МИНОБРНАУКИ РОССИИ
федеральное государственное бюджетное образовательное учреждение
высшего образования
«Самарский государственный технический университет»
(ФГБОУ ВО «СамГТУ»)

Кафедра «Вычислительная техника»

А.И. ПУГАЧЕВ, В.Д. ЛАПИР

Системное программное обеспечение

Лабораторный практикум

Самара
2020

Публикуется по решению методического совета института АиИТ
(протокол № 5 от 20.12.2019 г.)

УДК 004.45
ББК 32.973.26-018.2

Системное программное обеспечение: Лабораторный практикум / Сост. *А.И. Пугачев, В.Д. Лапир*. Самара: Самар. гос. техн. ун-т, 2020. - 104 с.

Изложены методы получения информации о состоянии ресурсов вычислительной системы, описаны используемые функции интерфейса Win32.

Приведен материал, необходимый для выполнения лабораторных работ по курсу «Системное программное обеспечение» в среде MS Visual Studio. Методические указания предназначены для обучающихся по программе бакалавриата направлений 09.03.01 «Информатика и вычислительная техника» и 09.03.04 «Программная инженерия» и родственных направлений, изучающих организацию и функционирование операционных систем MS Windows на системном уровне.

Рецензенты: к.х.н., доцент Чуваков А.В.
к.т.н., доцент Ефимушкина Н.В.

УДК 004.45
ББК 32.973.26-018.2

ОГЛАВЛЕНИЕ

ЛАБОРАТОРНАЯ РАБОТА № 1

ФУНКЦИИ ПОЛУЧЕНИЯ СИСТЕМНОЙ ИНФОРМАЦИИ

Цель работы – Получение практических навыков по программированию в Win32 API.

1. КРАТКИЕ ТЕОРЕТИЧЕСКИЕ СВЕДЕНИЯ

1.1 Создание консольного приложения в среде MS Visual Studio на языке C++

Консольные приложения – это программы, компилируемые в файлы типа *.exe, которые можно запускать на выполнение из командной строки как автономные приложения. Эти приложения работают только в текстовом режиме.

Для создания консольного приложения нужно запустить Microsoft Visual Studio, выбрать Создание проекта. В качестве шаблона (Templates) выбрать Visual C++, в качестве типа проекта (Project types) выбрать Консольное приложение Win32 (рисунок 1).

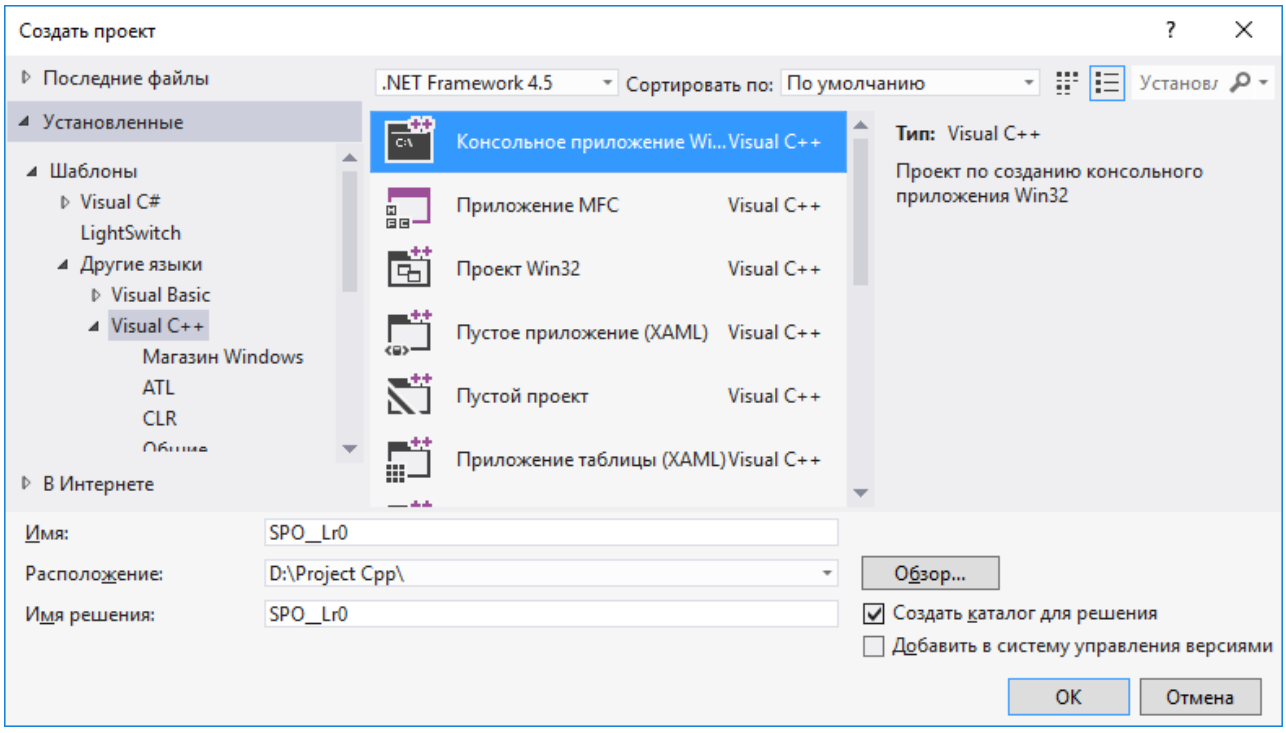


Рисунок 1 – Окно выбора типа шаблона

На рис. 1.1 показан процесс создания приложения. Далее нужно задать имя проекта. В примере – это SPO_Lr0. Это же имя используется для папки проекта, в которой автоматически будут созданы файлы проекта. Их назначение описано в файле ReadMe.txt (SPO_Lr0\SPO_Lr0\ReadMe.txt).

После окончания работы мастера, должно получиться следующее решение (рисунок 2) проекта SPO_Lr0.

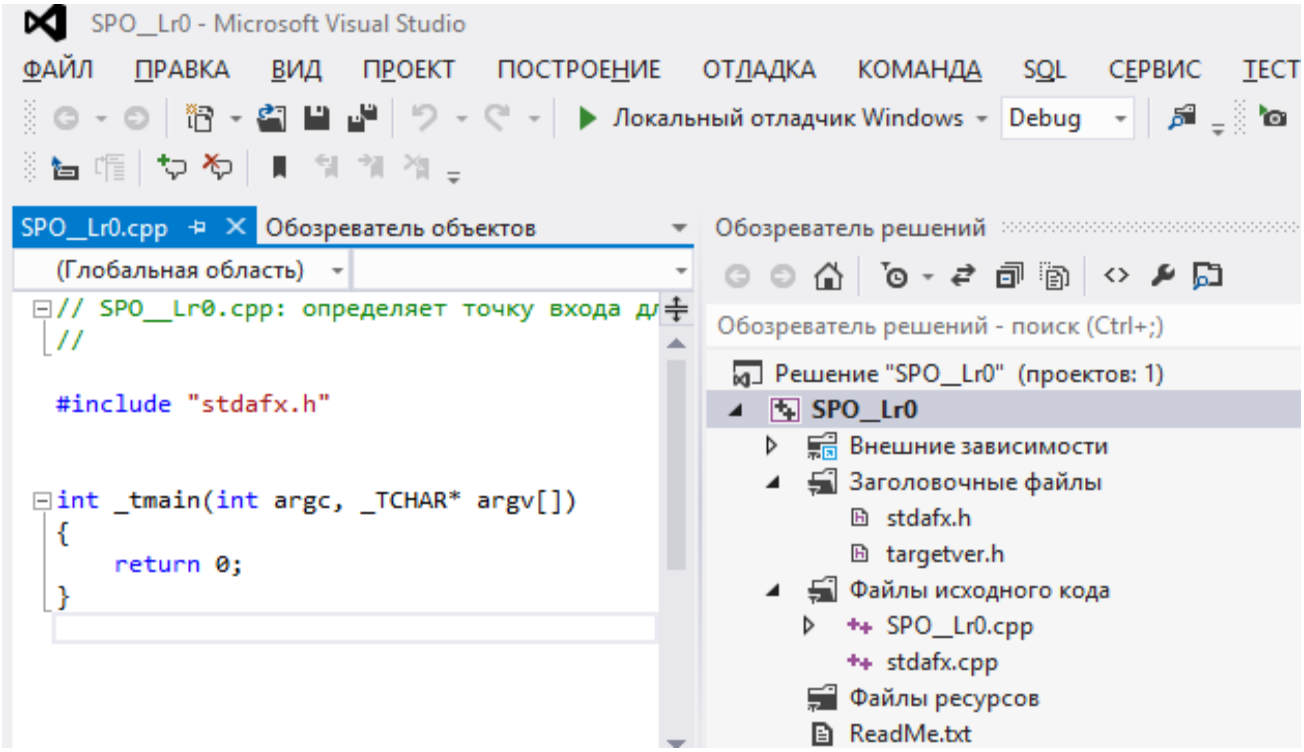


Рисунок 2 – Решение проекта

В большинстве консольных приложений требуется ввод данных с клавиатуры и вывод на экран. Добавление в начало программы строк

```
#include <iostream>
using namespace std;
```

позволит использовать в программе операторы cin для ввода данных и cout – для вывода информации на экран.

```
#include "stdafx.h"
#include <iostream>
using namespace std;
int _tmain(int argc, _TCHAR* argv[])
```

```
{
    cout << "\n My First Project \n"; // вывод строки на экран
    return 0;
}
```

Компиляция и запуск программы осуществляется из меню командой Отладка/Запуск без отладки или комбинацией Ctrl+F5.

Наиболее оптимальным способом русификации приложений является установка кодовой страницы с помощью функции setlocale(), объявленной в файле locale.h.

```
#include "stdafx.h"
#include <iostream>
#include <clocale>
using namespace std;

int _tmain(int argc, _TCHAR* argv[])
{   setlocale(LC_CTYPE, "rus");    // выбран русский язык
    cout <<"\n My First Проект \n";
    return 0;
}
```

1.2 Элементы программирования консольных приложений

1.2.1 Объявление переменных. Типы переменных

Переменная (идентификатор) – это именованная область в памяти компьютера, в которой может храниться некоторое значение. Имя переменной может состоять из латинских букв, цифр и символа подчеркивания, но начинаться с цифры не может. Заглавные и строчные буквы в именах различаются. Следует учитывать, что в C/C++ имеются ключевые слова (например, названия типов, операторов), и они не могут использоваться в качестве идентификаторов.

К базовым типам в C/C++ относятся: char – одна переменная может хранить один символ (или целое число от 0 до 255), массивы могут использоваться для работы со строками; int – целочисленная переменная; float – число с плавающей точкой (дробное); double – дробное число с двойной точностью; void – переменная без значения.

Пример:

```
int value;       // Переменная value целого типа, 4 байта
int i, j;
float Y;          // Переменная Y вещественного типа с плавающей точкой, 4 байта
double angle;  // Переменная angle вещественного типа с двойной точностью, 8 байт
```

Символы типа char кодируются в соответствие с таблицей ASCII (American Standard Code for Information Interchange).

```
char symbol;  // 1 байт
// инициализация
symbol = 56;    // 56 – код символа
symbol = 'a';
```

Массив символов:

```
  char text[20];         // массив из 20 символов
```

Непосредственное заполнение символьного массива из строковой константы возможно только при определении массива.

```
char str[] = "Text string";
```

Тип `wchar_t` определяет переменные символьного типа в кодировке Unicode (2 байта на символ).

```
wchar_t *wS = new wchar_t[30];   // массив символов Unicode
// инициализация значением константы – строки Unicode
wS = L"Пример инициализации строки в Unicode"; // строка Unicode
```

Префикс L означает, что строковая константа в коде Unicode.

1.2.2 Объявление и инициализация массивов

Массив может быть объявлен с помощью инициализатора – списка разделенных запятыми констант, задающих изначальные значения элементов. Он заключается в фигурные скобки. Длина списка может быть равна или меньше длины массива. Во втором случае оставшиеся элементы массива инициализируются нулями. Константы в инициализаторе должны быть совместимы с типом массива.

Пример инициализации массива целых чисел:

```
int a[5]={0,1,2,3,4};
```

Здесь размер массива задан равным 5. Также можно объявлять массивы без заранее установленных границ изменения индекса – безразмерные. Если при их объявлении использовать инициализатор, размер будет автоматически подсчитан компилятором. Пример:

```
int a[] = {1,2,3,4,5};
```

Массивы типа char можно инициализировать строковыми константами. Пример инициализации массива типа char:

```
// инициализация char
char login[8] = {'j','a','n','u','a','r','y','\0'};
char password[9]="february";
```

Данные строки являются нуль-терминированными, т.е. для обозначения их окончания используется символ '\0'. Для правильной работы с ними размер массива символов должен быть больше длины слова на 1. В приведенном примере слово "february" из 8 символов сохранено в массиве из 9 элементов.

Для строк, заданных массивами символов, стандартные операции присваивания и сравнения не поддерживаются. Тем не менее, можно присваивать значения таким строкам или сравнивать их посимвольно. Кроме того, в файле string.h объявлены встроенные функции для аналогичных операций. Так, функция strcpy(s1,s2) позволяет присвоить значение символьному массиву целиком, а сравнить такие массивы можно функцией strcmp(s1,s2). Функция strcat(s1,s2) осуществляет конкатенацию (склеивание) строк, функция strlen(s) возвращает длину строки. Для поиска в содержимом строки существуют функций strchr(s,ch) и strstr(s1,s2).

```
// инициализация массива копированием
strcpy_s(text, "New text");
strcpy_s(text, str);
```

1.2.3 Указатели

Указатель – это переменная, содержащая адрес другой переменной. При объявлении указателя используется *, кроме того,

указывается тип данных, на которые ссылается указатель.

```
<Тип_данных> *имя_указателя;
```

Применяя операцию * к указателю (операция разыменования), получаем значение, записанное по данному адресу. С помощью операции &, примененной к переменной, можно узнать адрес, по которому эта переменная хранится в памяти.

Пример. Объявление указателей и обращение к памяти. Значение переменной px – указатель (адрес памяти), а *px – данные, записанные по адресу px.

```
int x=100;    // переменная
cout << "\n x=" << x;
int *px;      // указатель на значение типа int
px = &x;      // указателю px присвоен адрес переменной x
*px = 200;   // Изменение значения, записанного по адресу px
```

При создании указателя память выделяется только для хранения адреса. Для выделения памяти под данные используется оператор new.

```
указатель = new <Тип>
```

Оператор new выделяет память в количестве, необходимом для хранения данных указанного типа. Указатель в левой части оператора присваивания должен быть указателем на тот же тип данных.

1.2.4 Указатели и массивы

Имя массива является указателем и содержит адрес первого элемента массива. Если, например, ms – имя массива, то *ms – значение первого элемента массива, а обращение ms[i] к i-му элементу массива есть сокращенная форма операции с указателями *(ms+i).

Пример:

```
const int N = 10;
int ms[N] = {0,10,20,30,40,50,60,70,80,90}; // массив из 10 элем.
int *v;     // указатель
v = ms + 5;
cout << *v; // будет выведено значение 50
```

Создание динамически размещаемого одномерного массива возможно с помощью оператора new:

```
p = new type [size]; // type – тип, size – количество элементов массива
```

Для удаления динамически размещаемого одномерного массива используется оператор delete []p, где p – указатель на блок памяти, ранее выделенный массиву, созданному при помощи оператора new.

Пример:

```
int *p;
int i;
p = new int [5]; // выделение памяти для 5 целых чисел
. . .
delete [ ] p;
```

1.2.5 Использование объектов класса string (строка)

В стандартной библиотеке C++ есть класс string с методами и переменными для организации работы со строками. Являясь частью стандартной библиотеки, объекты этого класса также являются частью стандартного пространства имен std. Класс string можно использовать, включив соответствующий заголовочный файл:

```
#include <string>
using namespace std;
```

Определение пустой строки типа string:

```
string st; // пустая строка
```

Объект типа string может быть инициализирован строкой символов при определении:

```
#include <string>
using namespace std;
string st2("Строка-объект\n");
```

Длину строки возвращает метод size() класса. Длина не включает завершающий нулевой символ.

```
cout << str2.size() << " символов\n";
```

Объект типа string можно инициализировать другим объектом того же типа. Ниже строка str3 инициализируется строкой str2:

```
string str3(str2);
```

Копирование строк можно выполнить с помощью обычной операции присваивания:

```
str2 = str3; // копируем str3 в str2
```

Для конкатенации строк используется операция сложения (+) или операция сложения с присваиванием (+=). Пример:

```
string s1("hello, ");
string s2("world\n");
string s3 = s1 + s2;
```

Чтобы добавить s2 в конец строки s1, следует написать:

```
s1 += s2;
```

Операция сложения может конкатенировать объекты класса string не только между собой, но и со строками встроенного типа char. Можно переписать предыдущий пример так, чтобы специальные символы и знаки препинания представлялись встроенным типом, а значимые слова – объектами класса string:

```
const char *pc = ", ";
string s1( "hello" );
string s2( "world" );
string s3 = s1 + pc + s2 + "\n";
```

Подобные выражения работают потому, что компилятор автоматически преобразует объекты встроенного типа в объекты класса string. Возможно и простое присваивание встроенной строки объекту string:

```
string s1;
const char *pc = "a character array";
s1 = pc;    // правильно
```

1.2.6 Определение структур

Структуры – это композитные (составные) типы данных, они позволяют объединить под одним именем совокупность данных разных типов. Рассмотрим следующее определение структуры:

```
struct Person
{     char name[20];
      char surname[20];
      int age;
};
```

Ключевое слово struct начинает определение структуры. Идентификатор Person – это тег (имя) структуры. Тег структуры используется при объявлении переменных в качестве наименования типа. Имена, объявленные в фигурных скобках – это элементы структуры. Элементы одной и той же структуры должны иметь уникальные имена, но две разные структуры могут содержать элементы с одинаковыми именами. Элементы структуры могут быть любого типа. Каждое определение элемента должно заканчиваться точкой с запятой.

Переменные типа структуры объявляются аналогично переменным других типов.

```
Person newPerson, personArray[10], *personPtr;
```

Здесь объявлена структура, массив структур и указатель на структуру типа Person.

Для доступа к элементам структуры (или класса) используются операции доступа к элементам – *операция точка* (.) и *операция стрелка* (->). Операция точка обращается к элементу структуры (или класса) по имени переменной объекта. Например, чтобы вывести значение элемента age структуры newPerson, используется оператор:

```
cout << newPerson.age << endl;
```

Для доступа к элементам структуры с помощью указателя используется операция стрелка. Присвоение указателю personPtr на структуру Person значения:

```
personPtr = new Person[50];
```

Присвоение элементу age структуры значения 18:

```
personPtr -> age=18;
```

1.2.7 Функции

Для более эффективного управления кодом используется разделение программы на отдельные логические модули – функции. Модульное программирование предполагает:

- объявление функции (создание прототипа), включающее имя функции, тип возвращаемого ею значения и типы формальных параметров функции;
- определение функции, включающее заголовок функции и последовательность операторов, реализующих функцию;
- вызов функции путем указания ее имени и замены формальных параметров значениями (фактическими параметрами).

