**: параметры командной строки. $1 содержит первый аргумент, $2 – второй, и так далее.
- **$@**: список всех аргументов командной строки.
- **$#**: количество аргументов командной строки.
- **$?**: код возврата (exit code) последней выполненной команды.
- **$$**: PID (идентификатор процесса) текущего shell.
- **$!**: PID последнего запущенного в фоновом режиме процесса.
- **$***: похоже на $@, но сохраняет аргументы как одну строку.
- **$IFS**: разделитель полей (Internal Field Separator). Это строка символов, которая определяет, где shell должен разбивать строки на поля, при чтении из ввода или из переменной.
- **$PWD**: текущий рабочий каталог (полный путь).
- **$OLDPWD**: предыдущий рабочий каталог.

Переменные окружения представляют собой переменные, которые могут быть установлены в системе операционной или виртуальной среде и доступны для всех процессов, запущенных в этой среде. Они могут содержать различную информацию о системе, пользователе, конфигурации и многое другое.

- **PATH**: содержит список каталогов, в которых операционная система ищет исполняемые файлы.
- **HOME**: путь к домашнему каталогу текущего пользователя.
- **USER**: имя текущего пользователя.
- **LANG**: определяет язык, используемый для отображения сообщений и форматирования даты и времени.
- **TERM**: определяет тип терминала, используемого для взаимодействия с пользователем.
- **PWD**: текущий рабочий каталог.
- **SHELL**: путь к оболочке, используемой текущим пользователем.
- **DISPLAY**: определяет, на каком дисплее должны отображаться графические приложения.
- **TMP, TEMP**: пути к временным каталогам.
- **EDITOR, VISUAL**: определяют текстовый редактор, используемый по умолчанию.
- **TZ**: определяет часовой пояс системы.
- **HOSTNAME**: имя хоста (имя компьютера).
- **PS1, PS2**: переменные, используемые для настройки приглашения командной строки.
- **LD_LIBRARY_PATH**: список каталогов, в которых операционная система ищет библиотеки, используемые программами.

- **JAVA_HOME**: путь к установленной JDK (Java Development Kit).

Эти переменные можно использовать в shell скриптах для доступа к информации о среде выполнения и для настройки поведения программы в соответствии с окружением. Для просмотра всех переменных окружения в вашей текущей среде выполнения вы можете использовать команду `env` в большинстве Unix-подобных систем.

Операторы ветвления в shell используются для принятия решений на основе условий и выполнения различных действий в зависимости от результатов этих условий.

- **if-then-else**:

```
if условие; then
    команды_если_условие_истинно
else
    команды_если_условие_ложно
fi
```

Пример:

```
if [ $x -gt 10 ]; then
    echo "x больше 10"
else
    echo "x не больше 10"
fi
```

- **if-then** (без else):

```
if условие; then
    команды_если_условие_истинно
fi
```

Пример:

```
if [ -e $FILE ]; then
    echo "$FILE существует"
fi
```

- **if-elif-else:**

```
if условие1; then
    команды_если_условие1_истинно
elif условие2; then
    команды_если_условие2_истинно
else
    команды_если_все_условия_ложны
fi
```

Пример:

```
if [ $x -gt 10 ]; then
    echo "x больше 10"
elif [ $x -eq 10 ]; then
    echo "x равно 10"
else
    echo "x меньше 10"
fi
```

- **case:**

```
case выражение in
    паттерн1)
        команды_если_паттерн1_соответствует
        ;;
    паттерн2)
        команды_если_паттерн2_соответствует
        ;;
    *)
        команды_если_нет_соответствия_ни_одному_паттерну
        ;;
esac
```

Пример:

```
case $VAR in
    "value1")
        echo "Переменная равна value1"
        ;;
    "value2")
        echo "Переменная равна value2"
        ;;
    *)
        echo "Переменная не равна ни value1, ни value2"
        ;;
esac
```

Эти операторы ветвления позволяют создавать гибкие скрипты, которые могут выполнять различные действия в зависимости от условий, и обеспечивают управление потоком выполнения программы в shell.

В контексте программирования «*предикат*» обычно означает выражение, которое оценивается как истинное или ложное. В языках программирования, включая shell, предикаты используются в операторах ветвления (например, в операторах `if`, `while`, `case` и других) для принятия решений на основе условий.

