Министерство науки и высшего образования Российской федерации

Федеральное государственное бюджетное образовательное учреждение
высшего образования
«Оренбургский государственный университет»

Кафедра программного обеспечения вычислительной техники
и автоматизированных систем

РЕШЕНИЕ ПРАКТИЧЕСКИХ ЗАДАЧ СИСТЕМНОГО ПРОГРАММИРОВАНИЯ

Методические указания

Составитель И.А. Щудро

Рекомендовано к изданию редакционно-издательским советом федерального государственного бюджетного образовательного учреждения высшего образования «Оренбургский государственный университет» для обучающихся по образовательной программе высшего образования по направлению подготовки 09.03.04 Программная инженерия

Оренбург
2021

УДК 004.451(076.5)
ББК 32.972.11я7
Р 47

Рецензент – кандидат технических наук, доцент Д.В. Горбачев

Р 47 **Решение практических задач системного программирования:** методические указания / составитель И.А. Щудро; Оренбургский гос. ун-т. Оренбург: ОГУ, 2021. – 97 с.

Методические указания содержат рекомендации для выполнения лабораторных работ по разработке программ системного программирования.

Методические указания предназначены для обучающихся по направлению подготовки 09.03.04 Программная инженерия при изучении дисциплины «Операционные системы и оболочки».

УДК 004.451(076.5)
ББК 32.972.11я7

Содержание

Введение

Настоящий лабораторный практикум предназначен для проведения исследований и получения практических навыков студентами направления 09.03.04 Программная инженерия при изучении дисциплины «Операционные системы и оболочки».

Лабораторный курс содержит шесть работ, рассчитанных на 16 часов аудиторных занятий. Предлагаемые задания охватывают основные исследовательские задачи курса.

Выполнение лабораторного практикума обеспечит формирование необходимой компетенции: «ОПК-6 Способен разрабатывать алгоритмы и программы, пригодные для практического использования, применять основы информатики и программирования к проектированию, конструированию и тестированию программных продуктов».

Достоинством практикума являются разработанные примеры решения предлагаемых задач.

В лабораторных работах решаются следующие задачи:

- управление процессами и потоками, их взаимодействие и синхронизация;
- использование механизмов виртуальной памяти и обмена сообщениями;
- управление каталогами и файлами.

Общие методические рекомендации по использованию лабораторных работ и методических указаний:

- к выполнению лабораторной работы следует приступать после ознакомления с теоретической частью соответствующего раздела, рекомендациями, приведенными в конкретной работе и прохождения коллоквиума;
- по результатам выполнения работы оформляется отчет установленной формы;
- дополнительная информация по лабораторным работам содержится в прилагаемом списке литературы.

Практикум рекомендован преподавателям для проведении лабораторных работ и практических занятий, а студентам – для аудиторного и самостоятельного освоения дисциплины «Операционные системы и оболочки».

1 Лабораторная работа №1. Управление процессами в ОС Windows

Цель работы: получение практических навыков создания и организации работы процессов и потоков в операционной системе Windows.

Задачи:

1. Выполнить программирование поставленной задачи на языке высокого уровня с использованием двух процессов;
2. Разработать отчет.

1.1 Теоретическая часть

В современных ОС пользователям предлагается несколько типов параллельной работы, основными из которых являются процессы и потоки. Процессы – это программы на этапе выполнения.

В ОС Windows создание процесса осуществляется с помощью вызова функции *Win*32 *API*, описываемой на языке Си следующим образом:

```
BOOL CreateProcess (
PCTSTR pszApplicationName. //имя исполняемого файла
PTSTR pszCommandLine, //командная строка
PSECURITY_ATTRIBUTES psaProcess, //атрибуты защиты процесса
PSECURITY_ATTRIBUTES psaThread, //атрибуты защиты потока
BOOL bInheritHandles, //наследование описателей
DWORD fwdCreate, //флаги процесса
PVOID pvEnvironment, //переменные окружения
PCTSTR pszCurDir, //текущий каталог
PSTARTUPINFO psiStrartInfo //начальная информация при создании процесса
PPROCESS_INFORMATION ppiProcInfo); //описатель процесса
```

Функция *CreateProcess* возвращает значение *TRUE*, если системе удастся создать процесс и начальный поток, при этом созданному объекту ядра будет присвоен уникальный идентификатор.

Процесс в ОС Windows можно завершить с помощью вызовов функций *Win*32 *API* под названием *ExitProcess* и *TerminateProcess*, описаные следующим образом:

```
VOID ExitProcess (
UINT fuExitCode); // код завершения процесса
```

Эта функция не возвращает значения.

```
VOID TerminateProcess (
HANDLE hProcess, // описатель завершаемого процесса
UINT fuExitCode); // код завершения процесса
```

Эта функция также не возвращает значения. Она отличается от *ExitProcess* тем, что ее может вызвать любой процесс и поток. Обе функции использовать нежелательно, поскольку процесс заканчивается, когда завершают работу все его потоки.

Следующий программный код просто создаст новый процесс и запустит калькулятор.

```
#include <windows.h>
void WinMain ()
{
STARTUPINFO start = { sizeof (start) };
PROCESS_INFORMATION procinfo;
TCHAR CommandLine[] = TEXT (“CALC”);

CreateProcess (NULL, CommandLine, NULL, NULL, FALSE, 0, NULL,
        NULL, &start, &procinfo);
}
```

1.2 Содержание отчета

- постановка задачи;
- блок – схема алгоритма работы программы;
- результаты работы программы;
- выводы о проделанной работе;
- листинг программного кода.

1.3 Варианты индивидуальных заданий

Вариант №1

Разработать две программы. Первая принимает от пользователя квадратную матрицу, осуществляет обход только крайних ее элементов, вычисляет их сумму, и выводит на экран. Вторая программа запускает первую в качестве вновь созданного процесса.

Вариант №2

Разработать две программы. Первая вычисляет число Фибоначчи по номеру, введенному пользователю, и формуле $F_i = F_{i-1} + F_{i-2}$, $F_0 = F_1 = 1$ и выводит его на экран. Вторая программа запускает первую в качестве вновь созданного процесса.

Вариант №3

Разработать две программы. Первая принимает от пользователя строку, хранящую знаковое целое число, и выводит на экран строковый эквивалент этого числа прописью (например, ввод «-1211» должен приводить к выводу «минус тысяча двести одиннадцать»). Вторая программа запускает первую в качестве вновь созданного процесса.

Вариант №4

Разработать две программы. Первая принимает от пользователя строку, хранящую число со знаком и плавающей точкой, и выводит на экран строковый эквивалент этого числа прописью (например, ввод «-12,11» должен приводить к выводу «минус двенадцать целых одиннадцать сотых»). Вторая программа запускает первую в качестве вновь созданного процесса.

Вариант №5

Разработать две программы. Первая принимает от пользователя две строки. Далее, если обе строки хранят целые числа со знаком, то на экран выводится сумма чисел, в противном случае – конкатенация двух введенных строк. Вторая про-

грамма запускает первую в качестве вновь созданного процесса.

Вариант №6

Разработать две программы. Первая принимает от пользователя две прямоугольных матрицы, а затем выводит на экран их сумму и произведение. Вторая программа запускает первую в качестве вновь созданного процесса.

Вариант №7

Разработать две программы. Первая принимает от пользователя одномерный целочисленный массив, упорядочивает его по возрастанию любым из так называемых «улучшенных алгоритмов» сортировки массивов и выводит на экран. Вторая программа запускает первую в качестве вновь созданного процесса.

Вариант №8

Разработать две программы. Первая принимает от пользователя одномерный массив чисел с плавающей точкой, упорядочивает его по убыванию любым из так называемых «улучшенных алгоритмов» сортировки массивов и выводит на экран. Вторая программа запускает первую в качестве вновь созданного процесса.

Вариант №9

Разработать две программы. Первая принимает от пользователя одномерный массив строк, упорядочивает его любым из так называемых «улучшенных алгоритмов» сортировки массивов и выводит на экран. Вторая программа запускает первую в качестве вновь созданного процесса.

Вариант №10

Разработать две программы. Первая принимает от пользователя квадратную матрицу, вычисляет сумму элементов, лежащих на главной и побочной диагоналях, и выводит на экран. Вторая программа запускает первую в качестве вновь созданного процесса.

Вариант №11

Разработать две программы. Первая принимает от пользователя квадратную матрицу, вычисляет сумму элементов, не лежащих на главной и побочной диагоналях, и результат выводит на экран. Вторая программа запускает первую в качестве вновь созданного процесса.

Вариант №12

Разработать две программы. Первая принимает от пользователя две даты – строки вида ЦЦ.ЦЦ.ЦЦЦЦ, где Ц – это любая цифра из диапазона [0-9]. Далее она вычисляет полное количество дней, прошедших между двумя введенными датами, и выводит его на экран. Вторая программа запускает первую в качестве вновь созданного процесса.

Вариант №13

Разработать две программы. Первая принимает от пользователя два значения времени – строки вида ЦЦ.ЦЦ.ЦЦ, где Ц – это любая цифра из диапазона [0-9]. Далее она вычисляет полное количество секунд, прошедших между двумя значениями времени, и выводит его на экран. Вторая программа запускает первую в качестве вновь созданного процесса.

Вариант №14

Разработать две программы. Первая принимает от пользователя две строки, осуществляет поиск вхождения второй строки в первую любым известным методом, кроме прямого, и выводит на экран значение индекса элемента первой строки, с которого началось совпадение, или -1 в противном случае. Вторая программа запускает первую в качестве вновь созданного процесса.

Вариант №15

Разработать две программы. Первая принимает от пользователя две строки, осуществляет поиск количества вхождений второй строки в первую любым известным методом, кроме прямого, и выводит на экран полученное значение. Вторая программа запускает первую в качестве вновь созданного процесса.

Вариант №16

Разработать две программы. Первая принимает от пользователя дату – строку вида ЦЦ.ЦЦ.ЦЦЦЦ, где Ц – это любая цифра из диапазона [0-9,] и выводит на экран число и месяц прописью, а за последними четырьмя – слово «года» (например, ввод «29.02.2008» приводит к выводу «Двадцать девятое февраля 2008 года»). Вторая программа запускает первую в качестве вновь созданного процесса.

Вариант №17

Разработать две программы. Первая принимает от пользователя значение времени – строку вида ЦЦ.ЦЦ.ЦЦ, где Ц – это любая цифра из диапазона [0-9] и выводит на экран значение часов минут и секунд прописью (например, ввод «12.01.20» приводит к выводу «двенадцать часов одна минута двадцать секунд»). Вторая программа запускает первую в качестве вновь созданного процесса.

Вариант №18

Разработать две программы. Первая принимает от пользователя строку из нулей и единиц – «битовую строку», инвертирует ее, выводит на экран значение инвертированной строки, переводит ее в число в десятичном формате и выводит полученное число на экран. Вторая программа запускает первую в качестве вновь созданного процесса.

Вариант №19

Разработать две программы. Первая принимает от пользователя строку из нулей и единиц – «битовую строку», осуществляет ее реверс, когда нули заменяются на единицы, а единицы на нули. Полученная строка выводится на экран, затем программа переводит ее в число в десятичном формате и выводит полученное число на экран. Вторая программа запускает первую в качестве вновь созданного процесса.

Вариант №20

Разработать две программы. Первая вычисляет факториал числа, введенного пользователем, по формуле $N! = N * (N - 1)!$, где 0 != 1, и выводит его на экран. Вторая программа запускает первую в качестве вновь созданного процесса.

1.4 Пример выполнения лабораторной работы

1.4.1 Задание

Разработать две программы. Первая вычисляет сумму и произведение чисел от *L* до *U*, где *L* – это нижняя граница диапазона, *U* – верхняя граница диапазона, границы вводятся пользователем, и выводит полученные значения на экран. Вторая программа запускает первую в качестве вновь созданного процесса.

1.4.2 Схема алгоритма.

Схема алгоритма программы представлена на рисунке 1.1

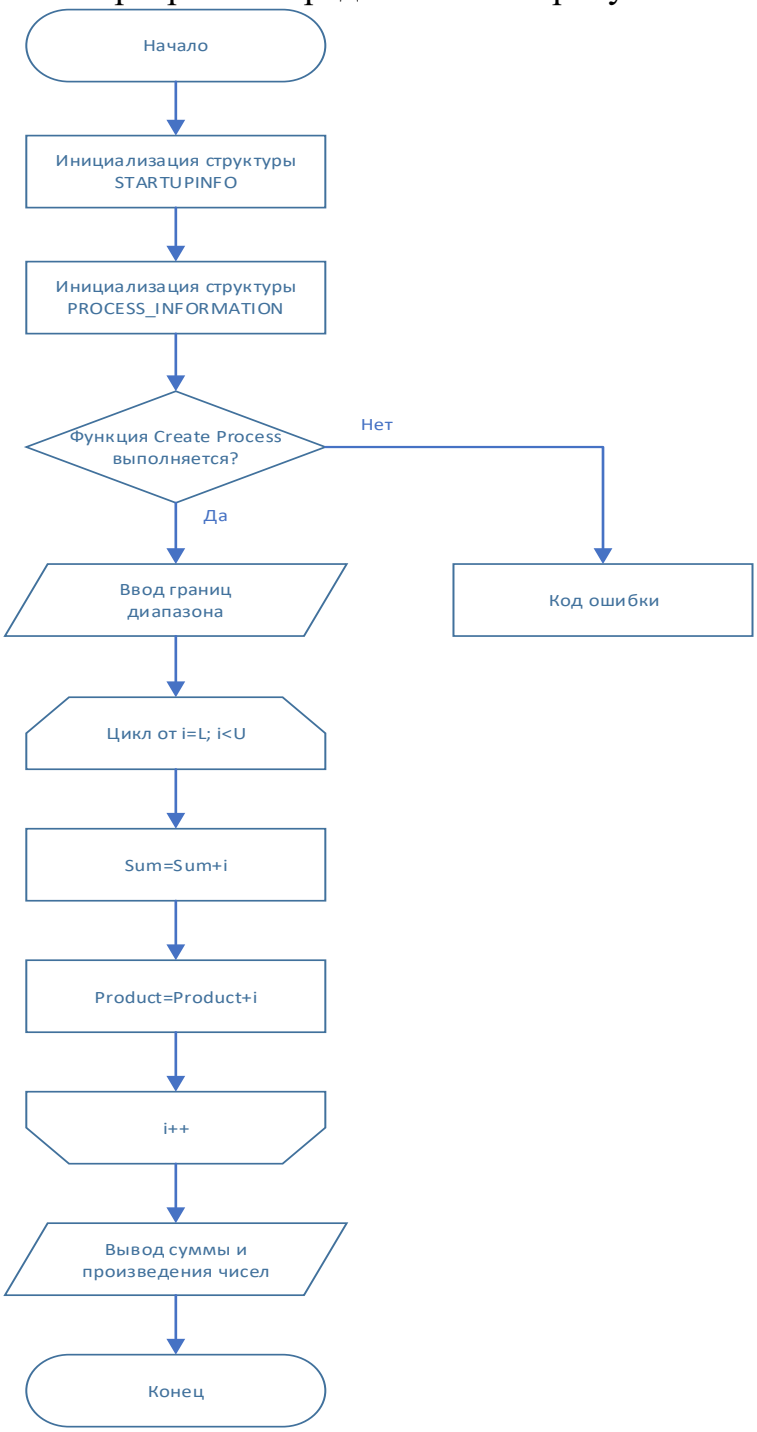

Рисунок 1.1 – Схема алгоритма программы

1.4.3 Код программы

Программа для запуска калькулятора в качестве вновь созданного процесса

```
#include <windows.h>
#include<iostream>
int main()
{
   setlocale(LC_ALL, "Rus");
   STARTUPINFO si ;
   PROCESS_INFORMATION pi;
   ZeroMemory(&si, sizeof(si));
   si.cb = sizeof(si);
   ZeroMemory(&pi, sizeof(pi));

   if (!CreateProcess(
      L"C:\\Users\\ПК\\OneDrive\\Desktop\\D\\repos\\PP LW 1.1\\Debug\\LW 1.1.exe",
      NULL, NULL, NULL, FALSE, 0, NULL, NULL, &si, &pi))
   {
      printf("CreateProcess failed (%d).\n", GetLastError());
      return 0;
   }
}
```

Основная программа (калькулятор)

```
#include <iostream>
using namespace std;
int L, U, Sum, Product=1;
int main()
{
   setlocale(LC_ALL, "Rus");
   cout << "Введите нижнюю границу диапазона"<<endl;
   cin >> L;
   cout << "Введите верхнюю границу диапазона"<<endl;
   cin >> U;
   for (int i = L; i < U; i++)
   {
     Sum = Sum+i;
     Product = Product * i;
   }
   cout << "Сумма чисел в диапазоне от "<< L <<" до "<< U <<" равна "<<Sum<<endl;
   cout << "Произведение чисел в диапазоне от " << L << " до " << U << " равно " << Product <<
endl;
}
```

1.4.4 Реализация программного кода

Реализация программного кода представлена на рисунке 1.2.

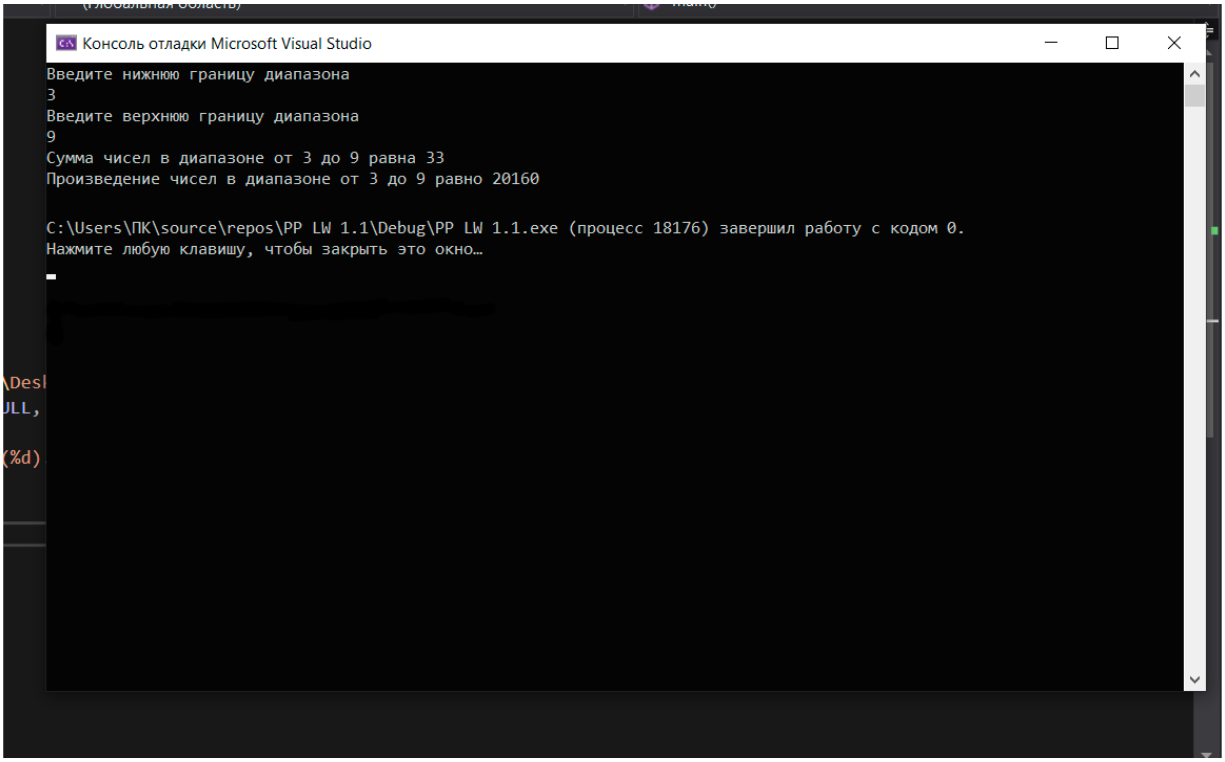

Рисунок 1.2 – Реализация программного кода

2 Лабораторная работа № 2. Разработка многопоточных приложений

Цель: получение практических навыков программной реализации многопоточных приложений в операционной системе Windows.

Задачи:

1. Выполнить программирование многопоточного приложения на языке высокого уровня;
2. Разработать отчет.

2.1 Теоретическая часть

В современных ОС пользователям предлагается несколько типов параллельной работы, основными из которых являются процессы и потоки. Процессы – это программы на этапе выполнения. Потоки – это меньшая единица работы. Однако с точки зрения распределения ресурсов именно потоки являются главными, поскольку им, а не процессам предоставляется на определенное время центральный процессор для выполнения какой-либо работы. Рассмотрим несколько функций *WinAPI*, выполняющих некоторые операции над потоками.

Функция *CreateThread* создает поток, для выполнения внутри адресного пространства вызывающего процесса.

```
HANDLE CreateThread (
PSECURITY_ATTRIBUTES lpThreadAttributes, // атрибуты защиты потока
DWORD dwStackSize,// начальный размер стека потока, в байтах
PTHREAD_START_ROUTINE lpStartAddress,// указатель на функцию потока
PVOID lpParameter,// параметр для нового потока
DWORD dwCreationFlags,// флаги создание потока
PDWORD lpThreadId //указатель на возвращаемый идентификатор потока
);
```

При успешном завершении функции возвращается дескриптор нового потока. При неудачном завершении функции возвращается *NULL*.

Параметры:

LpThreadAttributes – Указатель на структуру *SECURITY_ATTRIBUTES*, которая определяет, может ли возвращенный дескриптор быть унаследован дочерними процессами. Если *lpThreadAttributes NULL*, то дескриптор не может быть унаследован.

DwStackSize – Определяет размер, в байтах, стека для нового потока. Если определен 0, то размер стека по умолчанию равен размеру стека порождающего потока. Стек распределяется автоматически в пространстве памяти процесса, и освобождается при завершении потока.

LpStartAddress – Начальный адрес нового потока. Это обычно адрес функции, объявленной с соглашением о вызовах *WINAPI*, которое принимает одиночный 32-разрядный указатель как параметр и возвращает 32-разрядный код завершения. Прототип: *DWORD WINAPI ThreadFunc (LPVOID);*

LpParameter – Определяет единственное 32-разрядное значение параметра, передаваемое потоку.

DwCreationFlags – Определяет дополнительные флажки, которые управляют созданием потока. Если флажок определен *CREATE_SUSPENDED*, то поток создается в состоянии ожидания, и не будет выполняться, пока не будет вызвана функция *ResumeThread*. Если это значение нуль, то поток выполняется немедленно после создания.

LpThreadId – Указатель на 32-разрядную переменную, которая получает значение идентификатора потока

Поток можно завершить принудительно с помощью вызова следующих функций *Win*32 *API*:

```
VOID ExitThread (
DWORD ExitCode);// код завершения потока

VOID TerminateThread (
HANDLE hThread, // поток, который требуется завершить
DWORD ExitCode  // код завершения потока
 );
```

Для примера приведен фрагмент программы, которая вычисляет произведение всех чисел от 1 до 100.

```
// Функция потока
DWORD WINAPI ThreadFuction (PVOID Parametr)
{ int proizv = 1; // результат произведения
  int ii, *kk;
  kk = (int *) Parametr;
for (ii = *k; ii < (*kk) + 50; ii ++) proizv *= ii;
return proizv;
}
...
// код вызовов функций для создания потоков
DWORD idThread;
int k1 = 1; k2 = 51;
HANDLE h1, h2;
// Создается два потока в приостановленном состоянии
h1 = CreateThread (NULL, 0, ThreadFunction, &k1, CREATE_SUSPENDED, &idThread);
h2 = CreateThread (NULL, 0, ThreadFunction, &k2, CREATE_SUSPENDED, &idThread);
// Выполнение потоков
ResumeThread (h1);
ResumeThread (h2);
```

В данном коде встретилась весьма полезная *Win*32 *API* функция *ResumeThread*, которая возобновляет выполнение приостановленного потока и описанная как:

```
DWORD ResumeThread (
HANDLE hThread // поток, который требуется возобновить
);
```

Если вызов этой функции успешен, то возвращается предыдущее значение счетчика простоев данного потока, в противном случае – 0*xFFFFFFFF*.

Выполнение отдельного потока можно приостанавливать несколько раз (точно такое же число раз он должен возобновляться), а производится это вызовом функции *SuspendThread*, описанная как:

```
DWORD SuspendThread (
HANDLE hThread // поток, который требуется приостановить
);
```

Поток может сообщить ОС, чтобы она не выделяла ему процессор определенное время, указанное в миллисекундах:

```
VOID Sleep (DWORD MilliSeconds);
```

В ОС *Windows* 2000/*XP* есть также функция, которая находит и открывает поток по идентификатору:

```
HANDLE OpenThread (
DWORD DesiredAccess,
BOOL InheritHandle,
DWORD dwThreadId
);
```

2.2 Содержание отчета

- постановка задачи;
- блок – схема алгоритма работы программы;
- результаты работы программы;
- выводы о проделанной работе;
- листинг программного кода.

2.3 Варианты индивидуальных заданий

Вариант №1

Разработать программу, вычисляющую сумму крайних элементов квадратной матрицы. Выполнение данной операции оформляется как функция потока. Сначала осуществляется ввод элементов матрицы, затем запускается поток, и далее результат выводится на экран.

Вариант №2

Разработать программу, которая вычисляет число Фибоначчи по номеру, введенному пользователю, и формуле $F_i = F_{i-1} + F_{i-2}$, $F_0 = F_1 = 1$. Вычисление числа Фибоначчи оформить как функцию потока. По завершению функции потока программа выводит число на экран.

Вариант №3

Разработать программу для перевода целого числа со знаком в его строковый эквивалент прописью. Перевод числа оформить как функцию потока. Ввод числа происходит до запуска потока, а вывод строки – по его завершению. Например, ввод «-1211» должен приводить к выводу «минус тысяча двести одиннадцать».

Вариант №4

Разработать программу для перевода знакового числа с плавающей точкой в его строковый эквивалент прописью. Перевод числа оформить как функцию потока. Ввод числа происходит до запуска потока, а вывод строки – по его заверше-

нию. Например, ввод «-12.11» должен приводить к выводу «минус двенадцать целых одиннадцать сотых».

Вариант №5

Разработать программу, осуществляющую ввод двух строк, введенных пользователем. Далее, если обе строки хранят целые числа со знаком, то на экран выводится сумма чисел, в противном случае – конкатенация двух введенных строк. Проверку на соответствие строки целому числу, вычисление суммы чисел и конкатенации строк оформить как три разных функции потока. Ввод строк осуществляется до запуска всех потоков, а вывод результатов – после их завершения.