Пример использования функции:

```
#include <iostream>
using namespace std;
int sum(int, int); // Объявление функции sum - прототип

int main()
{       int a,b,c;
        cin >> a >> b;
        c = sum(a, b); // Вызов функции
        cout << "\n sum = " << c;
        return 0;
}

int sum(int a, int b) // Определение функции
{       return a+b; }
```

Объявление функции до обработки ее вызова определяет для компилятора имя функции и типы ее параметров. Прототип функции является оператором и должен заканчиваться точкой с запятой. В прототипе функции можно не приводить имена параметров. Достаточно указать список типов параметров.

```
<Возвращаемый_тип> Имя_функции(<список типов_параметров>);
```

Если блок определения функции размещается до блока, в котором производится вызов функции, то объявление прототипа не обязательно.

Пример:

```
#include <iostream>
using namespace std;

int sum(int a, int b) // Определение функции
{    return a+b; }

int main()
{    int a,b,c;
     cin >> a >> b;
     c = sum(a, b); // Вызов функции
     cout << "\n c = " << c;
     return 0;
}
```

Использование прототипов позволяет компилятору правильно обрабатывать возвращаемые значения функций, следить за соответствием количества аргументов при вызове функции, корректировать применение типов данных для аргументов функции и пытаться, если это возможно, выполнить приведение типов, в случае несоответствия типов при вызове.

Часто прототипы функций помещают в отдельные файлы – заголовочные файлы (эти файлы имеют расширение .h).

Определение функции начинается с заголовка. В отличие от прототипа функции, заголовок не должен заканчиваться точкой с запятой. В заголовке обязательно указываются имена параметров.

Тип возвращаемых функцией значений может быть указан как void, т.е. отсутствие возвращаемого значения. Такая функция может завершиться и вернуть управление в вызывающую программу оператором return без параметров, но может и не содержать его:

```
return;
```

Функция с другим типом возвращаемого значения должна содержать хотя бы один оператор return с параметром соответствующего типа, который и будет возвращаемым значением. В качестве параметра можно использовать результат вычисления выражения. Оператор завершает выполнение функции.

Синтаксис оператора:

```
return      <выражение>;
```

Переменные, перечисленные в заголовке функции, называются формальными параметрами. Эти переменные служат для передачи значений в функцию. При вызове функции формальные параметры заменяются значениями – фактическими параметрами.

Все переменные, объявленные в функции, включая параметры функции, действуют только в пределах данной функции. Во время вызова функции компилятор выделяет память для этих переменных, а по завершению выполнения функции эта память освобождается.

Параметры функции бывают двух видов: параметры-значения и параметры-ссылки. Параметры-значения функции могут рассматриваться как локальные переменные, для которых выделяется память в стеке при вызове функции и производится копирование значений фактических аргументов. При завершении работы функции происходит очистка стека от данных, принадлежащих этой функции, при этом значения переменных теряются. Такой способ передачи параметров в функцию не изменяет значения переменных в вызывающей функции, являющихся фактическими параметрами.

Пример передачи в функцию параметров-значений b и c:

```
int fn1(int b, float c);  // параметры-значения
```

Второй способ передачи параметров представляет собой передачу в функцию адреса фактического аргумента. Обращение к фактическому аргументу по адресу позволяет вызванной функции изменить его значение в вызвавшей эту функцию подпрограмме.

В качестве такого параметра можно использовать указатель на некоторую переменную:

```
int fn2(int *pb);       // параметр-указатель
```

Используя операцию косвенной адресации в теле функции, можно изменить значение этой переменной:

```
*pb = 6;
```

Другой способ передачи в функцию адреса переменной – передача параметра по ссылке. При этом в вызванной функции создается псевдоним исходной переменной, форма обращения к

такому параметру-ссылке такая же, как и к обычной переменной, а сам параметр передается с помощью адреса. Например:

```
int fn3(int &ab);  // параметр-ссылка
```

При этом обращение к переменной ab в теле функции fn3() такое же, как к обычной переменной типа int.

Пример. Передача по значению (функция обмена в синтаксисе языка С).

```
#include <iostream>
using namespace std;
void swapargs(int *px, int *py);

int main()
{       int i, j;
        i = 10; j = 19;
        cout << "i: " << i << ", ";
        cout << "j: " << j << "\n";
        swapargs(&i, &j);
        cout << "После перестановки: ";
        cout << "i: " << i << ", ";
        cout << "j: " << j << "\n"; return 0;
}

void swapargs(int *px, int *py)
{       int t;
        t = *px;*px = *py; *py = t;
}
```

1.2.8 Ссылочные переменные

В языке С++ введен новый составной тип данных – ссылочная переменная. Ссылка представляет собой имя, которое является псевдонимом для ранее объявленной переменной. Для объявления ссылочной переменной используется символ «&».

Пример:

```
#include <iostream>
using namespace std;
int main()
{       int x;
        int &r = x; // создание независимой ссылки
        x = 10;     // эти две инструкции
```

```
        r = 10;     // идентичны
        return 0;
}
```

Основное назначение ссылок – использование в качестве формальных параметров функций. Используя ссылку в качестве аргумента, функция работает с исходными данными, а не с их копиями.

Пример передачи аргументов по ссылке (функция, меняющая значения аргументов):

```
#include <iostream>
using namespace std;
void swapargs (int &x, int &y);

int main()
{       int i, j;
        i = 10;
        j = 19;
        cout << "i: " << i << ", "; cout << "j: " << j << "\n";
        swapargs(i, j);
        cout << "После перестановки: ";
        cout << "i: " << i << ", "; cout << "j: " << j << "\n"; return 0;
}

void swapargs (int &x, int &y)
{       int t;
        t = x; x = y; y = t;
}
```

При передаче массива в качестве параметра функции нужно после имени массива указать пустые квадратные скобки. Например,

```
int sum_arr(int arr[], int size_arr);
```

В вызове функции указать имя массива нужно уже без квадратных скобок:

```
sum = sum_arr(mass, N);
```

1.2.9 Потоковый ввод/вывод

В языке C++ производится ввод-вывод потоков байтов. Поток – это просто последовательность байтов. В операциях ввода байты

пересылаются от устройства (например, от клавиатуры, дисковода) в оперативную память. При выводе байты пересылаются из оперативной памяти на устройства (например, на экран дисплея).

Заголовочный файл iostream.h объявляет объекты cin и cout, которые соответствуют стандартным потокам ввода и вывода.

<< – операция – поместить в поток;

>> – операция – взять из потока;

\n – переход на новую строку (так же можно применять манипулятор потока endl, в отличие от '\n' этот манипулятор не только переводит строку, но и выгружает в поток содержимое буфера вывода).

Пример:

```
cout << "Test " << endl;
cout << "Test \n";
```

1.2.10 Консольный ввод-вывод средствами языка С, включенными в библиотеку С++

Процесс реализованного в функциях из С-библиотеки форматирования существенно проще, чем форматирование методами ввода и вывода языка С++.

Две функции printf_s() и scanf_s() выполняют соответственно консольный вывод и ввод. Этим функциям передается переменное число аргументов, но первым аргументом всегда является текстовая строка, называемая строкой формата. Она задает способ преобразования данных. Эти функции объявлены в файле stdio.h.

Функция printf_s() выдаст на экран всю форматную строку, в которой знак %xxx заменяется символами выводимой информации. Символы после % до первого разделителя рассматриваются как спецификация преобразования значения выводимой переменной. Имя переменной – это второй, третий и т.д. аргументы списка аргументов.

Спецификация преобразования задается в виде последовательности

% [флаги] [ширина] [.точность] <тип>

В таблице 1 перечисляются некоторые из флагов, задающих спецификацию преобразования.

Таблица 1

Флаги спецификации преобразований

Флаги	Действие
- + Пробел	Производится выравнивание влево выводимого значения в пределах выделенного поля. Правая сторона выделенного поля дополняется пробелами. По умолчанию устанавливается выравнивание вправо. Выводится знак числа символом '-' или '+'. Выводится пробел перед числом, если оно положительно. Для отрицательных чисел знак '-' выводится всегда.
#	Выводится идентификатор системы счисления для целых: - 0 перед числом, выводимым в восьмеричной системе счисления; - 0x или 0X для чисел, выводимых в шестнадцатеричной системе счисления; - ничего для чисел, выводимых в десятичной системе счисления. Выводится десятичная точка для чисел типа float.

Поле ширина воздействует только на вывод и определяет минимальную ширину поля в n символов. Если после преобразования ширины недостаточно, выводится столько символов, сколько есть. Незаполненные в пределах ширины позиции дополняются пробелами.

Поле тип определяет способ представления последовательности символов. Коды основных типов приведены в таблице 2.

Таблица 2

Типы

Тип преобразования	Тип переменной	Действие
c	char	При вводе из файла читается и передается переменной один байт. При выводе переменная преобразуется к типу char. В файл передается один байт

Тип преобразования	Тип переменной	Действие
d	int	Десятичное int со знаком
i	int	Десятичное int со знаком
o	int	Восьмеричное int unsigned
x	int	Шестнадцатеричное int unsigned; при выводе для числа используются символы 0 - F
s	char	При вводе принимает символы без преобразования до тех пор, пока не встретится '\n' или пока не достигнута специфицированная точность. В программу передаются символы до '\n' или пробела. При выводе выдает в поток все символы до тех пор, пока не встретится '\0' или пока не достигнута специфицированная точность

Функция scanf_s() считывает символы, вводимые с клавиатуры (из стандартного входного потока stdin) до нажатия клавиши ENTER, и помещает их в буфер. Затем по форматной строке (первый аргумент функции) определяется способ преобразования введенных символов в соответствии с заданными в ней спецификациями. Используется тот же способ записи спецификаций преобразования символов, что и в форматной строке функции printf_s().

Аргументы функции scanf_s(), следующие за форматной строкой, должны быть указателями на переменные, которые имеют типы, соответствующие спецификаторам в форматной строке. При выполнении функции преобразуемые значения помещаются по адресам, определяемым этими указателями.

Разделителем между вводимыми данными являются символы пробела, табуляции или конца строки. Поэтому в общем случае нет необходимости описывать ширину поля и точность вводимых данных.

Для scanf_s требуется также указывать размер буфера в символах типа char для каждого из входных параметров типа c и s. Размер

буфера передается как дополнительный параметр после указателя соответствующей переменной. Для этих целей может быть полезной функция sizeof(arg). Она возвращает размер аргумента arg в соответствии с его типом. Для строк типа массива символов возвращает размер в виде количества символов, включая символ "\n" конца строки.

Пример 1. Вывод значений двух переменных целого типа col, nm и строковой переменной name

```
#include <stdio.h>
{
    int col=234;
    int nm=77;
    char name[10] = "namestr\n";

    printf_s( "%4d %4d %25s", col, nm, name);
    // вывод переменных целого типа col, nm в 16-ричной системе счисления
    printf_s("%4x %4x %25s", col, nm, name);
}
```

Пример 2. Ввод значений различных переменных

```
char op;
int PID;
char fio[30];
// ввод значений символьной переменной и переменной целого типа
printf_s("Введите вид операции и PID процесса: \n");
scanf_s("%1c %4d", &op, 1, &PID);  // размер буфера 1 байт

// ввод строки – массива символов
printf_s("Введите фамилию, имя, отчество (лат) \n");
scanf_s("%29s", fio, sizeof(fio)); // здесь fio – указатель массива
```

Для вывода строк в кодировке Unicode используется функция wprintf_s(L”%s”, name_string). Например, для вывода строки wS из раздела 1.2.1 следует использовать эту функцию:

```
wprintf_s(L"%30s \n", wS);
```

1.2.11 Оператор if проверки условия

Синтаксис оператора:

```
if (выражение) оператор 1; [else оператор 2;]
```

В квадратные скобки заключена необязательная конструкция.

Первый этап выполнения оператора if – вычисление выражения. Дальнейшие действия зависят от того, является результат истиной или ложью. Даже если выражение не является логическим, оно считается ложным при результате 0 и истинным при любом другом. Далее:

- если выражение истинно, то выполняется оператор 1;
- если выражение ложно, то выполняется оператор 2;
- если выражение ложно и отсутствует оператор 2, то выполняется следующий за if оператор.

После выполнения оператора if управление передается следующему оператору программы, если последовательность выполнения операторов не будет принудительно нарушена использованием операторов перехода.

Пример:

```
int i=5;
int j=0;
if (i < j) i++; else { j = i-3; i++; }
// сравнение символов
char ch='\n';
if (ch=='\n') cout <<"\n ch= \\n";
```

Допускается использование вложенных операторов if. Оператор if может быть включен в конструкцию if или в конструкцию else другого оператора if.

1.2.12 Оператор for

Оператор for – это наиболее общий способ организации цикла. Он имеет следующий формат:

```
for ( <выражение 1> ; <выражение 2> ; <выражение 3> ) <тело цикла>;
```

Выражение 1 обычно используется для установки начального значения переменных, управляющих циклом (например, счетчика).

Выражение 2 задает условие, при котором тело цикла будет выполняться. Выражением 3 задается изменение управляющих циклом переменных после каждого выполнения его тела (например, увеличение значения счетчика на 1).

Схема выполнения оператора for:

1. Вычисляется выражение 1.
2. Вычисляется выражение 2.
3. Если значения выражения 2 отлично от нуля (истина), выполняется тело цикла, вычисляется выражение 3 и осуществляется переход к пункту 2; если выражение 2 равно нулю (ложь), то управление передается на оператор, следующий за оператором for.

Существенно то, что проверка условия всегда выполняется в начале цикла. Это значит, что тело цикла может ни разу не выполниться, если условие выполнения сразу будет ложным.

Пример – вычисление квадратов чисел от 1 до 9:

```
int main()
{       int i,b;
        for (i=1; i<10; i++) b = i*i;
        return 0;
}
```

1.2.13 Оператор while

Оператор while служит началом цикла с предусловием. Синтаксис такого цикла:

```
while (выражение) тело;
```

Выражение является условием цикла, оно может быть логическим или любым другим – если результат не равен нулю, оно будет считаться истинным. Если выражение ложно на момент передачи управления оператору while, цикл не выполнится ни разу. Т.е., оператор while работает по следующей схеме:

1. Вычисляется выражение.

2. Если выражение ложно, то выполнение оператора while заканчивается, и выполняется следующий по порядку оператор. Если выражение истинно, то выполняется тело оператора while.
3. Осуществляется переход к пункту 1.

Пример. Считывание вводимых с клавиатуры символов, завершающееся при обнаружении символа “.” (точка).

```
char ch = '\0'; // начальное значение
while (ch != '.') ch = getchar();
```

1.2.14 Оператор do while

Оператор do while – это цикл с постусловием. Он используется в тех случаях, когда тело цикла должно гарантированно выполниться хотя бы один раз. Синтаксис этого оператора:

```
do тело while (выражение);
```

Схема выполнения оператора do while:

1. Выполняется тело цикла (которое может быть составным оператором).
2. Вычисляется выражение.
3. Если выражение ложно, то выполнение оператора do while заканчивается, и выполняется следующий по порядку оператор. Если выражение истинно, то выполнение оператора продолжается с пункта 1.

Оператор break прерывает выполнение любого цикла независимо от условий. Оператор continue осуществляет пропуск следующей за ним части тела цикла с переходом на этап проверки условия. Циклы всех типов могут быть вложенными друг в друга, тогда операторы break и continue действуют на внутренний цикл, в теле которого встречаются.

Пример цикла while, вложенного в цикл do while:

```
int i,j,k;
i=0; j=0; k=0;
do
{       i++; j--;
        while (a[k] < i) k++;
} while (i<30 && j<-30);
```

1.2.15 Работа с файлами

Файл – это последовательность (поток) байтов, завершающаяся специальным символом – маркером конца файла. Каждому открытому файлу ставится в соответствие поток.

При успешном открытии файла с ним можно работать посредством указателя на структуру FILE, заполненную информацией о файле. Эта структура определена в stdio.h.

Оператор FILE *f объявляет, что переменная f является указателем на структуру типа FILE. С каждым файлом необходимо связать отдельную структуру FILE.

Вызов функции

```
fopen_s(&f, "c:\\result.txt", "w");
```

открывает файл. Возвращает код ошибки при выполнении открытия файла. Если это значение равно 0, значит, файл открыт успешно.

Первый параметр – указатель на файловый указатель. Его получит указатель на открытый файл. Второй параметр – имя файла, третий – режим его открытия. В примере использовано полное имя файла, начинающееся с пути от корневого каталога диска (c:\\). Режимы открытия файлов приведены в таблице 3.

Таблица 3

Режимы открытия файлов

Параметр	Режим
r	режим чтения данных из файла
w	режим записи в файл
a	добавление в конец файла (или создание, если файл не обнаружен)

Параметр	Режим
r+	открытие файла для обновления (чтение и запись)

После завершения работы с файлом следует использовать функцию fclose(f) для его закрытия. Ее единственный аргумент – это указатель файла.

Вызов функции

```
fprintf_s(f, "%d ", x);
```

записывает данные в файл, на который указывает первый параметр этой функции – переменная f. Далее в списке параметров следует форматная строка, как в функции printf_s(), а затем список переменных, значения которых нужно записать в файл в формате, заданном в форматной строке.

Пример записи в текстовый файл. Замечание: каталог c:\\aa должен существовать.

```
#include <stdio.h>

FILE *f;
void main()
{
 char *fname = "c:\\aa\\result.txt";
 if (fopen_s(&f,fname,"w")> 0)
   { printf_s("Ошибка открытия файла  %s",fname);
    return 0;
   }
 for (int i=0; i<10; i++)
  { x=i*2;   fprintf_s(f," %10d ",x);   }
 fclose(f);

 return 0;
}
```

Для чтения из файла используется функция fscanf_s(). Эта функция является аналогом функции scanf_s(), но с дополнительным параметром – указателем на файл, из которого будет производиться чтение. Так же, как и для scanf_s(), требуется указывать размер буфера для входных параметров типа c и s.

Пример чтения из файла:

```
FILE *fp;
int n; float x;
char ftype[20];

fopen_s(&fp, "d:\\result.txt", "r");  // для чтения
fscanf_s(fp, "%s", ftype, 20);        // 20 – размер буфера
fscanf_s(fp, "%d %f", &n, &x);
```

1.3 Функции получения системной информации

Интерфейс прикладного программирования (Application Programming Interface Win32 – Win32 API) – это программный интерфейс, который используется для управления 32-разрядными операционными системами Windows. Win32 API состоит из набора функций, предоставляющих программный доступ к возможностям операционной системы. В составе Win32 API более 3000 функций для реализации всех видов сервисов операционной системы.

API-функции Windows входят в состав динамически подключаемых библиотек. Для размещения API-функций использует несколько DLL. Большая часть функций Win32 API содержится в трех DLL:

KERNEL32.DLL – предоставляет порядка 700 функций для управления памятью, процессами и потоками;

USER32.DLL – содержит около 600 функций, предназначенных для управления пользовательским интерфейсом, например, создания окон и передачи сообщений;

GDI. DLL – содержит порядка 400 функций для отрисовки графических объектов, отображения текста и работы со шрифтами.

Основные API-функции из KERNEL32.DLL, служащие для получения системной информации, перечислены в таблице 4.