- **Сравнение чисел**:
  - `-eq`: равно
  - `-ne`: не равно
  - `-gt`: больше
  - `-lt`: меньше
  - `-ge`: больше или равно
  - `-le`: меньше или равно

Например:

```
if [ $x -gt 10 ]; then
    echo "x больше 10"
fi
```

- **Сравнение строк**:
  - `=`: равно
  - `!=`: не равно
  - `-z`: пустая строка

- `-n`: не пустая строка

Например:

```
if [ "$var" = "value" ]; then
    echo "Переменная равна value"
fi
```

- **Файловые предикаты**:
  - `-e`: файл существует
  - `-f`: обычный файл
  - `-d`: каталог
  - `-r`: доступен для чтения
  - `-w`: доступен для записи
  - `-x`: доступен для выполнения

Например:

```
if [ -e $FILE ]; then
    echo "$FILE существует"
fi
```

- **Логические операции**:
  - `-a`: логическое «и»
  - `-o`: логическое «или»
  - `!`: логическое «не»

Например:

```
if [ "$var" = "value1" -o "$var" = "value2" ]; then
    echo "Переменная равна value1 или value2"
fi
```

Эти предикаты позволяют создавать условия в операторах ветвления, которые позволяют программам принимать решения и выполнять различные действия на основе этих условий.

В shell скриптах часто используются циклы для повторения определенных действий определенное количество раз или до выполнения определенного условия.

- **Цикл while**:

```
while условие; do
    команды
done
```

Например:

```
counter=0
while [ $counter -lt 10 ]; do
    echo $counter
    ((counter++))
done
```

- **Цикл until**:

```
until условие; do
    команды
done
```

Например:

```
counter=0
until [ $counter -eq 10 ]; do
    echo $counter
    ((counter++))
done
```

- **Цикл for** (для перебора элементов в списке):

```
for переменная in список; do
    команды
done
```

Например:

```
for i in {1..5}; do
    echo $i
done
```

- **Цикл for** (для перебора элементов в массиве или списке):

```
for переменная in элемент1 элемент2 ... элементN; do
    команды
done
```

Например:

```
for color in red green blue; do
    echo "Color: $color"
done
```

- **Цикл select** (для создания интерактивного меню):

```
select переменная in список; do
    команды
done
```

Например:

```
select fruit in apple banana orange; do
    echo "Вы выбрали: $fruit"
    break
done
```

Эти циклы могут использоваться для автоматизации повторяющихся задач, обработки данных, итерации по спискам и многих других вещей в shell скриптах.

Вызов сторонних программ из shell скриптов – обычная практика, которая позволяет скриптам взаимодействовать с другими программами и выполнять различные задачи. Для вызова сторонней программы используются команды, которые могут быть исполнены внутри скрипта.

- **Вызов программы по её имени**:

```
программа_или_команда аргументы
```

Например:

```
ls -l
```

- **Использование переменных в качестве аргументов**:

```
переменная="аргументы"
программа_или_команда $переменная
```

Например:

```
directory="/path/to/directory"
ls $directory
```

- **Подстановка вывода другой команды**:

```
программа_или_команда $(другая_команда)
```

Например:

```
files_count=$(ls | wc -l)
echo "В текущем каталоге $files_count файлов"
```

- **Использование обратных кавычек для подстановки вывода команды** (устаревший метод):

```
программа_или_команда `другая_команда`
```

Например:

```
files_count=`ls | wc -l`
echo "В текущем каталоге $files_count файлов"
```

- **Пайплайны (|) для передачи вывода одной команды в качестве входных данных другой**:

```
команда1 | команда2
```

Например:

```
ls | grep ".txt"
```

- **Перенаправление вывода (stdout и stderr):**

```
программа_или_команда > файл
программа_или_команда >> файл (для добавления в конец файла)
```

Например:

```
ls > file_list.txt
```

- **Перенаправление ввода:**

```
программа_или_команда < файл
```

Например:

```
sort < unsorted_file.txt
```

Это основные способы вызова сторонних программ из shell скриптов. Они позволяют взаимодействовать со стандартными командами операционной системы, а также с любыми другими программами, доступными в вашей среде выполнения.

В shell скриптах можно выполнить математические вычисления несколькими способами.