Вариант №6

Разработать программу, вычисляющую сумму и произведение двух матриц. Выполнение этих операций оформить как две функции потока.
Сначала программа осуществляет ввод элементов матриц, далее запускает оба потока, а затем выводит результаты на экран.

Вариант №7

Разработать программу для упорядочивания одномерного целочисленного массива. Сортировка массива по возрастанию должна осуществляться любым из так называемых «улучшенных алгоритмов» сортировки и оформляется как функция потока. Сначала выполняется ввод элементов матрицы, затем запускается поток и далее – вывод упорядоченного массива.

Вариант №8

Разработать программу для упорядочивания одномерного массива чисел с плавающей точкой. Сортировка массива по возрастанию должна осуществляться любым из так называемых «улучшенных алгоритмов» сортировки и оформляется как функция потока. Сначала выполняется ввод элементов матрицы, затем запускается поток и далее – вывод упорядоченного массива.

Вариант №9

Разработать программу для упорядочивания одномерного массива строк. Сортировка массива по возрастанию должна осуществляться любым из так называемых «улучшенных алгоритмов» сортировки и оформляется как функция потока. Сначала выполняется ввод элементов матрицы, затем запускается поток и далее – вывод упорядоченного массива.

Вариант №10

Разработать программу для вычисления суммы элементов, лежащих на главной и побочной диагоналях квадратной матрицы. Выполнение этой операции оформляется как функция потока. Ввод элементов матрицы осуществляется до запуска потока, а вывод полученного значения – по его завершению.

Вариант №11

Разработать программу для вычисления суммы элементов, не лежащих на главной и побочной диагоналях квадратной матрицы. Выполнение этой операции оформляется как функция потока. Ввод элементов матрицы осуществляется до запуска потока, а вывод полученного значения – по его завершению.

Вариант №12

Разработать программу для вычисления полного количества дней, прошедших между двумя датами. Даты – это строки вида ЦЦ.ЦЦ.ЦЦЦЦ, где Ц – это любая цифра из диапазона [0-9]. Вычисление разницы между датами оформляется как функция потока. Сначала осуществляется ввод дат, затем запускается поток, и далее – результат выводится на экран.

Вариант №13

Разработать программу для вычисления полного количества секунд, прошедших между двумя значениями времени. Значение времени – это строки вида ЦЦ.ЦЦ.ЦЦ, где Ц – это любая цифра из диапазона [0-9]. Вычисление разницы между временами оформляется как функция потока. Сначала осуществляется ввод значений времени, затем запускается поток, и далее – результат выводится на экран.

Вариант №14

Разработать программу для поиска вхождения подстроки в строку. Эта операция оформляется как функция потока и реализует любой из известных методов поиска подстроки, кроме – прямого. Сначала осуществляется ввод двух строк, затем запускается поток, и далее выводятся результаты: значение индекса элемента первой строки, с которого началось совпадение, или -1 в противном случае.

Вариант №15

Разработать программу для подсчета количества вхождений подстроки в строку. Эта операция оформляется как функция потока и реализует любой из известных методов поиска подстроки, кроме прямого. Сначала осуществляется ввод двух строк, затем запускается поток, и далее выводятся результат – целое число.

Вариант №16

Разработать программу для получения строкового эквивалента даты прописью. Дата – это строка вида ЦЦ.ЦЦ.ЦЦЦЦ, где Ц – это любая цифра из диапазона [0-9]. Получение строкового эквивалента оформляется как функция потока. Сначала осуществляется ввод даты, затем запускается поток, и далее результат выводится на экран: число и месяц прописью, а за последними четырьмя – слово «года» (например, ввод «29.02.2008» приводит к выводу «Двадцать девятое февраля 2008 года»).

Вариант №17

Разработать программу для получения строкового эквивалента значения времени прописью. Время – это строка вида ЦЦ.ЦЦ.ЦЦ, где Ц – это любая цифра из диапазона [0-9]. Получение строкового эквивалента оформляется как функция потока. Сначала осуществляется ввод значения времени, затем запускается поток, и далее результат выводится на экран: значение часов минут и секунд прописью (например, ввод «12.01.20» приводит к выводу «двенадцать часов одна минута двадцать секунд»).

Вариант №18

Разработать программу, осуществляющую инвертирование битовой строки, а также ее перевод в десятичное число. Битовая строка – это строка, состоящая из нулей и единиц. Инвертирование битовой строки и перевод строки в десятичное число оформляется как две функции потока. Сначала осуществляется ввод битовой строки, затем запускаются два потока, и далее выводятся результаты.

Вариант №19

Разработать программу, осуществляющую реверс битовой строки, а также ее перевод в десятичное число. Битовая строка – это строка, состоящая из нулей и единиц. Реверс битовой строки (все нули заменяются на единицы, а единицы на нули) и перевод строки в десятичное число оформляется как две функции потока. Сначала осуществляется ввод битовой строки, затем запускаются два потока, и далее выводятся результаты.

Вариант №20

Разработать программу, осуществляющую вычисление факториала числа. Выполнение данной операции при $N! = N * (N - 1)!$, где $0! = 1$, оформляется как функция потока. Сначала осуществляется ввод числа, затем запускается поток, и далее результат выводится на экран.

2.4 Пример выполнения лабораторной работы

2.4.1 Задание.

Разработать две программы. Первая вычисляет сумму и произведение чисел от L до U, где L – это нижняя граница диапазона, U – верхняя граница диапазона, границы вводятся пользователем, и выводит полученные значения на экран. Вторая программа запускает первую в качестве вновь созданного процесса.

2.4.2 Схема алгоритма.

Схема алгоритма программы представлена на рисунке 2.1

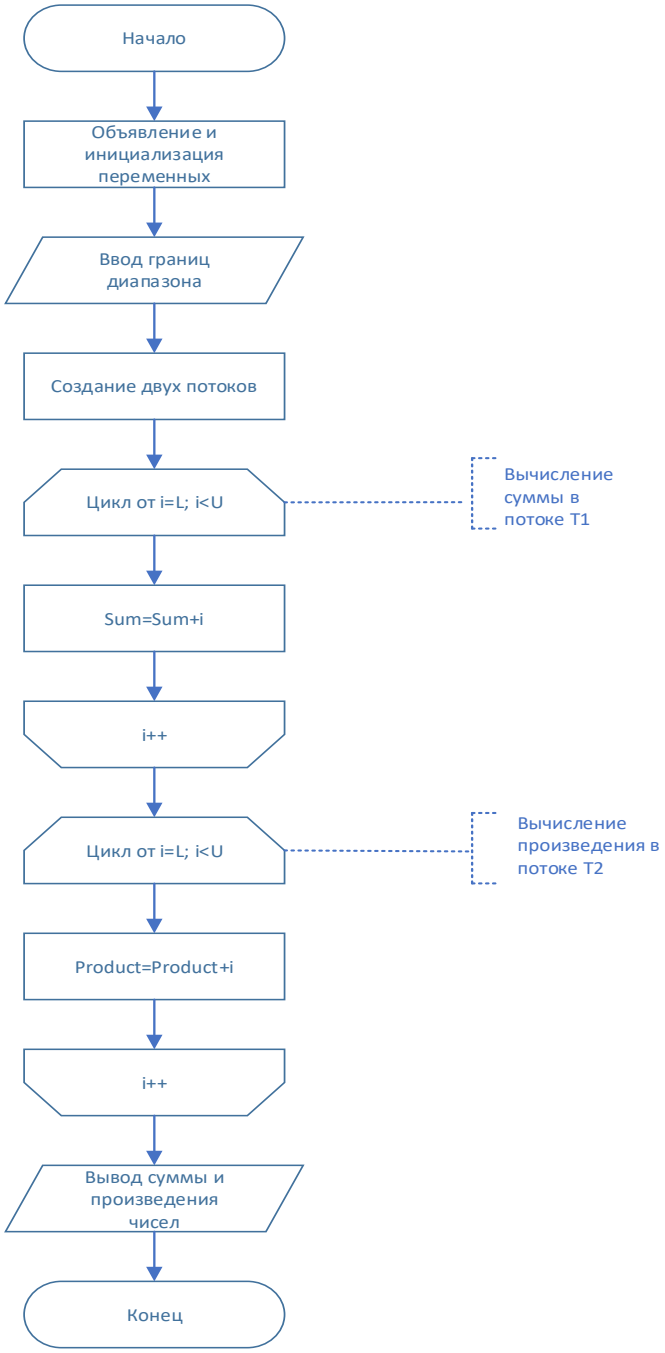

Рисунок 2.1 – Схема алгоритма программы

2.4.3 Код программы

```
#include <iostream>
#include <Windows.h>
#include <chrono>
#include <thread>
using namespace std;
void Cons(int L, int U, int& Sum)
{
   cout << "ID потока = " << this_thread::get_id() << endl;
   cout << "Началось выполнение функции нахождения суммы" << endl;
   for (int i = L; i < U; i++)
   {
```

```
        Sum += i;
    }
    cout << "Сумма чисел в диапазоне от " << L << " до " << U << " равна " << Sum << endl<<endl;
}
void Proizv(int L, int U, int& Product)
{
    this_thread::sleep_for(chrono::milliseconds(3000));  /*Задержка во времени выполнения функ-
ции в потоке для видимости, что она действительно выполняется в другом потоке*/
    cout << "ID потока = " << this_thread::get_id() << endl;
    cout << "Началось выполнение функции нахождения произведения" << endl;
    for (int i = L; i < U; i++)
    {
        Product *= i;
    }
    cout << "Произведение чисел в диапазоне от " << L << " до " << U << " равно " << Product <<
endl<<endl;
}
int main()
{
    int L, U, Sum=0, Product = 1;
    setlocale(LC_ALL, "Rus");
    cout << "Введите нижнюю границу диапазона" << endl;
    cin >> L;
    cout << "Введите верхнюю границу диапазона" << endl;
    cin >> U;
    cout << endl;
    thread T1(Cons, L, U, ref(Sum)); //создание двух отдельных потоков
    thread T2(Proizv, L, U, ref(Product));
    T1.join();
    T2.join();
    system ("pause");
    return 0;
}
```

2.4.4 Реализация программного кода

Реализация программного кода представлена на рисунке 2.2.

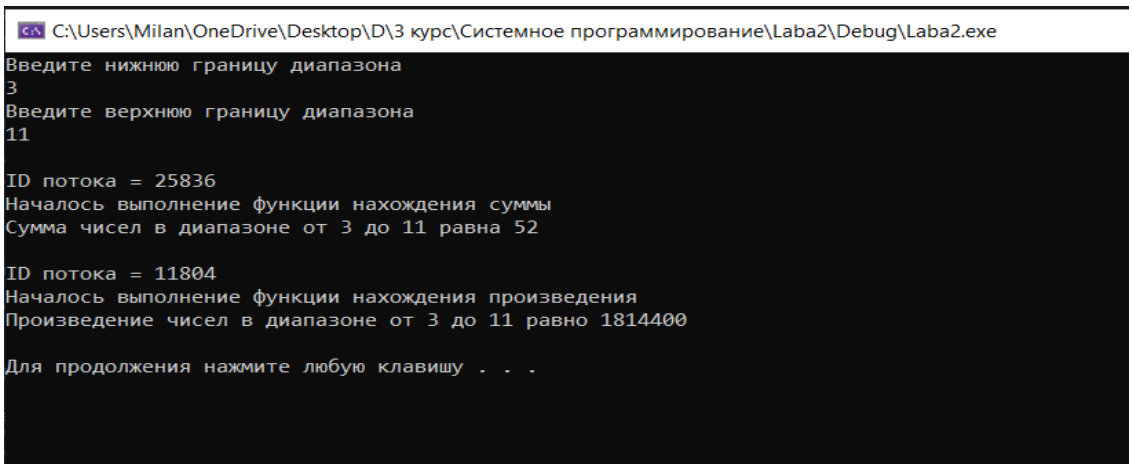

Рисунок 2.2 – Реализация программного кода

3 Лабораторная работа № 3. Управление приоритетами потоков

Цель: получение практических навыков использования приоритетов потоков в операционной системе Windows.

Задачи:

1. Выполнить программирование поставленной задачи на языке высокого уровня;
2. Разработать отчет.

3.1 Теоретическая часть

В основе многих алгоритмов планирования процессов лежит концепция приоритетного обслуживания. У потоков изначально известна характеристика – приоритет, на основании которого определяется порядок выполнения потоков. Приоритет – это число, характеризующее степень привилегированности потока при использовании ресурсов вычислительной машины, в том числе (и в первую очередь) времени процессора: чем выше приоритет, тем выше привилегии, а значит, поток меньше времени будет проводить в очередях. Приоритет может выражаться числом. В большинстве ОС, поддерживающих потоки, их приоритет непосредственно связан с приоритетом процесса, в рамках которого выполняется данный поток. Значение приоритета из описателя процесса используется при назначении приоритета потокам этого процесса.

Во многих ОС предусматривается возможность изменения приоритетов в течение жизни потока. Изменения приоритета могут происходить по инициативе самого потока, когда он обращается с соответствующим вызовом к ОС, или по инициативе пользователя, когда он выполняет соответствующую команду. Кроме того, ОС сама может изменять приоритеты потоков в зависимости от ситуации, складывающейся в системе. В последнем случае приоритеты называются динамическими, в отличие от неизменяемых, фиксированных приоритетов.

В ОС Windows процессу можно задать класс приоритета.
Поддерживается шесть классов приоритетов:

- простаивающий (*IDLE_PRIORITY_CLASS*), когда потоки выполняются, если система не занята другой работой;
- ниже нормального (*BELOW_NORMAL_PRIORITY_CLASS*);
- нормальный (*NORMAL_PRIORITY_CLASS*), у потоков нет особых требований;
- выше нормального (*ABOVE_NORMAL_PRIORITY_CLASS*);
- высокий (*HIGH_PRIORITY_CLASS*) – потоки немедленно реагируют на события;
- реального времени (*REALTIME_PRIORITY_CLASS*), где потоки тоже немедленно реагируют на события и могут вытеснять даже потоки ОС.

Класс приоритета устанавливается с помощью функции:

BOOL SetPriorityClass (
HANDLE Process,
DWORD Priority).

Например, процесс может поменять свой собственный класс приоритета:

SetProrityClass (GetCurrentProcess (),HIGH_PRIORITY_CLASS);

Чтобы узнать класс приоритета можно воспользоваться функцией:

DWORD GetPriorityClass (
HANDLE Process).

Выбрав класс приоритета, для процесса, не нужно забывать о потоках. В ОС Windows поддерживается семь относительных приоритетов потоков:

THREAD_PRIORITY_TIME_CRITICAL (уровень приоритета 31 в классе *REALTIME* и 15 в других); *THREAD_PRIORITY_HIGHEST* (на два уровня выше для текущего класса, для *NORMAL* уровень приоритета равен 10);

THREAD_PRIORITY_ABOVE_NORMAL (на один уровень выше, для *NORMAL* уровень приоритета равен 9);

THREAD_PRIORITY_NORMAL (обычный приоритет для класса, уровень равен 8);

THREAD_PRIORITY_BELOW_NORMAL (на один уровень ниже, для *NORMAL* уровень приоритета равен 7);

THREAD_PRIORITY_LOWEST (на два уровня ниже для текущего класса, для *NORMAL* уровень приоритета равен 6);

THREAD_PRIORITY_IDLE (16 – для *REALTIME* и 1 в других классах).

Задать относительный приоритет потока можно с помощью функции:

BOOL SetThreadPriority (
HANDLE Thread,
int Priority).

Узнать относительный приоритет потока позволяет функция:

int GetThreadPriority (
HANDLE Thread).

Следующий фрагмент кода показывает, как создать поток с относительным приоритетом *HIGH*, а по умолчанию поток создается с приоритетом *NORMAL*.

```
DWORD ThreadId;
HANDLE Thread = CreateThread (NULL, 0, ThreadFunction,
NULL, CREATE_SUSPENDED, &ThreadId);
SetThreadPriority (Thread, THREAD_PRIORITY_HIGH);
ResumeThread (Thread);
CloseHandle (Thread).
```

Иногда бывает полезным знать, сколько времени поток затратил на выполнение каких-либо операций. В *Windows NT*/2000/*XP* такая функция есть:

```
BOOL GetThreadTimes (
HANDLE Thread,
PFILETIME Created,// Время с 01.01.1601 в сотнях нс до создания потока
PFILETIME Exited,// Время с 01.01.1601 в сотнях нс до завершения потока
PFILETIME Kernel,// Время в сотнях нс, затраченное потоком на код ОС
PFILETIME User);// Время в сотнях нс, затраченное потоком на код программы.
```

3.2 Содержание отчета

- постановка задачи;
- блок – схема алгоритма работы программы;
- результаты работы программы;
- выводы о проделанной работе;
- листинг программного кода.

3.3 Варианты индивидуальных заданий

Вариант №1

Разработать программу, вычисляющую сумму крайних элементов квадратной матрицы. Выполнение данной операции оформляется как функция потока. Сначала осуществляется ввод элементов матрицы, затем запускается поток, и далее результат выводится на экран. Запустить программу два раза: первый раз с приоритетом потока *THREAD_PRIORITY_ABOVE_NORMAL*, второй – *THREAD_PRIORITY_HIGHEST*, и вывести на экран значения времени работы потока.

Вариант №2

Разработать программу, которая вычисляет число Фибоначчи по номеру, введенному пользователю, и формуле $F_i = F_{i-1} + F_{i-2}$, $F_0 = F_1 = 1$. Вычисление числа Фибоначчи оформить как функцию потока. По завершению функции потока программа выводит число на экран. Запустить программу два раза: первый раз с приоритетом потока *THREAD_PRIORITY_TIME_CRITICAL*, второй – *THREAD_PRIORITY_NORMAL*, и вывести на экран значения времени работы потока.

Вариант №3

Разработать программу для перевода целого числа со знаком в его строковый эквивалент прописью. Перевод числа оформить как функцию потока. Ввод числа происходит до запуска потока, а вывод строки – по его завершению. Например, ввод «-1211» должен приводить к выводу «минус тысяча двести одиннадцать». Запустить программу два раза: первый раз с приоритетом потока *THREAD_PRIORITY_ABOVE_NORMAL*, второй – *THREAD_PRIORITY_LOWEST*, и вывести на экран значения времени работы потока.

Вариант №4

Разработать программу для перевода знакового числа с плавающей точкой в его строковый эквивалент прописью. Перевод числа оформить как функцию потока. Ввод числа происходит до запуска потока, а вывод строки – по его завершению. Например, ввод «-12.11» должен приводить к выводу «минус двенадцать целых одиннадцать сотых». Запустить программу два раза: первый раз с приоритетом потока *THREAD_PRIORITY_ABOVE_NORMAL*, второй – *THREAD_PRIORITY_HIGHEST*, и вывести на экран значения времени работы потока.

Вариант №5

Разработать программу, осуществляющую ввод двух строк, введенных пользователем. Далее, если обе строки хранят целые числа со знаком, то на экран

выводится сумма чисел, в противном случае – конкатенация двух введенных строк. Проверку на соответствие строки целому числу, вычисление суммы чисел и конкатенации строк оформить как три разных функции
потока (с приоритетами, соответственно, *THREAD_PRIORITY_ABOVE_NORMAL, HREAD_PRIORITY_LOWEST* и *THREAD_PRIORITY_IDLE*). Ввод строк осуществляется до запуска всех потоков, а вывод результатов – после их завершения. Также выводятся значения времени работы каждого потока.

Вариант №6

Разработать программу, вычисляющую сумму и произведение двух матриц. Выполнение этих операций оформить как две функции потока. Сначала программа осуществляет ввод элементов матриц, далее запускает оба потока с приоритетами *THREAD_PRIORITY_IDLE* и *THREAD_PRIORITY_TIME_CRITICAL*, а затем выводит на экран результаты, а также значения времени работы каждого потока.

Вариант №7

Разработать программу для упорядочивания одномерного целочисленного массива. Сортировка массива по возрастанию должна осуществляться любым из так называемых «улучшенных алгоритмов» сортировки и оформляется как функция потока. Сначала выполняется ввод элементов матрицы, затем запускается поток и далее – вывод упорядоченного массива. Запустить программу два раза: первый раз с приоритетом потока *THREAD_PRIORITY_ABOVE_NORMAL*, второй – *THREAD_PRIORITY_LOWEST*, и вывести на экран значения времени работы потока.

Вариант №8

Разработать программу для упорядочивания одномерного массива чисел с плавающей точкой. Сортировка массива по возрастанию должна осуществляться любым из так называемых «улучшенных алгоритмов» сортировки и оформляется как функция потока. Сначала выполняется ввод элементов матрицы, затем запускается поток и далее – вывод упорядоченного массива. Запустить программу два раза: первый раз с приоритетом потока *THREAD_PRIORITY_ABOVE_NORMAL*, второй – *THREAD_PRIORITY_HIGHEST*, и вывести на экран значения времени работы потока.

Вариант №9

Разработать программу для упорядочивания одномерного массива строк. Сортировка массива по возрастанию должна осуществляться любым из так называемых «улучшенных алгоритмов» сортировки и оформляется как функция потока. Сначала выполняется ввод элементов матрицы, затем запускается поток и далее – вывод упорядоченного массива. Запустить программу два раза: первый раз с приоритетом потока *THREAD_PRIORITY_BELOW_NORMAL*, второй – *THREAD_PRIORITY_ABOVE_NORMAL*, и вывести на экран значения времени работы потока.

Вариант №10

Разработать программу для вычисления суммы элементов, лежащих на главной и побочной диагоналях квадратной матрицы. Выполнение этой операции оформляется как функция потока. Ввод элементов матрицы осуществляется до

запуска потока, а вывод полученного значения – по его завершению. Запустить программу два раза: первый раз с приоритетом потока *THREAD_PRIORITY_NORMAL*, второй – *THREAD_PRIORITY_LOWEST*, и вывести на экран значения времени работы потока.

Вариант №11

Разработать программу для вычисления суммы элементов, не лежащих на главной и побочной диагоналях квадратной матрицы. Выполнение этой операции оформляется как функция потока. Ввод элементов матрицы осуществляется до запуска потока, а вывод полученного значения – по его завершению. Запустить программу два раза: первый раз с приоритетом потока *THREAD_PRIORITY_NORMAL*, второй – *THREAD_PRIORITY_LOWEST,* и вывести на экран значения времени работы потока.

Вариант №12

Разработать программу для вычисления полного количества дней, прошедших между двумя датами. Даты – это строки вида ЦЦ.ЦЦ.ЦЦЦЦ, где Ц – это любая цифра из диапазона [0-9]. Вычисление разницы между датами оформляется как функция потока. Сначала осуществляется ввод дат, затем запускается поток, и далее – результат выводится на экран. Запустить программу два раза: первый раз с приоритетом потока *THREAD_PRIORITY_BELOW_NORMAL*, второй – *THREAD_PRIORITY_ABOVE_NORMAL*, и вывести на экран значения времени работы потока.

Вариант №13

Разработать программу для вычисления полного количества секунд, прошедших между двумя значениями времени. Значение времени – это строки вида ЦЦ.ЦЦ.ЦЦ, где Ц – это любая цифра из диапазона [0-9]. Вычисление разницы между временами оформляется как функция потока. Сначала осуществляется ввод значений времени, затем запускается поток, и далее – результат выводится на экран. Запустить программу два раза: первый раз с приоритетом потока *THREAD_PRIORITY_NORMAL*, второй – *THREAD_PRIORITY_LOWEST*, и вывести на экран значения времени работы потока.

Вариант №14

Разработать программу для поиска вхождения подстроки в строку. Эта операция оформляется как функция потока и реализует любой из известных методов поиска подстроки, кроме прямого. Сначала осуществляется ввод двух строк, затем запускается поток, и далее выводятся результаты: значение индекса элемента первой строки, с которого началось совпадение, или -1 в противном случае. Запустить программу два раза: первый раз с приоритетом потока *THREAD_PRIORITY_TIME_CRITICAL*, второй – *THREAD_PRIORITY_NORMAL*, и вывести на экран значения времени работы потока.

Вариант №15

Разработать программу для подсчета количества вхождений подстроки в строку. Эта операция оформляется как функция потока и реализует любой из известных методов поиска подстроки, кроме прямого. Сначала осуществляется ввод двух строк, затем запускается поток, и далее выводятся

результат – целое число. Запустить программу два раза: первый раз с приоритетом потока *THREAD_PRIORITY_ABOVE_NORMAL*, второй – *THREAD_PRIORITY_HIGHEST*, и вывести на экран значения времени работы потока.

Вариант №16

Разработать программу для получения строкового эквивалента даты прописью. Дата – это строка вида ЦЦ.ЦЦ.ЦЦЦЦ, где Ц – это любая цифра из диапазона [0-9]. Получение строкового эквивалента оформляется как функция потока. Сначала осуществляется ввод даты, затем запускается поток, и далее результат выводится на экран: число и месяц прописью, а за последними четырьмя – слово «года» (например, ввод «29.02.2008» приводит к выводу «Двадцать девятое февраля 2008 года»). Запустить программу два раза: первый раз с приоритетом потока *THREAD_PRIORITY_TIME_CRITICAL*, второй – *THREAD_PRIORITY_NORMAL*, и вывести на экран значения времени работы потока.

Вариант №17

Разработать программу для получения строкового эквивалента значения времени прописью. Время – это строка вида ЦЦ.ЦЦ.ЦЦ, где Ц – это любая цифра из диапазона [0-9]. Получение строкового эквивалента оформляется как функция потока. Сначала осуществляется ввод значения времени, затем запускается поток, и далее результат выводится на экран: значение часов минут и секунд прописью (например, ввод «12.01.20» приводит к выводу «двенадцать часов одна минута двадцать секунд»). Запустить программу два раза: первый раз с приоритетом потока *THREAD_PRIORITY_BELOW_NORMAL*, второй – *THREAD_PRIORITY_ ABOVE_NORMAL*, и вывести на экран значения времени работы потока.

Вариант №18

Разработать программу, осуществляющую инвертирование битовой строки, а также ее перевод в десятичное число. Битовая строка – это строка, состоящая из нулей и единиц. Инвертирование битовой строки и перевод строки в десятичное число оформляется как две функции потока. Сначала осуществляется ввод битовой строки, затем запускаются два потока (первый с приоритетом *THREAD_PRIORITY_HIGHEST*, другой – *THREAD_PRIORITY_IDLE*), и далее выводятся результаты, а также значения времени работы каждого потока.