Таблица 4

Основные функции получения системной информации

Функция	Возвращаемое значение
GetComputerName	Имя компьютера
GetUserName	Имя пользователя

Функция	Возвращаемое значение
GetWindowsDirectoty	Путь к каталогу Windows
GetSystemDirectory	Путь к системному каталогу
GlobalMemoryStatusEx	Информация об используемой системой памяти.

Функция GetComputerName используется для получения текущего имени компьютера.

```
BOOL GetComputerName
( LPTSTR lpBuffer,    // указатель на буфер для имени
  LPDWORD lpnSize   // указатель на размер буфера
);
```

Пример получения имени компьютера:

```
#include "stdafx.h"
#include "windows.h"
#include <iostream>
using namespace std;

void main()
{   setlocale(LC_CTYPE, "rus");
    TCHAR buffer[MAX_COMPUTERNAME_LENGTH+1];
    DWORD size = sizeof(buffer);
    GetComputerName(buffer, &size);
    wprintf_s(L"Компьютер %20s \n", buffer);
    getchar();
}
```

Функция GetUserName используется для получения текущего имени пользователя.

```
BOOL GetUserName
( LPTSTR lpBuffer, // указатель на буфер
LPDWORD nSize     // указатель на размер буфера
);
```

Функция GetWindowsDirectory служит для получения каталога Windows.

```
UNIT GetWindowsDirectory(
LPTSTR IpBuffer,        // указатель буфера каталога Windows.
UNIT nSize                 // размер буфера каталога.
 );
```

Если функция выполнится успешно, то она возвратит ненулевое значение.

Аналогичная функция GetSystemDirectory служит для получения системного каталога.

```
UNIT GetSystemDirectory(
LPTSTR IpBuffer,          // указатель буфера системного каталога
UNIT nSize                // размер буфера каталога
);
```

Сведения о конкретной платформе (совокупности аппаратно-программных средств) и параметрах виртуального адресного пространства предоставляет функция:

VOID GetSystemInfo (OUT LPSYSTEM_INFO lpSystemInfo).

Структура данных SystemInfo описана в файле winbase.h следующем образом:

```
typedef struct _SYSTEM_INFO {
  union {                        // объединение (или … или)
    DWORD dwOemId;       // не используется
    struct {
        WORD wProcessorArchitecture;
        WORD wReserved;   // не используется
     };
  };
  DWORD dwPageSize;     // размер страницы
  LPVOID lpMinimumApplicationAddress; //мин. адрес доступного адресного пространства
  LPVOID lpMaximumApplicationAddress; //макс. адрес доступного адресного пространства
  DWORD dwActiveProcessorMask;
  DWORD dwNumberOfProcessors;  // число процессоров
  DWORD dwProcessorType;          // устарел, не используется
  DWORD dwAllocationGranularity;   // гранулярность размещения регионов адресного
  // пространства процесса
  WORD wProcessorLevel;
  WORD wProcessorRevision;
} SYSTEM_INFO;
```

Поле wProcessorArchitecture указывает архитектуру процессора. Значение 0 соответствует архитектуре Intel (x86), значение 6 – архитектуре Intel Itanium (IA64). Поле dwProcessorType содержит код типа процессора. Например, код 586 означает PROCESSOR_INTEL_PENTIUM.

Поле wProcessorLevel определяет уровень процессора.

Поле wProcessorRevision указывает версию процессора.

Поле dwAllocationGranularity определяет гранулярность размещения регионов в адресном пространстве.

Пример вызова функции GetSystemInfo для получения размера страницы виртуальной памяти:

```
SYSTEM_INFO sysinf;
    GetSystemInfo(&sysinf);
    cout << sysinf.dwPageSize;
```

1.4 Работа с отладчиком Visual C++

Отладчик, встроенный в среду Microsoft Visual Studio, позволяет временно останавливать запущенную программу и просматривать значения переменных в определенный момент. Так, щелчок левой кнопкой мыши напротив строки исходного кода задает точку останова: отладчик остановит выполнение программы перед этой строкой.

Для того чтобы запустить программу в отладочном режиме, надо выбрать пункт меню Отладка/Начать отладку (F5) или нажать соответствующую кнопку на панели инструментов. После остановки программы в точке останова можно воспользоваться пунктом меню Отладка/Шаг с обходом (F10) для пошагового выполнения программы без захода в вызываемые функции.

Просмотреть значение переменной можно в окне Интерпретация (Ctrl+D,I), добавив туда ее имя (ввод с клавиатуры).

2. МЕТОДИКА ВЫПОЛНЕНИЯ

1. Базовые задания для всех бригад.

а. Создать консольное приложение.

б. В функции main() созданного приложения объявить переменную типа int и ввести ее значение, равное номеру бригады, с

помощью cin.

Объявить массивы символов и инициализировать их фамилиями студентов бригады.

Воспользовавшись объектом потокового вывода cout, вывести на экран текст «Бригада № » и далее значение указанной переменной типа int. С новой строки вывести в столбик фамилии членов бригады из массивов символов. Перейти на новую строку.

Повторить вывод той же информации, воспользовавшись функцией printf_s().

в. В эту же функцию main() вставить строки:

```
int mas[3] = { 1, 2, 3 };
int mmm[3] = { 4, 5, 6 };
```

С помощью функции printf_s() вывести на экран с новой строки значения элементов этих массивов, отделяя их друг от друга запятыми.

2. Реализовать инициализацию массива структур данных, а также вывод на экран и в текстовый файл отдельных элементов этого массива в соответствии со своим вариантом индивидуального задания, приведенным в таблице 5. Количество элементов массива равно 5. Параметры условия отбора вводить в клавиатуры.

Таблица 5

Варианты индивидуальных заданий для бригад

№ бригады	Структура	Поля	Условие отбора
1	Дом	Адрес, общая жилая площадь дома, количество жильцов	Дом, в котором жилая площадь на одного жильца менее s
2	Человек	Имя, фамилия, год рождения	Возраст человека более t лет
3	Треугольник	Длины сторон: a, b, c	Площадь треугольника менее S
4	Школьник	Фамилия, имя, возраст, пол	Мальчики старше t лет
5	Студент	Фамилия, имя, курс, вес	Студент n-го курса весом менее 50 кг

№ бригады	Структура	Поля	Условие отбора
6	Автомобиль	Марка, мощность, год выпуска	Автомобиль мощностью свыше p л.с., выпущенный после t-го года.
7	Отрезок	x1, y1, x2, y3	Длина отрезка меньше или равна L
8	Смартфон	Наименование, операционная система, емкость оперативной памяти	Емкость оперативной памяти свыше k1, но не более k2 (k2 > k1)
9	Работник	Фамилия, имя, зарплата, образование	Работник с высшим образованием и с зарплатой более w рублей
10	Рабочий	Фамилия, имя, должность, стаж	Слесарь, стаж свыше t лет

3. Разработать консольное приложение для определения и вывода системной информации в соответствии со своим вариантом индивидуального задания, приведенным в таблице 6.

Таблица 6

Варианты индивидуальных заданий для бригад

№ бригады	Системная информация
1	Имя пользователя, имя компьютера
2	Гранулярность размещения регионов в адресном пространстве, путь к системному каталогу
3	Имя компьютера, путь к каталогу Windows
4	Путь к системному каталогу, количество процессоров
5	Минимальный адрес доступного адресного пространства, архитектура процессора
6	Максимальный адрес доступного адресного пространства, размер страниц
7	Имя пользователя, размер страниц
8	Архитектура процессора, путь к каталогу Windows
9	Гранулярность размещения регионов в адресном пространстве, имя пользователя
10	Количество процессоров, максимальный адрес доступного адресного пространства

3. СОДЕРЖАНИЕ ОТЧЕТА

Отчет о работе должен включать:

1. Титульный лист с указанием номера и названия работы.
2. Общее и индивидуальное задание.
3. Листинги разработанных приложений.
4. Результаты выполнения индивидуального задания.

Контрольные вопросы

1. Пример консольного приложения для вывода на экран строки символов.
2. Как осуществляется русификация приложений?
3. Объявление и инициализация массивов.
4. Указатели и массивы.
5. Использование объектов класса string.
6. Определение структур.
7. Ссылочные переменные.
8. Потоковый ввод/вывод.
9. Консольный ввод-вывод средствами языка C.
10. Назначение строки формата в консольном выводе.
11. Вывод строк Unicode.
12. Формальные и фактические параметры функций.
13. Операторы проверки условия.
14. Операторы цикла.
15. Различие циклов с предусловием и с постусловием.
16. Ввод и вывод в файл.
17. Назначение функции GlobalMemoryStatusEx.
18. С помощью каких функций Win32 API можно получить имя компьютера, путь к каталогу Windows?

ЛАБОРАТОРНАЯ РАБОТА № 2

МОНИТОР ПРОЦЕССОВ И ПОТОКОВ

Цель работы – практическое знакомство с методикой использования функций Win32 API для получения информации о процессах, потоках, модулях и кучах ОС Windows в консольном приложении на языке C++.

1. КРАТКИЕ ТЕОРЕТИЧЕСКИЕ СВЕДЕНИЯ

1.1 Получение информации о процессах, выполняющихся в системе

Одна из основных задач при мониторинге ресурсов компьютера (а также локальной вычислительной сети) – получение списка процессов, выполняющихся в операционной системе. Операционная система Windows имеет встроенное системное средство для этого – Диспетчер задач. Также существует множество утилит от других разработчиков: Process Explorer, System Explorer и т. п. [4].

В Win32 API имеется ряд функций, которые позволяют получить информацию о выполняющихся в данный момент процессах. К их числу относятся такие функции, как CreateToolhelp32Snapshot, Process32First, Process32Next, Thread32First, Thread32Next, Module32First, Module32Next, Heap32ListFirst, Heap32ListNext [2]. Эти функции – из библиотеки Tool Help Library, входящей в Win32 API.

1.1.1 Использование функций CreateToolhelp32Snapshot и Process32xxxx для получения списка процессов

Получение информации о процессах, выполняющихся в ОС, начинается с создания снимка системы. Снимок (snapshot) содержит сведения о состоянии системы на момент его создания.

Снимок создается с помощью вызова функции CreateToolhelp32Snapshot (dwFlags, th32ProcessID). Первый

аргумент определяет вид информации, записываемой в снимок. Возможные значения dwFlags приведены в таблице 7 [1, 6].

Таблица 7

Значения флагов dwFlags функции CreateToolhelp32Snapshot

Значение флага	В снимок будет включаться
TH32CS_SNAPHEAPLIST	информация о динамически распределяемой памяти указанного процесса
TH32CS_SNAPPROCESS	список присутствующих в системе процессов
TH32CS_SNAPTHREAD	список присутствующих в системе потоков
TH32CS_SNAPMODULE	список модулей, принадлежащих указанному процессу
TH32CS_SNAPALL	список куч, процессов, потоков и модулей

Второй аргумент функции содержит идентификатор процесса, информация о котором необходима (в тех случаях, когда снимок должен возвращать список модулей или куч указанного процесса – иначе аргумент игнорируется).

Второй этап заключается в извлечении информации из снимка. В случае, когда в вызове функции CreateToolhelp32Snapshot аргументу dwFlags задано значение TH32CS_SNAPPROCESS (или TH32CS_SNAPALL), из полученного снимка с помощью функций

```
BOOL Process32First (hSnapshot, lpPE);
BOOL Process32Next (hSnapshot, lpPE);
```

можно извлечь информацию о процессах, присутствующих в системе.

Первым аргументом функций типа HANDLE должен быть дескриптор снимка, полученного функцией CreateToolhelp32Snapshot.

Второй аргумент типа LPPROCESSENTRY32 – указатель на структуру типа PROCESSENTRY32, поля которой описаны в таблице 8. Эта структура заполняется при успешном выполнении функции CreateToolhelp32Snapshot.

Таблица 8

Поля структуры типа PROCESSENTRY32

Поле	Описание
dwSize	Размер структуры в байтах
cntUsage	Количество ссылок на процесс, то есть количество потоков, которые в настоящий момент используют какие-либо данные процесса
th32ProcessID	Идентификатор процесса
pcPriClassBase	Базовый приоритет процесса
szExeFile	Имя файла, создавшего процесс
cntThreads	Число потоков процесса
h32ParentProcessID	Идентификатор процесса, родительского по отношению к текущему

Перед вызовом функций Process32xxxx полю dwSize необходимо присвоить значение размера структуры типа PROCESSENTRY32 в байтах (sizeof (PROCESSENTRY32)).

Функция Process32First позволяет получить информацию о первом процессе снимка. Возвращаемое значение TRUE соответствует успешному завершению функции. Для получения информации об остальных процессах необходимо последовательно вызывать функцию Process32Next, пока она не возвратит значение FALSE.

Пример. С помощью функций, рассмотренных выше, сформировать список имен процессов, выполняющихся в ОС.

Для подключения библиотеки Tool Help Library в список директив приложения необходимо включить директиву #include <tlhelp32.h>

Программа приложения для примера приведена ниже [6].

```
#include "stdafx.h"
#include <Windows.h>
#include <tlhelp32.h>
#include <clocale>
#include <iostream>
using namespace std;
```

```
int _tmain(int argc, _TCHAR* argv[])
{
// получение снимка процессов
HANDLE hSnap = CreateToolhelp32Snapshot(TH32CS_SNAPPROCESS,0);
PROCESSENTRY32 Pe;
Pe.dwSize = sizeof(PROCESSENTRY32);
if (Process32First(hSnap, &Pe))
  {
        do
        {       printf_s ("%6d", Pe.th32ProcessID);
                wprintf(L"%1s %-30s \n", " ", Pe.szExeFile);
        }
        while (Process32Next(hSnap, &Pe));
        CloseHandle(hSnap);
  }
}
```

На рисунке 3 показан результат выполнения приложения.

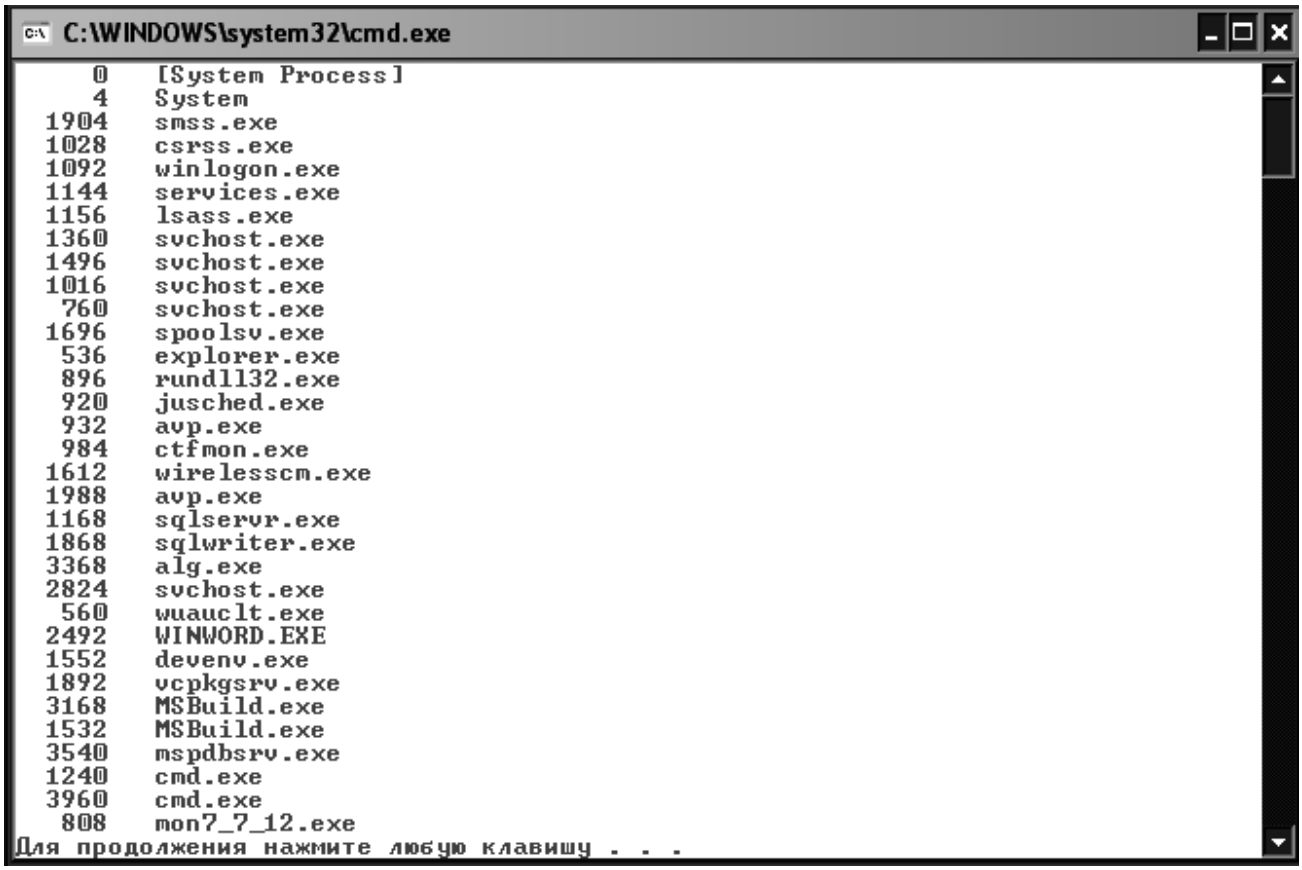


Рисунок 3 – Список выполняющихся процессов в ОС Windows

1.1.2 Использование функций CreateToolhelp32Snapshot и Thread32xxxx для получения сведений о приоритетах потоков

Для получения информации о потоках необходимо сначала с помощью вызова функции CreateToolhelp32Snapshot создать снимок потоков. Т.е. вызов функции должен быть с флагом TH32CS_SNAPTHREAD. Далее информацию о потоках можно извлечь с помощью функций:

BOOL Thread32First (hSnapshot, lpTE);

BOOL Thtead32Next (hSnapshot, lpTE);

Первый аргумент функций типа HANDLE – дескриптор снимка, возвращаемого функцией CreateToolHelp32Snapshot.

Второй аргумент типа LPTHREADENTRY32 – указатель на структуру типа THREADENTRY32, содержащую 7 полей. Наиболее важные поля структуры перечислены в таблице 9.

Таблица 9

Поля структуры типа THREADENTRY32

Поле	Описание
dwSize	Размер структуры в байтах.
th32ThreadID	Идентификатор потока, совместимый с идентификатором потока, возвращаемым функцией CreateProcess
th32OwnerProcessID	Идентификатор процесса – владельца потока
tpBasePri	Текущий приоритет потока
tpDeltaPri	Разность между текущим уровнем приоритета потока и базовым (уровнем, присвоенным потоку при создании)

Предварительно перед вызовом функций Thread32xxxx полю dwSize следует присвоить значение размера структуры типа THREADENTRY32 в байтах (sizeof (THREADENTRY32)).

Таким образом, из структуры THREADENTRY32 можно получить имя очередного потока и его текущий приоритет.

1.1.3 Использование функций CreateToolhelp32Snapshot и Module32xxxx для получения списка модулей

Модули процесса – это исполняемые модули, которые отображены (загружены) в его адресное пространство. Это могут быть динамические библиотеки (dll), драйверы (drv) и управляющие элементы (ocx), основной загрузочный модуль (exe) процесса, который иногда и называют собственно модулем.