- **Арифметическое выражение с помощью встроенной команды let:**

```
let "результат = выражение"
```

Например:

```
let "x = 10 + 5"
echo $x   # Выведет 15
```

- **Арифметическое выражение с помощью двойных круглых скобок ((...)):**

```
результат=$((выражение))
```

Например:

```
x=$((10 + 5))
echo $x   # Выведет 15
```

- **Использование команды expr:**

```
результат=$(expr выражение)
```

Например:

```
x=$(expr 10 + 5)
echo $x   # Выведет 15
```

- **Использование конструкции $((...)):**

```
результат=$((выражение))
```

Например:

```
x=$((10 + 5))
echo $x   # Выведет 15
```

- **Использование встроенной команды bc для вычисления выражений с плавающей запятой:**

```
результат=$(echo "выражение" | bc)
```

Например:

```
x=$(echo "10.5 * 3" | bc)
echo $x   # Выведет 31.5
```

Каждый из этих методов имеет свои особенности и подходит для разных типов вычислений. Например, `let` и `((...))` работают только с целыми числами, в то время как `bc` поддерживает вычисления с плавающей запятой.

В shell скриптах вы можете определять пользовательские функции для организации и структурирования вашего кода. Вот пример определения и вызова пользовательской функции:

```
# Определение функции
function my_function {
    echo "Привет, это моя пользовательская функция!"
}
```

```
# Вызов функции
my_function
```

Этот пример определяет функцию с именем `my_function`, которая выводит приветственное сообщение. После определения функции она вызывается с помощью `my_function`.

Ниже приведены некоторые дополнительные концепции, которые могут быть полезны при работе с пользовательскими функциями:

1. **Передача аргументов в функцию**:

```
function my_function {
    echo "Привет, $1!"
}

my_function "Мир"
```

Этот пример передает строку «Мир» в функцию `my_function`, которая выводит «Привет, Мир!».

2. **Возврат значений из функции:**

```
function add_numbers {
    local result=$(( $1 + $2 ))
    echo $result
}

sum=$(add_numbers 5 3)
echo "Сумма: $sum"
```

Этот пример определяет функцию `add_numbers`, которая принимает два аргумента, складывает их и возвращает результат. Результат сохраняется в переменной `sum`.

3. **Локальные переменные в функции:**

```
function my_function {
    local name="John"
    echo "Привет, $name!"
}

my_function
echo "Имя: $name"  # Это вызовет ошибку, потому что переменная name является локальной для функции
```

В этом примере переменная `name` является локальной для функции `my_function` и недоступна за её пределами.

4. **Рекурсивные функции:**

```
function factorial {
    if [ $1 -le 1 ]; then
        echo 1
    else
        local prev=$(factorial $(( $1 - 1 )))
        echo $(( $1 * $prev ))
    fi
```

```
}

result=$(factorial 5)
echo "Факториал 5: $result"
```

В этом примере определена рекурсивная функция для вычисления факториала числа. Функция вызывает саму себя до тех пор, пока не достигнет базового случая.

### *Порядок выполнения работы*

1. Переменные Shell:

○ Создать скрипт, который запрашивает у пользователя его имя и приветствует его с использованием этого имени.

○ Написать скрипт, который запрашивает у пользователя два числа и выводит их сумму.

2. Переменные окружения:

○ Написать скрипт, который выводит значения некоторых системных переменных окружения, таких как `$HOME`, `$PATH`, `$USER`, и т. д.

○ Создать скрипт, который выводит список всех переменных окружения.

3. Операторы ветвления:

○ Написать скрипт, который проверяет, является ли введенное пользователем число четным или нечетным, и выводит соответствующее сообщение.

○ Создать скрипт, который запрашивает у пользователя его возраст и сообщает, является ли он совершеннолетним или нет.

4. Оператор выбора:

○ Написать скрипт, который запрашивает у пользователя его любимый сезон и выводит сообщение о том, какие виды активностей связаны с этим сезоном.

○ Создатть скрипт, который проверяет введенную пользователем строку на наличие ключевого слова и выводит соответствующее сообщение.

5. Операторы циклов:

○ Написать скрипт, который выводит числа от 1 до 10 с использованием цикла `for`.

○ Создать скрипт, который запрашивает у пользователя число и выводит таблицу умножения этого числа.

6. Создание и использование пользовательских функций:

○ Создать функцию `greet`, которая принимает имя в качестве аргумента и выводит приветствие.

○ Вызвать функцию `greet` со своим именем.

7. Использование команд для работы с файлами:

○ Создать текстовый файл `example.txt`.

○ Записать в него строку «Пример текста».

○ Использовать команду `cat` для чтения содержимого файла `example.txt`.