Вариант №19

Разработать программу, осуществляющую реверс битовой строки, а также ее перевод в десятичное число. Битовая строка – это строка, состоящая из нулей и единиц. Реверс битовой строки (все нули заменяются на единицы, а единицы на нули) и перевод строки в десятичное число оформляется как две функции потока.

Сначала осуществляется ввод битовой строки, затем запускаются два потока с приоритетами *THREAD_PRIORITY_LOWEST*, второй – *THREAD_PRIORITY_ ABOVE_NORMAL*, и далее выводятся результаты, а также значения времени работы каждого потока.

Вариант №20

Разработать программу, осуществляющую вычисление факториала числа. Выполнение данной операции (по формуле $N! = N * (N – 1)!$, где 0 != 1) оформляет-

ся как функция потока. Сначала осуществляется ввод числа, затем запускается поток, и далее результат выводится на экран. Запустить программу два раза: первый раз с приоритетом потока *THREAD_PRIORITY_BELOW_NORMAL*, второй – *THREAD_PRIORITY_ABOVE_NORMAL*, и вывести на экран значения времени работы потока.

3.4 Пример выполнения лабораторной работы

3.4.1 Задание

Разработать программу, которая вычисляет сумму и произведение чисел от *L* до *U*, где *L* – это нижняя граница диапазона, *U* – верхняя граница диапазона. Вычисление суммы и произведения оформить как две функции потока. Значения границ диапазон вводятся пользователем, затем запускаются два требуемых потока (первый с приоритетом *THREAD_PRIORITY_HIGHEST*, другой – *THREAD_PRIORITY_IDLE*), а потом на экран выводится полученные значения, а также значения времени работы обоих потоков.

3.4.2 Схема алгоритма

Схема алгоритма программы представлена на рисунке 3.1

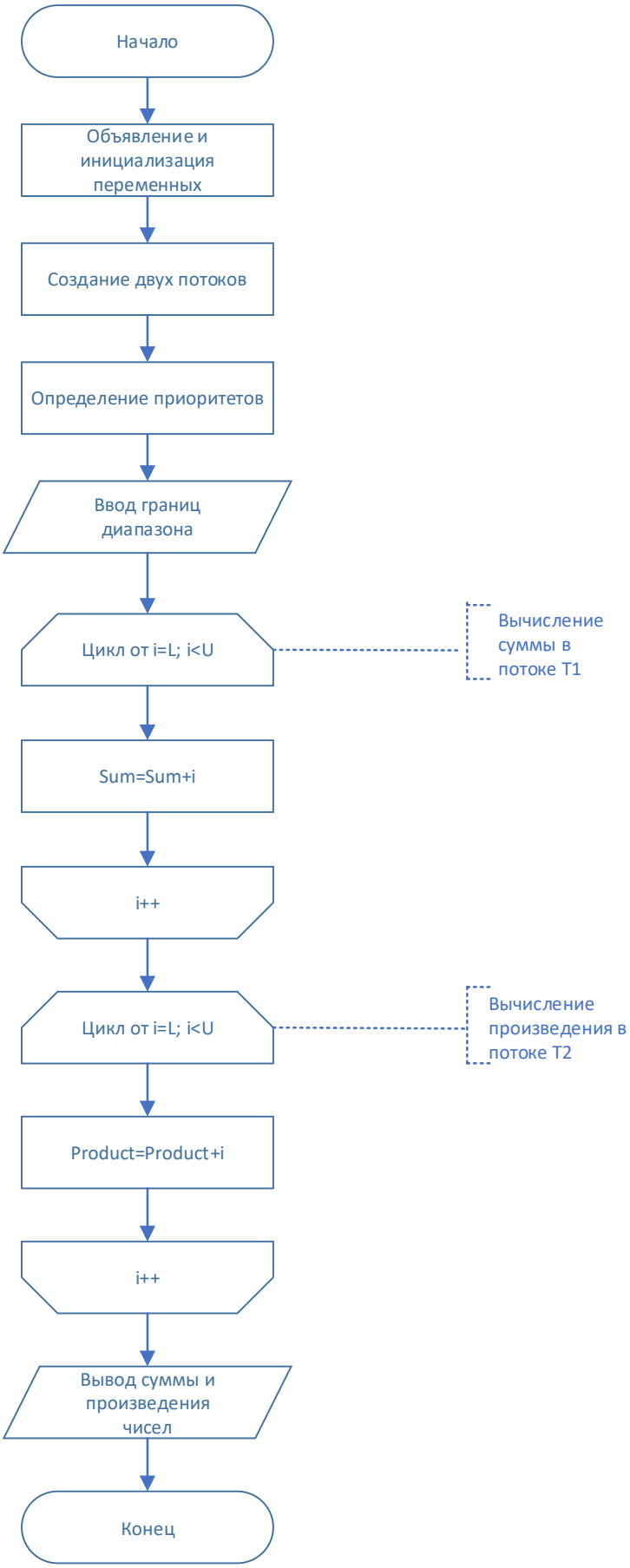

Рисунок 3.1 – Схема алгоритма программы

3.4.3 Код программы

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;
using System.Threading;
using System.Diagnostics;
namespace ConsoleApp1
{
   class Program
   {
      static UInt64 product = 1;
      static UInt64 sum = 0;
      static UInt64 a, b;
      //Создание двух отдельных потоков
      static Thread one = new Thread(new ThreadStart(Proizved));
      static Thread two = new Thread(new ThreadStart(Cons));
      static void Main(string[] args)
      {
         one.Priority = ThreadPriority.Lowest; //Низкий приоритет для функции вычисления произ-
ведения
         two.Priority = ThreadPriority.Highest; //Высший приоритет для функции вычисления сум-
мы
         Console.WriteLine("Введите диапазон значений");
         var read = Console.ReadLine().Split();
         a = Convert.ToUInt64(read[0]);
         b = Convert.ToUInt64(read[1]);
         one.Start();
         two.Start();
         Console.ReadKey();
      }
      public static void Proizved()
      {
         Stopwatch SW = new Stopwatch();
         SW.Start();
         for (UInt64 i = a; i < b; i++)
         {
            product = product * i;
         }
         SW.Stop();
         Console.WriteLine("Умножение равно: {0}, Время подсчета: {1}", product, Con-
vert.ToString(SW.ElapsedTicks));
      }
      public static void Cons()
      {
         Stopwatch SW = new Stopwatch();
         SW.Start();
         for (UInt64 i = a; i < b; i++)
         {
            sum += i;
```

```
        }
        SW.Stop();
        Console.WriteLine("Сложение равно: {0}, Время подсчета: {1}", sum, Con-
vert.ToString(SW.ElapsedTicks)); }
    }
}
```

3.4.4 Реализация программного кода

Реализация программного кода представлена на рисунке 3.2.

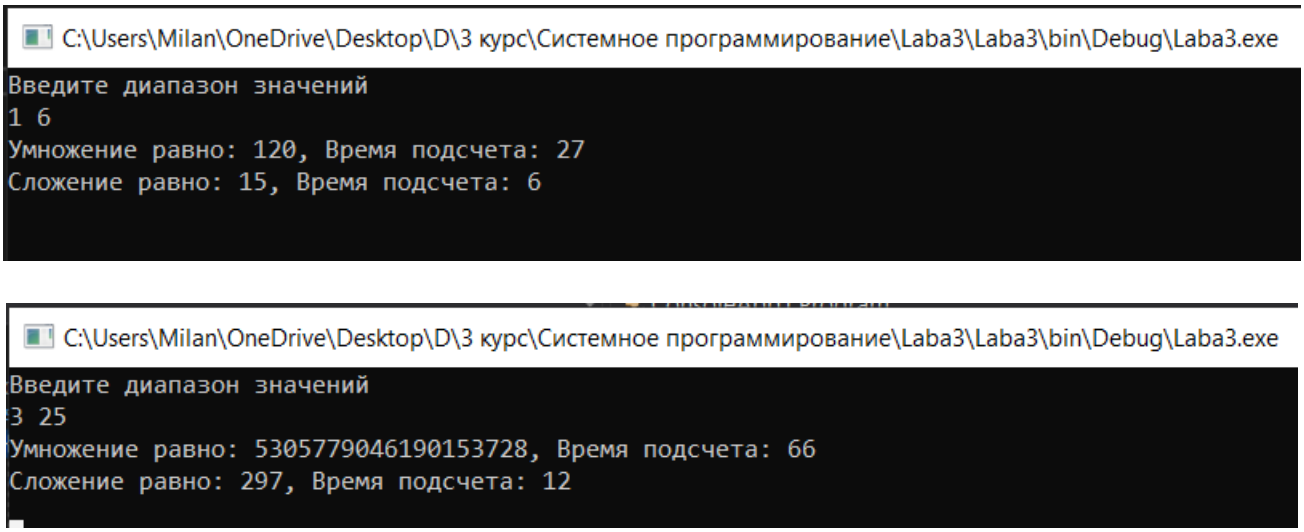

Рисунок 3.2 – Реализация программного кода

4 Лабораторная работа № 4. Синхронизация потоков в среде ОС Windows

Цель: получение практических навыков программирования объектов синхронизации, используемых в операционной системе Windows.

Задачи:

1. Выполнить программирование поставленной задачи на языке высокого уровня;
2. Разработать отчет.

4.1 Теоретическая часть

Существует обширный класс средств ОС, с помощью которых обеспечивается взаимная синхронизация процессов и потоков. Потребность в синхронизации потоков связана с совместным использованием ресурсов вычислительной системы. Синхронизация необходима для исключения гонок и тупиков при обмене данными между потоками, разделении данных, при доступе к центральному процессору и УВВ.

Во многих ОС эти средства называются средствами межпроцессного взаимодействия (*IPC*), поскольку обычно к ним относятся не только средства межпроцессной синхронизации, но и средства межпроцессного обмена данными.

Синхронизация заключается в согласовании скоростей процессов и потоков путем приостановки потока до наступления некоторого события и последующей его активизации при наступлении этого события. ОС может предоставлять обширный набор средств синхронизации.

Очень важным понятием синхронизации является понятие «критической секции» программы. Критическая секция – это часть программы, результат выполнения которой может непредсказуемо меняться, если переменные, относящиеся к этой части программ, изменяются другими потоками в то время, когда выполнение этого кода еще не закончено. Критическая секция всегда определяется по отношению к конкретным критическим данным, при несогласованном изменении которых могут возникнуть нежелательные эффекты. Во всех потоках, работающих с критическими данными, должна быть определена критическая секция, в общем случае состоящая из разных последовательностей команд. Взаимное исключение позволяет обеспечить нахождение только одного потока в критической секции.

Блокирующие переменные.

Для синхронизации потоков одного процесса программист может использовать глобальные блокирующие переменные. С этими переменными, к которым все потоки имеют прямой доступ, программист работает, не обращаясь к системным вызовам ОС. Каждому набору критических данных ставится в соответствие двоичная переменная, которой поток присваивает значение 0, когда он входит в критическую секцию, и значение 1, когда он ее покидает. Недостаток: во время нахождения одного потока в критической секции, другой поток, требующий тот же ре-

сурс, получив доступ к процессору, будет с завидной регулярностью опрашивать блокирующую переменную, бесполезно затрачивая процессорное время. Для устранения этого недостатка во многих ОС предусмотрены специальные системные вызовы для работы с критическими секциями.

В ОС Windows перед изменением критических данных, поток выполняет системный вызов *EnterCriticalSection*, в рамках которого сначала выполняет проверка блокирующей переменной, отражающей состояние ресурса. Если он занят (значение блокирующей переменной равно 0), он блокирует поток и делает отметку о том, что поток должен быть активизирован, когда соответствующий ресурс освободится. Поток, который в это время использует данный ресурс, после выхода из критической секции должен выполнить вызов *LeaveCriticalSection*, в результате блокирующая переменная получает значение 1 (ресурс свободен), а ОС просматривает очередь ожидающих этот ресурс потоков и переводит первый поток в состояние готовности. Эти, а также некоторые другие функции *Win32 API* для работы с критическими секциями приведены ниже:

VOID EnterCriticalSection (PCRITICAL_SECTION Section);
VOID LeaveCriticalSection (PCRITICAL_SECTION Section).

Когда ни один поток не использует критическую секцию, ее можно удалить:

VOID DeleteCriticalSection (PCRITICAL_SECTION Section);

Когда критическая секция является локальной, ее нужно инициализировать:

VOID InitializeCriticalSection (PCRITICAL_SECTION Section);

Попытаться войти в критическую секцию без блокирования потока можно с помощью функции:

BOOL TryEnterCriticalSection (PCRITICAL_SECTION Section);

Она возвращает *FALSE*, если ресурс занят другим потоком, и *TRUE*, если поток захватил нужный ресурс.

Для того чтобы организовать доступ к двум ресурсам, нужно создать две критические секции, что и демонстрирует следующий фрагмент кода:

```
int Numbers[500]; // первый разделяемый ресурс
CRITICAL_SECTION Nums;
double Doubles[500];
CRITICAL_SECTION DoubleNums; // второй разделяемый ресурс
DWORD ThFunction (PVOID Parametr) // функция потока
{
// Вход в обе критические секции
EnterCriticalSection (&Nums);
EnterCriticalSection (&DoubleNums);// В этом коде требуется одновременный доступ
 к обоим разделяемым ресурсам
for(int j = 0; j < 500; j ++) Doubles[j] = Numbers[j] = 500 – j:) // Покидаем
критические секции в обратном порядке
LeaveCriticalSection (&DoubleNums);
LeaveCriticalSection (&Nums);
return 0;
}
```

Эффект взаимной блокировки может возникнуть, если попытаться добавить еще одну функцию потока, где производится занятие тех же критических секций, но в обратном порядке:

```
DWORD ThFunction1 (PVOID Parametr) // функция потока
{ // Вход в обе критические секции
EnterCriticalSection (&DoubleNums);
EnterCriticalSection (&Nums);
 // В этом коде требуется одновременный доступ
 // к обоим разделяемым ресурсам
for(int j = 0; j < 500; j ++) Doubles[j] = Numbers[j] = 500 – j;)
 // Покидаем критические секции в обратном порядке
LeaveCriticalSection (&Nums);
LeaveCriticalSection (&DoubleNums);
return 0;
}
```

Здесь существует вероятность того, что *ThFunction* занимает критическую секцию *Nums*, а поток с функцией *ThFunction*1 захватывает *DoubleNums*. И теперь, какая бы функция не выполнялась, она не сумеет войти в другую, так необходимую ей критическую секцию.

Замечательное свойство критической секции заключается в том, что она не использует переход из режима пользователя в режим ядра. Следовательно, скорость ее работы достаточно высока, и этот объект может быть использован для большинства программ, требующих синхронизации. К недостаткам можно отнести то, что их можно использовать для синхронизации потоков, принадлежащих только одному и тому же процессу.

Обобщением блокирующих переменных являются, так называемые семафоры Дийкстры. Вместо двоичных переменных используются переменные, которые могут принимать целые неотрицательные значения. Такие переменные, используемые для синхронизации, получили название семафоров.

Для работы с семафорами вводятся два примитива (действия) *P* и *V*. Пусть переменная *S* представляет собой семафор, тогда действия *V*(*S*) и *P*(*S*) определяются так:

- *V*(*S*): переменная *S* увеличивается на 1. Выборка, инкремент и сохранение не могут быть прерваны. К переменной *S* нет доступа другим потокам во время выполнения этой операции.
- *P*(*S*): уменьшение *S*, если это возможно. Если *S* равно 0, то поток, вызывающий операцию *P*, пока декремент станет возможным. Проверка и уменьшение являются неделимой операцией.

В частном случае, когда семафор может принимать только значения 0 и 1, он превращается в блокирующую переменную, которую называют двоичным семафором.

Потоки с помощью специального системного вызова сообщают ОС, что они хотят синхронизировать свое выполнение с состоянием некоторого объекта (*WaitForSingleObject* в ОС Windows). Другой системный вызов может переводить объект в сигнальное состояние (например, *SetEvent* в ОС Windows).

Поток может ожидать установки сигнального состояния не одного объекта, а нескольких (*WaitForMultipleObjects*). При этом он может попросить ОС активизировать его при установке либо одного указанного объекта, либо всех. Поток может в качестве аргумента системного вызова ожидания указать также макси-

мальное время, которое он будет ожидать перехода объекта в сигнальное состояние, после чего ОС должна активизировать его в любом случае. Установки некоторого объекта в сигнальное состояние могут ожидать сразу несколько потоков. В зависимости от объекта в состояние готовности могут переводиться либо все ожидающие это событие потоки, либо один из них.

В ОС Windows есть набор функций, которые ожидают перехода в сигнальное состояние одного или нескольких объектов:

```
DWORD WaitForSingleObject (
HANDLE Object,
DWORD Milliseconds); // определяет ожидания,
// INFINITE - бесконечное ожидание
```

В следующем коде поток блокирован, пока не выполнится другой поток *Th*1:

```
WaitForSingleObject (Th1, INFINITE);
```

Второй пример демонстрирует значение таймаута, не равное *INFINITE*:

```
DWORD dw = WaitForSingleObject(Th1, 10000);
switch (dw) {
case WAIT_OBJECT_0: // поток завершил работу
break;
case WAIT_TIMEOUT: // поток не завершился через 10 сек.
break;
case WAIT_FAILED: // произошла какая-то ошибка
break;
}
```

Сразу несколько объектов или один из списка можно с помощью функции:

```
DWORD WaitForMultipleObjects (
DWORD Counter,// количество объектов ядра (от 1 до 64)
HANDLE *Objects,// массив описателей объектов
BOOL WaitForAll,// ожидать все (TRUE) или
// один из списка (FALSE)
DWORD Milliseconds); // определяет ожидания,
// INFINITE - бесконечное ожидание
```

Следующий код демонстрирует использование этой функции:

```
HANDLE hh[2];
hh[0] = Th1; hh[1] = Th2;
DWORD = WaitForMultipleObjects(2, hh, FALSE, 10000);
switch (dw) {
case WAIT_FAILED: // произошла какая-то ошибка
break;
case WAIT_TIMEOUT: // поток не завершился через 10 сек.
break;
case WAIT_OBJECT_0: // поток Th1 завершил работу
break;
case WAIT_OBJECT_0 + 1: // поток Th2 завершил работу
break;
}
```

В числе других в ОС можно встретить такие объекты как событие, мьютекс, системный семафор, таймер.

Объект – событие используется для того, чтобы оповестить другие потоки о

том, что некоторые действия завершены. События обычно используют в том случае, когда какой-то поток выполняет инициализацию, а затем сигнализирует другому потоку, что он может продолжить работу. Инициализирующий поток переводит «событие» в несигнальное состояние и приступает к своим операциям. Закончив, он сбрасывает «событие» в сигнальное состояние. Тогда другой поток, ждавший перехода события в сигнальное состояние, переводится в состояние готовности. В ОС Windows события содержат счетчик числа пользователей и две логических переменных: тип события и состояние. Эти объекты могут быть двух типов: с автосбросом и с ручным сбросом. Событие создается функцией:

```
HANDLE CreateEvent (
PSECURITY_ATTRIBUTES Attributes, // атрибуты защиты
BOOL ManualOrAuto, // ручной (TRUE) или
// автоматический (FALSE) сброс
BOOL Initial, // Начальное состояние: свободен (TRUE) или
// занят (FALSE)
PCTSTR Name); // Символьное имя объекта
```

Созданное событие может открываться с помощью функции:

```
HANDLE OpenEvent (
DWORD Access, // режим доступа
BOOL Inherit, // наследование
PCTSTR Name); // Символьное имя объекта
```

После создания события можно управлять его состоянием. Для этого существуют две функции. Перевод в сигнальное состояние осуществляется вызовом функции:

```
BOOL SetEvent (HANDLE Event);
```

Сменить его на занятое можно при помощи функции:

```
BOOL ResetEvent (HANDLE Event);
```

Когда поток успешно дождался события с автосбросом, он автоматически сбрасывается в занятое состояние, и для них, как правило, не нужно вызывать функцию *ResetEvent*.

Как используются события, показано в следующем фрагменте кода:

```
HANDLE Event1;
int WinMain(...)
{
// Создается событие с ручным сбросом в сигнальном состоянии
Event1 = CreateEvent (NULL, FALSE, FALSE, NULL);
// Создаются два потока, причем пропущены все параметры,
// кроме функции потока
HANDLE Th1 = CreateThread (..., Function1,...);
HANDLE Th2 = CreateThread (..., Function2,...);
...
// далее можно выполнять любые действия
...
CloseHandle (Event1);
}
DWORD WINAPI Function1 (PVOID Parametr)
{
WaitForSingleObject (Event1, INFINITE);
```

```
...
SetEvent (Event1);
return 0;
}
DWORD WINAPI Function2 (PVOID Parametr)
{
WaitForSingleObject (Event1, INFINITE);
...
SetEvent (Event1);
return 0;
}
```

Ожидаемый таймер – это объект, который самостоятельно переходит в сигнальное состояние в определенное время или через регулярные промежутки времени. Объект создается функцией:

```
HANDLE CreateWaitableTimer (
PSECURITY_ATTRIBUTES Attributes, // атрибуты защиты
BOOL ManualOrAuto,
// ручной (TRUE) или автоматический (FALSE) сброс состояния
PCTSTR Name); // Символьное имя объекта
```

Ожидаемый таймер всегда создается в несигнальном состоянии. Чтобы перевести таймер в сигнальное состояние достаточно вызвать функцию:

```
HANDLE SetWaitableTimer (
HANDLE Timer, // нужный таймер
const LARGE_INTEGER *DueTime,// Когда таймер должен сработать впервые
LONG Period, // сколько будет срабатывать таймер,
// 0 – для одного срабатывания
PTIMERAPCROUTINE ComplRoutine,
// процедура для асинхронного вызова
PVOID ArgumentsToRoutine,
// аргумент для процедуры асинхронного вызова
BOOL Resume);
// необходим для компьютеров с поддержкой спящего режима
```

Чтобы перевести ожидаемый таймер в несигнальное состояние, нужно вызвать:

```
HANDLE CancelWaitableTimer (
HANDLE Timer); // нужный таймер
```

В следующем фрагменте кода таймер настраивается так, чтобы сработать в первый раз 28 февраля 2021 года в 17.30, и после этого – каждые три часа, то есть 10800000 миллисекунд:

```
HANDLE Timer1;
SYSTEMTIME Time1;
FILETIME LocalTime, UTC_Time;
LARGE_INTEGER IntUTC;
// создается таймер с автосбросом
Timer1 = CreateWaitableTimer (NULL, FALSE, NULL);
// задаются параметры для таймера
Time1.wYear = 2021; Time1.wMonth = 2; Time1.wDay = 28;
Time1.wHour = 17; Time1.wMinute = 30; Time1.wSecond = 0;
Time1.wMilliseconds = 0;
SystemTimeToFileTime (&Time1, &LocalTime);
```

```
LocalFileTimeToFileTime (&LocalTime, &UTC_Time);
IntUTC.LowPart = UTC_Time.dwLowDateTime;
IntUTC.HighPart = UTC_Time.dwHighDateTime;
// наконец устанавливается таймер
SetWaitableTimer (Timer1, &IntUTC, 10800000, NULL, NULL, FALSE);
...
CloseHandle (Timer1);
```

Можно также устанавливать время срабатывания не в абсолютных единицах, а в относительных, которые рассчитываются в блоках по 100 нс (то есть 0,1 сек. равна миллиону таких блоков), число при этом должно быть отрицательным. В следующем коде показано, как установить таймер на срабатывание через 20 секунд после вызова соответствующей функции:

```
HANDLE Timer1;
LARGE_INTEGER LargeInt;
// создается таймер с автосбросом
Timer1 = CreateWaitableTimer (NULL, FALSE, NULL);
// задаются параметры для таймера,
// который должен сработать через 20 сек.
// Время берется в блоках по 100 нс.
int UnitsBySeconds = 10000000;
LargeInt = -20 * UnitsBySeconds;
// наконец устанавливается таймер,
// который сработает вначале через 20 сек,
// а потом каждые три часа
SetWaitableTimer (Timer1, &LargeInt, 10800000, NULL, NULL, FALSE);
...
CloseHandle (Timer1);
```

Последнее замечание относительно ожидаемых таймеров. Когда освобождается таймер с ручным сбросом, то все потоки, ожидавшие данный объект, возобновляют свою работу, а когда в сигнальное состояние переходит таймер с автосбросом, возобновляется выполнение только одного потока.

Семафор в целом был описан выше. Что касается ОС Windows, то семафор в них является объектом ядра, и чаще всего такие объекты используются для учета ресурсов. В них помимо остальных параметров, характерных для многих объектов ядра, есть еще два специфичных: один используется для установки максимально возможного числа ресурсов, а второй – это счетчик настоящего количества ресурсов. Для семафоров определены следующие правила работы:

1. Семафор переходит в сигнальное состояние, если значение счетчика ресурсов больше 0.
2. Семафор занят, если значение счетчика равно 0.
3. Не допускается установка отрицательного значения счетчика.
4. Счетчик не может иметь значение, большее максимального числа ресурсов.

Семафор создается вызовом функции:

```
HANDLE CreateSemaphore (
PSECURITY_ATTRIBUTES Attributes, // атрибуты защиты
LONG Initial, // количество ресурсов, доступных изначально
LONG Maximum, // максимальное количество ресурсов
```

PCTSTR Name); // Символьное имя объекта

Получить описатель существующего семафора можно с помощью функции:

HANDLE OpenSemaphore (
DWORD Access, // режим доступа
BOOL Inherit, // наследование
PCTSTR Name); // Символьное имя объекта

Поток может увеличить счетчик настоящего количества ресурсов, вызвав функцию:

BOOL ReleaseSemaphore (
HANDLE Semaphore, // описатель объекта-семафора
LONG ReleaseCount,
// насколько увеличить счетчик ресурсов (обычно 1)
PLONG PreviousCount);
// исходное значение счетчика (обычно передают NULL)

Процесс может использовать семафор, чтобы ограничить число окон, которые он создаст. Сначала, он использует функцию *CreateSemaphore*, чтобы создать семафор и определить начальное и максимальное значения счетчика:

```
HANDLE Semaphore; LONG Max = 12, PreviousCount;
// создание семафора с одинаковыми значениями счетчиков равными 12
Semaphore = CreateSemaphore(NULL, cMax, cMax, NULL);
// безымянный семафор
if (Semaphore == NULL)
{ // проверка ошибок
}
```

Прежде, чем любой поток создает новое окно, он вызывает функцию WaitForSingleObject, чтобы определить, разрешает ли текущий счетчик семафора создание дополнительных окон. Параметр блокировки времени функции ожидания установлен на 0:

```
DWORD WaitResult;
WaitResult = WaitForSingleObject(Semaphore, 0);
switch (WaitResult) {
// Семафор свободен
case WAIT_OBJECT_0:
// можно создать следующее окно
break;
// Семафор занят, время прошло
case WAIT_TIMEOUT:
// Не создавать следующее окно
break;
}
```

Когда поток закрывает окно, он вызывает функцию *ReleaseSemaphore*, чтобы увеличить счетчик семафора:

```
if (!ReleaseSemaphore(Semaphore, 1, NULL))
{ // Возникла ошибка
}
```

Объект ядра «мьютексы» гарантируют потокам взаимоисключающий доступ к единственному ресурсу. Отсюда и пошло название объекта. Поток, пытаясь получить доступ к критическим данным, выполнил соответствующий системный вызов. Мьютекс находится в сигнальном состоянии, в этом случае поток тут же

становится его владельцем, устанавливая его в несигнальное состояние, и входит в критическую секцию. После того, как поток выполнил работы с критическими данными, он отдает мьютекс, устанавливая его в сигнальное состояние. В этот момент мьютекс свободен и не принадлежит ни одному потоку. Если какой-либо поток ожидает его освобождения, то он становится следующим владельцем этого мьютекса, одновременно мьютекс переходит в несигнальное состояние. Таким образом, эти объекты должны содержать счетчик числа пользователей, счетчик рекурсии и идентификатор потока, а для мьютекса определены следующие правила:

1. Если идентификатор потока равен 0, мьютекс находится в сигнальном состоянии и не захвачен ни одним потоком.

2. Если идентификатор потока не равен 0, мьютекс захвачен одним потоком и находится в несигнальном состоянии.

3. Мьютексы могут нарушать правила, действующие в ОС.

Процесс может создать мьютекс вызовом функции:

HANDLE CreateMutex (
PSECURITY_ATTRIBUTES Attributes, // атрибуты защиты
BOOL InitialOwner, // начальное состояние мьютекса
PCTSTR Name); // Символьное имя объекта

Получить описатель существующего мьютекса можно с помощью функции:

HANDLE OpenMutex (
DWORD Access, // режим доступа
BOOL Inherit, // наследование
PCTSTR Name); // Символьное имя объекта

Параметр InitialOwner определяет начальное состояние мьютекса. Если передается FALSE, то мьютекс находится в сигнальном состоянии и не принадлежит ни одному потоку, причем идентификатор потока и счетчик рекурсии равны нулю. Если передается TRUE, то счетчик рекурсии становится равным 1, а идентификатор потока в мьютексе становится равным идентификатору вызвавшего потока, и теперь мьютекс находится в несигнальном состоянии. Поток получает доступ к ресурсу, вызывая какую-либо ожидающую функцию с передачей ей описатель мьютекса. Если она определяет, что мьютекс занят, то вызывающий поток переходит в состояние ожидания.

Это запоминается, и когда идентификатор потока обнуляется, в него записывается идентификатор ждущего потока, счетчику рекурсии присваивается при этом значение 1. После этого ожидающий поток может быть активизирован.

Надо отметить, что система обязательно проверит, не совпадает ли идентификатор потока у мьютекса с идентификатором потока, ожидающего мьютекса. В случае совпадения система выделит потоку процессорное время, хотя мьютекс еще занят. Счетчик рекурсии увеличивается на 1, когда поток захватывает мьютекс, и значение этого счетчика больше 1 только в том случае, если поток захватывает один и тот же объект несколько раз. Когда ожидание мьютекса завершается, поток получает монопольный доступ к ресурсу. Все остальные потоки переходят в состояние ожидания. По окончании работы с ресурсом, поток обязан освободить мьютекс вызовом функции

BOOL ReleaseMutex (HANDLE Mutex);

Данная функция уменьшает счетчик рекурсии на 1, и если мьютекс передавался во владение поток несколько раз, он должен вызвать *ReleaseMutex* такое же число раз, чтобы, в конце концов, обнулить счетчик рекурсии.

Когда это произойдет, идентификатор потока также станет равным 0, и мьютекс станет свободным. Система проверит, нет ли ожидающих потоков, и выберет один из них, чтобы передать тому мьютекс.

Как видно, этот объект обладает одним замечательным свойством, которого нет у остальных синхронизирующих объектов: мьютекс запоминает поток, которому он принадлежит. Если посторонний поток попытается вызвать *ReleaseMutex*, то эта функция просто вернет *FALSE*. Если же какой-то поток завершится, не успев освободить мьютекс, то считается, что произошел отказ от мьютекса, и система переведет его в сигнальное состояние.

```
HANDLE Thread1, Thread2;
HANDLE Mutex1;
int WinMain(...)
{
Mutex1 = CreateMutex (NULL, FALSE, “Mutex1”);
// Создается мьютекс
// Создаются два потока, причем пропущены все параметры,
// кроме функции потока
HANDLE Th1 = CreateThread (..., Function1,...);
HANDLE Th2 = CreateThread (..., Function2,...);
// далее можно выполнять любые действия
...
CloseHandle (Mutex1);
}
DWORD WINAPI Function1 (PVOID Parametr)
{
WaitForSingleObject (Mutex1, INFINITE);
...
ReleaseMutex (Mutex1);
return 0;
}
DWORD WINAPI Function2 (PVOID Parametr)
{
WaitForSingleObject (Mutex1, INFINITE);
...
ReleaseMutex (Mutex1);
return 0;
}
```

4.2 Содержание отчета

- постановка задачи;
- блок – схема алгоритма работы программы;
- результаты работы программы;
- выводы о проделанной работе;
- листинг программного кода.

4.3 Варианты индивидуальных заданий

Вариант №1

В пансионе отдыхают и предаются размышлениям 5 философов, пронумерованные от 1 до 5. В столовой расположен круглый стол, вокруг которого расставлены 5 стульев, также пронумерованные от 1 до 5. На столе находится одна большая тарелка со спагетти, которая пополняется бесконечно, также там расставлены 5 тарелок, в которые накладывается спагетти, и 5 вилок, назначение которых очевидно.

Для того чтобы пообедать, философ входит в столовую и садится на стул со своим номером. При этом есть философ сможет только в том случае, если свободны две вилки – справа и слева от его тарелки. При выполнении этого условия философ поднимает одновременно обе вилки и может поглощать пищу в течение какого-то заданного времени. В противном случае, философу приходится ждать освобождения обеих вилок.

Пообедав, философ кладет обе вилки на стол одновременно и уходит.

Описанный процесс происходит бесконечно.

Воспользоваться объектами синхронизации типа «ожидаемый таймер».

Вариант №2

В пансионе отдыхают и предаются размышлениям 5 философов, пронумерованные от 1 до 5. В столовой расположен круглый стол, вокруг которого расставлены 5 стульев, также пронумерованные от 1 до 5. На столе находится одна большая тарелка со спагетти, которая пополняется бесконечно, также там расставлены 5 тарелок, в которые накладывается спагетти, и 5 вилок, назначение которых очевидно.

Для того чтобы пообедать, философ входит в столовую и садится на стул со своим номером. При этом есть философ сможет только в том случае, если свободны две вилки – справа и слева от его тарелки. При выполнении этого условия философ поднимает одновременно обе вилки и может поглощать пищу в течение какого-то заданного времени. В противном случае, философу приходится ждать освобождения обеих вилок.

Пообедав, философ кладет обе вилки на стол одновременно и уходит.

Описанный процесс происходит бесконечно.

Воспользоваться объектами синхронизации типа «событие».

Вариант №3

В пансионе отдыхают и предаются размышлениям 5 философов, пронумерованные от 1 до 5. В столовой расположен круглый стол, вокруг которого расставлены 5 стульев, также пронумерованные от 1 до 5. На столе находится одна большая тарелка со спагетти, которая пополняется бесконечно, также там расставлены 5 тарелок, в которые накладывается спагетти, и 5 вилок, назначение которых очевидно.

Для того чтобы пообедать, философ входит в столовую и садится на стул со своим номером. При этом есть философ сможет только в том случае, если свободны две вилки – справа и слева от его тарелки. При выполнении этого условия философ поднимает одновременно обе вилки и может поглощать пищу в течение ка-

кого-то заданного времени. В противном случае, философу приходится ждать освобождения обеих вилок.

Пообедав, философ кладет обе вилки на стол одновременно и уходит.

Описанный процесс происходит бесконечно.

Воспользоваться объектами синхронизации типа «семафор».

Вариант №4

В пансионе отдыхают и предаются размышлениям 5 философов, пронумерованные от 1 до 5. В столовой расположен круглый стол, вокруг которого расставлены 5 стульев, также пронумерованные от 1 до 5. На столе находится одна большая тарелка со спагетти, которая пополняется бесконечно, также там расставлены 5 тарелок, в которые накладывается спагетти, и 5 вилок, назначение которых очевидно.

Для того чтобы пообедать, философ входит в столовую и садится на стул со своим номером. При этом есть философ сможет только в том случае, если свободны две вилки – справа и слева от его тарелки. При выполнении этого условия философ поднимает одновременно обе вилки и может поглощать пищу в течение какого-то заданного времени. В противном случае, философу приходится ждать освобождения обеих вилок.

Пообедав, философ кладет обе вилки на стол одновременно и уходит.

Описанный процесс происходит бесконечно.

Воспользоваться объектами синхронизации типа «мьютекс».

Вариант №5

В пансионе отдыхают и предаются размышлениям 5 философов, пронумерованные от 1 до 5. В столовой расположен круглый стол, вокруг которого расставлены 5 стульев, также пронумерованные от 1 до 5. На столе находится одна большая тарелка со спагетти, которая пополняется бесконечно, также там расставлены 5 тарелок, в которые накладывается спагетти, и 5 вилок, назначение которых очевидно.

Для того чтобы пообедать, философ входит в столовую и садится на стул со своим номером. При этом есть философ сможет только в том случае, если свободны две вилки – справа и слева от его тарелки. При выполнении этого условия философ поднимает одновременно обе вилки и может поглощать пищу в течение какого-то заданного времени. В противном случае, философу приходится ждать освобождения обеих вилок.

Пообедав, философ кладет обе вилки на стол одновременно и уходит.

Описанный процесс происходит бесконечно.

Воспользоваться объектами синхронизации типа «критическая секция».

Вариант №6

В парикмахерской расположено единственное кресло, на котором спит парикмахер, и несколько стульев для клиентов.

Когда клиент приходит в парикмахерскую, он будит парикмахера, садится в кресло. Стрижка производится в течение заданного времени. Если же кресло занято другим клиентом, то вновь прибывший клиент занимает любой свободный стул и ожидает своей очереди (клиенты обслуживаются в порядке очередности,

например, времени прибытия). Если все стулья заняты, то клиент поворачивается и уходит.

Когда обслужены все клиенты, парикмахер садится в кресло и снова засыпает.

Описанный процесс происходит бесконечно.

Воспользоваться объектами синхронизации типа «ожидаемый таймер».

Вариант №7

В парикмахерской расположено единственное кресло, на котором спит парикмахер, и несколько стульев для клиентов.

Когда клиент приходит в парикмахерскую, он будит парикмахера, садится в кресло. Стрижка производится в течение заданного времени. Если же кресло занято другим клиентом, то вновь прибывший клиент занимает любой свободный стул и ожидает своей очереди (клиенты обслуживаются в порядке очередности, например, времени прибытия). Если все стулья заняты, то клиент поворачивается и уходит.

Когда обслужены все клиенты, парикмахер садится в кресло и снова засыпает. Описанный процесс происходит бесконечно.

Воспользоваться объектами синхронизации типа «событие».

Вариант №8

В парикмахерской расположено единственное кресло, на котором спит парикмахер, и несколько стульев для клиентов.

Когда клиент приходит в парикмахерскую, он будит парикмахера, садится в кресло. Стрижка производится в течение заданного времени. Если же кресло занято другим клиентом, то вновь прибывший клиент занимает любой свободный стул и ожидает своей очереди (клиенты обслуживаются в порядке очередности, например, времени прибытия). Если все стулья заняты, то клиент поворачивается и уходит.

Когда обслужены все клиенты, парикмахер садится в кресло и снова засыпает.

Описанный процесс происходит бесконечно.

Воспользоваться объектами синхронизации типа «семафор».

Вариант №9

В парикмахерской расположено единственное кресло, на котором спит парикмахер, и несколько стульев для клиентов.

Когда клиент приходит в парикмахерскую, он будит парикмахера, садится в кресло. Стрижка производится в течение заданного времени. Если же кресло занято другим клиентом, то вновь прибывший клиент занимает любой свободный стул и ожидает своей очереди (клиенты обслуживаются в порядке очередности, например, времени прибытия). Если все стулья заняты, то клиент поворачивается и уходит.

Когда обслужены все клиенты, парикмахер садится в кресло и снова засыпает.

Описанный процесс происходит бесконечно.

Воспользоваться объектами синхронизации типа «мьютекс».

Вариант №10

В парикмахерской расположено единственное кресло, на котором спит парикмахер, и несколько стульев для клиентов.

Когда клиент приходит в парикмахерскую, он будит парикмахера, садится в кресло. Стрижка производится в течение заданного времени. Если же кресло занято другим клиентом, то вновь прибывший клиент занимает любой свободный стул и ожидает своей очереди (клиенты обслуживаются в порядке очередности, например, времени прибытия). Если все стулья заняты, то клиент поворачивается и уходит.

Когда обслужены все клиенты, парикмахер садится в кресло и снова засыпает.

Описанный процесс происходит бесконечно.

Воспользоваться объектами синхронизации типа «критическая секция».

Вариант №11

Рассмотрим взаимодействие двух потоков, один из которых пишет данные в буферный пул, а другой считывает их из пула. Буферный пул состоит из N буферов, каждый содержит одну запись. В общем случае поток-писатель и поток-читатель имеют разные скорости и обращаются к пулу с переменной интенсивностью. Для правильной работы поток-писатель приостанавливается, когда все буферы заняты, и переходит в активное состояние при наличии хотя бы одного свободного буфера. Поток-читатель приостанавливается, когда все буферы пусты, и активизируется, когда появляется, по крайней мере, одна запись.

Описанный процесс происходит бесконечно.

Воспользоваться объектами синхронизации типа «ожидаемый таймер».

Вариант №12

Рассмотрим взаимодействие двух потоков, один из которых пишет данные в буферный пул, а другой считывает их из пула. Буферный пул состоит из N буферов, каждый содержит одну запись. В общем случае поток-писатель и поток-читатель имеют разные скорости и обращаются к пулу с переменной интенсивностью. Для правильной работы поток-писатель приостанавливается, когда все буферы заняты, и переходит в активное состояние при наличии хотя бы одного свободного буфера. Поток-читатель приостанавливается, когда все буферы пусты, и активизируется, когда появляется, по крайней мере, одна запись.

Описанный процесс происходит бесконечно.

Воспользоваться объектами синхронизации типа «событие».

Вариант №13

Рассмотрим взаимодействие двух потоков, один из которых пишет данные в буферный пул, а другой считывает их из пула. Буферный пул состоит из N буферов, каждый содержит одну запись. В общем случае поток-писатель и поток-читатель имеют разные скорости и обращаются к пулу с переменной интенсивностью. Для правильной работы поток-писатель приостанавливается, когда все буферы заняты, и переходит в активное состояние при наличии хотя бы одного свободного буфера. Поток-читатель приостанавливается, когда все буферы пусты, и активизируется, когда появляется, по крайней мере, одна запись.

Описанный процесс происходит бесконечно.

Воспользоваться объектами синхронизации типа «семафор».

Вариант №14

Рассмотрим взаимодействие двух потоков, один из которых пишет данные в буферный пул, а другой считывает их из пула. Буферный пул состоит из N буферов, каждый содержит одну запись. В общем случае поток-писатель и поток-читатель имеют разные скорости и обращаются к пулу с переменной интенсивностью. Для правильной работы поток-писатель приостанавливается, когда все буферы заняты, и переходит в активное состояние при наличии хотя бы одного свободного буфера. Поток-читатель приостанавливается, когда все буферы пусты, и активизируется, когда появляется, по крайней мере, одна запись.

Описанный процесс происходит бесконечно.

Воспользоваться объектами синхронизации типа «мьютекс».

Вариант №15

Рассмотрим взаимодействие двух потоков, один из которых пишет данные в буферный пул, а другой считывает их из пула. Буферный пул состоит из N буферов, каждый содержит одну запись. В общем случае поток-писатель и поток-читатель имеют разные скорости и обращаются к пулу с переменной интенсивностью. Для правильной работы поток-писатель приостанавливается, когда все буферы заняты, и переходит в активное состояние при наличии хотя бы одного свободного буфера. Поток-читатель приостанавливается, когда все буферы пусты, и активизируется, когда появляется, по крайней мере, одна запись.

Описанный процесс происходит бесконечно.

Воспользоваться объектами синхронизации типа «критическая секция».

Вариант №16

В пансионе отдыхают и предаются размышлениям 5 философов, пронумерованные от 1 до 5. В столовой расположен круглый стол, вокруг которого расставлены 5 стульев, также пронумерованные от 1 до 5. На столе находится одна большая тарелка со спагетти, которая пополняется бесконечно, также там расставлены 5 тарелок, в которые накладывается спагетти, и 5 вилок, назначение которых очевидно.

Для того чтобы пообедать, философ входит в столовую и садится на любой стул. При этом есть философ сможет только в том случае, если свободны две вилки – справа и слева от его тарелки. При выполнении этого условия философ поднимает одновременно обе вилки и может поглощать пищу в течение какого-то заданного времени. В противном случае, философу приходится ждать освобождения обеих вилок.

Пообедав, философ кладет обе вилки на стол одновременно и уходит.

Описанный процесс происходит бесконечно.

Воспользоваться объектами синхронизации типа «ожидаемый таймер».

Вариант №17

В пансионе отдыхают и предаются размышлениям 5 философов, пронумерованные от 1 до 5. В столовой расположен круглый стол, вокруг которого расставлены 5 стульев, также пронумерованные от 1 до 5. На столе находится одна

большая тарелка со спагетти, которая пополняется бесконечно, также там расставлены 5 тарелок, в которые накладывается спагетти, и 5 вилок, назначение которых очевидно.

Для того чтобы пообедать, философ входит в столовую и садится на любой стул. При этом есть философ сможет только в том случае, если свободны две вилки – справа и слева от его тарелки. При выполнении этого условия философ поднимает одновременно обе вилки и может поглощать пищу в течение какого-то заданного времени. В противном случае, философу приходится ждать освобождения обеих вилок.

Пообедав, философ кладет обе вилки на стол одновременно и уходит.

Описанный процесс происходит бесконечно.

Воспользоваться объектами синхронизации типа «событие».

Вариант №18

В пансионе отдыхают и предаются размышлениям 5 философов, пронумерованные от 1 до 5. В столовой расположен круглый стол, вокруг которого расставлены 5 стульев, также пронумерованные от 1 до 5. На столе находится одна большая тарелка со спагетти, которая пополняется бесконечно, также там расставлены 5 тарелок, в которые накладывается спагетти, и 5 вилок, назначение которых очевидно.

Для того чтобы пообедать, философ входит в столовую и садится на любой стул. При этом есть философ сможет только в том случае, если свободны две вилки – справа и слева от его тарелки. При выполнении этого условия философ поднимает одновременно обе вилки и может поглощать пищу в течение какого-то заданного времени. В противном случае, философу приходится ждать освобождения обеих вилок.

Пообедав, философ кладет обе вилки на стол одновременно и уходит.

Описанный процесс происходит бесконечно.

Воспользоваться объектами синхронизации типа «семафор».

Вариант №19

В парикмахерской расположено единственное кресло, на котором спит парикмахер, и несколько стульев для клиентов, которые делятся на два класса – обычные и «приближенные». Сначала всегда обслуживаются «приближенные» клиенты, и только после этого парикмахер может работать с обычными клиентами.

Когда клиент приходит в парикмахерскую, он будит парикмахера, садится в кресло. Стрижка производится в течение заданного времени. Если же кресло занято другим клиентом, то вновь прибывший клиент занимает любой свободный стул и ожидает своей очереди. Далее клиенты обслуживаются в порядке приоритета и очередности (времени прибытия). Если все стулья заняты, то клиент поворачивается и уходит.

Когда обслужены все клиенты, парикмахер садится в кресло и снова засыпает.

Описанный процесс происходит бесконечно.

Воспользоваться объектами синхронизации типа «ожидаемый таймер».

Вариант №20

В пансионе отдыхают и предаются размышлениям 5 философов, пронумерованные от 1 до 5. В столовой расположен круглый стол, вокруг которого расставлены 5 стульев, также пронумерованные от 1 до 5. На столе находится одна большая тарелка со спагетти, которая пополняется бесконечно, также там расставлены 5 тарелок, в которые накладывается спагетти, и 5 вилок, назначение которых очевидно.

Для того чтобы пообедать, философ входит в столовую и садится на любой стул. При этом есть философ сможет только в том случае, если свободны две вилки – справа и слева от его тарелки. При выполнении этого условия философ поднимает одновременно обе вилки и может поглощать пищу в течение какого-то заданного времени. В противном случае, философу приходится ждать освобождения обеих вилок.

Пообедав, философ кладет обе вилки на стол одновременно и уходит.

Описанный процесс происходит бесконечно.

Воспользоваться объектами синхронизации типа «критическая секция».

4.4 Пример выполнения лабораторной работы

4.4.1 Задание

В пансионе отдыхают и предаются размышлениям 5 философов, пронумерованные от 1 до 5. В столовой расположен круглый стол, вокруг которого расставлены 5 стульев, также пронумерованные от 1 до 5. На столе находится одна большая тарелка со спагетти, которая пополняется бесконечно, также там расставлены 5 тарелок, в которые накладывается спагетти, и 5 вилок, назначение которых очевидно.

Для того чтобы пообедать, философ входит в столовую и садится на любой стул. При этом есть философ сможет только в том случае, если свободны две вилки – справа и слева от его тарелки. При выполнении этого условия философ поднимает одновременно обе вилки и может поглощать пищу в течение какого-то заданного времени. В противном случае, философу приходится ждать освобождения обеих вилок.

Пообедав, философ кладет обе вилки на стол одновременно и уходит.

Описанный процесс происходит бесконечно.

Воспользоваться объектами синхронизации типа «мьютекс».

4.4.2 Схема алгоритма

Блок – схемы алгоритмов программы представлены на рисунках 4.1, 4.2.

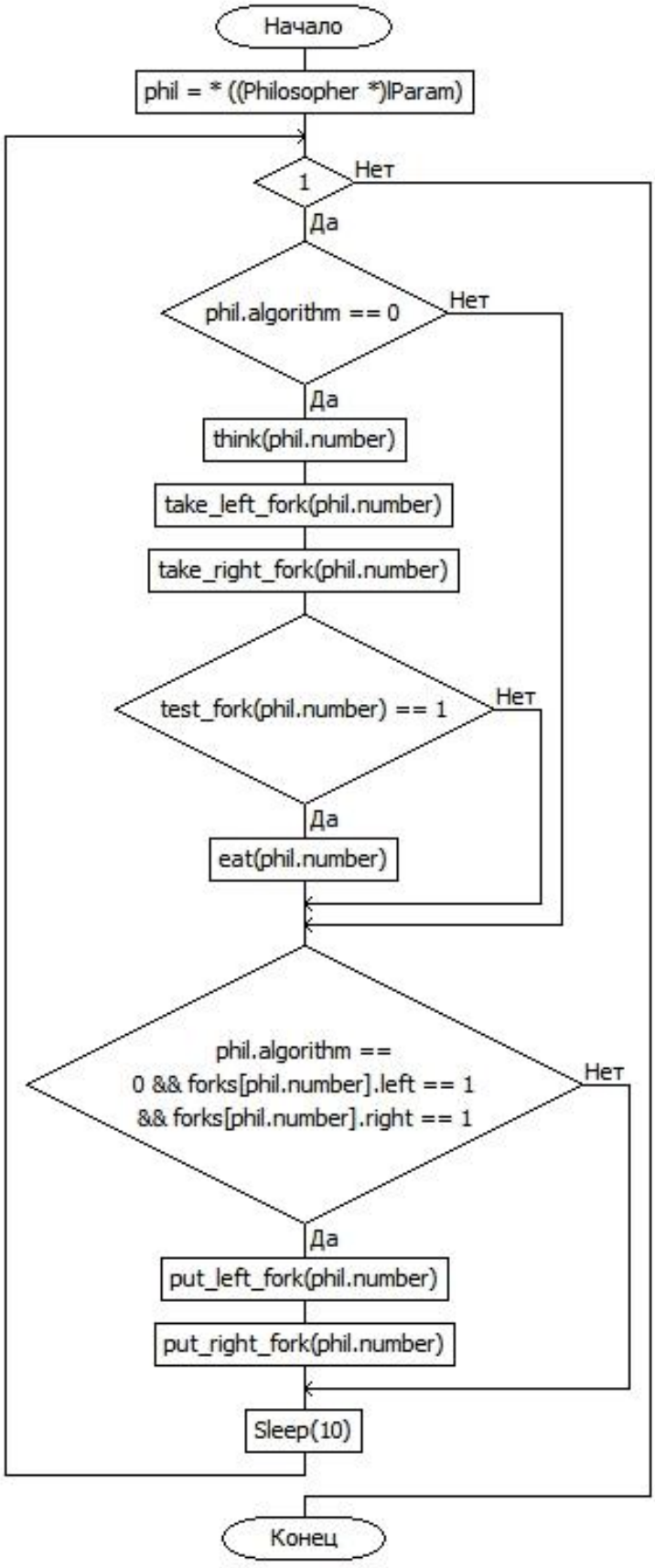

Рисунок 4.1 – Блок – схема модели философа

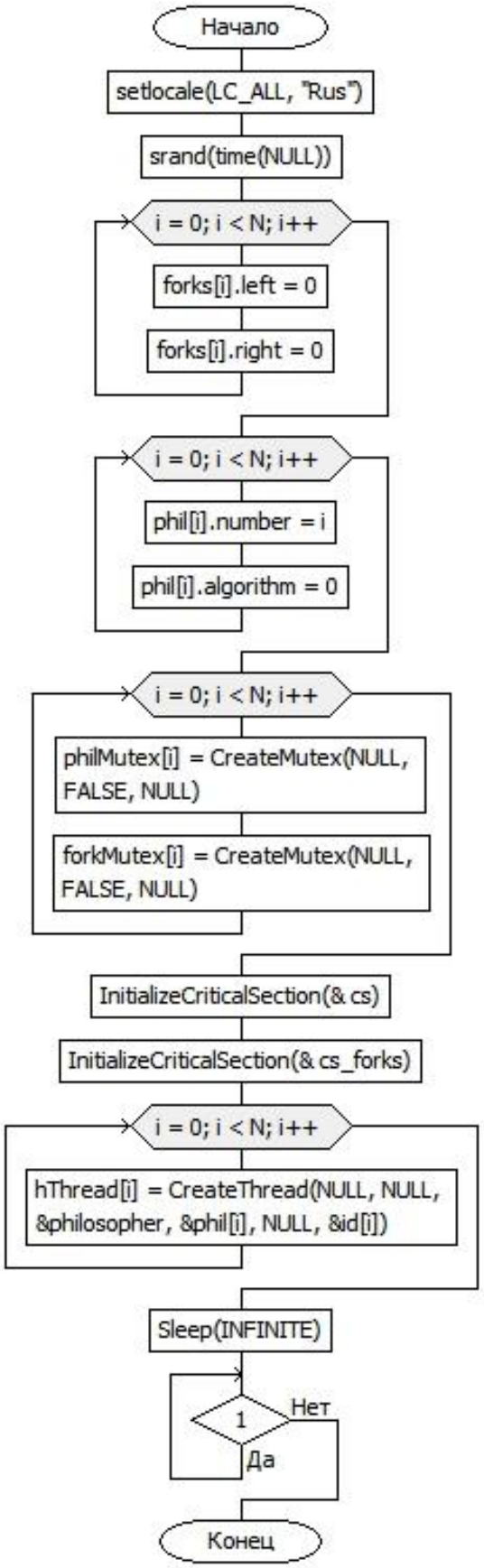

Рисунок 4.2 – Блок – схема главной функции

4.4.3 Код программы