Сведения о модулях какого-либо процесса можно получить из снимка, формируемого функцией CreateToolhelp32Snapshot с аргументами TH32CS_SNAPMODULE и PID, где PID – переменная, содержащая идентификатор выбранного процесса.

Для последовательного доступа к структурам типа MODULEENTRY32, содержащим информацию о модулях процесса, необходимо использовать функции

BOOL Module32First (hSnapshot, lpME);
BOOL Module32Next (hSnapshot, lpME).

Аргумент hSnapshot типа HANDLE – дескриптор созданного снимка, lpME типа lpMODULEENTRY32 – указатель на структуру типа MODULEENTRY32, поля которой описаны в таблице 10.

Таблица 10

Поля структуры типа MODULEENTRY32

Поле	Описание
dwSize	Размер структуры в байтах
th32ModuleID	Не используется, всегда равно 1
th32ProcessID	Идентификатор процесса, владеющего модулем
GlblcntUsage	Счетчик ссылок на модуль в системе
ProccntUsage	Счетчик ссылок на модуль в контексте процесса
modBaseAddr	Базовый адрес модуля
modBaseSize	Базовый размер модуля в байтах
hModule	Дескриптор модуля
szModule	Имя модуля
szExePath	Путь размещения модуля

Перед вызовом функций Module32xxxx полю dwSize необходимо присвоить значение размера структуры типа MODULEENTRY32 в байтах (sizeof (MODULEENTRY32)).

Чтобы получения сведения о модулях, используемых одним из процессов, необходимо:

- получить снимок всех выполняющихся процессов;
- извлечь из снимка имена процессов и их идентификаторы, вывести их на экран;
- выбрать идентификатор одного из прикладных процессов и ввести его с клавиатуры;
- используя введенный идентификатор в качестве второго аргумента функции CreateToolhelp32Snapshot, получить снимок списка модулей выбранного процесса;
- с помощью функций Module32First и Module32Next извлечь из снимка модулей информацию о каждом из них.

1.2 Получение дополнительной информации о процессах

Для получения сведений о времени работы процессов, использовании ими памяти и других ресурсов существует ряд функций Win32 API [6]. Краткая информация о некоторых из них приведена в таблице 11.

Таблица 11

Функции для получения информации о процессах

Функция	Описание
GetProcessTimes	Извлекает сведения о времени для указанного процесса
GetProcessHandleCount	Возвращает количество открытых дескрипторов, принадлежащих указанному процессу
GetProcessMemoryInfo	Получает информацию об использовании памяти указанного процесса

Функция	Описание
GetProcessWorkingSetSize	Получает минимальный и максимальный размеры рабочего множества страниц указанного процесса в байтах (множество виртуальных страниц процесса, находящихся в памяти).

1.2.1 Получение информации о времени выполнения процессов

Используя функцию GetProcessTimes для любого процесса, можно получить информацию о времени (CreationTime) запуска (создания), времени (ExitTime) завершения, времени (KernelTime) работы процесса в режиме ядра и времени (UserTime) работы процесса пользователя.

Первым аргументом функции

BOOL GetProcessTimes(hProcess, lpCreationTime, lpExitTime,
lpKernelTime, lpUserTime)

должен быть дескриптор процесса. lpCreationTime, lpExitTime, lpKernelTime, lpUserTime – ссылки типа LPFILETIME на структуры типа FILETIME. В таких структурах функция GetProcessTimes формирует информацию о времени.

Время, возвращаемое функцией, отсчитывается в 100-наносекундных интервалах с 12:00 1.01.1601 по Гринвичу (универсальное глобальное время (UTC)). Если в момент вызова GetProcessTimes процесс не завершился, содержание структуры ExitTime будет неопределенным.

Для представления времени в привычном формате используются функции FileTimeToLocalFileTime и FileTimeToSystemTime.

Функция BOOL FileTimeToLocalFileTime (lpFileTime, lpLocalFileTime) преобразует универсальное глобальное время формата FILETIME в локальное время такого же формата. Для локального времени используются текущие параметры настройки для часового пояса и летнего времени.

Функция BOOL WINAPI FileTimeToSystemTime (lpFileTime, lpSystemTime) преобразует время формата FILETIME во время формата SYSTEMTIME.

Структура SYSTEMTIME включает 8 полей типа WORD, определяющих системную дату и системное время. Имена и назначение полей приведены в таблице 12.

Таблица 12

Поля структуры SYSTEMTIME

Поле	Описание
wYear	Год
wMonth	Месяц в году
wDayOfWeek	День недели (число, 0 –воскресенье)
wDay	День в месяце
wHour	Часы
wMinute	Минуты
wSecond	Секунды
wMilliseconds	Миллисекунды

Используя поля структуры SYSTEMTIME, можно получить дату и время старта или завершения процесса с точностью до миллисекунды.

Пример использования перечисленных функций:

```
GetProcessTimes(hProc,&CreationTime,&ExitTime,&KernelTime,&UserTime);
FileTimeToLocalFileTime(&CreationTime,&LocCreationTime);

FileTimeToSystemTime(&LocCreationTime,&sysCreation);
FileTimeToSystemTime(&KernelTime,&sysKernel);
```

Для получения дескриптора процесса в общем случае следует использовать функцию OpenProcess():

```
hProc=OpenProcess(PROCESS_QUERY_INFORMATION, FALSE, P.th32ProcessID);
```

Дескриптор текущего процесса возвращается функцией GetCurrentProcess().

1.2.2 Получение информации об используемой процессом памяти

При создании каждого процесса в его распоряжении оказывается собственное виртуальное адресное пространство (ВАП). Обычно процессы используют только небольшую его часть. Поэтому физическая память выделяется не для всех виртуальных страниц, а только для ограниченного их количества. Такое подмножество виртуальных страниц процесса, расположенных в физической памяти, называется рабочим набором процесса (working set).

Функция Win32 API BOOL GetProcessWorkingSetSize(hProcess, lpMinimumWorkingSetSize, lpMaximumWorkingSetSize) возвращает в своих параметрах указатели lpMinimumWorkingSetSize и lpMaximumWorkingSetSize типа PSIZE_T соответственно на минимальный и максимальный размеры рабочего множества страниц в байтах для процесса с дескриптором hProcess.

Функция

```
BOOL GetProcessMemoryInfo(
    HANDLE hProcess,
    PPROCESS_MEMORY_COUNTERS ppsmemCounters,
    DWORD cb)
```

из библиотеки psapi.lib возвращает информацию об используемой процессом памяти.

Для подключения библиотеки psapi.lib в приложение необходимо добавить директивы #include <psapi.h> и #pragma comment (lib,"psapi.lib").

Процесс задается дескриптором hProcess. Второй параметр функции – структура типа PPROCESS_MEMORY_COUNTERS, содержащая информацию об использовании памяти. Третий параметр – размер структуры в байтах. Наименование и назначение ее полей приведено в таблице 13.

Таблица 13

Поля структуры PROCESS_MEMORY_COUNTERS

Поле	Описание
DWORD cb	Размер структуры
DWORD PageFaultCount	Количество страничных ошибок, возникающих при попытке обратиться к странице, отсутствующей в рабочем наборе
SIZE_T PeakWorkingSetSize	Пиковый размер рабочего набора в байтах
SIZE_T WorkingSetSize	Используемый размер рабочего набора в байтах
SIZE_T QuotaPeakPagedPoolUsage	Максимальный размер выгружаемого пула памяти
SIZE_T QuotaPagedPoolUsage	Размер выгружаемого пула памяти
SIZE_T QuotaPeakNonPagedPoolUsage	Максимальный размер невыгружаемого пула памяти
SIZE_T QuotaNonPagedPoolUsage	Размер невыгружаемого пула памяти
SIZE_T PagefileUsage	Использование страничного файла
SIZE_T PeakPagefileUsage	Максимальный размер использования файла подкачки

Пул (pool) – это область системной памяти, в которой компоненты операционной системы получают пространство для выполнения своих задач. Страницы невыгружаемого пула (nonpaged pool) не могут выгружаться в файл подкачки, они остаются в оперативной памяти в течение всего периода времени, на который они выделены. Страницы выгружаемого пула (paged pool) могут выгружаться из памяти в файл подкачки, когда система не осуществляет к ним доступ в течение длительного времени [1].

1.2.3 Оценка загрузки процессора процессом с использованием счетчиков производительности

Операционная система Windows при работе создает и поддерживает ряд счетчиков производительности. Их значения заносятся в реестр. Использование их данных позволяет оценивать в том числе загрузку процессора каждым процессом.

Различают счетчики производительности локального компьютера и счетчики производительности компьютеров локальной сети. Каждый счетчик относится к определенной категории. Например, имеются такие категории, как "Процессор", "Процесс", "Память", "Система" и т. д. Для некоторых из них есть несколько экземпляров счетчиков. Каждый счетчик имеет определенное функциональное назначение. Например, имеются счетчики категории "Процесс": "% загруженности процессора"; "% работы в пользовательском режиме" и другие [6].

Ознакомиться с категориями, экземплярами и назначением счетчиков производительности можно с помощью Монитора ресурсов (утилита perfmon).

Для доступа к значениям счетчиков используется библиотека PDH (Performance Data Helper).

Функция PdhOpenQuery(NULL, 0, &hQuery) из PDH создает новый запрос, который используется для управления сбором данных о производительности. hQuery – дескриптор запроса, к которому требуется добавить счетчик.

Библиотека PDH включает также следующие функции:

- PdhAddCounter(hQuery, szCounterPath, 0, &hCounter);
- PdhCollectQueryData(hQuery);
- PdhGetFormattedCounterValue(hCounter, PDH_FMT_DOUBLE, 0, &value).

Первая функция используется для добавления нового счетчика. Параметр hQuery типа HQUERY – дескриптор запроса. Параметр szCounterPath типа TCHAR должен содержать полный путь к

счетчику. Этот путь может быть задан строкой вида L"\\Процесс(name)\\% загруженности процессора", где name – имя файла без расширения, создавшего процесс. Но удобнее путь предварительно сформировать с помощью специальной функции PdhMakeCounterPath.

Параметр hCounter – дескриптор счетчика, который добавляется к запросу.

Функция PdhCollectQueryData служит для многократного обращения к выбранному счетчику для снятия измерений. Ее параметр – дескриптор запроса.

Третья функция выполняет форматирование данных, полученных счетчиком с дескриптором hCounter, в соответствии с заданным форматом. В примере задан формат PDH_FMT_DOUBLE – числа с плавающей точкой. Функция заносит результат форматирования в структуру value типа _PDH_FMT_COUNTERVALUE.

Ниже приведен пример простейшей программы для периодического вывода общей загруженности процессора.

```
#include "stdafx.h"
#include <stdio.h>
#include <conio.h>
#include <string>
#include <pdh.h>
#pragma comment(lib, "pdh.lib")

using namespace std;

int _tmain(int argc, _TCHAR* argv[])
{   setlocale(LC_CTYPE, "rus"); // русификация вывода
    PDH_STATUS  pdhResult   = 0;
    TCHAR       szCounterPath[1024];
    DWORD       dwPathSize  = 1024;
    PDH_COUNTER_PATH_ELEMENTS Cpe;  // структура с элементами пути
    HQUERY hQuery;
    HQUERY hCounter;
   _PDH_FMT_COUNTERVALUE pfCi;    // структура со значениями счетчика
    DWORD       dwType  = 0;
   // создание запроса для сбора данных о производительности
   pdhResult = PdhOpenQuery(NULL, 0, &hQuery);
```

```
    // заполнение структуры Cpe
    Cpe.szMachineName     = 0l;
    Cpe.szObjectName      = _T("Процессор");      // или "Процесс”
    Cpe.szInstanceName    = _T("_Total");          //  "0", …, или, например, “WINWORD”
    Cpe.szParentInstance  = NULL;
    Cpe.dwInstanceIndex   = 0;
    Cpe.szCounterName     = _T("% загруженности процессора");
    pdhResult = PdhMakeCounterPath(&Cpe, szCounterPath, &dwPathSize, 0);

    pdhResult = PdhAddCounter(hQuery, szCounterPath, 0, &hCounter);

    while (TRUE)
    {   Sleep(1000);
        pdhResult = PdhCollectQueryData(hQuery);
        ZeroMemory(&pfci, sizeof(pfCi));  // заполняет структуру нулями
        // форматирование и вывод
        pdhResult =
        PdhGetFormattedCounterValue(hCounter, PDH_FMT_DOUBLE, &dwType, &pfCi);
        if (pdhResult == ERROR_SUCCESS)
                wprintf(L"Загруженность процессора: %i%%\r\n", (int)pfCi.doubleValue);
    }

    PdhRemoveCounter(hCounter); // удаляем счетчик
    PdhCloseQuery(hQuery);          // закрываем запрос
    return 0;
}
```

Подробное описание перечисленных выше и использованных в примере функций можно найти в MSDN.

2. МЕТОДИКА ВЫПОЛНЕНИЯ

1. Выполнить базовые задания для всех бригад:

– вывести на экран список процессов, выполняющихся в системе.
– для выбранного процесса, задаваемого посредством ввода с клавиатуры его идентификатора, вывести сведения о его приоритете и количестве потоков.
– для выбранного прикладного процесса, задаваемого посредством ввода с клавиатуры его идентификатора, вывести время его работы в режиме ядра и в режиме пользователя.

2. Выполнить индивидуальное задание для бригады согласно таблице 14.

Таблица 14

Варианты индивидуальных заданий для бригад

№ бригады	Вариант задания
1	1. Для выбранного процесса вывести сведения об используемых им модулях: имя модуля и его размер. Идентификатор процесса вводить с клавиатуры. 2. Для выбранного прикладного процесса вывести количество страничных ошибок. Идентификатор процесса вводить с клавиатуры.
2	1. Вывести список имен выполняющихся процессов с указанием PID и количества потоков, упорядочить список по количеству потоков в процессах. 2. Для выбранного процесса вывести размер выгружаемого и невыгружаемого пулов памяти. Идентификатор процесса вводить с клавиатуры.
3	1. Для выбранного процесса вывести список имен дочерних процессов. Идентификатор процесса вводить с клавиатуры. 2. Для выбранного прикладного процесса вывести текущий и максимальный размер используемой оперативной памяти. Идентификатор процесса вводить с клавиатуры.
4	1. Вывести список выполняющихся процессов с указанием PID, имени, базового приоритета, количества потоков и используемых модулей. 2. Для выбранного прикладного процесса вывести размер рабочего множества страниц. Идентификатор процесса вводить с клавиатуры.
5	1. Вывести список имен выполняющихся процессов, у которых есть потомки. 2. Для выбранного прикладного процесса вывести время его старта в формате мин:сек. Идентификатор процесса вводить с клавиатуры.
6	1. Вывести информацию о загрузке в процентах каждого ядра процессора. 2. Для выбранного прикладного процесса вывести время его пребывания в системе с момента запуска в формате мин:сек. Идентификатор процесса вводить с клавиатуры.

№ бригады	Вариант задания
7	1. Вывести список выполняющихся процессов с указанием PID, имени, объема используемой оперативной памяти. 2. Для выбранного прикладного процесса вывести текущий и максимальный размер выгружаемого пула страниц. Идентификатор процесса вводить с клавиатуры.
8	1. Вывести информацию о загрузке процессора каждым процессом. 2. Для выбранного прикладного процесса вывести текущий и максимальный размер невыгружаемого пула страниц. Идентификатор процесса вводить с клавиатуры.
9	1. Для выбранного прикладного процесса вывести используемый размер рабочего набора и его максимальное значение использование файла подкачки в килобайтах. Идентификатор процесса вводить с клавиатуры. 2. Вывести список имен прикладных процессов, использовавших более T мс времени процессора. Значение T задавать с клавиатуры при запуске программы.
10	1. Вывести информацию о загрузке в процентах процессора. 2. Вывести список имен прикладных процессов, использовавших более M байт оперативной памяти. Значение M задавать с клавиатуры при запуске программы.

Сравнить результаты с данными Диспетчера задач.

3. СОДЕРЖАНИЕ ОТЧЕТА

Отчет о работе должен включать:

1. Перечень в виде таблицы использованных при выполнении работы функций Win32 API и их назначение.
2. Описание алгоритма и тексты программной реализации общих и индивидуальных заданий.
3. Результаты, полученные при выполнении заданий.
4. Комментарии и выводы по каждому выполненному заданию.

Контрольные вопросы

1. Атрибуты (описатели, характеристики) процесса.
2. Какие функции Win32 API необходимо использовать, чтобы сформировать список потоков процесса и их текущие приоритеты?
3. Функции Win32 API для получения сведений о выполняющихся в системе процессах и используемых ими ресурсах.
4. Назначение и использование функции OpenProcess.
5. Алгоритм работы приложения для получения списка имен выполняющихся процессов.
6. Функции Win32 API для получения информации о времени выполнения процессов и потоков.
7. Функции Win32 API и порядок их вызова для получения списка модулей заданного процесса.
8. Что называется рабочим набором процесса?
9. Функции Win32 API для получения информации об используемой процессом памяти.
10. Функции Win32 API для преобразования времени из формата FILETIME в общепринятый формат и методика их использования.

ЛАБОРАТОРНАЯ РАБОТА № 3

ПРОЦЕССЫ И ПОТОКИ. ИССЛЕДОВАНИЕ ДИСПЕТЧЕРИЗАЦИИ ПОТОКОВ

Цель работы – знакомство со средствами создания и управления процессами и потоками, исследование алгоритма диспетчеризации потоков операционной системы Windows.

1. КРАТКИЕ ТЕОРЕТИЧЕСКИЕ СВЕДЕНИЯ

1.1 Функции Win32 для создания и управления процессами

В ОС Windows запуск приложения приводит к созданию нового процесса. Для создания нового процесса используется функция Win32 API CreateProcess(lpApplicationName, lpCommandLine,
lpProcessAttributes, lpThreadAttributes,
bInheritHandles, dwCreationFlags, lpEnvironment,
lpCurrentDirectory, lpStartupInfo,
lpProcessInformation) типа BOOL.

lpApplicationName (тип LPCTSTR) – указатель на строку с именем exe-модуля. Параметр может быть задан значением NULL. Тогда имя модуля должно быть в lpCommandLine самым первым элементом.

lpCommandLine (тип LPCTSTR) – указатель на командную строку. Через нее передаются параметры. Здесь можно указать и путь, и имя модуля.

Параметр lpProcessAttributes (тип LPSECURITY_ATTRIBUTES) определяет атрибуты защиты для нового приложения. Если указать NULL, то система сделает это по умолчанию.

Параметр lpThreadAttributes типа LPSECURITY_ATTRIBUTES. Здесь определяются атрибуты защиты для первого потока, созданного приложением. Значение NULL приводит к установке по умолчанию.

bInheritHandles (тип BOOL) – флаг возможности наследования дескриптора процесса, производящего запуск. Если bInheritHandles = TRUE, каждый создаваемый процесс наследует дескриптор родительского процесса.

dwCreationFlags (тип DWORD) – флаг способа создания процесса и его класса приоритета.

lpEnvironment (тип LPVOID) указывает на блок среды. Если NULL, то будет использован блок среды родительского процесса. Блок среды – это список переменных <имя=значение> в виде строк с нулевым окончанием.

lpCurrentDirectory (тип LPCTSTR) – указатель на строку, определяющую текущий диск и каталог для дочернего процесса. Если NULL, то будет использован диск и каталог процесса-родителя.