***Контрольные вопросы***

1. *Основы работы с командной строкой:*
   - Какие команды используются для навигации по файловой системе в Shell?
   - Как узнать текущий рабочий каталог в командной строке?
   - Какие команды используются для управления файлами и каталогами?
2. *Перенаправление ввода/вывода и конвейеры:*
   - Какие символы используются для перенаправления вывода команды в файл?
   - Чем отличаются символы > и >> при перенаправлении вывода?
   - Что такое конвейер (pipe) в Shell, и как он используется?
3. *Написание скриптов на Shell:*
   - Как объявить переменную в Shell скрипте?
   - Как передать аргументы в скрипт при его вызове?
   - Какие операторы управления используются в Shell скриптах, и как они работают?
4. *Управление процессами:*
   - Как узнать список запущенных процессов в системе?
   - Как отправить сигнал процессу для его завершения?
   - Как перевести процесс в фоновый режим или вернуть его в передний план?
5. *Безопасность и права доступа:*
   - Как изменить права доступа к файлу или каталогу в Shell?
   - Как изменить владельца и группу файла или каталога?
   - Какие команды используются для работы с пользователями и группами в Unix/Linux?
6. *Практические навыки:*
   - Как написать скрипт, который будет искать все файлы в заданном каталоге с расширением «.txt» и выводить их список?
   - Как создать скрипт, который будет резервировать копию заданного каталога и его содержимого?
   - Как можно использовать командный интерпретатор для автоматизации рутинных задач на вашем компьютере?

# ЛАБОРАТОРНАЯ РАБОТА 5. РАБОТА С ФАЙЛАМИ В LINUX

**Цель работы**: Изучение основных аспектов работы с файлами в операционной системе Linux в контексте системного программирования.

***Основные теоретические сведения***

Пример 1. Создание файла:

```c
#include <stdio.h>
#include <fcntl.h>
#include <unistd.h>

int main() {
    int fd;
    fd = open("example.txt", O_CREAT | O_WRONLY, 0644);
    if (fd == -1) {
        perror("open");
        return 1;
    }
    printf("Файл создан успешно!\n");
    close(fd);
    return 0;
}
```

Пример 2. Запись данных в файл:

```c
#include <stdio.h>
#include <fcntl.h>
#include <unistd.h>
#include <string.h>

int main() {
    int fd;
    char buffer[100] = "Привет, мир!";
    fd = open("example.txt", O_WRONLY | O_APPEND);
    if (fd == -1) {
        perror("open");
        return 1;
    }
    if (write(fd, buffer, strlen(buffer)) == -1) {
        perror("write");
        return 1;
    }
    printf("Данные успешно записаны в файл!\n");
    close(fd);
    return 0;
}
```

Пример 3. Чтение данных из файла:

```c
#include <stdio.h>
#include <fcntl.h>
#include <unistd.h>
#include <string.h>

int main() {
    int fd;
    char buffer[100];
    fd = open("example.txt", O_RDONLY);
```

```
    if (fd == -1) {
        perror("open");
        return 1;
    }
    if (read(fd, buffer, sizeof(buffer)) == -1) {
        perror("read");
        return 1;
    }
    printf("Прочитанные данные: %s\n", buffer);
    close(fd);
    return 0;
}
```

Пример 4. Изменение размера файла:

```
#include <stdio.h>
#include <fcntl.h>
#include <unistd.h>
#include <string.h>

int main() {
    int fd;
    fd = open("example.txt", O_RDWR);
    if (fd == -1) {
        perror("open");
        return 1;
    }
    if (ftruncate(fd, 10) == -1) {
        perror("ftruncate");
        return 1;
    }
    printf("Размер файла изменен успешно!\n");
    close(fd);
    return 0;
}
```

Пример 5. Удаление файла:

```
#include <stdio.h>
#include <unistd.h>

int main() {
    if (unlink("example.txt") == -1) {
        perror("unlink");
        return 1;
    }
    printf("Файл успешно удален!\n");
    return 0;
}
```

### *Порядок выполнения работы*

1. Создать простую программу на языке Си, использующую системный вызов `open()`, чтобы создать новый файл.
2. Модифицировать программу так, чтобы она записывала строку в файл, используя системный вызов `write()`.
3. Добавить код для чтения данных из файла, используя системный вызов `read()`.
4. Добавить возможность изменять размер файла, используя системный вызов `ftruncate()`.
5. Написать программу для удаления файла с помощью системного вызова `unlink()`.