```
#include <process.h>
#include <stdio.h>
#include<windows.h>
#include<iostream>
#include<time.h>
using namespace std;
#define N 5            //Число философов
#define LEFT (i-1)%N    //Левый сосед философа с номером i
#define RIGHT (i+1)%N   //Правый сосед философа сномером i
#define THINKING 0      //Философ размышляет
#define HUNGRY 1        //Философ получается получить вилки
#define EATING 2        //Философ ест
int state[N];          //Состояния каждого философа
struct Philosopher      //Описание философа: номер, алгоритм
{
        int number;
        int algorithm;
};
struct Forks           //Описывает количество вилок у философа
{
        int left;          //0-нет вилки
        int right;          //1-есть вилка
}forks[N];
CRITICAL_SECTION cs;       //Для критических секций: синхрон. процессов
CRITICAL_SECTION cs_forks;  //и синхр. вилок
HANDLE philMutex[N];    //Каждому философу по мьютексу
HANDLE forkMutex[N];    //и каждой вилке
void think(int i)       //Моделирует размышления философа
{
        EnterCriticalSection(&cs);    //Вход в критическую секцию
        cout << "Философ номер " << i << " размышляет" << endl;
        LeaveCriticalSection(&cs);    //Выход из критической секции
}
void eat(int i)         //Моделирует обед философа
{
        EnterCriticalSection(&cs);    //Вход в критическую секцию
        cout << "Философ номер " << i << " ест" << endl;
        LeaveCriticalSection(&cs);    //Выход из критической секции
}
void test(int i)    //Проверка возможности начать философом обед
{
        if (state[i] == HUNGRY && state[LEFT] != EATING && state[RIGHT] != EATING)
        {
                state[i] = EATING;
                ReleaseMutex(philMutex[i]);
        }
}
void take_forks(int i)  //Взятие вилок
{
        EnterCriticalSection(&cs_forks);            //Вход в критическую секцию
```

```
        state[i] = HUNGRY;                          //Фиксация наличия голодного философа
        test(i);                                    //Попытка получить две вилки
        LeaveCriticalSection(&cs_forks);            //Выход из критической секции
        WaitForSingleObject(philMutex[i], INFINITE);  //Блокировка если вилок не досталось
}
void put_forks(int i)   //Философ кладет вилки обратно
{
        EnterCriticalSection(&cs_forks);  //Вход в критическую секцию
        state[i] = THINKING;              //Философ перестал есть
        test(LEFT);                       //Проверить может ли есть сосед слева
        test(RIGHT);                      //Проверить может ли есть сосед справа
        LeaveCriticalSection(&cs_forks);  //Выход из критической секции
}
void take_left_fork(int i)  //Попытка взять левую вилку
{
        EnterCriticalSection(&cs);
        WaitForSingleObject(forkMutex[LEFT], INFINITE);   //Если вилка свободна
        forks[i].left = 1;                                //Берем ее
        LeaveCriticalSection(&cs);
}
void put_left_fork(int i)   //Кладем левую вилку
{
        WaitForSingleObject(forkMutex[LEFT], INFINITE);
        forks[i].left = 0;          //Кладем вилку
        ReleaseMutex(forkMutex[LEFT]);
}
void take_right_fork(int i) //Попытка взять правую вилку
{
        EnterCriticalSection(&cs);
        WaitForSingleObject(forkMutex[RIGHT], INFINITE);  //Если вилка свободна
        forks[i].right = 1;                               //Берем ее
        LeaveCriticalSection(&cs);
}
void put_right_fork(int i)  //Кладем правую вилку
{
        WaitForSingleObject(forkMutex[RIGHT], INFINITE);
        forks[i].right = 0;         //Кладем вилку
        ReleaseMutex(forkMutex[RIGHT]);
}
int test_fork(int i)    //Проверяем наличи у философа обеих вилок
{
        if (forks[i].left == 1 && forks[i].right == 1) //Если обе
        {
                ReleaseMutex(forkMutex[LEFT]);    //Разрешаем положить вилки
                ReleaseMutex(forkMutex[RIGHT]);   //
                return 1;                         //Едим
        }
        else
                return 0;
}
DWORD WINAPI philosopher (void* lParam)  //Собственно модель философа
```

```
{
	Philosopher phil = *((Philosopher*)lParam);  //Получаем модель философа
	while (1)    //Моделируем обед
	{   //Берем вилки
		if (phil.algorithm == 0)   //Берем первой левую вилку
		{
			think(phil.number); //Думаем
			take_left_fork(phil.number);    //Берем левую
			take_right_fork(phil.number);   //и правую вилки
			if (test_fork(phil.number) == 1)          //Если обе
				eat(phil.number);                 //то едим
		}
		//кладем вилки
		if (phil.algorithm == 0 && forks[phil.number].left == 1 && forks[phil.number].right
== 1)
		{
			put_left_fork(phil.number);     //Кладем левую
			put_right_fork(phil.number);    //и правую по первому алгоритму
		}
			Sleep(10);  //Даем время на переключение контекста
	}
}
int main()
{
	setlocale(LC_ALL, "Rus");
	srand(time(NULL));
	Philosopher phil[N];
	for (int i = 0; i < N; i++)  //Первоначально у философов нет вилок
	{
		forks[i].left = 0;
		forks[i].right = 0;
	}
	//cout << "Please input number of algorithm(0-left, 1-random, 2-right: " << endl;
	for (int i = 0; i < N; i++)  //Задаем алгоритмы взятия вилок
	{
		phil[i].number = i;
		//cout << "Philosopher number " << phil[i].number << ": ";
		phil[i].algorithm = 0;
	}
	for (int i = 0; i < N; i++)  //Создаем мьютексы
	{
		philMutex[i] = CreateMutex(NULL, FALSE, NULL);
		forkMutex[i] = CreateMutex(NULL, FALSE, NULL);
	}
	InitializeCriticalSection(&cs);       //Инициализируем
	InitializeCriticalSection(&cs_forks); //критические секции
	DWORD id[N];       //Идентификаторы потоков
	HANDLE hThread[N];  //Потоки
	for (int i = 0; i < N; i++)//Создаем потоки
	{
		hThread[i] = CreateThread(NULL, NULL, &philosopher, &phil[i], NULL, &id[i]);
```

```
}
Sleep(INFINITE);    //Чтобы потоки успели выполнится с корректными значениями
while (1);
}
```

4.4.4 Реализация программного кода

Реализация программного кода представлена на рисунке 4.3.

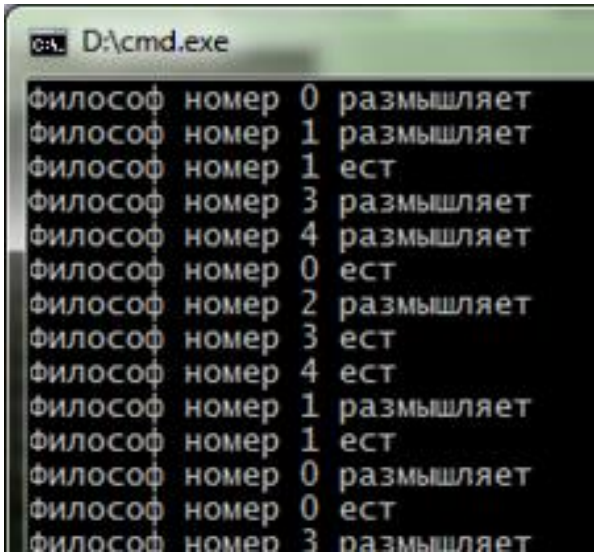

Рисунок 4.3 – Реализация программного кода

5 Лабораторная работа № 5. Использование механизма виртуальной памяти в ОС Windows

Цель: Изучение виртуальной памяти в операционной системе Windows.

Задачи:

1. Выполнить программирование поставленной задачи на языке высокого уровня с использованием механизма управления виртуальной памятью;
2. Разработать отчет.

5.1 Теоретическая часть

Виртуальное адресное пространство каждого процесса в ОС Windows организовано следующим образом. В момент своего создания, оно почти полностью пусто. Для использования какой-то его части, необходимо выделить в нем определенные регионы с помощью функции VirtualAlloc, эта операция называется резервированием. При этом ОС должна выравнивать начало региона в соответствии с так называемой гранулярностью выделения памяти, которая составляет на текущий момент времени 64 Кб. Также система должна учитывать, что размер региона должен быть кратен размеру страницы. Для процессоров Pentuim размер страницы составляет 4 Кб. Иными словами, если процесс попытается зарезервировать 10 Кб, то будет выделен регион размером 12 Кб. Когда регион становится не нужен, его необходимо освободить вызовом функции *VirtualFree*.

Чтобы использовать выделенный регион виртуального адресного пространства (ВАП), для него необходимо также выделить физическую память, спроецировав ее на регион. Эта операция называется передачей физической памяти и осуществляется с помощью функции *VirtualAlloc*. Для физической памяти определяют ее возврат, что выполняется с помощью функции *VitualFree*. Обе упомянутые функции будут описаны ниже.

Отдельным страницам физической памяти можно устанавливать атрибуты защиты:

PAGE_NOACCESS – Любая операция вызовет нарушение доступа;

PAGE_READONLY – Попытки записи или исполнения могут вызвать нарушение доступа;

PAGE_READWRITE – Попытки исполнить содержимое страницы вызывают нарушение доступа;

PAGE_EXECUTE – Чтение и запись могут вызвать нарушение доступа;

PAGE_EXECUTE_READ – Нарушение доступа при попытке записи;

PAGE_EXECUTE_READWRITE – Возможны любые операции;

PAGE_WRITECOPY – При исполнении этой страницы нарушение доступа, при записи процессу дается личная копия страницы;

PAGE_EXECUTE_WRITECOPY – Любые операции, при записи процессу дается личная копия страницы;

PAGE_NOCACHE – Отключает кэширование страницы (флаг);

PAGE_WRITECOMBINE – Объединение нескольких операций записи (флаг);

PAGE_GUARD – Используется для получения информации о записи на какую-либо страницу (флаг).

Узнать размеры страницы, гранулярность выделения памяти и другие параметры ОС можно с помощью функции *Win*32 *API:*

VOID GetSystemInfo (LPSYSTEM_INFO SysInfo).

В эту функцию в качестве параметра передается адрес структуры данных, описанной следующим образом:

```
typedef struct {
union {
DWORD Oem; // не используется
struct {
WORD ProcArchitecture; // типа архитектуры процессора
WORD Reserved; // не используется
};
};
DWORD PageSize; // размер страницы в байтах
LPVOID MinApplnAddress; // минимальный адрес доступного ВАП
LPVOID MaxApplnAddress; // максимальный адрес доступного ВАП
DWORD_PTR ActiveProcessors; // процессоры, выполняющие потоки
DWORD NumberOfProc; // количество установленных процессоров
DWORD ProcType; // тип процессора
DWORD Granularity; // гранулярность
WORD ProcLevel, ProcRevizion;
// дополнительные параметры
} SYSTEM_INFO, *LPSYSTEM_INFO.
```

Если есть необходимость узнать параметры, которые имеют отношение к памяти (а их всего четыре), достаточно выполнить всего два оператора:

```
SYSTEM_INFO SysInfo;
GetSystemInfo (&SysInfo).
```

После этого можно спокойно просматривать содержимое полей структуры *SysInfo*, где и будут находиться искомые системные параметры.

Следующая функция позволяет отслеживать состояние памяти на текущий момент времени, однако она имеет довольно странное название:

VOID GlobalMemoryStatus (LPMEMORY_STATUS MemStat).

Как видно, ей необходимо передать адрес некоторой структуры, которая описана следующим образом:

```
typedef struct {
DWORD Length; // размер структуры в байтах
DWORD MemLoad;
// Занятость подсистемы управления памятью (0 - 100)
SIZE_T TotalPhysMem; // объем физической памяти
SIZE_T AvailablePhysMem; // объем свободной физической памяти
SIZE_T TotalPageFile; // максимальный размер файла подкачки
SIZE_T AvailablePageFile;
// размер свободного места в файле подкачки
SIZE_T TotalVirtual; // максимальный размер ВАП процесса
SIZE_T AvailableVirtual; // размер доступного ВАП процесса
```

*} MEMORY_STATUS, *LPMEMORY_STATUS.*

Если нужно получить какие-то параметры памяти, потребуется еще несколько операторов, но предварительно потребуется установить длину всей структуры:

```
MEMORY_STATUS MemStat;
MemStat.Length = {sizeof (MemStat)};
GlobalMemoryStatus (&SysInfo).
```

В дальнейшем можно смело пользоваться значениями остальных полей структуры *MemStat*.

В ОС Windows есть функция, которая позволяет получать информацию о конкретном участке памяти в пределах ВАП процесса:

```
DWORD VirtualQuery (
LPCVOID Address, // адрес участка памяти
PMEMORY_BASIC_INFORMATION MemBase,
// адрес структуры с информацией о памяти
DWORD Length); // размер структуры с информацией о памяти
```

Этой функции требуется адрес структуры типа *MEMORY_BASIC_INFORMATION*, описанной как:

```
typedef struct {
PVOID Base;
// Address, округленный до адреса, кратного размеру страницы
PVOID AllocBase;
// Базовый адрес региона, в который входит адрес Address
DWORD AllocProtection; // атрибут защиты для региона
SIZE_T RegionSize;
// размер страниц, имеющих одинаковые атрибуты
DWORD State; // состояние всех смежных страниц
DWORD Protection; // атрибуты защиты всех смежных страниц
DWORD Type; // тип физической памяти всех смежных страниц
} MEMORY_BASIC_INFORMATION, *PMEMORY_BASIC_INFORMATION.
```

В том случае, если понадобится вывести информацию о регионах ВАП текущего процесса, можно это сделать следующим образом, однако весьма «грубо»:

```
// сначала необходимо получить размер страницы для данной системы
SYSTEM_INFO SysInfo;
GetSystemInfo (&SysInfo);
PVOID BaseAddr = NULL;
MEMORY_BASIC_INFORMATION MemBase;
for (;;) {
if(VirtalQuery(BaseAddr,&MemBase,sizeof(MemBase)) !=
sizeof(MemBase)) break;
printf ("%Lp\t", MemBase.AllocBase); // Базовый адрес
printf ("%d\t", MemBase.RegionSize / SysInfo.PageSize);
// Страниц на регион
switch (MemBase.State) { // Состояние региона
case MEM_FREE: printf ("Свободный\t"); break;
case MEM_RESERVE: printf ("Зарезервирован, память не передана\t"); break;
case MEM_COMMIT: printf ("Зарезервирован,
память передана\t"); break;
}
```

```
switch (MemBase.Protection) { // Атрибуты защиты региона
case PAGE_NOACCESS: printf (“----\t”); break;
case PAGE_READONLY: printf (“R---\t”); break;
case PAGE_READWRITE: printf (“RW--\t”); break;
case PAGE_EXECUTE: printf (“--E-\t”); break;
case PAGE_EXECUTE_READ: printf (“R-E-\t”); break;
case PAGE_EXECUTE_ READWRITE: printf (“RWE-\t”); break;
case PAGE_EXECUTE_ WRITECOPY: printf (“RWEC\t”); break;}
switch (MemBase.Type) { // Тип региона
case MEM_FREE: printf (“Свободный”); break;
case MEM_PRIVATE: printf (“Закрытый”); break;
case MEM_IMAGE: printf (“Образ файла”); break;
case MEM_MAPPED: printf (“Отображаемый файл”); break;
default: printf (“Неизвестно”);}
printf (“\n”);
BaseAddr = MemBase.BaseAddress+ MemBase.RegionSize;}
```

Виртуальная память очень удобна для работы с большими массивами данных. Для малых по размеру объектов больше подходят так называемые «кучи». Если есть надобность в обмене данными между процессами, то ОС Windows предлагает еще один механизм – отображаемые на память файлы.

О них более подробно будет рассказано там, где речь пойдет о подсистеме управления файлами. Ниже будут описаны функции для управления кучами. Функции, имеющие дело с виртуальной памятью, позволяют резервировать регион, отдавать ему физическую память и устанавливать параметры защиты.

Для резервирования предназначена функция:

```
PVOID VirtualAlloc (
PVOID Address, // адрес, где система должна резервировать память
SIZE_T Size, // размер региона, который надо зарезервировать
DWORD AllocType,
//тип резервирования (резервировать или передать физ. память)
DWORD Protect);   // атрибут защиты (PAGE_*).
```

Функция возвращает *NULL*, если не удалось выделить память для региона. Для резервирования достаточно при вызове указать тип *MEM_RESERVE*. Если теперь надо передать ему физическую память, нужно еще раз вызвать *VirtualAlloc* с уже знакомым флагом доступа *MEM_COMMIT*. Можно выполнить обе операции одновременно:

```
PVOID Region = VirtualAlloc (NULL, 25 * 1024,
MEM_RESERVE | MEM_COMMIT, PAGE_EXECUTE_READWRITE).
```

Здесь содержится запрос на выделение региона размером 25 Кб и передачи ему физической памяти. Поскольку первый параметр функции равен *NULL*, ОС попытается найти подходящее место, просматривая все ВАП.

Регион и переданная ему память получат одинаковый атрибут защиты *PAGE_EXECUTE_READWRITE*.

Для освобождения зарезервированного региона или возврата физической памяти можно пользоваться функцией:

```
BOOL VirtualFree (
PVOID Address,  // адрес, где система зарезервировала память
SIZE_T Size,    // размер региона, который был зарезервирован
```

DWORD FreeType); //тип освобождения

Для освобождения региона нужно вызвать *VirtualFree* с его адресом, в *Size* указать 0, поскольку система знает размер региона, а в *FreeType* – *MEM_RELEASE*. Если же возникла необходимость просто вернуть часть физической памяти, то *Address* должен адресовать первую возвращаемую страницу, *Size* – количество освобождаемых байтов, а *FreeType* – идентификатор *MEM_COMMIT*.

PVOID Region;
VirtualFree (Region, 0, MEM_RELEASE);

Атрибуты защиты страницы памяти можно вызовом функции:

PVOID VirtualProtect (
PVOID Address, // адрес, где система зарезервировала память
SIZE_T Size, // число байтов, для которых меняется защита
DWORD NewProtection, // новые атрибуты защиты
PDWORD OldProtecttion) //старые атрибуты защиты.

Для изменения атрибутов защиты у тех страниц, которые принадлежат разным регионам функцию *VirtualProtect* использовать нельзя.

*PVOID Region = VirtualAlloc (NULL, 25 * 1024,*
MEM_RESERVE | MEM_COMMIT, PAGE_EXECUTE_READWRITE);
DWORD OldProt;
*VirtualProtect (Region, 3 * 1024, PAGE_READONLY, &OldProt).*

Еще один механизм управления памятью – «куча» — это также регион зарезервированного адресного пространства. Этому региону большая часть физической памяти не передается. В зависимости оттого, что делает программа со своими данными, специализированный «менеджер куч» передает региону физическую память или возвращает страницы.

В ОС Windows при инициализации процесса в его ВАП создается стандартная куча, размер которой 1 Мб. Описатель этой кучи можно при помощи вызова функции:

HANDLE GetProcessHeap ().

Можно создать и дополнительные кучи, для этого нужна функция:

HANDLE HeapCreate (
DWORD Options, // способ выполнения операций над кучей
SIZE_T StartSize, // начальное количество байтов в куче
SIZE_T MaxSize) // максимальное количество байтов в куче.

Если в *Options* указан 0, то к куче могут одновременно обращаться несколько потоков. Атрибут *HEAP_NO_SERIALIZE* позволяет потоку осуществлять доступ к куче монопольно, однако пользоваться таким способом не рекомендуется. Другой флаг *HEAP_GENERATE_EXCEPTIONS* при ошибке обращения к куче дает системе возможность уведомлять программы об этом. Если в третьем параметре *MaxSize* указать значение больше 0, то будет создана нерасширяемая куча именно такого размера, в противном случае система резервирует регион и может расширять его до максимального размера. Данная функция в случае успеха возвращает описатель вновь созданной кучи.

Для выделения блока памяти из кучи необходимо вызвать функцию:

PVOID HeapAlloc (
HANDLE Heap, // описатель кучи

DWORD Flags, // флаги выделения памяти
SIZE_T Bytes) // количество выделяемых байтов.

Если в качестве флага указан *HEAP_ZERO_MEMORY*, функция возвратит блок памяти, заполненный нулями. Назначение *HEAP_GENERATE_EXCEPTIONS* и *HEAP_NO_SERIALIZE* очевидно.

Иногда требуется изменить размер выделенного блока памяти: уменьшить или увеличить. Для этого вызывается функция:

PVOID HeapReAlloc (
HANDLE Heap, // описатель кучи
DWORD Flags, // флаги изменения памяти
PVOID Memory, // текущий адрес блока
SIZE_T Bytes) // новый размер в байтах.

Возможны четыре значения флага: *HEAP_NO_SERIALIZE*, *HEAP_GENERATE_EXCEPTIONS*, *HEAP_ZERO_MEMORY* и *HEAP_REALLOC_IN_PLACE_ONLY*. Два первых из них знакомы по *HeapAlloc*. При использовании *HEAP_ZERO_MEMORY* нулями заполняются только дополнительный байты. Флаг *HEAP_REALLOC_IN_PLACE_ONLY* говорит о том, что блок перемещать внутри кучи нельзя. Возвращает эта функция адрес нового блока либо *NULL*, если не удалось изменить размер.

После того, как выделен блок памяти, можно узнать его размер:

SIZE_T HeapSize (
HANDLE Heap, // описатель кучи
DWORD Flags, // флаги изменения памяти (0 или HEAP_NO_SERIALIZE)
PVOID Memory) // текущий адрес блока.

После того, как блок перестал быть нужным, его освобождают функцией:

BOOL HeapFree (
HANDLE Heap, // описатель кучи
DWORD Flags, // флаги изменения памяти (0 или HEAP_NO_SERIALIZE)
PVOID Memory) // текущий адрес блока.

В случае успеха эта функция возвращает *TRUE*. Аналогично работает функция *HeapDestroy*, которая освобождает все блоки памяти внутри кучи и возвратит системе регион, занятый кучей.

BOOL HeapDestroy (HANDLE Heap) // описатель кучи.

Небольшой фрагмент кода демонстрирует использование некоторых из этих функций:

```
// получение описателя стандартной кучи активного процесса
HANDLE SysHeap = GetProcessHeap ();
UINT MAX_ALLOCATIONS = 15;
// максимальное количество выделений памяти
UINT NumOfAllocations = 0;
// текущее количество выделений памяти
for (;;) {
// выделение из кучи 2 Кб памяти
if (HeapAlloc (SysHeap, 0, 2 * 1024) == NULL) break;
else ++ NumOfAllocations;
// условие прерывания цикла
if (NumOfAllocation == MAX_ALLOCATIONS) break;
}
```

```
// вывод соответствующих сообщений в зависимости от ситуации
if (NumOfAllocations == 0)
printf (“Память из кучи не выделялась.”);
else printf (“Память из кучи выделялась %d раз.”,
NumOfAllocations).
```

В ОС Windows NT/2000/XP есть пара функций *Win*32 *API*, которые позволяют блокировать (или зафиксировать) и разблокировать страницу в оперативной памяти. Функция *VirtualLock* позволяет предотвратить запись памяти на диск.