Параметр lpStartupInfo (тип LPSTARTUPINFO) используется для настройки свойств процесса, например, заголовка и расположения окон.

lpProcessInformation (тип LPPROCESS_INFORMATION) – указатель структуры с информацией о процессе. Заполняется операционной системой.

Для завершения процесса используются функции ExitProcess и TerminateProcess.

Функция ExitProcess(uExitCode) обеспечивает нормальное завершение процесса.

Функция TerminateProcess(hProcess, uExitCode) используется для безусловного завершения процесса. Первый аргумент функции – дескриптор процесса типа HANDLE, который возвращается функцией, создавшей процесс, а второй аргумент – код возврата типа UINT.

Значение дескриптора необходимо получить с помощью функции

```
OpenProcess (unsigned long dwDesiredAcccess, // флаг доступа
int bInheritHandle,                  // флаг наследования дескриптора
unsigned long dwProcessId)    // идентификатор процесса
```

Некоторые значения первого параметра функции OpenProcess приведены в таблице 15 [1].

Таблица 15

Некоторые значения первого параметра функции OpenProcess

Вид доступа	Описание
PROCESS_ALL_ACCESS	Задает все возможные флаги доступа к объекту «процесс»
PROCESS_SET_INFORMATION	Разрешает использование дескриптора процесса в функции SetPriorityClass для задания класса приоритета процесса
PROCESS_TERMINATE	Разрешает использование дескриптора процесса в функции TerminateProcess для завершения процесса.

В некоторых случаях функция OpenProcess не сможет вернуть дескриптор, например, если в ОС предусмотрена защита от остановки системного процесса (службы) пользовательским приложением.

Для управления выполнением процесса используются две функции:

– HANDLE GetCurrentProcess (void) для получения дескриптора текущего процесса;

– BOOL SetPriorityClass (HANDLE hProcess, DWORD dwPriorityClass) для изменения класса приоритета процесса.

Параметр dwPriorityClass может принимать значения только из множества перечисленных в таблице 16 [4, 6].

Таблица 16

Классы приоритетов

Класс приоритета	Применение
HIGH_PRIORITY_CLASS	Для процесса, выполняющего критичные по времени задачи
IDLE_PRIORITY_CLASS	Для процесса, потоки которого должны выполняться, когда нет более приоритетных задач – например, хранитель экрана

Класс приоритета	Применение
NORMAL_PRIORITY_CLASS	Для большинства процессов (значение по умолчанию)
REALTIME_PRIORITY_CLASS	Наибольшее возможное значение класса приоритета

1.2 Функции Win32 для создания и управления потоками

В ОС Windows при создании процесса он автоматически создается единственный программный *поток* (Thread), который во время исполнения может создавать другие потоки.

Для создания потока служит функция Win32 API CreateThread(lpThreadAttributes, dwStackSize, StartFunction, lpParameter, dwCreationFlags, lpThreadId) [6].

lpThreadAttributes (тип LPSECURITY_ATTRIBUTES) – указатель на атрибуты безопасности. Типичное значение – NULL.

dwStackSize (тип SIZE_T) – начальный размер стека. Если этот параметр равен 0, то используется размер по умолчанию.

lpStartAddress (тип LPTHREAD_START_ROUTINE) – указатель функции, которая определена в приложении, выполняемом потоком. В данном случае указатель определяет начальный адрес функции.

lpParameter (тип LPVOID) – указатель параметра функции потока.

CreationFlags (тип DWORD) – флаг управления созданием потока. Если задать $\text{CreationFlags} = 0$, то поток начнет выполняться сразу после создания.

lpThreadId (тип LPDWORD) – указатель возвращаемого идентификатора потока. Если задано значение параметра NULL, то идентификатор потока отсутствует.

Функция возвращает дескриптор созданного потока (тип HANDLE).

Пример простейшего приложения, выполняющего три потока, приведен ниже.

```
#include "stdafx.h"
#include "Windows.h"

int cnt1, cnt2;

DWORD WINAPI Thread1(LPVOID lpParam)
{	cnt1++;
	return 0;
}

DWORD WINAPI Thread2(LPVOID lpParam)
{	cnt2++;
	return 0;
}

int _tmain(int argc, _TCHAR* argv[])
{
	HANDLE h1, h2;
	cnt1 = 0; cnt2 = 0;
	h1 = CreateThread(NULL, 0, Thread1, &cnt1, 0, NULL);
	h2 = CreateThread(NULL, 0, Thread2, &cnt2, 0, NULL);
	printf_s("cnt1 = %5i, cnt2 = %5i\n", cnt1, cnt2);
	system("pause");
	return 0;
}
```

Для завершения потока служат функции ExitThread и TerminateThread.

Функция VOID WINAPI ExitThread(DWORD dwExitCode) используется для обычного (штатного) завершения потока.

Функция BOOL WINAPI TerminateThread(HANDLE hThread, DWORD dwExitCode) завершает поток аварийно, без освобождения принадлежащих ему ресурсов.

Для управления приоритетом потока используются функции GetCurrentThread и SetThreadPriority.

Функция GetCurrentThread(void) возвращает дескриптора текущего потока.

Функция SetThreadPriority(hThread, nPriority) типа BOOL WINAPI устанавливает относительный приоритет потока в пределах процесса. hThread (тип HANDLE) – дескриптор потока. nPriority –

относительный приоритет потока, задается константой из следующего списка:

– THREAD_PRIORITY_IDLE;
– THREAD_PRIORITY_LOWEST;
– THREAD_PRIORITY_BELOW_NORMAL;
– THREAD_PRIORITY_NORMAL;
– THREAD_PRIORITY_ABOVE_NORMAL;
– THREAD_PRIORITY_HIGHEST;
– THREAD_PRIORITY_TIME_CRITICAL.

Приоритеты в этом списке следуют в порядке возрастания.

2. МЕТОДИКА ВЫПОЛНЕНИЯ

1. Для исследования алгоритма диспетчеризации потоков разработать тестовое приложение, содержащее три потока – главный и два дополнительных, приоритеты которых нужно будет изменять в ходе выполнения работы (при запуске каждого потока). Дополнительные потоки должны выполнять вычисление какого-либо математического выражения в бесконечном цикле (для создания нагрузки на процессор), а главный – отображать количество выполненных итераций циклов за фиксированное время, например, за последнюю секунду. Для задания заметной нагрузки процессору рекомендуется в вычисляемое выражение включать тригонометрические функции, причем их аргумент должен изменяться в цикле. Приложение должно выводить эти значения раз в секунду и сбрасывать счетчики итераций.

Если запущенное тестовое приложение будет единственным выполняемым на компьютере приложением, а загрузкой процессора другими неактивными приложениями и системными потоками можно пренебречь, то количество выполненных потоками приложения итераций позволит оценить время, затраченное процессором на выполнение каждого дополнительного потока, и сделать выводы о

влиянии приоритета потока на количество получаемых этим потоком квантов процессорного времени.

2. Исследовать зависимость количества итераций от приоритета одного потока при трех значениях приоритета – ниже нормального, нормальный, выше нормального. При выполнении задания приоритет второго потока установить нормальный.

Повторить эксперимент, установив для приложения с помощью Диспетчера задач ограничение на использование только одного ядра процессора.

Построить семейство графиков зависимости количества выполняемых итераций первого потока от его приоритета при разных условиях: при доступности всех ядер процессора; при доступности только одного ядра.

3. Запустить две копии тестового приложения. Исследовать зависимость производительности (числа итераций) потоков приложения от приоритета одного потока при разной активности окна приложения. Приоритет второго потока должен сохраняться неизменным. Изменять активное окно вручную. Ограничивать количество ядер не обязательно.

4. Для исследования влияния загрузки процессора на зависимость количества итераций одного потока от его приоритета добавить задержку на 5 мс в цикл каждого потока. Приоритет второго потока сохранять неизменным. Выполнить эксперимент с ограничением на использование только одного ядра процессора и без ограничения. Построить семейство графиков и сравнить влияние приоритета первого потока на количество выполняемых итераций в зависимости от доступности ядер процессора.

5. Повторить предыдущий эксперимент, уменьшив задержку до 1 мс. Построить семейство графиков вида: число итераций в зависимости приоритета первого потока при разном числе ядер.

6. С помощью Диспетчера задач получить количество потоков тестового приложения и оценить загрузку процессора.

3. СОДЕРЖАНИЕ ОТЧЕТА

Отчет о работе должен включать:

1. Исходный код тестового приложения, разработанного в п.1.
2. Таблицы и построенные по ним семейства графиков для заданий п. 2–5, выводы по каждому эксперименту.
3. Скриншот Диспетчера задач с загрузкой процессора и количеством потоков приложения (п. 6).
4. Выводы: обобщение результатов экспериментов; формулировка критерия влияния приоритета потоков на их производительность.

Контрольные вопросы

1. Функции Win32, используемые для управления выполнением процессов.
2. Функции Win32, используемые для создания процессов.
3. Функции Win32, используемые для завершения процессов.
4. Функции Win32, используемые для управления приоритетом потока.
5. Функции Win32, используемые для создания и завершения потоков.
6. Какие классы приоритета процессов Вы знаете?
7. Составляющие приоритета потока.
8. Характер влияния приоритета потока на количество выполняемых вычислений (производительность).
9. Влияние загрузки ЦП на характер зависимости производительности потока от его приоритета.
10. Голодание потока с низким приоритетом.
11. Характер влияния активности окна приложения на производительность потоков приложения.

ЛАБОРАТОРНАЯ РАБОТА № 4

СРЕДСТВА СИНХРОНИЗАЦИИ ПОТОКОВ

Цель работы – практическое знакомство с методами синхронизации двух потоков одного процесса с помощью критических секций и объектов ядра ОС MS Windows – мьютексов, семафоров и событий, а также с тупиками и их распознаванием средствами ОС Windows.

1. КРАТКИЕ ТЕОРЕТИЧЕСКИЕ СВЕДЕНИЯ

В многопоточной многозадачной ОС при одновременной модификации неразделяемого ресурса (например, глобальной переменной) двумя и более потоками возможна потеря выполненных изменений. Для правильной работы многопоточных приложений необходимо обеспечить поочередный доступ потоков к участкам кода, выполняющим изменение и запись значений переменной в память (критическим участкам).

Участок программного кода, в котором поток обращается к общему неразделяемому ресурсу с целью его модификации, называется *критическим участком.*

Правильная работа программы, содержащей критические участки, возможна только при поочередном выполнении потоками программы своих критических участков.

Для решения этой задачи могут использоваться как простые средства – критические секции, обеспечивающие поочередный доступ к критическим участкам потоков одного процесса, так и более сложные средства – такие, как объекты ядра ОС мьютексы (Mutex – сокр. от mutually exclusive – взаимно исключающий). Мьютексы могут решать такую же задачу для потоков, созданных как одним, так и различными процессами. Поочередный доступ потоков к своим критическим участкам можно также обеспечить с помощью универсальных средств синхронизации – семафоров и событий [2, 3].

1.1 Критические секции

Критические секции – средство синхронизации потоков одного приложения. Критические секции обеспечивают поочередный доступ потоков к критическим участкам программы. Для использования критической секции ее необходимо создать, то есть объявить глобальную переменную типа CRITICAL_SECTION и инициализировать ее с помощью вызова функции InitializeCriticalSection. Например, так:

```
CRITICAL_SECTION CrSec;
InitializeCriticalSection(&CrSec);
```

Для обеспечения поочередного доступа потоков к критическому участку в начале критического участка вызывается функция EnterCriticalSection, а в конце – функция LeaveCriticalSection. Параметром в обоих случаях должен быть указатель ранее созданной переменной типа CRITICAL_SECTION.

После окончания использования критической секции она должна быть уничтожена вызовом функции DeleteCriticalSection с таким же параметром.

1.2 Мьютексы

Мьютекс – объект ядра, используемый как средство синхронизации доступа потоков одного или нескольких приложений к критическому участку. Мьютекс создается с помощью функции [3]

```
HANDLE CreateMutex(
  LPSECURITY_ATTRIBUTES lpMutexAttributes, //обычно это NULL
  BOOL bInitialOwner, // обычно FALSE
  LPCTSTR lpName).   // ссылка на имя мьютекса
```

Функция возвращает дескриптор мьютекса.

Перед входом в критический участок поток должен проверить возможность входа с помощью вызова функции

```
DWORD WaitForSingleObject(
  HANDLE hMutex,
```

DWORD dwMilliseconds).

В конце критического участка с помощью вызова функции

BOOL ReleaseMutex(HANDLE hMutex)

поток должен сообщить операционной системе о завершении его выполнения.

Объект ядра Mutex удаляется функцией CloseHandle(hObject) типа BOOL, где параметр hObject – дескриптор мьютекса.

1.3 Семафоры

Семафор – объект ядра, используемый как универсальное средство управления ресурсами. Как правило, семафор используется для учета неразделяемых ресурсов.

Семафор создается с помощью функции [5, 6]

```
HANDLE CreateSemaphore(
LPSECURITY_ATTRIBUTES lpSemaphoreAttributes,
    // атрибут защиты, обычно равен NULL
LONG lInitialCount,   // начальное значение счетчика свободных
                      // ресурсов
LONG lMaximumCount, //макс. количество свободных ресурсов
LPCTSTR lpName    // ссылка на имя семафора).
```

Функция возвращает дескриптор семафора.

В данной работе семафор будет использоваться для синхронизации двух конкурирующих потоков одного приложения. Поэтому создаваемый семафор может быть безымянным (значение lpName NULL), а доступ потоков к нему будет осуществляться с помощью дескриптора.

В начале критического участка вызывается функция

```
DWORD WaitForSingleObject(
  HANDLE hSemaphore,
  DWORD dwMilliseconds),
```

а в конце – функция

BOOL ReleaseSemaphore(
HANDLE hSemaphore,
LONG lReleaseCount,
LPLONG lpPreviousCount).

Удаление объекта Semaphore выполняет функция CloseHandle(hObject) типа BOOL, где hObject – дескриптор семафора.

1.4 События

Объект ядра событие создается с помощью функции

```
HANDLE CreateEvent(
  LPSECURITY_ATTRIBUTES lpEventAttributes, // обычно NULL
  BOOL bManualReset, // тип сброса события
  BOOL InitialState,      // начальное состояние, True – сигнальное
  LPTSTR lpName)        // имя события, обычно NULL.
```

Параметр bManualReset определяет тип сброса события: TRUE – ручной сброс, FALSE – автоматический.

Если событие создано со значением параметра bManualReset = FALSE, то оно работает в режиме автосброса. Тогда в начале критического участка вызывается функция

```
DWORD WINAPI WaitForSingleObject(
HANDLE hHandle,
 DWORD dwMilliseconds),
```

а в конце – функция BOOL SetEvent (HANDLE hEvent), которая переводит событие в сигнальное состояние (событие – завершение критического участка).

Удаление объекта ядра Event выполняет функция

```
BOOL CloseHandle(HANDLE hObject).
```

2. МЕТОДИКА ВЫПОЛНЕНИЯ

1. Ознакомиться с приведенным ниже текстом программы консольного приложения, в котором создаются 2 потока одного процесса, выполняющие увеличение глобальной переменной. Для синхронизации доступа конкурирующих потоков к переменной используется неименованный объект ядра – mutex.

```
#include "stdafx.h"
#include <windows.h>
#include <iostream>
using namespace std;

int global = 100; // глобальная переменная, значение которой увеличивают потоки
DWORD Tid1, Tid2;
HANDLE hT1, hT2;
HANDLE hMx;
// функция первого потока
DWORD WINAPI ThreadProc1( LPVOID lpParameter )
{
	int i, j;
	SetThreadPriority (ht1,THREAD_PRIORITY_TIME_CRITICAL);
	for (j=1; j <= 12; j++)
	{
		WaitForSingleObject(hMx, INFINITE);
		// начало критического участка
		i = global;
		i++;
		Sleep (1);
		global = i;
		// конец критического участка
		printf_s( "%4s %4d \n", " 1 th", i );
		ReleaseMutex(hMx);
	}
return 0;
}
// функция второго потока
DWORD WINAPI ThreadProc2 (LPVOID lpParameter)
{
	int i, j;
	for (j=1; j <= 12; j++)
	{
		WaitForSingleObject hMx, INFINITE);
		// начало критического участка
		i = global;
		i++;
```

```
                Sleep (1);
                global = i;
                // конец критического участка
                printf_s( "%4s %4d %4d \n", " 2 th", i, j );
                ReleaseMutex(hMx);
        }
return 0;
}
// основной поток
int _tmain(int argc, _TCHAR* argv[])
{
        HANDLE msh[2];
        hMx = CreateMutex(NULL, FALSE, NULL);
        hT1 = CreateThread(NULL, 0, &ThreadProc1, NULL, 0, &Tid1);
        hT2 = CreateThread(NULL, 0, &ThreadProc2, NULL, 0, &Tid2);
        msh[0] = hT1;
        msh[1] = hT2;
        // ожидание завершения работы потоков
        if (WaitForMultipleObjects(2,msh,TRUE,1000)==WAIT_TIMEOUT)
                printf_s("Over time");
        return 0;
}
```

В рассмотренном приложении два потока ThreadProc1 и ThreadProc2 при каждом выполнении цикла увеличивают значение глобальной переменной global от начального значения 100 на 1. Число повторений цикла в каждом потоке равно 12.

При правильной работе приложения конечным значением переменной global должно стать число 100+12+12=124. Однако без использования средств синхронизации такого значения получить в общем случае не удается, так как потоки могут перезаписывать значения переменной global. Поэтому в приложении для синхронизации работы потоков используется мьютекс. Критические участки потоков ThreadProc1 и ThreadProc2 выделены вызовом функций WaitForSingleObject и ReleaseMutex.

Результаты увеличения переменной global в потоках выводятся на экран.

2. Выполнить приложение и убедиться в том, что после его выполнения значение переменной global будет равно 124.

3. Отключить синхронизацию, закомментировав вызов функций WaitForSingleObject и ReleaseMutex. Выполнить приложение и убедиться в том, что после его выполнения значение переменной global будет менее 124. Объяснить причину изменений.

4. Закомментировать вызовы функций Sleep(1) в функциях потоков. Выполнить приложение и убедиться в том, что после его выполнения значение переменной global изменилось. Объяснить причину изменения.

5. Завершить работу первого потока после выполнения 5 циклов. Для этого вставить в цикл функции потока ThreadProc1 строку

```
if (j==5) TerminateThread(GetCurrentThread(),0);
```

Выполнить приложение и убедиться в том, что первый поток выполнился 5 раз, а второй – 12 раз. Объяснить причины такого поведения потоков.

6. Выполнить индивидуальные задания для бригад согласно таблице 17.

Таблица 17

Индивидуальные задания для бригад

№№ бригад	Средства синхронизации потоков
1, 2	Семафор
3, 4	Критическая секция
5, 6	Событие с автоматическим сбросом
7, 8, 9, 10	Событие с ручным сбросом

Изменить текст исходного приложения, удалив из него обращения к средству синхронизации Mutex и добавив указанные в таблице 17 средства синхронизации (согласно вариантам). Затем повторно выполнить пункты 2–5.