### *Контрольные вопросы*

1. Какие основные команды Linux используются для работы с файлами и директориями?
2. Какие системные вызовы предоставляет язык программирования C для работы с файлами?
3. Что такое файловый дескриптор? Какова его роль в системном программировании?
4. Какие функции языка C используются для открытия файла?
5. Какие флаги могут быть переданы функции `open()` для управления режимом открытия файла?
6. Какие системные вызовы используются для чтения и записи данных в файл?
7. Какие функции C используются для перемещения указателя текущей позиции в файле?
8. Какие системные вызовы позволяют перемещаться по директориям и работать с ними?
9. Что такое права доступа к файлам в Linux? Как они устанавливаются и изменяются?
10. Какие системные вызовы позволяют управлять правами доступа к файлам?
11. Какие методы используются для проверки успешности выполнения системных вызовов в языке C?
12. Какие основные функции используются для удаления файлов и директорий в Linux?
13. Какие методы можно использовать для обработки ошибок при работе с файлами в языке C?
14. В чем состоит различие между абсолютным и относительным путями к файлам и директориям?
15. Какие средства предоставляет операционная система Linux для поиска файлов по различным критериям?

# ЛАБОРАТОРНАЯ РАБОТА 6.
# РАБОТА С ПРОЦЕССАМИ В LINUX

**Цель работы**: Изучение основных аспектов работы с процессами в операционной системе Linux с целью приобретения практических навыков управления процессами, мониторинга и анализа их работы.

### *Основные теоретические сведения*

Пример 1. Создание нового процесса с помощью системного вызова `fork()`:

```
#include <stdio.h>
#include <unistd.h>

int main() {
    pid_t pid;

    pid = fork();

    if (pid < 0) {
        fprintf(stderr, "Ошибка при вызове fork()\n");
        return 1;
    } else if (pid == 0) {
        printf("Это дочерний процесс (PID: %d)\n", getpid());
    } else {
        printf("Это родительский процесс (PID: %d), дочерний процесс создан
(PID: %d)\n", getpid(), pid);
    }

    return 0;
}
```

Пример 2. Использование системного вызова `exec()` для выполнения программы в созданном процессе:

```
#include <stdio.h>
#include <unistd.h>

int main() {
    pid_t pid;

    pid = fork();

    if (pid < 0) {
        fprintf(stderr, "Ошибка при вызове fork()\n");
        return 1;
    } else if (pid == 0) {
        printf("Это дочерний процесс (PID: %d)\n", getpid());
```

```
        execl("/bin/ls", "ls", "-l", NULL);
    } else {
        printf("Это родительский процесс (PID: %d), дочерний процесс создан
(PID: %d)\n", getpid(), pid);
    }

    return 0;
}
```

Пример 3. Использование каналов (pipes) для обмена данными между процессами:

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>

int main() {
    int pipefd[2];
    pid_t pid;
    char buffer[20];

    if (pipe(pipefd) == -1) {
        perror("pipe");
        exit(EXIT_FAILURE);
    }

    pid = fork();

    if (pid == -1) {
        perror("fork");
        exit(EXIT_FAILURE);
    }

    if (pid == 0) {
        close(pipefd[0]);  // Закрываем конец для чтения в дочернем процессе
        write(pipefd[1], "Привет от дочернего процесса", 30);
        close(pipefd[1]);  // Закрываем конец для записи в дочернем процессе
        exit(EXIT_SUCCESS);
    } else {
        close(pipefd[1]);  // Закрываем конец для записи в родительском
процессе
        read(pipefd[0], buffer, sizeof(buffer));
        printf("Родительский процесс прочитал: %s\n", buffer);
        close(pipefd[0]);  // Закрываем конец для чтения в родительском
процессе
    }

    return 0;
}
```

Пример 4. Использование семафоров (semaphores) для синхронизации между процессами:

```c
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/sem.h>

#define SEM_KEY 1234

union semun {
    int val;
    struct semid_ds *buf;
    unsigned short *array;
};

int main() {
    int sem_id;
    struct sembuf sops;

    // Создание семафора
    sem_id = semget(SEM_KEY, 1, IPC_CREAT | 0666);
    if (sem_id == -1) {
        perror("semget");
        exit(EXIT_FAILURE);
    }