```
BOOL VirtualLock (
LPVOID Address,   // адрес начала памяти
SIZE_T Size);    // количество байтов
```

Если фиксация больше не нужна, то ее можно убрать функцией:

```
BOOL VirtualUnlock (
LPVOID Address,   // адрес начала памяти.
SIZE_T Size);     // количество байтов
```

Обе функции возвращают ненулевое значение в случае успеха.

Следующий фрагмент демонстрирует их использование:

```
int MEMSIZE = 4096;
PVOID Mem = NULL;
int num;
Mem = VirtualAlloc(NULL,4 * 1024, MEM_RESERVE,
PAGE_EXECUTE_READWRITE);
if (Mem != NULL) {
if (VirtualLock (Mem, MEMSIZE)) printf ("Привязка\n");
else printf (“Ошибка привязки”);
scanf (“%d”, &num);
if (VirtualUnlock (Mem, MEMSIZE))
printf ("Привязка снята\n");
else printf ("Ошибка снятия привязки\n");
if (VirtualFree (Mem, 0, MEM_RELEASE))
printf ("Память освобождена\n");
else printf ("Память не освобождена\n");
}
else printf ("Память не выделена\n");
}
```

5.2 Содержание отчета

- постановка задачи;
- блок – схема алгоритма работы программы;
- результаты работы программы;
- выводы о проделанной работе;
- листинг программного кода.

5.3 Варианты индивидуальных заданий

Вариант №1

Разработать программу, которая демонстрирует управление структурами данных типа «стек», элементы которого занимают 10 кб. Операции, выполняемые над стеком:

- проверить, стек пуст/не пуст;
- втолкнуть элемент;
- вытолкнуть элемент;
- просмотреть вершину стека;
- обменять значения двух верхних элементов стека.

Воспользоваться механизмом управления разделами виртуальной памятью.

Вариант №2

Разработать программу, которая демонстрирует управление структурами данных типа «стек», элементы которого занимают 15 кб. Операции, выполняемые над стеком:

- проверить, стек пуст/не пуст;
- втолкнуть элемент;
- вытолкнуть элемент;
- просмотреть вершину стека;
- продублировать вершину стека.

Воспользоваться механизмом управления разделами виртуальной памятью.

Вариант №3

Разработать программу, которая демонстрирует управление структурами данных типа «линейный однонаправленный список» (*L*1-*list*), элементы которого занимают 12 кб. Операции, выполняемые над списком (при этом определяется указатель списка и элемент списка за указателем):

- проверить, список пуст/не пуст;
- установить указатель в начало списка;
- добавить элемент за указателем;
- удалить элемент за указателем;
- просмотреть элемент за указателем;
- переместить указатель вправо;
- обменять значения конца списка и элемента за указателем.

Воспользоваться механизмом управления разделами виртуальной памятью.

Вариант №4

Разработать программу, которая демонстрирует управление структурами данных типа «очередь», элементы которого занимают 10 кб. Операции, выполняемые над очередью:

- проверить, очередь пуста/не пуста;
- добавить элемент в хвост очереди;
- удалить элемент из головы очереди;
- просмотреть голову очереди;
- обменять значения из головы и хвоста очереди.

Воспользоваться механизмом управления разделами виртуальной памятью.

Вариант №5

Разработать программу, которая демонстрирует управление структурами данных типа «очередь», элементы которого занимают 15 кб. Операции, выполняемые над очередью:

- проверить, очередь пуста/не пуста;
- добавить элемент в хвост очереди;
- удалить элемент из головы очереди;
- просмотреть голову очереди;
- продублировать хвост очереди.

Воспользоваться механизмом управления разделами виртуальной памятью.

Вариант №6

Разработать программу, которая демонстрирует управление структурами данных типа «очередь», элементы которого занимают 12 кб. Операции, выполняемые над очередью:

- проверить, очередь пуста/не пуста;
- добавить элемент в хвост очереди;
- удалить элемент из головы очереди;
- просмотреть голову очереди;
- продублировать голову очереди.

Воспользоваться механизмом управления разделами виртуальной памятью.

Вариант №7

Разработать программу, которая демонстрирует управление структурами данных типа «дек» (очередь с двумя концами), элементы которого занимают 10 кб. Операции, выполняемые над деком:

- проверить, дек пуст/не пуст;
- добавить элемент в левый конец дека;
- добавить элемент в правый конец дека;
- удалить элемент слева;
- удалить элемент справа;
- просмотреть элемент слева;
- просмотреть элемент справа.

Воспользоваться механизмом управления разделами виртуальной памятью.

Вариант №8

Разработать программу, которая демонстрирует управление структурами данных типа «ограниченный слева дек» (очередь с двумя концами), элементы которого занимают 15 кб. Операции, выполняемые над деком:

- проверить, дек пуст/не пуст;
- добавить элемент в левый конец дека;
- добавить элемент в правый конец дека;
- удалить элемент справа;
- просмотреть элемент справа.

Воспользоваться механизмом управления разделами виртуальной памятью.

Вариант №9

Разработать программу, которая демонстрирует управление структурами данных типа «ограниченный справа дек» (очередь с двумя концами), элементы которого занимают 12 кб. Операции, выполняемые над деком:

- проверить, дек пуст/не пуст;
- добавить элемент в левый конец дека;
- добавить элемент в правый конец дека;
- удалить элемент слева;
- просмотреть элемент слева.

Воспользоваться механизмом управления разделами виртуальной памятью.

Вариант №10

Разработать программу, которая демонстрирует управление структурами данных типа «линейный однонаправленный список» (*L*1-*list*), элементы которого занимают 10 кб. Операции, выполняемые над списком (при этом определяется указатель списка и элемент списка за указателем):

- проверить, список пуст/не пуст;
- установить указатель в начало списка;
- добавить элемент за указателем;
- удалить элемент за указателем;
- просмотреть элемент за указателем;
- переместить указатель вправо.

Воспользоваться механизмом управления разделами виртуальной памятью.

Вариант №11

Разработать программу, которая демонстрирует управление структурами данных типа «линейный двунаправленный список» (*L*2-*list*), элементы которого занимают 15 кб. Операции, выполняемые над списком (при этом определяется указатель списка, элемент списка за указателем и элемент до указателя):

- проверить, список пуст/не пуст;
- установить указатель в начало списка;
- установить указатель в конец списка;
- добавить элемент за указателем;
- добавить элемент до указателя;
- удалить элемент за указателем;
- удалить элемент до указателя;
- просмотреть элемент за указателем;
- просмотреть элемент до указателя;
- переместить указатель вправо;
- переместить указатель влево.

Воспользоваться механизмом управления разделами виртуальной памятью.

Вариант №12

Разработать программу, которая демонстрирует управление структурами данных типа «динамический вектор» (одномерный массив), элементы которого занимают 12 кб. Операции, выполняемые над вектором (при этом определяются начало и конец вектора, индекс элемента вектора):

- проверить, вектор пуст/не пуст;
- прочитать элемент с указанным индексом;
- изменить значение элемента с указанным индексом;
- добавить элемент в конец вектора;
- опустошить вектор.

Воспользоваться механизмом управления разделами виртуальной памятью.

Вариант №13

Разработать программу, которая демонстрирует управление структурами данных типа «последовательность» (файл в оперативной памяти), элементы которого занимают 10 кб. Операции, выполняемые над последовательностью (при этом определяются указатель на текущий элемент, начало и конец последовательности):

- проверить, последовательность пуста/не пуста;
- установить указатель в начало последовательности;
- прочитать элемент последовательности;
- добавить элемент в конец последовательности;
- опустошить последовательность.

Воспользоваться механизмом управления разделами виртуальной памятью.

Вариант №14

Разработать программу, которая демонстрирует управление структурами данных типа «кольцевой однонаправленный список», элементы которого занимают 15 кб. Операции, выполняемые над списком (при этом определяется указатель списка, который может автоматически перемещаться на начало списка, если достигнут его конец, и элемент списка за указателем):

- проверить, список пуст/не пуст;
- установить указатель в начало списка;
- добавить элемент за указателем;
- удалить элемент за указателем;
- просмотреть элемент за указателем;
- переместить указатель вправо.

Воспользоваться механизмом управления разделами виртуальной памятью.

Вариант №15

Разработать программу, которая демонстрирует управление структурами данных типа «кольцевой двунаправленный список», элементы которого занимают 12 кб. Операции, выполняемые над списком (при этом определяется указатель списка, который может автоматически перемещаться на начало списка, если до-

стигнут его конец, и в конец списка – в случае достижения его начала, а также элемент списка за указателем и элемент до указателя):

- проверить, список пуст/не пуст;
- установить указатель в начало списка;
- установить указатель в конец списка;
- добавить элемент за указателем;
- добавить элемент до указателя;
- удалить элемент за указателем;
- удалить элемент до указателя;
- просмотреть элемент за указателем;
- просмотреть элемент до указателя;
- переместить указатель вправо;
- переместить указатель влево.

Воспользоваться механизмом управления разделами виртуальной памятью.

Вариант №16

Разработать программу, которая демонстрирует управление структурами данных типа «стек», элементы которого занимают 12 кб. Операции, выполняемые над стеком:

- проверить, стек пуст/не пуст;
- втолкнуть элемент;
- вытолкнуть элемент;
- просмотреть вершину стека;
- обменять значения двух верхних элементов стека.

Воспользоваться механизмом управления разделами виртуальной памятью.

Вариант №17

Разработать программу, которая демонстрирует управление структурами данных типа «сбалансированное двоичное упорядоченное дерево», элементы которого занимают 15 кб. Операции, выполняемые над деревом (при этом определяется один узел дерева, являющийся его корнем, все значения в узлах дерева разные, длины путей от корня до всех узлов-листьев отличаются не более, чем на единицу):

- проверить, дерево пусто/не пусто;
- добавить элемент в дерево;
- удалить элемент из дерева;
- найти элемент с заданным значением;
- опустошить дерево.

Воспользоваться механизмом управления разделами виртуальной памятью.

Вариант №18

Разработать программу, которая демонстрирует управление структурами данных типа «неупорядоченное 2-3-дерево», элементы которого занимают 12 кб. Операции, выполняемые над деревом (при этом определяется один узел дерева,

являющийся его корнем, все значения в узлах дерева разные, количество потомков каждого узла – не более трех):

- проверить, дерево пусто/не пусто;
- добавить элемент в дерево;
- удалить элемент из дерева;
- найти элемент с заданным значением;
- опустошить дерево.

Воспользоваться механизмом управления разделами виртуальной памятью.

Вариант №19

Разработать программу, которая демонстрирует управление структурами данных типа «линейный двунаправленный список» (*L2-list*), элементы которого занимают 10 кб. Операции, выполняемые над списком (при этом определяется указатель списка, элемент списка за указателем и элемент до указателя):

- проверить, список пуст/не пуст;
- установить указатель в начало списка;
- установить указатель в конец списка;
- обменять значения элементов за указателем и до указателя, если это возможно;
- добавить элемент за указателем;
- добавить элемент до указателя;
- удалить элемент за указателем;
- удалить элемент до указателя;
- просмотреть элемент за указателем;
- просмотреть элемент до указателя;
- переместить указатель вправо;
- переместить указатель влево.

Воспользоваться механизмом управления разделами виртуальной памятью.

Вариант №20

Разработать программу, которая демонстрирует управление структурами данных типа «линейный однонаправленный список» (*L1-list*), элементы которого занимают 15 кб. Операции, выполняемые над списком (при этом определяется указатель списка и элемент списка за указателем):

- проверить, список пуст/не пуст;
- установить указатель в начало списка;
- добавить элемент за указателем;
- удалить элемент за указателем;
- просмотреть элемент за указателем;
- переместить указатель вправо;
- обменять значения начала списка и элемента за указателем.

Воспользоваться механизмом управления разделами виртуальной памятью.

Разработать программу, которая демонстрирует управление структурами данных типа «стек», элементы которого занимают 12 кб. Операции, выполняемые над стеком:

- проверить, стек пуст/не пуст;
- втолкнуть элемент;
- вытолкнуть элемент;
- просмотреть вершину стека;
- обменять значения двух верхних элементов стека.

5.4 Пример выполнения лабораторной работы

5.4.1 Задание

Разработать программу, которая демонстрирует управление структурами данных типа «двоичное упорядоченное дерево», элементы которого занимают 10 кб. Операции, выполняемые над деревом (при этом определяется один узел дерева, являющийся его корнем, все значения в узлах дерева разные):

- проверить, дерево пусто/не пусто;
- добавить элемент в дерево;
- удалить элемент из дерева;
- найти элемент с заданным значением;
- опустошить дерево.

Воспользоваться механизмом управления разделами виртуальной памятью.

5.4.2 Схема алгоритма.

Схемы алгоритмов добавления и удаления узла дерева представлены на рисунках 5.1, 5.2.

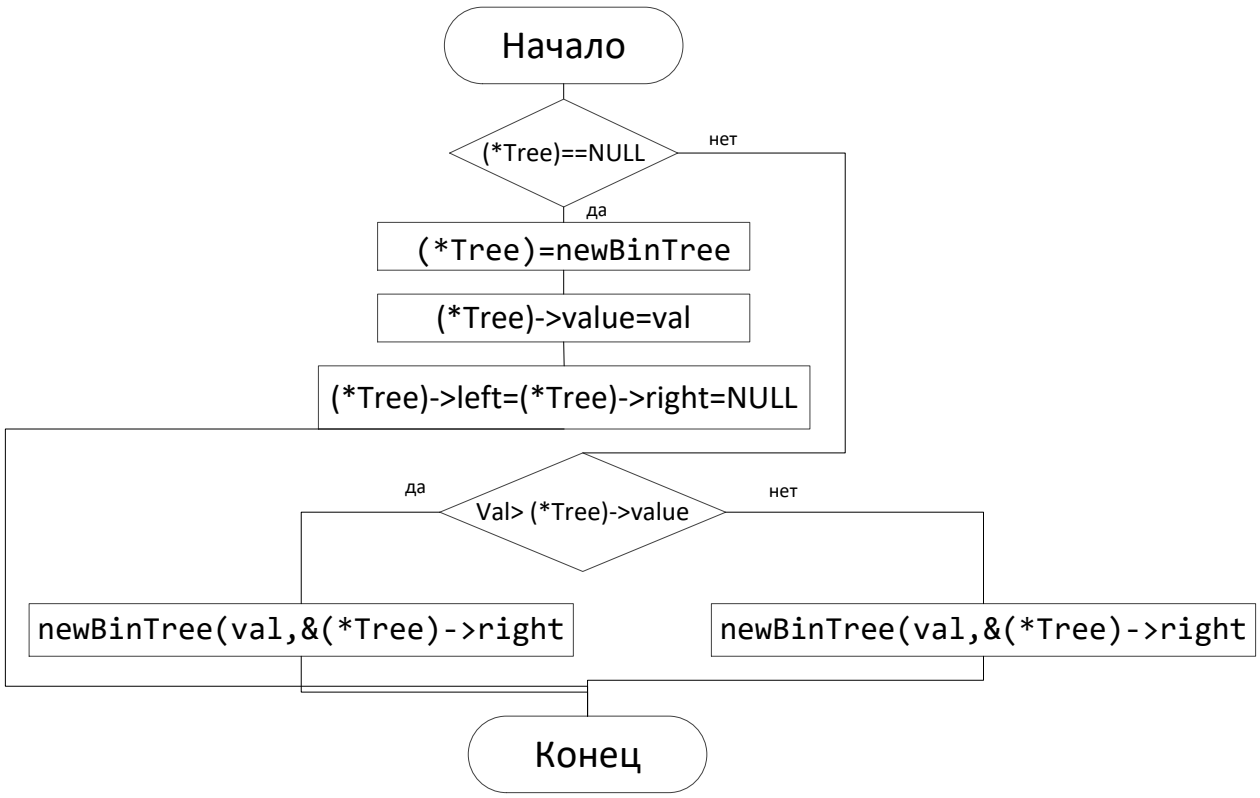

Рисунок 5.1 – Блок – схема функции добавления узла дерева

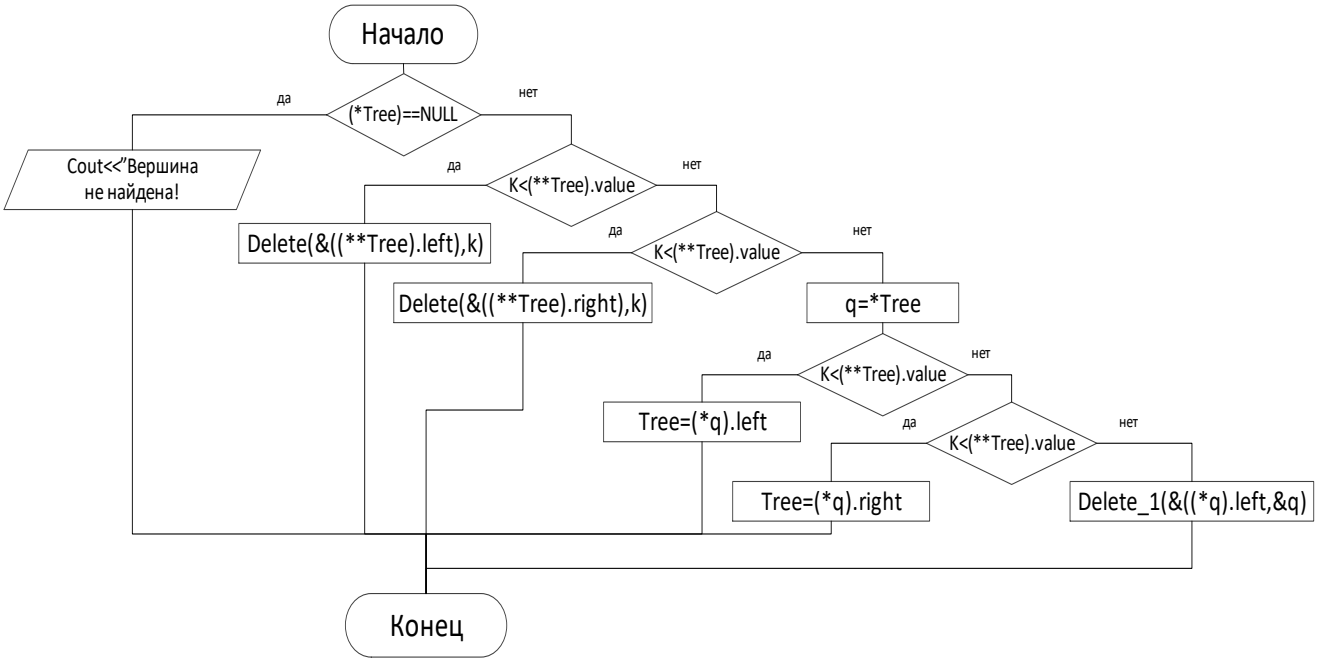

Рисунок 5.2 – Блок – схема функции удаления узла дерева

5.4.3 Код программы

```
#include <iostream>
#include <queue>
using namespace std;
struct Node //Структура дерева
{
	int field;
	struct Node* left;
	struct Node* right;
};
Node* Add_Node(Node* tree, int value) //Добавление узла в дерево
{
	if (tree == NULL) //Если дерева нет, то формируем ко-рень
	{
		tree = new Node; //Выделяем память под узел
		tree->field = value; //Заносим данные
		tree->left = NULL; //Ветви инициализируем пустотой
		tree->right = NULL;
	}
	else
	{
		if (value < tree->field) //Условие добавления ле-вого потомка
		{
			tree->left = Add_Node(tree->left, value);
		}
		else //Условие добавления правого потомка
		{
			tree->right = Add_Node(tree->right, value);
		}
	}
	return(tree);
```

```
}
Node* Search_Min(Node* tree)
{
	if (tree->left == NULL)
	{
		return tree;
	}
	return Search_Min(tree->left);
}
Node* Delete_Node(Node* tree, int value)
{
	if (tree == NULL) //Если дерево пустое, то возвращаем NULL
	{
		cout << "Вершина не найдена!\n";
		return tree;
	}
	else { cout << "Вершина удалена!\n"; }
	if (value < tree->field) //Если удаляемый элемент в левом поддереве
	{
		tree->left = Delete_Node(tree->left, value); //Используем рекурсию для левого под-
дерева
	}
	else if (value > tree->field) //Если удаляемый элемент в правом поддереве
	{
		tree->right = Delete_Node(tree->right, value); //Используем рекурсию для правого
поддерева
	}
	else if (tree->left != NULL && tree->right != NULL) //Если удаляемый элемент в корне и
имеет два дочерних узла
	{
		tree->field = Search_Min(tree->right)->field; //Ищем самый левый узел првого под-
дерева и заменяем им корневой
		tree->right = Delete_Node(tree->right, tree->field); //Удаляем самый левый узел пра-
вого поддерева
	}
	else if (tree->left != NULL) //Если удаляемый узел имеет левый дочерний узел
	{
		tree = tree->left; //Заменяем его потомком
	}
	else //Если удаляемый узел имеет правый дочерний узел
	{
		tree = tree->right; //Заменяем его потомком
	}
	return tree; //Удаляем найденный элемент
}
Node* Delete_Tree(Node* tree) //Удаление дерева
{
	if (tree != NULL)
	{
		Delete_Tree(tree->left);
		Delete_Tree(tree->right);
```

```
                delete tree;
        }
        return NULL;
}
//Вывод дерева
Node* Print(Node* tree, int l)
{
        int i;
        if (tree != NULL)
        {
                Print((
                        tree->right), l + 1);
                for (i = 1; i <= l; i++) cout << "   ";
                cout << (tree->field) << endl;
                Print((tree->left), l + 1);
        }
        return tree;
}
Node* Find(Node* tree, int key)
{
        if (tree == NULL) //Если дерево пустое
        {
                return NULL; //Элемент не найден
        }
        if (tree->field == key) //Если ключи совпали, то эле-мент найден
        {
                return tree;
        }
        if (key < tree->field) //Если необходимый ключ <= дан-ному, то идем влево
        {
                if (tree->left != NULL) //Если есть левый дочерний элемент, то используем рекур-
сию
                {
                        return Find(tree->left, key);
                }
                else
                {
                        return NULL; //Элемент не найден
                }
        }
        else
        {
                if (tree->right != NULL) //Если есть правый дочер-ний элемент, то используем ре-
курсию
                {
                        return Find(tree->right, key);
                }
                else
                {
                        return NULL; //Элемент не найден
                }
```

```
	}
}
int main()
{
	setlocale(LC_ALL, "Rus");
	int n = 0, check = 0, sw = -1, field = 0, count = 0;
	struct Node* root = NULL;
	//cout << "Введите количество узлов дерева: ";
	//cin >> count;
	cout << "Введите значения узлов: " << endl;
//	for (int i = 0; i < count; i++)
//	{
//		cin >> n;
//		root = Add_Node(root, n);
	//}
	for (int i = 0; i < 640; i++) {
		n = rand();
		root = Add_Node(root, n);
	}
	while (check == 0)
	{
		cout << "Выберите действие: \n1 - Просмотр дерева \n2 - Добавление узла\n3 - Удаление узла\n4 - Удаление дерева\n5 - Поиск элемента дерева\n6 - Проверка на пустоту\n0 - Выход из программы\n";
		cin >> sw;
		switch (sw)
		{
		case 1: //Добавление узла
			// Просмотр дерева
			cout << "Бинарное дерево поиска:" << endl;
			Print(root, 0);
			break;
		case 2: //Добавление узла
			cout << "Введите значение: ";
			cin >> field;
			Add_Node(root, field);
			cout << "Узел добавлен!" << endl;
			break;
		case 3: //Удаление узла
			if (root != NULL)
			{
				cout << "Введите значение: ";
				cin >> field;
				Delete_Node(root, field);
			}
			else
			{
				cout << "Дерева не существует!";
			}
			cout << endl;
			break;
```

```
		case 4: //Удаление дерева
			if (root != NULL)
			{
				Delete_Tree(root);
				root = NULL;
				cout << "Дерево удалено!" << endl;
			}
			else
			{
				cout << "Дерева не существует!";
			}
			cout << endl;
			break;
		case 5: //Поиск элемента дерева
			if (root != NULL)
			{
				cout << "Введите значение: ";
				cin >> field;
				if (Find(root, field) != NULL)
				{
					cout << "Элемент найден";
				}
				else
				{
					cout << "Такого элемента не существует!";
				}
			}
			else
			{
				cout << "Дерева не существует!";
			}
			cout << endl;
			break;
		case 0: //Выход из программы
			system("pause");
			return 0;
			break;
		default:
			cout << "Ошибка ввода!";
			cout << endl;
			break;
		}
	}
}
```

5.4.4 Реализация программного кода

Отдельные фрагменты реализации программного кода представлены на рисунках 5.3 – 5.7.

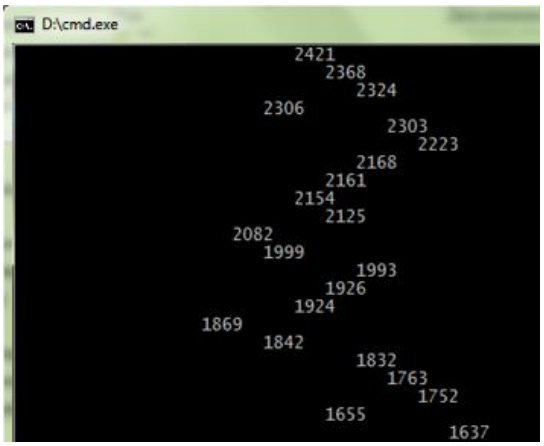

Рисунок 5.3 – Просмотр дерева

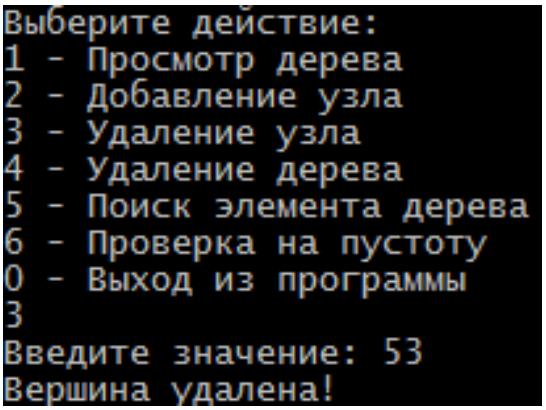

Рисунок 5.4 – Удаление узла

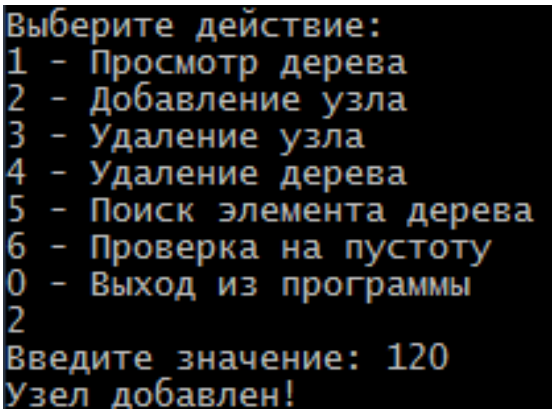

Рисунок 5.5 – Добавление узла

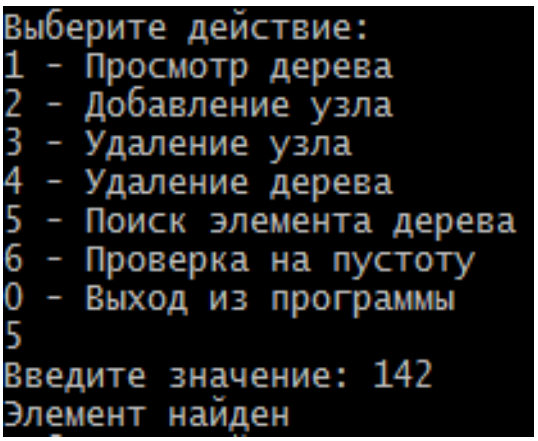

Рисунок 5.6 – Поиск узла

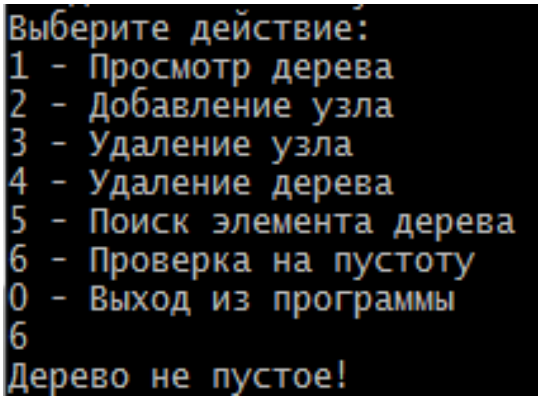

Рисунок 5.7 – Проверка на пустоту

6 Лабораторная работа № 6. Использование механизма обмена сообщениями для управления окнами

Цель: Получение практических навыков в программировании задач механизма обмена сообщениями для управления окнами в операционной системе Windows.

Задачи:

1. Выполнить программирование поставленной задачи на языке высокого уровня с использованием главных и дочерних окон, различных элементов управления, стилей и обмена сообщениями;
2. Разработать отчет.

6.1 Теоретическая часть

Программы в ОС Windows управляются сообщениями. Все действия пользователей перехватываются системой и преобразуются в сообщения, направляемые программе, которая владеет окнами, к которым обращены действия пользователя. У каждой программы, состоящей хотя бы из одного потока, есть несколько собственных очередей сообщений: очередь асинхронных сообщений, очередь синхронных сообщений, очередь ответных сообщений и очередь виртуального ввода, куда и направляются все сообщения, касающиеся действий с окнами. В каждой программе есть главный цикл, который состоит из получения следующего сообщения и его обработки с помощью внутренней процедуры, соответствующей данному типу сообщений. Иногда некоторые сообщения вызывает эти процедуры, минуя очереди сообщений.

В ОС Windows регистрацию класса окна производит функция:

*ATOM RegisterClass (CONST WNDCLASS *WndClass); // указатель на структуру с данными класса.*

Ниже перечислены стили окон:

- *WS_BORDER* – создание окна с рамкой;
- *WS_CAPTION* – создание окна с заголовком (невозможно использовать одновременно со стилем *WS_DLGFRAME*);
- *WS_CHILDWINDOW* – создание дочернего окна (невозможно использовать одновременно со стилем *WS_POPUP*);
- *WS_DISABLED* – создание недоступного окна;
- *WS_DLGFRAME* – создание окна с двойной рамкой, без заголовка;
- *WS_GROUP* – объединение элементов управления в группы;
- *WS_HSCROLL* – создание окна с горизонтальной полосой прокрутки;
- *WS_MAXIMIZE* – создание окна максимального размера;
- *WS_MAXIMIZEBOX* – создание окна с кнопкой развертывания окна;
- *WS_ICONIC* – создание первоначально свернутого окна (используется только со стилем *WS_OWERLAPPED*);
- *WS_MINIMIZEBOX* – создание окна с кнопкой свертывания;

- *WS_OVERLAPPED* – создание перекрывающегося окна (имеет заголовок и *WS_TILED* рамку);
- *WS_OVERLAPPEDWINDOW* – создание перекрывающегося окна со стилями *WS_OVERLAPPED*, *WS_CAPTION*, *WS_SYSMENU*, WS_THICKFRAME, *WS_MINIMIZEBOX*, *WS_MAXIMIZEBOX;*
- *WS_POPUP* – создание всплывающего окна (невозможно использовать совместно со стилем *WS_CHILD*);
- *WS_POPUPWINDOW* – создание всплывающего окна, имеющего стили *WS_BORDER*, *WS_POPUP*, *WS_SYSMENU;*
- *WS_SYSMENU* – создание окна с кнопкой системного меню (можно используется только с окнами, имеющими строку заголовка);
- *WS_TABSTOP* – определение элементов управления, переход к которым может быть выполнен по клавише *TAB;*
- *WS_THICKFRAME* – создание окна с рамкой, используемой для изменения *WS_SIZEBOX* размера окна;
- *WS_VISIBLE* – создание первоначально неотображаемого окна;
- *WS_VSCROLL* – создание окна с вертикальной полосой прокрутки.

Функция *CreateWindow* используется программой для создания окна:

```
HWND CreateWindow (
LPCTSTR ClassName, // указатель на зарегистрированное имя класса
LPCTSTR WindowName, // указатель на имя окна
DWORD dwStyle, // стиль окна
int x, // горизонтальная позиция окна
int y, // вертикальная позиция окна
int Width, // ширина окна
int Height, // высота окна
HWND WndParent, // дескриптор родительского или окна владельца
HMENU Menu, // дескриптор меню или идентификатор дочернего окна
HANDLE Instance, // дескриптор экземпляра программы
LPVOID Param) // указатель на данные создания окна.
```

Следующие предопределенные классы элементов управления могут быть определены в параметре *ClassName*:

BUTTON (Кнопка);

COMBOBOX (Комбинированное окно);

EDIT (окно редактирования);

LISTBOX (окно со списком);

MDICLIENT;

SCROLLBAR (линейка прокрутки);

STATIC (статический элемент).

Стили кнопок (в классе *BUTTON*), которые могут быть определены в параметре *dwStyle*:

BS_STATE – Создает кнопку, которая является такой же, как окошко для флажка, за исключением того, что поле окна может стать недоступным так же, как это делается при установке флажка ("галочки") проверки (*checked*) или при отмене его. Используйте недоступное состояние, чтобы показать, что состояние окошка для флажка не определено.

BS_AUTOSTATE – создание кнопки;

BS_AUTOCHECKBOX – создание кнопки, которая также является окошком для флажка, за исключением того, что состояние установки флажка проверки автоматически переключается между установленным и не установленным параметром, каждый раз, когда пользователь выбирает эту кнопку.

BS_AUTORADIOBUTTON – создает кнопку, которая то же, что и "радио" кнопка, за исключением того, что, когда пользователь выбирает её, Windows автоматически устанавливает состояние кнопки в режим контроля флажка, отметив ее "галочкой" и автоматически устанавливает проверку состояния для всех других кнопок в той же самой группе без проверки флажка.

BS_CHECKBOX – создает маленькое, пустое окошко для флажка с текстом. По умолчанию, текст отображается справа от окошка. Чтобы отображать текст слева от окошка, объедините этот флажок со стилем *BS_LEFTTEXT* (или с эквивалентным стилем *BS_RIGHTBUTTON*).

BS_DEFPUSHBUTTON – создает командную кнопку, которая ведет себя подобно кнопке стиля *BS_PUSHBUTTON* и к тому же имеет жирную черную рамку. Если кнопка находится в диалоговом окне, пользователь может выбрать кнопку, нажав клавишу *ENTER*, даже тогда, когда кнопка не имеет фокуса ввода. Этот стиль полезен для предоставления пользователю возможности быстро выбрать наиболее подходящую (заданную по умолчанию) опцию.

BS_GROUPBOX – создает прямоугольник, в котором могут быть сгруппированы другие элементы управления. Любой текст, связанный с этим стилем, отображается в верхнем левом угле прямоугольника.

BS_LEFTTEXT – помещает текст слева от "радио" кнопки или окошечка-переключателя, когда объединен со стилем переключателя или "радио" кнопкой. Тот же самое, что и стиль BS_RIGHTBUTTON.

BS_OWNERDRAW – создает кнопку, представляемую владельцем. Окно владельца принимает сообщение *WM_MEASUREITEM*, когда кнопка создана и сообщение WM_DRAWITEM, когда внешний вид кнопки изменился.
Не объединяйте стиль *BS_OWNERDRAW* с любыми другими стилями кнопки.

BS_PUSHBUTTON – создает командную кнопку, которая отправляет сообщение *WM_COMMAND* окну владельца, когда пользователь выбирает эту кнопку.

BS_RADIOBUTTON – создает маленький кружок с текстом. По умолчанию, текст отображается справа от кружка. Чтобы отображать текст слева от кружка, объедините этот флажок со стилем *BS_LEFTTEXT* (или его эквивалентом - стилем *BS_RIGHTBUTTON*). Используйте "радио" кнопки для групп связанного, но взаимоисключающего выбора.

BS_USERBUTTON – устаревшая, но предусматривающая совместимость с 16-разрядными версиями Windows. Базирующиеся на Win32 прикладные программы должны использовать BS_OWNERDRAW взамен этого параметра.

BS_BITMAP – определяет, что кнопка отображает точечный рисунок.

BS_BOTTOM – помещает текст внизу прямоугольника кнопки.

BS_CENTER – выравнивает текст горизонтально по центру в прямоугольнике кнопки.

BS_ICON – определяет, что кнопка отображается как значок.

BS_LEFT – выравнивает слева текст в прямоугольнике кнопки. Однако, если кнопка - окошечко-переключатель или "радио" кнопка, которые не имеет стиля *BS_RIGHTBUTTON*, текст остается выровненным справа от переключателя или радиокнопки.

BS_MULTILINE – переносит по словам текст кнопки в дополнительные строки, если текстовая строка слишком длинна, чтобы поместиться в одной строке в прямоугольнике кнопки.

BS_NOTIFY – дает возможность кнопке послать уведомительные сообщения *BN_DBLCLK*, *BN_KILLFOCUS* и *BN_SETFOCUS* в её родительское окно. Обратите внимание, что кнопки посылают уведомительное сообщение *BN_CLICKED* независимо от того, имеет ли она этот стиль.

BS_PUSHLIKE – создает кнопку (типа переключателя, переключателя с тремя состояниями или "радио" кнопки) имеющую вид и действующую подобно командной кнопке.

BS_RIGHT – выровненный справа текст в прямоугольнике кнопки. Однако, если кнопка – окно для флажка или "радио" кнопка, которая не имеет стиля *BS_RIGHTBUTTON*, текст выровнен по правому краю справа от окошка для флажка или "радио" кнопки.

BS_RIGHTBUTTON – устанавливает кружок "радио" кнопки или квадрат окошка для флажка справа от прямоугольника кнопки. Тот же самый стиль, что и *BS_LEFTTEXT*.

BS_TEXT – определяет, что кнопка отображает текст.

BS_TOP – размешает текст вверху прямоугольника кнопки.

BS_VCENTER – размещает текст в середине (вертикально) прямоугольника кнопки.

Ниже перечислены стили комбинированного окна (в классе *COMBOBOX*), которые могут быть определены в параметре *dwStyle*:

CBS_AUTOHSCROLL – Автоматически прокручивает текст в поле редактирования текста вправо, когда пользователь вводит с клавиатуры символ в конце строки. Если этот стиль не установлен, принимается только текст, который помещается внутри прямоугольной границы поля.

CBS_DISABLENOSCROLL – В окне со списком показывает вертикальную линейку прокрутки заблокированной, когда поле окна содержит недостаточно элементов для прокрутки. Без этого стиля, линейка прокрутки скрыта, если окно со списком содержит недостаточно элементов.

CBS_DROPDOWN – Подобен *CBS_SIMPLE*, за исключением того, что окно со списком не отображается, пока пользователь не выберет значок рядом с полем редактирования текста.

CBS_DROPDOWNLIST – Подобен *CBS_DROPDOWN*, за исключением того, что поле редактирования текста заменено статическим текстовым элементом, который отображает текущий выбор в окне со списком.

CBS_HASSTRINGS – Определяет, что представляемое владельцем комбинированное окно содержит элементы, состоящие из строк. Комбинированное окно

поддерживает память и адрес для строк, так что прикладная программа может использовать сообщение *CB_GETLBTEXT*, чтобы восстановить текст для отдельного элемента.

CBS_LOWERCASE – Преобразовывает в нижний регистр любые символы верхнего регистра, введенные в поле редактирования текста комбинированного окна.

CBS_NOINTEGRALHEIGHT – определяет, что размер комбинированного окна – это точный размер, определенный прикладной программой, когда она создала комбинированное окно. Обычно, *Windows* устанавливает размеры комбинированного окна так, чтобы оно не отображало элементы частично.

CBS_OEMCONVERT – преобразует текст, введенный в поле редактирования текста комбинированного окна. Текст преобразуется из набора символов *Windows* в набор символов *OEM*, а затем обратно в набор *Windows*. Это гарантирует соответствующее символьное преобразование, когда прикладная программа вызывает функцию *CharToOem*, чтобы преобразовать строку *Windows* в комбинированном окне в символы *OEM*. Этот стиль наиболее полезен для комбинированных окон, которые содержат имена файлов и применяются только в комбинированных окнах, созданных со стилем *CBS_SIMPLE* или *CBS_DROPDOWN*.

CBS_OWNERDRAWFIXED – определяет, что владелец окна со списком ответственен за прорисовку его содержания и что элементы в окне со списком все равной высоты. Окно владельца принимает сообщение *WM_MEASUREITEM*, когда комбинированное окно создано, а сообщение *WM_DRAWITEM*, когда внешний вид комбинированного окна изменился.

CBS_OWNERDRAWVARIABLE – определяет, что владелец окна со списком ответственен за прорисовку его содержания и что элементы в окне со списком являются переменными по высоте. Окно владельца принимает сообщение *WM_MEASUREITEM* для каждого элемента комбинированного окна, когда Вы создаете комбинированное окно; окно владельца принимает сообщение *WM_DRAWITEM* тогда, когда изменился внешний вид комбинированного окна.

CBS_SIMPLE – всегда отображать окно со списком. Текущий выбор в окне со списком отображается в поле редактирования текста.

CBS_SORT – автоматически сортирует строки, введенные в окно со списком.

CBS_UPPERCASE – преобразовывает любые символы нижнего регистра в символы верхнего регистра, введенные в поле редактирования текста комбинированного окна.

Нижеперечисленные стили поля редактирования текста (в классе *EDIT*) могут быть определены в параметре *dwStyle*:

ES_AUTOHSCROLL – автоматически прокручивает текст вправо на 10 символов, когда пользователь напечатает символ в конце строчки. Когда пользователь нажимает клавишу *ENTER*, управление прокручивает весь текст обратно, чтобы установить нуль.

ES_AUTOVSCROLL – автоматически перемещает текст вверх на одну страницу, когда пользователь нажимает клавишу *ENTER* на последней строчке.

ES_CENTER – Выравнивает по центру текст в много строковом поле редактирования текста.

ES_LEFT – Выравнивание текста слева.

ES_LOWERCASE – Преобразовывает все символы в нижний регистр, поскольку они печатаются внутри поля редактирования текста.

ES_MULTILINE – Обозначает много строковое окно редактирования текста.

ES_NOHIDESEL – Отрицает заданное по умолчанию поведение для поля редактирования текста. Заданное по умолчанию поведение скрывает выбор, когда элемент управления теряет фокус ввода и инвертирует выбор, когда панель управления принимает фокус ввода. Если Вы определяете *ES_NOHIDESEL*, выбранный текст инвертируется, даже если панель управления не имеет фокуса.

ES_NUMBER – Позволяет ввести в поле редактирования только цифры.

ES_OEMCONVERT – Преобразует текст, введенный в окно редактирования. Текст преобразуется из набора символов Windows - в набор символов *OEM*, а затем обратно - в набор Windows. Это гарантирует соответствующее символьное преобразование, когда из прикладной программы вызывается функция *CharToOem*, чтобы преобразовать строку Windows в окне редактирования в символы *OEM*. Этот стиль наиболее полезен для окон редактирования текста, которые содержат имена файлов.

ES_PASSWORD – Отображает звездочку (*) вместо каждого символа, введенного с клавиатуры в окно редактирования. Вы можете использовать сообщение *EM_SETPASSWORDCHAR*, чтобы заменить ею символ, который отображается.

ES_READONLY – Не допускает пользователя к вводу или редактированию текста в окне редактирования.

ES_RIGHT – Выравнивает по правому краю текст в многострочном окне редактирования.

ES_UPPERCASE – Преобразует все символы в символы верхнего регистра, когда они вводятся в окно редактирования.

Следующие стили элемента управления окна со списком (в классе *LISTBOX*) могут быть определены в параметре *dwStyle*:

− *LBS_DISABLENOSCROLL* – Показывает заблокированную вертикальную линейку прокрутки в окне со списком, когда поле окна не содержит достаточно элементов для прокрутки. Если Вы не определяете этот стиль, линейка прокрутки скрыта, когда окно со списком не содержит достаточно элементов.

− *LBS_EXTENDEDSEL* – Позволяет многочисленным элементам быть выбранными, при помощи использования клавиши *SHIFT* и мыши или специальной комбинации клавиш.

− *LBS_HASSTRINGS* – Определяет, что окно со списком содержит элементы, состоящие из строк. Окно со списком сохраняет память и адреса строк, так что прикладная программа может использовать сообщение *LB_GETTEXT*, чтобы восстановить текст для отдельного элемента. По умолчанию, все окна со списком за исключением окон со списком, предоставленных владельцем, имеют этот стиль.

Вы можете создать предоставляемое владельцем окно со списком с ним или без этого стиля.

− *LBS_MULTICOLUMN* – Определяет много столбцовое окно со списком, которое прокручивается горизонтально. Сообщение *LB_SETCOLUMNWIDTH* устанавливает ширину столбцов.

− *LBS_MULTIPLESEL* – Включает или выключает выбор последовательности символов каждый раз, когда пользователь одним или двойным щелчком мыши активизирует строку символов в окне со списком. Пользователь может выбрать любое число строк.

− *LBS_NODATA* – Определяет "отсутствие данных" в окне со списком. Этот стиль определяется тогда, когда число элементов в окне со списком может превысить одну тысячу. Окно со списком с "отсутствующими данными" должно иметь также стиль *LBS_OWNERDRAWFIXED*, но не должно иметь стилей *LBS_SORT* или *LBS_HASSTRINGS*. Окно со списком с "отсутствующими данными" имеет сходство с окном со списком предоставляемым владельцем за исключением того, что оно не содержит ни строковых или растровых данных для элемента. Команды "добавить", "вставить" или "удалить" элемент всегда игнорируют любые передаваемые элементы данных; запрос на поиск строки внутри окна всегда терпит неудачу. Windows посылает сообщение *WM_DRAWITEM* окну владельцу, когда элемент должен быть прорисован. Элемент ID (*itemID*) член структуры *DRAWITEMSTRUCT*, переданный с сообщением *WM_DRAWITEM*, определяет номер строки элемента, который будет прорисован. Окно списка с "отсутствующими данными" не посылает сообщение *WM_DELETEITEM*.

− *LBS_NOINTEGRALHEIGHT* – Определяет, что размер окна со списком соответствует размеру, определенному прикладной программой, когда она создавала окно со списком. Обычно, Windows устанавливает величину окна со списком так, чтобы оно не отображало элементы частично.

− *LBS_NOREDRAW* – Определяет, что вид окна со списком не модифицируется, когда производятся изменения. Вы можете в любое время изменить этот стиль, посылая сообщение *WM_SETREDRAW*.

− *LBS_NOSEL* – Определяет, что окно со списком содержит элементы, которые могут просматриваться, но не выбираться.

− *LBS_NOTIFY* – Сообщает родительскому окну о входящем сообщении всякий раз, когда пользователь щелкает мышью или дважды щелкает по строке в окне списка.

− *LBS_OWNERDRAWFIXED* – Определяет, что владелец окна со списком ответственен за прорисовку его содержания и что элементы в окне со списком появляются одинаковой высоты. Окно владельца принимает сообщение *WM_MEASUREITEM*, когда окно со списком создано, а сообщение *WM_DRAWITEM*, когда внешний вид окна изменился.

− *LBS_OWNERDRAWVARIABLE* – Определяет, что владелец окна со списком ответственен за прорисовку его содержания и что элементы в окне со списком появляются переменными по высоте. Окно владельца принимает сообщение

WM_MEASUREITEM для каждого элемента в окне со списком, когда оно создано, а сообщение *WM_DRAWITEM*, когда внешний вид окна изменился.

− *LBS_SORT* – Сортирует строки в окне со списком по алфавиту.

− *LBS_STANDARD* – Сортирует строки в окне со списком в алфавитном порядке. Родительское окно принимает входящее сообщение всякий раз, когда пользователь щелкает мышью или дважды щелкает по строке. Окно со списком имеет, рамку со всех сторон.

− *LBS_USETABSTOPS* – Дает возможность окну списка распознавать и развернуть символы в виде таблицы при прорисовке его строк. По умолчанию таблица занимает 32 единицы измерения диалогового окна. Единица измерения диалогового окна - горизонтальное или вертикальное расстояние. Одна горизонтальная единица диалогового окна равна четвертой части текущей единицы измерения габаритов диалогового окна. Windows вычисляет эти единицы измерения, основанные на высоте и ширине шрифта существующей системы. Функция *GetDialogBaseUnits* возвращает значение текущей базовой единицы измерения диалогового окна в пикселях.

− *LBS_WANTKEYBOARDINPUT* – Определяет, что владелец окна списка принимает сообщения *WM_VKEYTOITEM* всякий раз, когда пользователь нажимает клавишу, а окно со списком имеет фокус ввода. Это дает возможность прикладной программе выполнить специальную обработку при вводе с клавиатуры.

− Следующие стили линейки прокрутки (в классе *SCROLLBAR*) могут быть определены в *dwStyle* параметре:

− *SBS_BOTTOMALIGN* – Выравнивает нижнюю кромку линейки прокрутки с нижней кромкой прямоугольника, определенного параметрами *x*, *y*, *Width* и *Height*. Линейка прокрутки имеет заданную по умолчанию высоту для системных линеек прокрутки. Используйте этот стиль со стилем *SBS_HORZ*.

− *SBS_HORZ* – Обозначает горизонтальную линейку прокрутки. Если стили ни *SBS_BOTTOMALIGN*, ни *SBS_TOPALIGN* не определены, линейка прокрутки имеет высоту, ширину и позицию, определенные *x*, *y*, *Width* и *Height*.

− *SBS_LEFTALIGN* – Выравнивает левый край линейки прокрутки с левым краем прямоугольника, определенного параметрами *x*, *y*, *Width* и *Height*. Линейка прокрутки имеет заданную по умолчанию ширину для системных линеек прокрутки. Используйте этот стиль с *SBS_VERT* стилем.

− *SBS_RIGHTALIGN* – Выравнивает правый край линейки прокрутки с правым краем прямоугольника, определенного параметрами *x*, *y*, *Width* и *Height*. Линейка прокрутки имеет заданную по умолчанию ширину для системных линеек прокрутки. Используйте этот стиль с SBS_VERT стилем.

− *SBS_SIZEBOX* – Обозначает размер окна. Если Вы не определяете ни *SBS_SIZEBOXBOTTOMRIGHTALIGN*, ни *SBS_SIZEBOXTOPLEFTALIGN* стиль, размер окна имеет высоту, ширину и позицию, определенную параметрами *x*, *y*, *Width* и *Height*.

− *SBS_SIZEBOXBOTTOMRIGHTALIGN* – Выравнивает размер нижнего правого угла окна с нижним правым углом прямоугольника, определенного парамет-

рами *x*, *y*, *Width* и *Height*. Размер окна имеет заданный по умолчанию размер для системы размера окон. Используйте этот стиль с *SBS_SIZEBOX* стилем.

− *SBS_SIZEBOXTOPLEFTALIGN* – Выравнивает размер верхнего левого угла окна с левым верхним углом прямоугольника, определенного параметрами *x*, *y*, *Width* и *Height*. Размер окна имеет заданный по умолчанию размер для системы размера окон. Используйте этот стиль с *SBS_SIZEBOX* стилем.

− *SBS_SIZEGRIP* – Подобен стилю *SBS_SIZEBOX*, но с выпуклой рамкой.

− *SBS_TOPALIGN* – Выравнивает верхний край линейки прокрутки с верхним краем прямоугольника, определенного параметрами *x*, *y*, *Width* и *Height*. Линейка прокрутки имеет заданную по умолчанию высоту для системы линеек прокрутки. Используйте этот стиль с *SBS_HORZ* стилем.

− *SBS_VERT* – Обозначает вертикальную линейку прокрутки. Если Вы не определяете ни *SBS_RIGHTALIGN*, ни *SBS_LEFTALIGN* стиль, линейка прокрутки имеет высоту, ширину и позицию, определенную параметрами *x*, *y*, *Width* и *Height*.

− Следующие стили статического элемента управления (в классе *STATIC*) могут быть определены в параметре *dwStyle*. Статический элемент управления может иметь только один из этих стилей:

− *SS_BITMAP* – Определяет, что в статическом элементе управления должен отобразиться точечный рисунок. Текст кода ошибки - имя точечного рисунка (не имя файла) определенного в другом месте файла ресурса. Стиль игнорирует параметры *Width* и *Height*; элемент управления автоматически устанавливает собственные размеры, чтобы поместить точечный рисунок.

− *SS_BLACKFRAME* – Определяет окно с рамкой, использующей тот же самый цвет, как и у рамки основного окна. Этот цвет черный по умолчанию в системе цветов Windows.

− *SS_BLACKRECT* – Определяет прямоугольник, заполненный текущим цветом рамки окна. По умолчанию этот цвет черный в системе цветов *Windows*.

− *SS_CENTER* – Определяет простой прямоугольник и выравнивает по центру текст кода ошибки в прямоугольнике. Текст форматируется перед отображением его на экране. Слова, которые выходят за пределы конца строки автоматически переносятся в начало следующей центрированной строки.

− *SS_GRAYFRAME* – Определяет поле окна с рамкой, выведенной тем же самым цветом, что и экранный фон (рабочий стол). По умолчанию в системе цветов *Windows* этот цвет серый.

− *SS_GRAYRECT* – Определяет прямоугольник, заполненный текущим экранным цветом фона. По умолчанию в системе цветов Windows этот цвет серый.

− *SS_ICON* – Определяет пиктограмму, отображаемую в диалоговом окне. Данный текст – имя пиктограммы (не имя файла) определенный в другом месте файла ресурса. Стиль игнорирует параметры *nWidth* и *nHeight*; пиктограмма автоматически устанавливает свою величину.

− *SS_LEFT* – Определяет простой прямоугольник и выравнивание по левому краю текста, помещенного в прямоугольнике. Текст форматируется перед его

отображением. Слова, которые выходят за пределы конца строки автоматически переносятся в начало следующей, выровненной по левой границе строки.

− *SS_LEFTNOWORDWRAP* – Определяет простой прямоугольник и выравнивание по левому краю текста, помещенного в прямоугольнике. Планшеты расширяются, но слова не переносятся. Текст, который выходит за пределы конца строки, отсекается.

− *SS_NOTIFY* – Посылает родительскому окну уведомительные сообщения *STN_CLICKED* и *STN_DBLCLK*, когда пользователь щелкает или дважды щелкает мышью по элементу управления.

− *SS_RIGHT* – Определяет простой прямоугольник и выравнивает по правому краю текста помещенный в прямоугольнике. Текст форматируется перед его отображением на экране. Слова, которые выходят за пределы конца строки автоматически переносятся в начало следующей выровненной по правой границе строки.

− *SS_RIGHTIMAGE* - Определяет, что угол правой нижней части статического элемента управления со стилем *SS_BITMAP* или *SS_ICON* должен остаться фиксированным, когда элемент управления изменяется. Только верхняя и левая стороны корректируются, чтобы поместить новый точечный рисунок или пиктограмму.

− *SS_SIMPLE* – Определяет простой прямоугольник и отображает одиночную строку выровненного по левой границе текста в прямоугольнике.
Текстовая строка не может быть, сокращена или изменена в любом случае. Родительское окно панели управления или диалоговое окно не должны обрабатывать сообщение *WM_CTLCOLORSTATIC*.

− *SS_WHITEFRAME* – Определяет поле окна с рамкой, выведенной тем же самым цветом как фон окна. По умолчанию, в системе цветов Windows - этот цвет белый.

− *SS_WHITERECT* – Определяет прямоугольник, заполненный текущим цветом фона окна. По умолчанию, в системе цветов Windows – этот цвет белый.

− Ниже перечислены стили диалогового окна, которые могут быть определены в параметре *dwStyle*:

− *DS_ABSALIGN* – Указывает, что координаты диалогового окна – экранные координаты; иначе, Windows принимает их за координаты пользователя.

− *DS_CENTER* – Выравнивает по центру диалоговое окно в рабочей области; то есть в области, не загораживаемой панелью.

− *DS_CENTERMOUSE* – Выравнивает по центру курсор мыши в диалоговом окне.

− *DS_CONTROL* – Создает диалоговое окно, которое работает также как дочернее окно другого диалогового окна, очень похожее на страницу в окне свойств. Этот стиль позволяет пользователю перемещаться среди элементов управления дочернего диалогового окна, использовать его клавиши-ускорители, и так далее.

− *DS_MODALFRAME* – Создает диалоговое окно с модальной рамкой диалогового окна, которая может быть объединена со строкой заголовка и меню окна, путем определения стилей *WS_CAPTION* и *WS_SYSMENU*.

− *DS_NOFAILCREATE* – Создает диалоговое окно, даже если происходят ошибки - например, если дочернее окно не может быть создано или если система не может создать специальный сегмент данных для элементов редактирования.

− *DS_NOIDLEMSG* – Подавляет сообщения *WM_ENTERIDLE*, которое Windows иначе послал бы владельцу диалогового окна, в то время как диалоговое окно отображается на экране.

− *DS_RECURSE* – Стиль диалогового окна подобно элементу управления диалоговых окон.

Функция *ShowWindow* устанавливает режим отображения окна:

```
BOOL ShowWindow (
HWND hWnd,    // указатель на окно
int CmdShow); // режим.
```

Функция *UpdateWindow* обновляет указанное окно, посылая ему сообщение *WM_PAINT*. Это сообщение посылается непосредственно процедуре указанного окна, обходя очередь других сообщений.