7. Используя средство синхронизации, выбранное из таблицы 17 по номеру бригады, создать тупик с участием двух потоков, использующих по два неразделяемых ресурса каждый. Первый поток должен занять первый ресурс, затем второй ресурс. Далее этот поток должен освободить ресурсы в любом порядке.

Второй поток должен занять сначала второй ресурс, затем первый. После этого он должен освободить ресурсы.

Признаком возникновения тупика служит вывод сообщения "Over time" при превышении временем ожидания завершения потоков ThreadProc1 и ThreadProc2 заданного значения 1000 мс.

3. СОДЕРЖАНИЕ ОТЧЕТА

Отчет о работе должен включать:

1. Тексты разработанных приложений.
2. Задания и результаты, полученные при их выполнении.
3. Письменный ответ на вопрос: «В чем состоит отличие поведения разработанных приложений, использующих для доступа потоков к критическим участкам мьютексы, критические секции, семафоры и события при досрочном завершении одного из потоков?».
4. Выводы

Контрольные вопросы

1. Функции Win32, использованные при выполнении работы.
2. В чем состоит отличие критического участка от критической секции?
3. В каких случаях требуется синхронизация потоков, которым необходим доступ к одному и тому же ресурсу?
4. Каким образом организуется синхронизация потоков с помощью событий?
5. Синхронизация потоков с помощью критических секций.
6. Синхронизация потоков с помощью семафоров.
7. Какова роль вызова функции ReleaseMutex в синхронизации потоков с помощью мьютексов?
8. В чем состоит отличие поведения разработанных приложений, использующих для доступа потоков к критическим участкам мьютексы, критические секции, семафоры и события?
9. Понятие тупика двух потоков.

ЛАБОРАТОРНАЯ РАБОТА № 5

ВЫДЕЛЕНИЕ ВИРТУАЛЬНОЙ ПАМЯТИ

Цель работы – знакомство с функциями Win32 и структурами данных, используемыми для управления виртуальной памятью.

1. КРАТКИЕ ТЕОРЕТИЧЕСКИЕ СВЕДЕНИЯ

1.1 Механизмы выделения виртуальной памяти

В Win32 используются четыре механизма выделения виртуальной памяти [3, 6]:

- выделение регионов виртуальной памяти для работы с крупными массивами объектов и структур (по размеру превышающих одну страницу памяти);
- кучи (heaps) – используются при работе со значительным количеством элементов данных небольшого размера (меньше одной страницы памяти);
- файлы, проецируемые в память – для работы с интенсивными потоками данных, а также для обеспечения совместного доступа приложений;
- Address Windowing Extensions (AWE) – отображение виртуального адресного пространства, размер которой превышает 4 Гбайт, на окно размером 4 Гбайт физической памяти.

Далее рассмотрены первые три механизма.

1.2 Функции для анализа виртуальной памяти

Получить информацию о текущем состоянии физической и виртуальной памяти, используемой в 32-х и 64-хразрядных ОС Windows, можно с помощью функции
BOOL GlobalMemoryStatusEx(LPMEMORYSTATUSEX lpBuffer).

Параметр lpBuffer – это указатель на структуру типа MEMORYSTATUSEX, которая определена так

```
typedef struct _MEMORYSTATUSEX {
    DWORD     dwLength;
    DWORD     dwMemoryLoad;
    DWORDLONG ullTotalPhys;
    DWORDLONG ullAvailPhys;
    DWORDLONG ullTotalPageFile;
    DWORDLONG ullAvailPageFile;
    DWORDLONG ullTotalVirtual;
    DWORDLONG ullAvailVirtual;
    DWORDLONG ullAvailExtendedVirtual;
} MEMORYSTATUSEX, *LPMEMORYSTATUSEX;
```

Назначение полей этой структуры дано в таблице 18 [6].

Таблица 18

Назначение полей структуры MEMORYSTATUSEX

Наименование поля	Содержимое поля
dwLength	Размер структуры типа MEMORYSTATUSEX в байтах
dwMemoryLoad	Процент используемой физической памяти
ullTotalPhys	Общий размер физической памяти в байтах
ullAvailPhys	Общий размер физической памяти в байтах, доступной для выделения
ullTotalPageFile	Максимальное количество байтов, которое может содержаться в страничном файле на жестком диске (или дисках)
ullAvailPageFile	Максимальное количество байтов, которое может быть передано процессу из страничного файла
ullTotalVirtual	Размер виртуального адресного пространства вызывающего процесса в байтах
ullAvailVirtual	Суммарный объем незарезервированной и свободной памяти в пользовательской части адресного пространства процесса, вызывающего функцию GlobalMemoryStatus, в байтах

Элемент ullAvailExtendedVirtual не используется, его значение равно 0.

Перед вызовом функции GlobalMemoryStatusEx в поле dwLength необходимо занести размер структуры MEMORYSTATUSEX в байтах, вычисленный с помощью функции sizeof.

Пример. Вызов функции GlobalMemoryStatus для получения объема оперативной памяти:

```
MEMORYSTATUSEX MemInfo;
MemInfo.dwLength = sizeof (MemInfo);
GlobalMemoryStatusEx (&MemInfo);
printf_s (" Размер ОП %d Кбайт ", MemInfo. ullTotalPhys / 1024);
```

Информацию о состоянии области виртуальной памяти текущего процесса (размер, тип памяти, атрибуты защиты) можно получить с помощью функции VirtualQuery, которая имеет следующий прототип:

```
DWORD VirtualQuery (
LPCVOID lpAddress,                          // адрес области
PMEMORY_BASIC_INFORMATION lpBuffer, // указатель буфера
DWORD dwLength                              // размер буфера
).
```

Параметр lpAddress должен указывать на ту область виртуальной памяти, информацию о которой нужно получить. При этом размер области определяется количеством последовательных виртуальных страниц, имеющих одинаковые атрибуты (состояние и тип защиты). Параметр lpBuffer – это указатель на структуру типа MEMORY_BASIC_INFORMATION, в которую функция VirtualQuery помещает информацию об области виртуальной памяти, указанной параметром lpAddress. Параметр dwLength должен содержать размер структуры типа MEMORY_BASIC_INFORMATION.

Параметр lpBuffer является указателем на структуру типа MEMORY_BASIC_INFORMATION, назначение полей которой приведено в таблице 19 [6].

Таблица 19

Назначение полей структуры MEMORY_BASIC_INFORMATION

Тип и наименование поля	Содержимое поля
PVOID BaseAddress	Указатель базового адреса региона страниц области виртуальной памяти (значение параметра lpAddress, округленное до значения, кратного размеру страницы).
PVOID AllocationBase	Указатель на базовый адрес диапазона страниц, выделенных функцией VirtualAlloc. Страница, на которую указывает элемент BaseAddress, содержится в пределах этого диапазона.
DWORD AllocationProtect	Начальное значение атрибута защиты региона, присвоенное ему при резервировании. Значения в виде констант защиты памяти: PAGE_EXECUTE; PAGE_EXECUTE_READ; PAGE_EXECUTE_READWRITE; PAGE_READONLY и других.
DWORD RegionSize	Суммарный размер (в байтах) группы страниц, начинающихся с базового адреса и имеющих те же атрибуты защиты, состояние и тип, что и страница, обнаруженная по адресу lpAddress.
DWORD State	Указывает состояние всех смежных страниц, имеющих те же атрибуты защиты, состояние и тип, что и страница, расположенная по адресу lpAddress. При MEM_FREE элементы AllocationBase, AllocationProtect, Protect и Type содержат неопределенные значения, а при MEM_RESERVE неопределенное значение содержит элемент Protect.
DWORD Protect	Атрибут защиты доступа к страницам региона. Принимает значение из списка значений AllocationProtect.
DWORD Type	Тип страниц региона виртуальной памяти.

Поле State региона может получить значение MEM_COMMIT, MEM_FREE или MEM_RESERVE.

MEM_COMMIT указывает созданные страницы, для физического хранения которых была выделена область в памяти или в файле подкачки на диске.

MEM_FREE указывает свободные страницы, не доступные для вызывающего процесса и которые могут быть ему выделены.

MEM_RESERVE указывает зарезервированные страницы, для которых выделяется диапазон адресов ВАП процесса без какого-либо физического выделения памяти.

Значением поля Type могут быть константы:

MEM_IMAGE – исполняемый код;

MEM_MAPPED – файл, проецируемый в память;

MEM_PRIVATE – память, принадлежащая процессу.

При вызове функция VirtualQuery заполняет структуру MEMORY_BASIC_INFORMATION и возвращает количество байтов, записанных в нее. Если возвращается нулевое значение, значит, информация о запрошенном участке памяти не получена.

Сканируя виртуальное адресное пространство в диапазоне от минимального до максимального адреса, можно построить карту виртуальной памяти процесса (таблицу с базовыми адресами регионов, их размерами и другими характеристиками).

Пример. Использование функции VirtualQuery для просмотра виртуального адресного пространства текущего процесса и вывода адресов и размеров зарезервированных регионов.

```
MEMORY_BASIC_INFORMATION mbi;
DWORD pb ;     // базовый адрес
pb = 4096;
        while (VirtualQuery (LPCVOID(pb), &mbi,
                sizeof (MEMORY_BASIC_INFORMATION)) == sizeof (mbi))
{
        if (mbi.State == MEM_RESERVE)
        printf_s("%08x %8d\n", mbi.BaseAddress, mbi.RegionSize);
        pb = pb + mbi.RegionSize;
}
```

1.3 Выделение виртуальной памяти

Регионом в адресном пространстве процесса называется совокупность смежных страниц виртуальной памяти, имеющих одинаковый статус, тип и атрибут защиты.

Для резервирования региона в виртуальном адресном пространстве процесса (первый механизм управления памятью) используется функция

```
LPVOID VirtualAlloc (
  LPVOID lpAddress,  // начальный адрес резервируемого
                     // региона виртуальной памяти,
  SIZE_T dwSize,     // размер региона
  DWORD flAllocationType,    // вид операции
  DWORD flProtect    // тип защиты доступа
).
```

Первый параметр задает начальный адрес региона в адресном пространстве. При этом адрес округляется до ближайшей границы блока размером 64 Кбайт (гранулярность размещения). Если параметр равен NULL, этот адрес определяет система.

Если запрос выполнен успешно, функция возвращает базовый адрес зарезервированного региона.

Параметр dwSize задает размер резервируемого региона в байтах. Заданный размер увеличивается до величины, кратной размеру страницы.

Третий параметр определяет вид операции: MEM_RESERVE – зарезервировать регион; MEM_COMMIT – передать региону физическую память. Можно одновременно зарезервировать регион и передать ему физическую память. Например,

```
VirtualAlloc (NULL, 100*1024, MEM_RESERVE | MEM_COMMIT, PAGE_READWRITE);
```

Если параметру flAllocationType задается значение MEM_TOP_DOWN, то это указывает системе на необходимость резервирования региона в верхних адресах виртуального адресного

пространства с целью уменьшения фрагментации виртуальной памяти.

Последний параметр flProtect определяет атрибут защиты страниц региона. Задается константами:

- PAGE_READWRITE – чтение и запись;
- PAGE_READONLY – только чтение;
- PAGE_EXECUTE – только исполнение программного кода;
- PAGE_EXECUTE_READ – исполнение и чтение;
- PAGE_EXECUTE_READWRITE – исполнение, чтение и запись;
- PAGE_NOACCESS – запрещен любой вид доступа;
- PAGE_GUARD – сигнализация доступа к странице;
- PAGE_NOCACHE – отмена кэширования для страницы памяти.

Для возврата физической памяти, спроецированной на регион, или освобождения всего региона адресного пространства используется функция

```
BOOL VirtualFree (LPVOID lpAddress,    // адрес региона
DWORD dwSize,              // размер региона
DWORD dwFreeType       // тип операции освобождения
).
```

В случае использования этой функции для освобождения всей переданной региону физической памяти в параметр lpAddress необходимо передать базовый адрес региона, установить параметр dwSize = 0, так как системе известен размер региона. В третьем параметре следует передать MEM_RELEASE – это приведет к возврату системе всей физической памяти, спроецированной на регион, и освобождению самого региона.

1.4 Кучи

Куча (Heap) – это регион зарезервированного адресного пространства [3, 6]. Первоначально большей его части физическая память не передается. После того, как программа занимает эту

область под данные, диспетчер, управляющий кучами, передает ей страницы физической памяти. При освобождении страниц в куче диспетчер возвращает физическую память системе.

Каждый процесс имеет одну или несколько куч. При создании процессу всегда выделяется одна куча, называемая кучей по умолчанию. Процесс может получить ее дескриптор, вызвав функцию HANDLE GetProcessHeap ().

Для создания каждой дополнительной кучи процесс должен вызвать функцию

```
HANDLE HeapCreate(DWORD flOptions,// флаги создания
    DWORD dwInitialSize,          // начальный размер кучи
    DWORD dwMaximumSize           // максимальный размер кучи
).
```

Параметр flOptions может принимать в качестве значения одну из двух констант:

- HEAP_GENERATE_EXCEPTIONS – указывает, что неудача рассматриваемой операции должна вызвать ошибку исполнения программы;
- HEAP_NO_SERIALIZE – задает отключить синхронизацию при работе с кучей.

Может принимать и значение 0. Тогда доступ к куче организуется последовательно.

Второй параметр указывает, сколько байтов будет передано куче изначально.

Третий параметр ограничивает максимальный размер кучи. Может быть равен нулю, тогда размер резервируемого региона увеличивается системой при необходимости и ограничен лишь объемом свободной памяти.

Возвращаемое функцией HeapCreate значение – дескриптор созданной кучи.

Выделение блока памяти из кучи осуществляется функцией

```
LPVOID HeapAlloc(
```

```
    HANDLE hHeap,      // дескриптор кучи
    DWORD dwFlags,     // флаги выделения памяти
    DWORD dwBytes      // количество размещаемых байтов
).
```

Первый параметр – дескриптор кучи, возвращаемый функциями GetProcessHeap() или HeapCreate().

Второй параметр влияет на характер выделения памяти. Так, если dwFlags равен HEAP_ZERO_MEMORY = 8, то выделяемый блок заполняется нулями.

Третий параметр определяет для кучи объем выделяемой памяти в байтах.

Освобождение блока памяти осуществляется функцией

```
BOOL HeapFree(
    HANDLE hHeap,      // дескриптор кучи
    DWORD dwFlags,     // флаги освобождения памяти
    LPVOID lpMem       // указатель на освобождаемую память
).
```

Уничтожение кучи осуществляется вызовом функции

```
BOOL HeapDestroy(HANDLE hHeap).
```

1.5 Файлы, проецируемые в память

Проецирование в память файла – это отображение его в виртуальное адресное пространство процесса. Проецируемый в память файл позволяет резервировать регион адресного пространства и передавать ему физическую память. При этом физическая память не выделяется из системного страничного файла, а берется из файла, уже находящегося на диске. Если файл спроецирован в память, то становится возможным обращаться к нему так, будто он целиком в нее загружен [4, 6]. Этот механизм управления памятью применяется:

- для загрузки и исполнения EXE- и DLL-файлов;
- для доступа к файлу данных, размещенному на диске;

– для обеспечения совместного доступа разных процессов к общим данным.

Для подготовки к использованию файла, проецируемого в память, нужно выполнить три операции:

– создать с помощью вызова функции CreateFile объект ядра «файл», соответствующий дисковому файлу, который нужно спроецировать в память;
– создать с помощью вызова функции CreateFileMapping объект ядра «проецируемый файл», чтобы сообщить системе размер файла и способ доступа к нему;
– с помощью функции MapViewOfFile указать системе, как спроецировать в адресное пространство процесса созданный объект «проецируемый файл» – целиком или только его часть.

Функция создания объект ядра «файл» определена так

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

Параметр lpFileName – ссылка на строку с именем файла (при необходимости вместе с путем), который создается или открывается. dwDesiredAccess задает режим доступа к файлу: чтение, запись или и чтение и запись. dwShareMode задает режим совместного доступа к файлу. Если dwShareMode = 0, то совместный доступ невозможен.

Параметр lpSecurityAttributes – ссылка на структуру SECURITY_ATTRIBUTES, которая устанавливает, может ли дескриптор создаваемого объекта наследоваться дочерними процессами (если NULL, то – нет).

dwCreationDistribution – атрибут, который определяет действия с существующим или создаваемым файлом (CREATE_NEW – создает новый, OPEN_EXISTING – открывает существующий).

Параметр dwFlagsAndAttributes включает атрибуты и флаги файла. Дескриптор hTemplateFile определяет шаблон создаваемого файла (может быть NULL).

Функции CreateFileMapping создает или открывает именованный или безымянный объект ядра «проецируемый файл» (File Mapping) для файла, заданного своим дескриптором.

```
HANDLE CreateFileMapping(
  HANDLE hFile,                  // дескриптор файла
  LPSECURITY_ATTRIBUTES lpAttributes,
  DWORD flProtect,
  DWORD dwMaximumSizeHigh,
  DWORD dwMaximumSizeLow,
  LPCTSTR lpName                 // имя объекта
).
```

Ссылка lpAttributes аналогична ссылке lpSecurityAttributes предыдущей функции. flProtect устанавливает вид защиты страниц проецируемого в память файла. Например, PAGE_READWRITE – разрешает чтение и запись страниц.

Параметры dwMaximumSizeHigh и dwMaximumSizeLow типа DWORD должны содержать старшую и младшую часть максимального размера объекта «проецируемый файл». Если они равны NULL, в них автоматически помещается размер отображаемого файла.

lpName – указатель на имя проецируемого файла. Следует задавать для совместного доступа к объекту несколькими процессами. Иначе может быть NULL.

Функции MapViewOfFile проецирует «проецируемый файл» в адресное пространство вызывающего процесса.

```
LPVOID MapViewOfFile(
```

```
  HANDLE hFileMappingObject,
  DWORD dwDesiredAccess,     // режим доступа
  DWORD dwFileOffsetHigh,    // старшее DWORD смещения
  DWORD dwFileOffsetLow,     // младшее DWORD смещения
  SIZE_T dwNumberOfBytesToMap // число отображаемых байтов
);
```

hFileMappingObject – дескриптор объекта «проецируемый файл), созданного функцией CreateFileMapping. dwDesiredAccess – устанавливает режим доступа к отображаемому файлу. Значение FILE_MAP_WRITE устанавливает режим чтения и записи.

dwFileOffsetHigh и dwFileOffsetLow – старшее и младшее слово смещения внутри файла, откуда начинается отображение в адресное пространство процесса, с учетом гранулярности. dwNumberOfBytesToMap – количество отображаемых в память байт.

При успешном завершении функция MapViewOfFile возвращает начальный адрес выделенного региона памяти.

Для завершения работы с файлом, проецируемым в память, также следует выполнить три операции:

- освободить адресное пространство процесса от объекта ядра «проецируемый файл», используя функцию UnmapViewOfFile;
- закрыть с помощью вызова функции CloseHandle объект «проецируемый файл»;
- закрыть объект ядра «файл», вызвав функцию CloseHandle().

Функция BOOL UnmapViewOfFile(LPCVOID lpBaseAddress), используя начальный адрес региона в адресном пространстве процесса, отменяет его отображение.