    // Установка начального значения семафора
    union semun arg;
    arg.val = 1;
    if (semctl(sem_id, 0, SETVAL, arg) == -1) {
        perror("semctl");
        exit(EXIT_FAILURE);
    }

    pid_t pid = fork();
    if (pid == -1) {
        perror("fork");
        exit(EXIT_FAILURE);
    }

    if (pid == 0) {
        sops.sem_num = 0;
        sops.sem_op = -1;  // Уменьшаем семафор
        sops.sem_flg = 0;
        semop(sem_id, &sops, 1);  // Захватываем семафор

        printf("Дочерний процесс захватил семафор\n");

        sops.sem_op = 1;  // Увеличиваем семафор
```

```
        semop(sem_id, &sops, 1);  // Освобождаем семафор
    } else {
        sops.sem_num = 0;
        sops.sem_op = -1;  // Уменьшаем семафор
        sops.sem_flg = 0;
        semop(sem_id, &sops, 1);  // Захватываем семафор

        printf("Родительский процесс захватил семафор\n");

        sops.sem_op = 1;  // Увеличиваем семафор
        semop(sem_id, &sops, 1);  // Освобождаем семафор
    }

    return 0;
}
```

### *Порядок выполнения работы*

1. Создать программу на языке Си, которая использует системные вызовы fork() для создания нового процесса.
2. Использовать системные вызовы exec() в дочернем процессе для выполнения другой программы (например, ls, ps, или ваша собственная программа).
3. Изучить механизм каналов (pipes) и реализовать коммуникацию между родительским и дочерним процессами через канал.
4. Изучить семафоры (semaphores) и реализуйте пример синхронизации между несколькими процессами.

### *Контрольные вопросы*

1. Что такое процесс в операционной системе Linux?
2. Каковы основные характеристики процесса (идентификатор процесса, родительский процесс, состояние процесса)?
3. Как создать новый процесс с использованием системного вызова fork()?
4. Как процесс может выполнить другую программу с помощью системного вызова exec()?
5. Какие команды и утилиты Linux используются для работы с процессами (например, ps, top, kill) и какие ключи часто используются с ними?
6. Какие механизмы коммуникации между процессами существуют в Linux?
7. Какие действия могут быть выполнены над процессом с помощью команды kill?
8. Какие утилиты мониторинга можно использовать для анализа работы процессов в реальном времени?
9. Как можно определить, сколько ресурсов (памяти, CPU и т. д.) потребляет определенный процесс?
10. Какие меры безопасности могут быть применены при работе с процессами в Linux?

# ЛИТЕРАТУРА


1. Гунько А.В. Программирование (в среде Windows) : учебное пособие / А.В. Гунько. – Новосибирск : Новосибирский государственный технический университет, 2019. – 155 с.
2. Лав Р. Linux. Системное программирование. / Р. Лав. – 2-е изд. – Санкт-Петербург : Питер, 2018. – 448 с.
3. Молчанов А. Ю. Системное программное обеспечение: учебник для вузов. / А.Ю. Молчанов. – 3-е изд. – Санкт-Петербург : Питер, 2021. – 400 с.
4. Побегайло А. Системное программирование в Windows / А. Побегайло. – Санкт-Петербург : БХВ-Петербург, 2006. – 1056 с.
5. Собель М. Linux. Администрирование и системное программирование. / М. Собель. – 2-е изд. – Санкт-Петербург : Питер, 2011. – 880 с.
6. Харт, Джонсон, М. Системное программирование в среде Windows, 5-е изд.: Пер. с англ. – М. : Издательский дом «Вильямс», 2005. – 592 с.
7. Хэвиленд Кейт. Системное программирование в UNIX. – Москва : ДМК Пресс, 2015. – 368 с.

Учебное издание

# СИСТЕМНОЕ ПРОГРАММИРОВАНИЕ

Методические указания по выполнению лабораторных работ

Составители:
Бизюк Андрей Николаевич
Соколова Анна Сергеевна

Редактор *Р.А. Никифорова*
Корректор *А.С. Прокопюк*
Компьютерная верстка *А.С. Соколова*

---

Подписано к печати 18.06.2024. Усл. печ. листов 3,9.
Уч.-изд. листов 3,8. Заказ № 154.

Учреждение образования «Витебский государственный технологический университет»
210038, г. Витебск, Московский пр., 72.
Отпечатано на ризографе учреждения образования
«Витебский государственный технологический университет».
Свидетельство о государственной регистрации издателя, изготовителя,
распространителя печатных изданий № 1/172 от 12 февраля 2014 г.
Свидетельство о государственной регистрации издателя, изготовителя,
распространителя печатных изданий № 3/1497 от 30 мая 2017 г.