BOOL CloseHandle(HANDLE hObject) закрывает объект по его дескриптору.

2. МЕТОДИКА ВЫПОЛНЕНИЯ

1. Разработать два консольных приложения с функционалом, указанным в таблице 20 для соответствующего варианта.

Таблица 20

Индивидуальные задания для бригад

Номер бригады	Содержание задания
1, 7	1. Вывести информацию о параметрах и состоянии виртуальной памяти, количестве и общем объеме свободных регионов выполняющегося процесса. Построить карту виртуальной памяти выполняющегося процесса в диапазоне 0-1 Гбайт (адрес, размер региона, статус). 2. Операции с памятью – резервирование региона в верхних адресах ВАП с передачей ему физической памяти через 10 секунд. Размер региона указывать в байтах посредством ввода с клавиатуры.
2, 8	1. Вывести информацию о параметрах и состоянии виртуальной памяти (размер страничного файла (файла подкачки) и суммарный объем всех зарезервированных регионов в адресном пространстве процесса), количестве зарезервированных регионов выполняющегося процесса. Построить карту виртуальной памяти выполняющегося процесса в диапазоне 1-2 Гбайт (адрес, размер региона, атрибут защиты). 2. Операции с памятью – резервирование региона в ВАП и передача ему физической памяти. Размер региона указывать в байтах посредством ввода с клавиатуры.
3, 9	1. Вывести информацию о параметрах и состоянии виртуальной памяти (гранулярность, диапазон адресов ВАП, размер виртуального адресного пространства вызывающего процесса), количестве и общем объеме зарезервированных регионов выполняющегося процесса. 2. Операции с кучами – создать новую кучу процесса и разместить в ней *N* блоков по *K* байт в каждом блоке. Значения *N* и *K* вводить с клавиатуры. *K* не должно быть больше 512 Кбайт.

Номер бригады	Содержание задания
4, 10	1. Вывести информацию о параметрах и состоянии виртуальной памяти (гранулярность, диапазон адресов ВАП, общий объем физической памяти), количестве и общем объеме зарезервированных регионов выполняющегося процесса. 2. Операции с памятью – резервирование региона в ВАП с передачей ему физической памяти через 10 секунд. Размер региона указывать в байтах посредством ввода с клавиатуры.
5, 11	1. Вывести информацию о параметрах и состоянии виртуальной памяти (гранулярность, диапазон адресов ВАП, процент использования физической памяти), количестве и общем объеме зарезервированных регионов выполняющегося процесса. 2. Операции с памятью – резервирование региона памяти в верхних адресах ВАП. Размер региона указывать в байтах посредством ввода с клавиатуры.
6, 12	1. Вывести информацию о параметрах и состоянии виртуальной памяти выполняющегося процесса (размер страничного файла (файла подкачки) и суммарный объем всех зарезервированных регионов в адресном пространстве процесса), количестве зарезервированных регионов. 2. Создать проецируемый файл в доступном для записи каталоге. Занести в его отображаемый регион строку «To be or not to be?». Закрыть файл. Найти его в выбранном каталоге файлов и проверить его содержимое с помощью Блокнота.

2. Проверить работу приложений. Обратить внимание на изменение состояния ВАП процесса после резервирования региона и выделения ему физической памяти.

3. Проанализировать изменение параметров приложения с помощью Диспетчера задач. Для анализа изменений при резервировании памяти следует использовать имеющийся в Диспетчере задач Монитор ресурсов (раздел Память).

3. СОДЕРЖАНИЕ ОТЧЕТА

Отчет о работе должен включать:

1. Задания и тексты приложений, разработанных в п. 1.
2. Результаты работы приложений и их сопоставление с данными Монитора ресурсов.
3. Выводы.

Контрольные вопросы

1. Функции, используемые для получения подробной информации о состоянии виртуальной памяти.
2. Функции управления виртуальной памятью.
3. Использование куч при разработке приложений.
4. Функции размещения информации в куче.
5. Необходимость создания дополнительных куч процесса.
6. Функции создания и удаления дополнительных куч процесса.
7. Функции, используемые для получения общей информации о состоянии виртуальной памяти.
8. Каков диапазон адресов виртуального адресного пространства, возвращаемый функцией VirtualQuery?
9. Что такое гранулярность выделения памяти?
10. Для каких целей при создании объекта ядра «проецируемый файл» ему назначается имя?

ЛАБОРАТОРНАЯ РАБОТА № 6

ДИНАМИЧЕСКИ ЗАГРУЖАЕМЫЕ БИБЛИОТЕКИ (DLL)

Цель работы – освоение методики создания и использования динамически загружаемых библиотек (DLL) в среде Microsoft Visual Studio.

1. КРАТКИЕ ТЕОРЕТИЧЕСКИЕ СВЕДЕНИЯ

1.1 DLL и их роль в Win32

Библиотеки DLL (Dynamic Link Library) составляют основу архитектуры операционной системы Microsoft Windows любой версии. К областям применения динамически компонуемых библиотек относятся CPL-файлы (компоненты панели управления), серверы OLE и т.д.

Когда в 32-разрядной версии Windows запускается программа, операционная система создает новый процесс с 2 Гбайт виртуальной памяти, в которых эта программа и будет выполняться. Затем ОС создает объект отображения исполняемого файла, который проецирует этот EXE-файл в адресное пространство его процесса. Следующим шагом ОС, получив из образа исполняемого файла список используемых им библиотек по известному смещению от начала, находит и проецирует в то же адресное пространство соответствующие файлы DLL. Для каждой библиотеки создается отдельный объект проецирования файла. Для этих объектов повторяется считывание списка используемых DLL. Если в нем есть библиотеки, отсутствующие в адресном пространстве рассматриваемого процесса, их поиск, проецирование и поиск обращений к дополнительным библиотекам повторяется и для них, рекурсивно [6].

Когда программа заканчивает работу, образы всех DLL удаляются из памяти. Образ исполняемого файла и объект главного процесса также удаляется, и процесс завершается.

Динамически компонуемые библиотеки могут использоваться совместно разными приложениями. В простейшем случае, DLL содержит в себе набор функций схожего назначения. Среди функций могут быть как служащие интерфейсом связи с объектами внутри DLL, так и осуществляющие требуемые действия сами по себе. Обращение к экспортируемым DLL-функциям возможно как из ОС, так и из прикладных программ или других DLL.

1.2 Создание проекта DLL

Для создания динамически компонуемой библиотеки в среде Visual Studio необходимо выбрать: Файл - Создать - Проект - Приложение Win32 (консольное либо проект Win32). После выбора, ввести имя и нажать ОК, Далее, и выбрать Тип приложения «библиотека DLL». После этого открывается файл, в который необходимо ввести текст DLL-модуля.

Чтобы в результате компиляции была создана DLL, а не EXE-файл, в настройках компиляции следует указать, что необходимо получить именно Dynamic Link Library. Для этого функцию надо объявить как экспортируемую из DLL. Делается это посредством добавления слева от функции модификатора extern "C" или extern "C++".

Пример кода файла Lb5Dll.cpp проекта DLL.

```
#include "stdafx.h"
#include <iostream>
#include <string>
using namespace std;

extern "C" __declspec(dllexport) string Prefix(char * s, char ch)
{
	string str = s;
	str = ch + str;
	return str;
}
```

В этом примере функция Prefix типа string добавляет символ в начало строки. После компиляции DLL будут созданы два файла: Lb5Dll.dll и Lb5Dll.lib.

Затем необходимо создать файл заголовка с описанием прототипов функций, находящихся в DLL-файле (Меню: Проект/Добавить новый элемент). В данном случае заголовочный файл Lb5Dll.h будет следующего содержания:

```
#include "stdafx.h"
#include <string>
using namespace std;

string Prefix(char * s, char ch);
```

Заголовочный файл нужно добавить в программу, в которой планируется использование созданной библиотеки DLL.

1.3 Вызов функций из DLL

Проект, в котором будут использованы функции разработанной библиотеки, создается обычным образом. В Visual Studio это: Файл - Создать - Проект - Консольное Приложение Win32 - Готово. В проект нужно включить файлы ***.dll, ***.lib и ***.h библиотеки.

В Visual Studio подключение lib-файла выполняется с помощью меню в следующем порядке: Проект/Свойства/Компоновщик/Ввод/ Дополнительные зависимости. Туда добавляется путь к lib-файлу. Проще дописать туда имя lib-файла (рисунок 4), тогда он должен располагаться там же, где и cpp-файл проекта. Туда же нужно поместить и заголовочный файл библиотеки. Dll-файл следует разместить в каталоге Debug вместе с exe-файлом приложения.

В программе в части определений нужно объявить подключение заголовочного файла библиотеки.

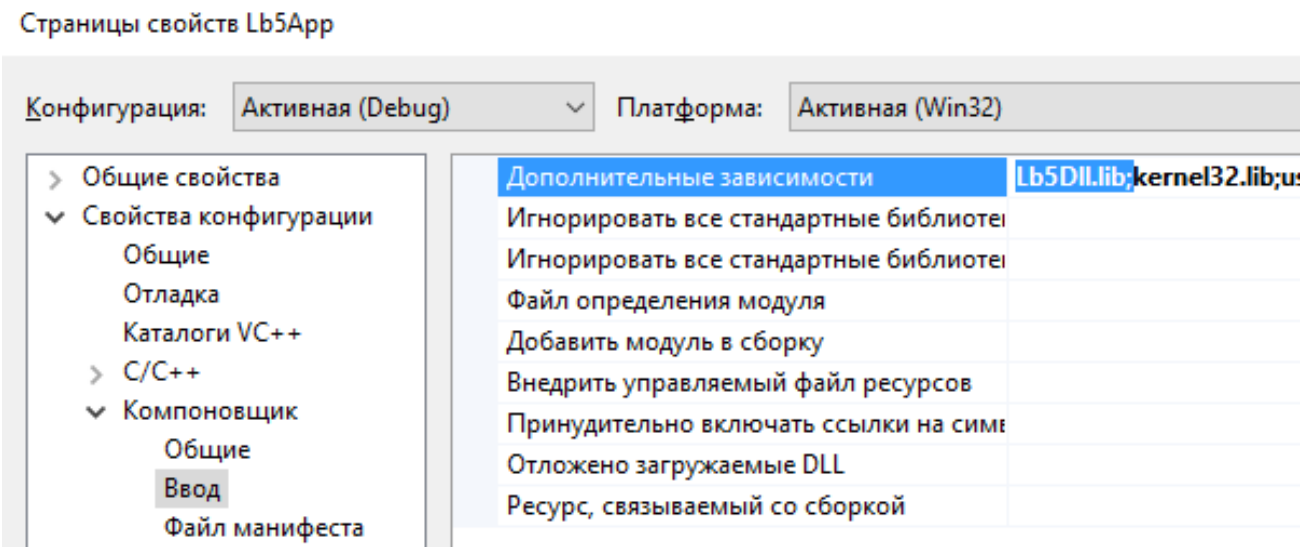


Рисунок 4 – Путь к lib-файлу для компоновщика

Ниже приведен пример программы Lb5App, подключающей библиотеку Lb5Dll и вызывающей ее функцию Prefix.

```
#include "stdafx.h"
#include <string>
#include <conio.h>

#include "Lb5Dll.h"
using namespace std;

int _tmain(int argc, _TCHAR* argv[])
{       setlocale(LC_CTYPE, "rus");
        char * tp = {" 007"};
        char pr = '№';
        string st = Prefix(tp, pr);
        printf_s("Агент = %5s", st.c_str());
        _getch();

        return 0;
}
```

1.4 Загрузка DLL

Существует два способа загрузки DLL: с явной и неявной компоновкой.

При неявной компоновке функции загружаемой библиотеки добавляются в секцию импорта вызывающего файла. При запуске такого файла загрузчик операционной системы анализирует секцию импорта и подключает все указанные в ней библиотеки. Пример проекта с неявной компоновкой приведен в разделе 1.3.

Неявная компоновка популярна благодаря своей простоте, но при выборе этого способа следует учитывать его недостатки и ограничения:

1. Все подключенные DLL всегда загружаются при запуске программы, хотя в процессе ее выполнения к ним может не быть ни одного обращения.
2. Поиск DLL производится в следующем порядке: в каталоге, где содержится вызывающий файл; в текущем каталоге процесса; в каталоге %Windows%\System32 (для 32-разрядной системы); в основном каталоге %Windows%; в каталогах из переменной окружения PATH. Вручную установить другой путь или порядок поиска невозможно.
3. Если хотя бы одна из требуемых DLL не будет найдена, загрузка программы прерывается ошибкой "*Dynamic link library could not be found*".

Для подключения DLL к программе этим способом достаточно передать компоновщику имя библиотеки импорта, чтобы тот использовал ее при сборке. Это самое простое решение данной задачи.

Явная компоновка необходима в тех случаях, когда требуется обойти перечисленные ограничения. Ее недостатком является определенное усложнение кода. Достоинства – подгрузка библиотек по мере необходимости и возможность программно обрабатывать ситуации их отсутствия. Также явная компоновка позволяет прямо указывать абсолютный или относительный путь к файлам DLL.

Явное подключение DLL осуществляется функцией LoadLibrary, возвращающей дескриптор библиотеки с заданным именем. Этот

дескриптор требуется при вызове всех остальных функций работы с библиотеками, поэтому его нужно сохранить в переменной.

Загрузка функций из DLL осуществляется с помощью вызова функции GetProcAddress.

Пример вызова функции из явно подключенной библиотеки:

```
typedef void (__stdcall * MYPROC)();    // определение типа
HMODULE hLib;
hLib = LoadLibrary(L"MyProject\MyDll.dll");
if (hLib) // проверка успешной загрузки библиотеки
{  cout << "Library loaded successfully." << endl;
   // загрузка функции
   MYPROC MyFunction = (MYPROC)GetProcAddress(hLib, "MyFunction");
   if (MyFunction)
   {    cout << "Function loaded successfully." << endl;
      MyFunction();  // вызов функции из загруженной библиотеки
   }
   FreeLibrary(hLib);
 }
```

2. МЕТОДИКА ВЫПОЛНЕНИЯ

1. Создать DLL, содержащую набор функций в соответствии с таблицей 21.

Таблица 21

Варианты индивидуальных заданий для бригад

Номер бригады	Возвращаемые функциями значения (s, s1 и s2, N, x, x1, x2, x3 – аргументы функций)
1	количество слов в строке s
2	количество вхождений строки s1 в строку s2
3	количество гласных букв в строке s
4	s=s1+s1+s1… – конкатенация аргумента s1 N раз
5	s=s1+s2+s1 (конкатенация строк); y=min (x1, x2, x3); x1, x2, x3 – целые числа
6	s=s1+s2 (конкатенация строк); y=sin(x)*cos(2x);
7, 8	длина s1+s2 (длина конкатенации строк); m=N! (факториал аргумента)
9, 10	класс строки s: 1– идентификатор; 2 – целое число; 3 – произвольная строка

2. Разработать приложение, использующее функции из разработанной библиотеки. Библиотеку подключить посредством неявной компоновки.

3. Разработать приложение, использующее функции из разработанной библиотеки. Библиотеку подключить посредством явной компоновки.

3. СОДЕРЖАНИЕ ОТЧЕТА

Отчет о работе должен включать:

1. Исходный код разработанной DLL.
2. Исходный код приложений, использующих DLL посредством неявной и явной компоновки.
3. Индивидуальные задания и результаты работы приложений.
4. Выводы.

Контрольные вопросы

1. Области применения DLL.
2. Структура DLL.
3. Создание DLL средствами среды MS Visual Studio.
4. Для чего используется заголовочный файл DLL?
5. Достоинства и недостатки явного способа загрузки DLL.
6. Достоинства и недостатки неявного способа загрузки DLL.
7. Программная реализация явного способа загрузки DLL.
8. Программная реализация неявного способа загрузки DLL.
9. Функции Win32, используемые при явной загрузке DLL.

ЛАБОРАТОРНАЯ РАБОТА № 7

ОБМЕН ДАННЫМИ МЕЖДУ ПРОЦЕССАМИ

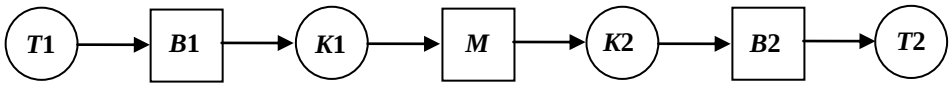
Цель работы – практическое знакомство со средствами передачи данных между процессами, выполняющимися на одном компьютере.

1. КРАТКИЕ ТЕОРЕТИЧЕСКИЕ СВЕДЕНИЯ

1.1 Способы передачи данных между процессами

Обмен данными между параллельными процессами (Interprocess Communications – IPC) подразумевает, что данные передаются от одного потока, выполняемого в контексте своего процесса, к другому потоку, выполняемому в контексте другого процесса. Соответственно, первый поток (посылающий данные) называется отправителем, а второй (получающий данные) – получателем или адресатом.

Потоки одного процесса могут обмениваться данными с помощью глобальных переменных и средств синхронизации потоков. Из-за невозможности использования общих переменных потоками разных процессов возникает необходимость в более сложных способах передачи данных. Для этого существуют специальные средства ОС, при использовании которых между процессами создается канал передачи данных. Схема канала приведена на рисунке 5 [6].



*T*1, *T*2 — пользовательские потоки
*B*1, *B*2 — буферы
*K*1, *K*2 — потоки ядра операционной системы
M — общая память

Рисунок 5 – Схема канала передачи данных

Канал передачи данных включает входной и выходной буферы памяти, потоки ядра операционной системы и общую память, доступ к которой имеют оба потока ядра. Канал передачи данных работает следующим образом:

– первый поток *K*1 ядра операционной системы читает данные из входного буфера *B*1 и записывает их в общую память *M*;
– второй поток *K*2 ядра читает данные из общей памяти *M* и записывает их в буфер *B*2.

Пользовательские потоки *T*1 и *T*2 посредством вызова функций ядра операционной системы имеют доступ к буферам *B*1 и *B*2 соответственно. Поэтому пересылка данных из потока *T*1 в поток *T*2 происходит следующим образом:

– пользовательский поток *T*1 записывает данные в буфер *B*1, используя специальную функцию ядра операционной системы;
– поток *K*1 ядра операционной системы читает данные из буфера *B*1 и записывает их в общую память *M*;
– поток *K*2 ядра операционной системы читает данные из общей памяти *M* и записывает их в буфер *B*2;
– пользовательский поток *T*2 читает данные из буфера *B*2.

Из рисунка 5 видно, что обмен данными между пользовательскими потоками может быть организован через цепочку взаимодействующих потоков, которые обмениваются данными через общую только для них память.

Обмен информацией по сети также может происходить через канал передачи данных, работа которого организована по рассмотренной схеме.

Простейший канал передачи данных между процессами можно организовать без поддержки операционной системы. Достаточно вместо потоков ядра ОС и общей памяти, показанных на рисунке 5, использовать файл, доступный для записи потоку одного процесса и для чтения потоку другого процесса.

Обмен данными между параллельно выполняющимися процессами может осуществляться двумя способами: потоком и сообщениями [6].

При передаче данных потоком, т.е. непрерывной последовательностью байтов, общая память *M,* доступная потокам ядра операционной системы, может и отсутствовать. Тогда пересылка данных выполняется одним потоком ядра непосредственно из буфера *B*1 в буфер *B*2.

Сообщение – это группа байтов фиксированного размера, Обмен с помощью сообщений называется передачей данных сообщениями.

1.2. Виды связей между процессами

Для передачи данных межу процессами прежде всего необходимо установить связь между ними. Связь между процессами может устанавливаться как на физическом (или аппаратном), так и на логическом (или программном) уровнях. В зависимости от направления передачи данных различают два вида связей:

- полудуплексная связь, т.е. когда данные могут передаваться только в одном направлении;
- дуплексная связь, т.е. когда данные могут передаваться в обоих направлениях.

При полудуплексной связи используются следующие топологии связи:

- между собой связаны только два процесса;
- один процесс связан с *N* процессами;
- каждый из *N* процессов связан с одним процессом;
- каждый из *N* процессов связан с множеством из *M* процессов.

Кроме того, обмен между процессами может быть синхронным и асинхронным.

Если поток-отправитель, отправив сообщение, блокируется до тех пор, пока не получит сообщение от потока-получателя с подтверждением приема, то такой обмен, называется синхронным.

В противном случае обмен сообщениями называется асинхронным.

Обмен данными потоком всегда происходит синхронным образом, так как в этом случае между отправителем и адресатом устанавливается непосредственная связь.

1.3 Использование буфера обмена

В качестве области памяти для обмена данными может использоваться буфер обмена (clipboard). Передачу данных между процессами с использованием буфера обмена можно рассматривать как организованный программным способом канал.

Для записи информации в буфер обмена сначала вызовом OpenClipboard типа BOOL необходимо его открыть и связать с текущим процессом. Параметром функции должен быть дескриптор hWndNewOwner того окна, которое должно быть связано с буфером обмена. Для окна текущего процесса параметр имеет значение NULL.

Функцией BOOL EmptyClipboard без параметров буфер очищается.

Далее нужно подготовить данные для передачи в буфер обмена. Для этого сначала функцией GlobalAlloc(GMEM_MOVEABLE, Size) выделяется блок глобальной памяти размером Size. Функция возвращает дескриптор hMem этой области памяти. Вызов функции GlobalLock(hMem) блокирует область от перемещения и возвращает указатель на нее.

Копирование данных в глобальный блок памяти по ее указателю выполняет функция

```
void CopyMemory(
  PVOID  Destination,
  const VOID   *Source,
```

SIZE_T Length

).

Destination – указатель на начальный адрес принимающего блока памяти, *Source – указатель на начальный адрес блока памяти для копирования, Length – размер блока памяти для копирования, в байтах.

После копирования блокировка перемещения блока памяти отменяется вызовом функции GlobalUnlock(hMem).

Для программного заполнения буфера обмена служит функция SetClipboardData.

HANDLE SetClipboardData(UINT uFormat,

HANDLE hMem).

Параметр uFormat устанавливает формат буфера обмена. В Win32 API имеется большое число стандартных форматов буфера обмена. Один из них CF_TEXT задает текстовый формат ANSI. hMem – это дескриптор блока глобальной памяти, содержащего данные в заданном формате.

Функция SetClipboardData возвращает дескриптор области данных.

После заполнения буфера обмена его следует закрыть вызовом функции CloseClipboard без параметров.

Чтобы какой-либо процесс получил доступ в буферу обмена для считывания хранящихся в нем данных, необходимо прежде всего функцией OpenClipboard открыть буфер обмена. Функция IsClipboardFormatAvailable(UINT uFormat) проверяет, содержит ли буфер обмена данные в заданном формате.

Для считывания данных служит функция GetClipboardData.

HANDLE GetClipboardData(

UINT uFormat).

uFormat задает формат представления данных в буфере. Возвращает функция дескриптор блока глобальной памяти со считанными данными заданном формате.

Используя дескриптор с помощью GlobalLock(hMem) можно получить указатель блока глобальной памяти и скопировать находящиеся в нем данные буфера обмена в локальный блок памяти процесса.

В завершении функцией CloseClipboard процесс-получатель должен закрыть буфер обмена.

1.4 Использование файлов, проецируемых в память

Еще одним программно организованным способом передачи данных между приложениями является использование области в страничном файле, доступ к которой могут иметь несколько процессов.

Для совместного использования файла, первый процесс должен создать или открыть файл с помощью функции CreateFile. Затем необходимо создать объект «проецируемый файл», используя функцию CreateFileMapping, для которой нужно определить дескриптор файла и имя для объекта «проецируемый файл». Имена объектов «проецируемый файл» совместно используют одно и то же пространство имен. Поэтому функции CreateFileMapping и OpenFileMapping завершаются ошибкой, если они устанавливают имя, которое используется объектом другого типа [6].

Функция OpenFileMapping определена так

```
HANDLE OpenFileMapping(
  DWORD dwDesiredAccess, // режим доступа
  BOOL bInheritHandle,       // флажок наследования
  LPCTSTR lpName              // имя объекта
).
```

Чтобы совместно использовать память, не связанную с каким-либо файлом, первый процесс должен использовать вызов функции CreateFileMapping со значением INVALID_HANDLE_VALUE параметра, задающего дескриптор файла hFile. Тогда

соответствующий объект «проецируемый файл» получит доступ к памяти, который поддерживается системным файлом подкачки. Размер объекта следует установить больше нуля [3].

Для получения дескриптора объекта «проецируемый файл», созданного первым процессом, другой процесс может использовать функцию OpenFileMapping с именем объекта. Оно указывается как имя совместно используемой памяти (named shared memory). Если объект «проецируемый файл» не имеет имени, то процесс может получить его дескриптор через посредство наследования или дублирования.

Процессы, которые совместно используют файлы, должны создавать представления файла, используя функцию MapViewOfFile. Они также должны координировать свой доступ, используя семафоры, мьютексы или события. Прототип функции MapViewOfFile имеет вид [1]:

```
LPVOID MapViewOfFile(
  HANDLE hFileMappingObject,  // дескриптор объекта
                              // «проецируемый файл»
  DWORD dwDesiredAccess,    // режим доступа
  DWORD dwFileOffsetHigh,   // старшее DWORD смещения
  DWORD dwFileOffsetLow,    // младшее DWORD смещения
  SIZE_T dwNumberOfBytesToMap // число отображаемых байтов
).
```

Совместно используемый объект «проецируемый файл» не должен разрушаться до тех пор, пока все процессы, которые используют его, не закроют свои дескрипторы, связанные с этим объектом, используя функцию CloseHandle.

1.5 Использование именованного канала

Именованным каналом называется объект ядра операционной системы, который обеспечивает передачу данных между процессами,

выполняющимися на компьютерах одной локальной сети. Процесс, создающий именованный канал, называется сервером именованного канала. Процессы, которые связываются с именованным каналом, называются клиентами именованного канала [6].

Именованный канал имеет имя, которое используется клиентами для связи с именованным каналом. Обмен может быть как полудуплексным, так и дуплексным синхронным и асинхронным. Передача данных может осуществляться как потоком, так и сообщениями.

Порядок работы с именованными каналами включает:

- создание сервером именованного канала;
- соединение сервера с экземпляром именованного канала;
- соединение клиента с экземпляром именованного канала;
- обмен данными по именованному каналу;
- отсоединение сервера от экземпляра именованного канала;
- закрытие именованного канала клиентом и сервером.

Сервер назначает именованному каналу имя в соответствии с универсальными правилами именования. Формат имени канала выглядит следующим образом:

\\Сервер\Pipe\ИмяКанала

Элемент имени Сервер указывает компьютер, на котором работает сервер именованного канала. Элемент Pipe должен быть строкой «Pipe», а ИмяКанала – уникальным именем, назначенным именованному каналу. Для сервера, работающего на локальном компьютере, используется псевдоним локального компьютера \\.\

1.5.1 Создание сервером именованного канала

Для создания сервером именованного канала используется функция CreateNamedPipe. Ее прототип имеет вид [6]:

```
HANDLE CreateNamedPipe (
  LPCTSTR lpName,            // указатель на имя канала
```

```
  DWORD dwOpenMode,    // режим открытия канала
  DWORD dwPipeMode,    // режим работы канала
  DWORD nMaxInstances, //макс. количество экземпляров канала
  DWORD nOutBufferSize, // размер выходного буфера в байтах
  DWORD nInBufferSize,   // размер входного буфера в байтах
  DWORD nDefaultTimeOut, // время ожидания в миллисекундах
  LPSECURITY_ATTRIBUTES lpSecurityAttributes // адрес
)                              // переменной для атрибутов защиты
```

Параметр dwOpenMode задает режим, в котором открывается канал. Режим работы канала при открытии (ориентированный на передачу потоком или сообщениями) задается, соответственно, константами PIPE_TYPE_BYTE или PIPE_TYPE_MESSAGE.

Параметр dwPipeMode, определяет режим работы канала. В этом параметре можно указать перечисленные выше константы PIPE_TYPE_BYTE, PIPE_TYPE_MESSAGE, а также PIPE_READMODE_BYTE, PIPE_READMODE_MESSAGE, PIPE_WAIT и PIPE_NOWAIT. Для всех реализаций канала необходимо указывать один и тот же набор констант.

Пример вызова данной функции для создания серверной части канала на локальной машине:

```
hPipe = CreateNamedPipe (L"\\.\PIPE\test", // Имя канала = 'test'.
  PIPE_ACCESS_DUPLEX |          // Дуплексный обмен
  FILE_FLAG_OVERLAPPED,         // Асинхронный ввод-вывод
  PIPE_WAIT |                   // Ожидать сообщений
  PIPE_READMODE_MESSAGE |    // Обмен сообщениями
  PIPE_TYPE_MESSAGE,
  PIPE_UNLIMITED_INSTANCES, // Максимальное число экземпляров канала
  OUT_BUF_SIZE, // Размеры буферов чтения/записи. 0 – размер по умолчанию
   IN_BUF_SIZE,
  TIME_OUT,       // Тайм-аут
NULL);            // Атрибуты безопасности
```

При первом вызове CreateNamedPipe с указанием какого-либо имени создается первый экземпляр именованного канала с этим именем и задается поведение всех последующих экземпляров этого канала. Повторно вызывая CreateNamedPipe, сервер может создавать

дополнительные экземпляры именованного канала, максимальное число которых должно задаваться при первом вызове CreateNamedPipe.

1.5.2 Соединение сервера с клиентом

Создав минимум один экземпляр именованного канала, сервер вызывом функции ConnectNamedPipe создает соединение с этим экземпляром, после чего именованный канал позволяет устанавливать соединения и с клиентами.

```
BOOL ConnectNamedPipe(
HANDLE hNamedPipe,        // дескриптор именованного канала
LPOVERLAPPED lpOverlapped); // адрес структуры OVERLAPPED.
```

Функция ConnectNamedPipe может выполняться как синхронно, так и асинхронно, и она не завершится, пока клиент не установит соединение через данный экземпляр именованного канала (или не возникнет ошибка).

Если используется только синхронный обмен, то в качестве значения lpOverlapped можно указать NULL.

В случае успешного завершения эта функция возвращает TRUE.

1.5.3 Соединение клиентов с именованным каналом

Прежде чем соединяться с именованным каналом, клиент должен определить, доступен ли какой-либо экземпляр этого канала для соединения. С этой целью клиент должен вызвать функцию WaitNamedPipe. Если заранее известно, что сервер вызвал функцию ConnectNamedPipe, функцию WaitNamedPipe можно не вызывать и сразу подключаться к именованному каналу.

Для подключения к серверу клиенты именованного канала используют функцию CreateFile или CallNamedPipe, указывая при вызове имя созданного сервером канала. Если клиенту разрешен

доступ к именованному каналу, он получает описатель, представляющий клиентскую сторону именованного канала, и функция ConnectNamedPipe, вызванная сервером, завершается.

В случае успешного завершения эта функция возвращает дескриптор именованного канала.

После того, как соединение по именованному каналу установлено, клиент и сервер могут использовать его для чтения и записи данных с помощью функций ReadFile и WriteFile. Именованные каналы поддерживают как синхронную, так и асинхронную передачу сообщений.

1.5.4 Обмен данными по именованному каналу

Для обмена данными по именованному каналу используются функции ReadFile и WriteFile [6].

Запись данных в открытый канал выполняется с помощью функции WriteFile, аналогично записи в обычный файл:

```
HANDLE hNamedPipe;
DWORD cbWritten;
char szBuf [256];
WriteFile(hNamedPipe, szBuf, strlen(szBuf) + 1, &cbWritten, NULL);
```

Через первый параметр функции WriteFile передается дескриптор экземпляра именованного канала. Через второй параметр передается адрес буфера, данные из которого будут записаны в канал. Размер этого буфера указывается при помощи третьего параметра. Предпоследний параметр используется для определения количества байтов данных, действительно записанных в канал. И, наконец, последний параметр задан как NULL, поэтому запись будет выполняться в синхронном режиме.

Для чтения данных из канала можно воспользоваться функцией ReadFile, например, так:

```
HANDLE hNamedPipe;
DWORD cbRead;
char szBuf[256];
ReadFile(hNamedPipe, szBuf, 512, &cbRead, NULL);
```

В примере данные, прочитанные из канала с дескриптором hNamedPipe, будут записаны в буфер szBuf, имеющий размер 512 байт. Количество действительно прочитанных байтов данных будет сохранено функцией ReadFile в переменной cbRead. Так как последний параметр функции указан как NULL, используется синхронный режим работы без перекрытия.

Разрушение экземпляра канала осуществляется вызовом функции DisconnectNamedPipe(hPipe).

2. МЕТОДИКА ВЫПОЛНЕНИЯ

1. Разработать приложение, выполняющее занесение текста в буфер обмена, и другое приложение, получающее данные из буфера обмена. Для занесения в буфер новых данных можно использовать стандартные приложения MS Windows.

Запустить оба приложения и произвести обмен.

2. Разработать два приложения, использующие для передачи данных общую область в страничном файле.

Запустить оба приложения и произвести обмен.

3. Разработать два приложения для передачи данных через именованный канал. Метод передачи – любой, передавать из одного приложения в другое можно любую строку.

Запустить оба приложения и произвести обмен

3. СОДЕРЖАНИЕ ОТЧЕТА

Отчет о работе должен включать:

1. Описание алгоритмов работы приложений, разработанных в п. 1-3 задания на выполнение работы.
2. Листинги разработанных приложений.
3. Результаты выполнения заданий.
4. Выводы.

Контрольные вопросы

1. Виды связей между процессами.
2. Способы передачи данных между процессами.
3. Чем отличается полудуплексный обмен от дуплексного?
4. Асинхронный обмен данными.
5. Синхронный обмен данными.
6. Файлы, проецируемые в память.
7. Буфер обмена.
8. Принцип работы функций ReadFile и WriteFile.
9. Создание именованного канала.

БИБЛИОГРАФИЧЕСКИЙ СПИСОК

1. http://msdn.microsoft.com/ru-ru/ [Электронный ресурс].
2. Рихтер Дж. Windows для профессионалов. 4-е изд. – СПб.: Питер, 2008 – 720с. – ISBN 978-5-7502-0360-4.
3. Рихтер Дж., Назар К. Windows via C/C++. Программирование на языке Visual C++. – СПб.: Питер, 2009 – 896 с. – ISBN 978-5-7502-0367-3.
4. Руссинович М., Соломон Д., Ионеску А., Йосифович П. Внутреннее устройство Windows [Текст]: – 7-е изд. – СПб.: Питер, 2018. - 944 с.: ил. – ISBN 978-5-4461-0663-9.
5. Побегайло А.П. Системное программирование в Windows [Текст]. СПб.: БХВ-Петербург, 2006. – 1056 с. – ISBN 5-49157-792-3.
6. Системное программное обеспечение: Метод. указ. к лаб. работам / Самар. гос. техн. ун-т; Сост. А.А. Тихомиров, А.М. Кистанов. Самара, 2013. – 80 с.

Приложение 1 Типы данных Win32 API

В таблице П1 приведены сведения о наиболее часто используемых типах C++ и Win32 API.

Таблица П1

Типы данных Win32 API

Тип данных	Описание
BOOL, BOOLEAN	Булевское (значения TRUE или FALSE).
BYTE	Байт (8 бит).
CHAR	8-битный символ Windows (ANSI).
DWORD, DWORD32	32-битное целое без знака. Диапазон значений от 0 до 4294967295 в десятичной системе счисления.
DWORDLONG	64-битное целое без знака. Диапазон значений от 0 до 18446744073709551615 в десятичной системе счисления.
DWORD_PTR	Указатель переменной типа DWORD.
FLOAT	Вещественное с плавающей точкой.
HANDLE	Дескриптор (хэндл) объекта.
HDC	Дескриптор контекста устройства (DC).
HFILE	Дескриптор файла, открытого функцией OpenFile или CreateFile.
HGDIOBJ	Дескриптор объекта GDI.
HGLOBAL	Дескриптор глобального блока памяти.
HLOCAL	Дескриптор локального блока памяти.
HMODULE	Дескриптор модуля. Это тип для базового адреса модуля в памяти.
HRGN	Дескриптор региона.
INT, INT32	32-битное целое со знаком. Диапазон значений от -2147483648 до 2147483647 в десятичной системе счисления.
INT_PTR	Указатель переменной типа INT.
LPCSTR	Указатель на строку символов
LPCTSTR	Указатель на строку символов, включая UNICODE
PVOID	Нетипизированный указатель.
_TCHAR	Универсальный символьный тип как для кодировки _UNICODE, так и для ASCII.

Приложение 2 Запуск программы из командной строки

При запуске программы через командную строку можно передать в нее какую-либо информацию через параметры argc и argv[] главной функции main. В этом случае ее заголовок должен иметь вид

```
int main(int argc, char* argv[])
```

Параметр argc не вводится в командной строке. Он автоматически получает значение в виде количества значений в командной строке элементов массива argv[], передаваемых в функцию main. Параметр argv[] – это массив указателей на строки. Причем argc[0] автоматически получает значение в виде указателя на строку, содержащую полный путь к exe-файлу программы. Через командную строку можно передать только данные строкового типа.

Ниже приведен пример простейшей программы, в которой предусмотрен ввод параметров в командной строке.

```
#include "stdafx.h"
#include "Windows.h"
#include <iostream>
using namespace std;

int main(int argc, char* argv[])
{
	for (int i=0; i < argc; i++) cout << i << ", " << argv[i] << endl;
	system("pause");
	return 0;
}
```

Значения элементов массива *argv в командной строке должны разделяться пробелами.

Системное программное обеспечение:
Лабораторный практикум

Составители*: Пугачев Анатолий Иванович*
Лапир Владимир Давидович

В авторской редакции

Подписано в печать 19.06.2020
Формат 60x84 $^{1}/_{16}$. Бум. офсетная.
Печать офсетная.
Усл. п. л. 4,6
Тираж 50 экз.

Федеральное государственное бюджетное образовательное учреждение
высшего профессионального образования
"Самарский государственный технический университет"
443100, г. Самара, ул. Молодогвардейская, 244.
Главный корпус

Отпечатано в типографии
Самарского государственного технического университета
443100, г. Самара, ул. Молодогвардейская, 244. Корпус № 8