```
BOOL UpdateWindow (HWND Wnd);  // указатель на окно.
```

Функция *DestroyWindow* уничтожает определенное окно. Функция посылает сообщения *WM_DESTROY* и *WM_NCDESTROY* окну, чтобы дезактивировать его и удалить фокус клавиатуры. Функция также уничтожает меню окна, очищает очередь потоков сообщений, уничтожает таймеры, удаляет монопольное использование буфера обмена и разрывает цепочку просмотра окон буфера обмена (если окно имеет наверху цепочку просмотров). Если определенное окно – родитель или владелец окон, *DestroyWindow* автоматически уничтожает связанные дочерние или находящиеся в собственности окна, когда она уничтожает окно владельца или родителя. Функция сначала уничтожает дочерние или находящиеся в собственности окна, и затем она уничтожает окно владельца или родителя.

```
BOOL DestroyWindow (HWND hWnd);  // дескриптор для уничтожения окна.
```

С набором сообщений работает несколько функций *Win32 API*.
Сообщения ставятся в очередь асинхронных сообщений при помощи функции:

```
BOOL PostMessage (
HWND Window,    // дескриптор окна
UINT Message,   // передаваемое сообщение
WPARAM Parametr,
LPARAM Par);
```

При вызове этой функции определяется поток, создавший окно с дескриптором Windows. Далее выделяется память для параметров сообщения и производится добавление в очередь асинхронных сообщений. Возврат из этой функции происходит немедленно, и вероятно, что окно даже не получит данное сообщение. Кстати, упомянутая выше функция *PostQuitMessage* тоже добавляет сообщение в очередь асинхронных сообщений потока.

Оконное сообщение можно отправить оконной процедуре вызовом функции *SendMessage*, которая производит добавление в очередь синхронных сообщений.

BOOL SendMessage (
HWND Window, // дескриптор окна
UINT Message, // передаваемое сообщение
WPARAM Parametr,
LPARAM Par);

Оконная процедура обработает сообщение *Message*, и только после этого вернет управление. Пока сообщение не обработано, поток находится в состоянии ожидания. Поток, обрабатывающий синхронное сообщение, может содержать бесконечный цикл, тогда поток-отправитель со спокойной совестью может «зависнуть». Избежать таких ситуаций можно при помощи
нескольких функций *WinAPI*. Одна из них ожидает в течение некоторого времени ответа от другого потока.

BOOL SendMessageTimeout (
HWND Window, // дескриптор окна
UINT Message, // передаваемое сообщение
WPARAM Parametr,
LPARAM Par,
UINT Flags, // флаги
UINT Timeout, // время ожидания ответа, в миллисекундах
PDWORD_PTR Result); // возвращаемое значение оконной процедуры

Вторая функция также предназначена для отправки оконных сообщений:

BOOL SendMessageCallback (
HWND Window, // дескриптор окна
UINT Message, // передаваемое сообщение
WPARAM Parametr,
LPARAM Par,
SENDASYNCPROC PrcResCallBack, // асинхронная процедура
ULONG_PTR Data); // данные для передачи оконной процедуре

При вызове потоком этой функции сообщение добавляется в очередь синхронных сообщений потоком-приемником, а управление сразу же возвращается потоку-отправителю. По окончании обработки сообщения приемник асинхронно отправляет свое сообщение в очередь ответных сообщений. Чтобы поток-отправитель смог получить этот ответ должна быть предусмотрена функция, имеющая следующий прототип:

VOID CALLBACK ResCallBack (
HWND Window, // дескриптор окна
UINT Message, // передаваемое сообщение
ULONG_PTR Data, // данные для передачи оконной процедуре
LRESULT Res); // результат обработки сообщения от оконной процедуры.

Адрес этой функции передается в параметре *PrcResCallBack* функции *SendMessageCallback*. При вызове *ResCallBack* ей передается одноименный параметр *Data* из *SendMessageCallback*. Параметр *Res* содержит результат обработки сообщения от оконной процедуры.

Еще одна функция *Win*32 *API*, предназначенная для обмена сообщениями между потоками:

BOOL SendNotifyMessage (
HWND Window, // дескриптор окна
UINT Message, // передаваемое сообщение

WPARAM Parametr,
LPARAM Par);

Она также добавляет элемент в очередь синхронных сообщений и немедленно возвращает управление вызывающему потоку, то есть ее поведение подобно функции *PostMessage*. Однако есть и отличия. Если *SendNotifyMessage* посылает сообщение окну другого потока, то они извлекаются из очереди раньше сообщений, отправленных *PostMessage*, то есть имеет перед ними приоритет.

Если же посылается сообщение тому окну, которое создано потоком-отправителем, то управление не возвращается до окончания обработки сообщения, что очень похоже на поведение функции *SendMessage*.

Вообще, большинство синхронных сообщений посылается окну просто для уведомления об изменении состояния, чтобы оно отреагировало перед продолжением работы. Например, таковым является сообщение *WM_DESTROY*. Система не прерывает работу, чтобы процедура окна могла его обработать, тогда как сообщение *WM_CREATE* вызывает перерыв до окончания его обработки.

Четвертая функция *Win API*, связанная с обработкой сообщений, вызывается потоком, принимающим оконное сообщение:

BOOL ReplyMessage (LRESULT Res).

Ответ (результат обработки) асинхронно помещается в очередь ответных сообщений потока-отправителя, а тот может теперь получить результат с помощью параметра *Res* и продолжить свою работу. Функция возвращает *TRUE*, если обрабатывается межпоточное сообщение, а *FALSE* используется для внутрипоточных сообщений. Для того чтобы узнать является ли сообщение меж- или внутрипоточным, используется функция:

BOOL InSendMessage (LRESULT Res).

Возвращаемое значение: *TRUE*, если обрабатывается межпоточное синхронное сообщение, и *FALSE* при обработке синхронного и асинхронного внутрипоточного сообщения.

Некоторые из функций, предназначенных для посылки и приема сообщений, были описаны в одной из более ранних частей нашего курса, поэтому повторяться мы не будем.

Перечень некоторых сообщений, которые могут посылаться окну:

- *WM_MOUSEMOVE* – курсор меняет свою позицию. При этом в процедуре обработки сообщений надо обязательно учитывать, что ее "первый" (если быть более точным, то третий) параметр *Parametr* будет содержать флаги виртуальных клавиш, а "второй" (четвертый, *Prm*) – координаты курсора. Для их получения надо воспользоваться макросом *MAKEPOINTS*.
- *WM_NCMOUSEMOVE* – курсор перемещается над неклиентской областью окна. Параметры обозначают координаты курсора и так называемые *hit-test* значения, они будут описаны ниже.
- *WM_NCHITTEST* – перемещается курсор мыши или нажимается/освобождается кнопка мыши. Параметры: координаты курсора. Возвращает *hit-test* значения.

- *WM_NCLBUTTONUP* – пользователь отжимает левую кнопку мыши, ее курсор находится над неклиентской областью окна. Параметры: *hit-test* значение и позиция курсора.
- *WM_NCLBUTTONDOWN* – пользователь нажимает левую кнопку мыши, ее курсор находится над неклиентской областью окна. Параметры: *hit-test* значение и позиция курсора.
- *WM_NCRBUTTONUP* и *WM_NCRBUTTONDOWN* – аналогичны двум предыдущим, только "работают" с правой кнопкой мыши.
- *WM_NCHITTEST* – перемещается курсор мыши или нажимается/отжимается одна из ее кнопок.

Параметры - координаты курсора.

Hit-test значения демонстрируют, в каком месте окна находится курсор:

– *HTBORDER* – на границе окна;
– *HTBOTTOM* – на нижней горизонтальной границе окна;
– *HTBOTTOMLEFT* – в левом нижнем углу границы окна;
– *HTBOTTOMRIGHT* – в правом нижнем углу границы окна;
– *HTCAPTION* – на заголовке окна;
– *HTCLIEN T*– на клиентской области окна;
– *HTERROR* – фон экрана или разделительная линия между двумя окнами;
– *HTGROWBOX* – в области изменения размеров окна;
– *HTHSCROLL* – на горизонтальной полосе прокрутки;
– *HTLEFT* – на левой границе окна;
– *HTMENU* – в меню;
– *HTREDUCE* – на кнопке "Свернуть";
– *HTRIGHT* – на правой границе окна;
– *HTSIZE* – области изменения размеров окна;
– *HTSYSMENU* – на кнопке закрытия дочернего окна;
– *HTTOP* – на верхней горизонтальной границе окна;
– *HTTOPLEFT* – в левом верхнем углу границы окна;
– *HTTOPRIGHT* – в правом верхнем углу границы окна;
– *HTTRANSPARENT* – на окне в данный момент перекрытым другим окном;
– *HTVSCROLL* – на вертикальной полосе прокрутки;
– *HTZOOM* – на кнопке "Развернуть".

Функция вывода окна на передний план:

BOOL SetForegroundWindow (HWND Window).

Для получения дескриптора окна, находящегося на переднем плане, используется следующая функция:

HWND GetForegroundWindow ().

Фрагменты кода, иллюстрирующие использование некоторых из описанных функций:

```
HWND win1;
DWORD CurrThread, WinThread;
CurrThread = GetCurrentThreadId ();
```

WinThread = GetWindowThreadProcesId (GetForegroundWindow (),
AttachThreadInput (CurrThread, TRUE);
...
SetForegroundWindow (win1).

Функция получения экранных координат указанного окна:

BOOL GetWindowRect (
HWND Wnd, // идентификатор окна
LPRECT Rect); // адрес структуры с координатами окна.

Получение координат клиентской части окна, при равенстве координат левой верхней точки нулю:

BOOL GetClientRect (
HWND Wnd, // идентификатор окна
LPRECT Rect); // адрес структуры с координатами окна

Определение координаты левой верхней и правой нижней точек прямоугольника:

typedef struct _RECT {
LONG left;
LONG top;
LONG right;
LONG bottom;
} RECT;

Определение принадлежности точки прямоугольнику:

BOOL PtInRect(
CONST RECT *Rect, // адрес структуры-прямоугольника
POINT aPoint); // структура с точкой.

Определение структуры с точкой:

typedef struct tagPOINT {
LONG x;
LONG y;
} POINT;

Функция для получения экранных координат курсора мыши:

BOOL GetCursorPos (LPPOINT aPoint); // адрес структуры с координатами курсора.

Изменение некоторых параметров окна производится функцией:

BOOL MoveWindow(
HWND Wnd, // Описатель окна
int Hor, // горизонтальная координата новой позиции окна
int Vert, // вертикальная координата новой позиции окна
int Width, // новая ширина
int Height, // новая высота
BOOL Repaint); // Определяет, будет ли окно перерисовываться (TRUE)

Функция для получения и установки параметров полосы прокрутки:

BOOL GetScrollInfo(
HWND Wnd, // описатель окна с полосой прокрутки
int Bar, // тип полосы прокрутки (SB_HORZ -горизонтальная,
// SB_VERT -вертикальная)
LPSCROLLINFO ScrollInfo);// указатель на структуру с параметрами скроллинга
int SetScrollInfo(
HWND Wnd, // описатель окна
int fnBar, //тип полосы прокрутки (SB_HORZ -горизонтальная,
// SB_VERT - вертикальная)

LPSCROLLINFO ScrollInfo, // указатель на структуру с параметрами скроллинга
BOOL fRedraw);// флаг перерисовки (TRUE - перерисисовывать).

Функция для прокручивания содержимого клиентской области окна:

```
int ScrollWindowEx(
HWND Wnd,   // описатель окна
int dx,  // горизонтальная величина скроллинга.
int dy,     // вертикальная величина скроллинга.
CONST RECT *Scroll,//структуры с прямоугольником прокрутки,
// TRUE - прокручивать все
CONST RECT *Clip, // адрес структуры с прямоугольником-клипом
HRGN RegionUpdate, // обычно NULL
LPRECT Update, // обычно NULL
UINT flags); // параметры прокрутки, обычно SW_ERASE
```

Группа функций позволяющая направлять и сбрасывать "мышиные" сообщения, связанные с конкретными окнами:

```
// Установка захвата сообщений
HWND SetCapture(HWND hWnd); // описатель окна
HWND GetCapture(VOID) // получение идентификатора окна
BOOL ReleaseCapture(VOID)  // сброс перехвата сообщений
```

6.2 Содержание отчета

- постановка задачи;
- блок – схема алгоритма работы программы;
- результаты работы программы;
- выводы о проделанной работе;
- листинг программного кода.

6.3 Варианты индивидуальных заданий

Вариант №1

Разработать программу, которая демонстрирует эффект «убегания окна от курсора» при его попадании на клиентскую область окна. Завершение «убегания» достигается с помощью двойного щелчка кнопки мыши.

Вариант №2

Разработать программу, которая демонстрирует эффект «прилипания окна к курсору» при его попадании на клиентскую область окна. «Отлипание окна от курсора» производится последовательным перемещением курсора вверх, а потом вниз.

Вариант №3

Разработать программу, которая демонстрирует эффект «убегания окна от курсора» при его попадании на не клиентскую область окна. Завершение «убегания» достигается с помощью двойного щелчка кнопки мыши.

Вариант №4

Разработать программу, которая демонстрирует эффект «прилипания окна к курсору» при его попадании на заголовок окна. Дальнейшее перемещение влево-вправо блокируется, окно может перемещаться только вверх и вниз. «Отлипание окна от курсора» производится двойным щелчком кнопки мыши.

Вариант №5

Разработать программу, которая демонстрирует эффект «прилипания окна к курсору» при его попадании на заголовок окна. Дальнейшее перемещение вверх-вниз блокируется, окно может перемещаться только влево и вправо. «Отлипание окна от курсора» производится двойным щелчком кнопки мыши.

Вариант №6

Разработать программу, которая создает окно с горизонтальной и вертикальной полосами прокрутки. При щелчке кнопки мыши по заголовку окна блокируется вертикальная прокрутка. Блокировка снимается при щелчке кнопки мыши по вертикальной полосе.

Вариант №7

Разработать программу, которая создает окно с горизонтальной и вертикальной полосами прокрутки. При щелчке кнопки мыши по заголовку окна блокируется горизонтальная прокрутка. Блокировка снимается при щелчке кнопки мыши по горизонтальной полосе.

Вариант №8

Разработать программу, которая создает два окна. Действия по сворачиванию, разворачиванию и закрытию одного окна должны выполняться над обоими окнами.

Вариант №9

Разработать программу, которая демонстрирует эффект «прилипания окна к курсору» при его попадании на полосу прокрутки и последующем двойном щелчке кнопки мыши по какой-либо из полос прокрутки. «Отлипание окна от курсора» производится повторным двойным щелчком кнопки мыши по полосе прокрутки.

Вариант №10

Разработать программу, которая демонстрирует эффект «прилипания окна к курсору» при его попадании и последующем щелчке кнопки мыши по какой-либо из полос прокрутки. «Отлипание окна от курсора» производится трехкратным щелчком кнопки мыши по заголовку окна.

Вариант №11

Разработать программу, которая демонстрирует эффект «прилипания окна к курсору» при его попадании и последующем щелчке кнопки мыши по заголовку окна. «Отлипание окна от курсора» производится трехкратным щелчком кнопки мыши по заголовку окна.

Вариант №12

Разработать программу, которая создает окно с горизонтальной и вертикальной полосами прокрутки. При щелчке кнопки мыши по вертикальной полосе прокрутки окна блокируется горизонтальная прокрутка. Блокировка снимается при щелчке кнопки мыши по вертикальной полосе.

Вариант №13

Разработать программу, которая создает окно с горизонтальной и вертикальной полосами прокрутки. При щелчке кнопки мыши по горизонтальной полосе прокрутки окна блокируется вертикальная прокрутка. Блокировка снимается при щелчке кнопки мыши по горизонтальной полосе.

Вариант №14

Разработать программу, которая создает окно. Закрытие окна должно выполняться щелчком кнопки по значку разворачивания окна, при этом должно создаваться новое окно с таким же стилем. Однократный щелчок кнопки по значку закрытия не приводит к выполнению этого действия. Двойной щелчок по этому значку завершает работу программы.

Вариант №15

Разработать программу, которая создает три окна. Действия по сворачиванию и разворачиванию одного окна должны выполняться над обоими окнами.

Вариант №16

Разработать программу, которая демонстрирует эффект «прилипания окна к курсору» при его попадании и последующем щелчке кнопки мыши по горизонтальной полосе прокрутки. «Отлипание окна от курсора» производится последовательным перемещением курсора влево-вправо-вверх.

Вариант №17

Разработать программу, которая демонстрирует эффект «прилипания окна к курсору» при его попадании и последующем двойном щелчке кнопки мыши по заголовку окна. «Отлипание окна от курсора» производится повторным двойным щелчком кнопки мыши по заголовку окна.

Вариант №18

Разработать программу, которая создает окно. Закрытие окна должно выполняться щелчком кнопки по значку закрытия окна, при этом должно создаваться новое окно с таким же стилем. Двойной щелчок по этому значку завершает работу программы.

Вариант №19

Разработать программу, которая создает три окна. Действия по сворачиванию и, закрытию одного окна должны выполняться над обоими окнами.

Вариант №20

Разработать программу, которая демонстрирует эффект «прилипания окна к курсору» при его попадании и последующем щелчке кнопки мыши по вертикальной полосе прокрутки. «Отлипание окна от курсора» производится повторным щелчком кнопки мыши по этой же полосе прокрутки.

6.4 Пример выполнения лабораторной работы

6.4.1 Задание

Разработать программу, которая создает окно. Закрытие окна должно выполняться щелчком кнопки по значку сворачивания, при этом должно создаваться новое окно с таким же стилем. Однократный щелчок кнопки по значку закрытия не приводит к выполнению этого действия. Двойной щелчок по этому значку завершает работу программы.

6.4.2 Схема алгоритма.

Схема алгоритма программы представлена на рисунке 6.1

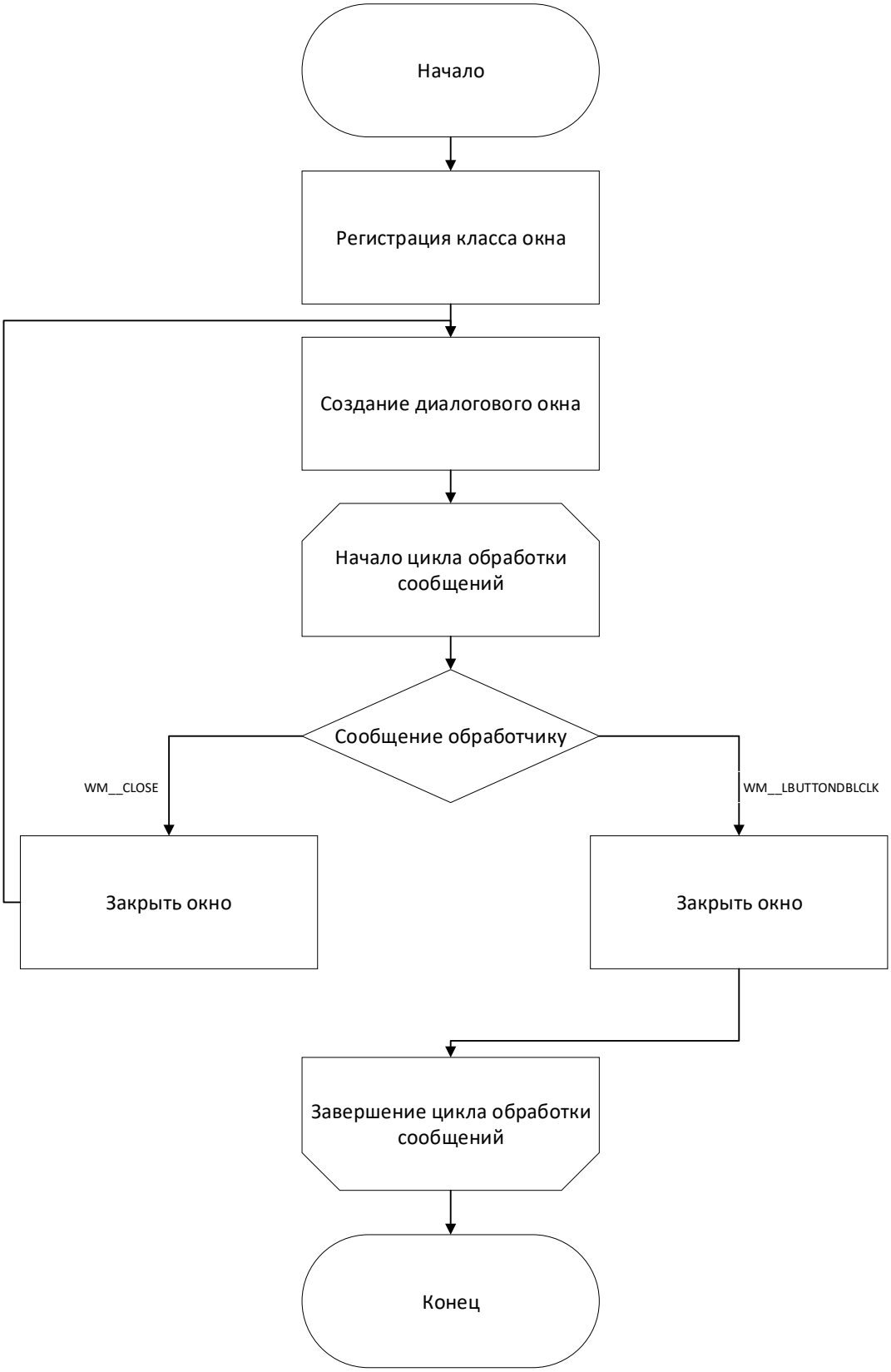

Рисунок 6.1 – Схема алгоритма управления окнами

6.4.3 Код программы.

```
#include <windows.h>
// объявление функций
LRESULT CALLBACK WndProc(HWND, UINT, WPARAM, LPARAM);
ATOM RegMyWindowClass(HINSTANCE, LPCTSTR);
WINDOWPLACEMENT wnd1;
HWND hWnd;
int check = 0, check1 = 0;
// функция вхождений программы WinMain
int APIENTRY WinMain(HINSTANCE hInstance,
   HINSTANCE        hPrevInstance,
   LPSTR            lpCmdLine,
   int              nCmdShow)
{
   // имя будущего класса
   LPCTSTR lpzClass = TEXT("My Window Class!");
   // регистрация класса
   if (!RegMyWindowClass(hInstance, lpzClass))
      return 1;
   // вычисление координат центра экрана
   RECT screen_rect;
   GetWindowRect(GetDesktopWindow(), &screen_rect); // разрешение экрана
   int x = screen_rect.right / 2 - 150;
   int y = screen_rect.bottom / 2 - 75;
   // создание диалогового окна
   if (check1 == 0) {
      hWnd = CreateWindow(lpzClass, TEXT("Dialog Window"),
         WS_OVERLAPPEDWINDOW | WS_VISIBLE, x, y, 300, 250, NULL, NULL,
         hInstance, NULL);
      check1++;
   }
   // если окно не создано, описатель будет равен 0
   if (!hWnd) return 2;
   // цикл сообщений приложения
   MSG msg = { 0 };    // структура сообщения
   int iGetOk = 0;   // переменная состояния
   while ((iGetOk = GetMessage(&msg, NULL, 0, 0)) != 0) // цикл сообщений
   {
      if (iGetOk == -1) return 3;  // если GetMessage вернул ошибку - выход
      TranslateMessage(&msg);
      DispatchMessage(&msg);
      GetWindowPlacement(hWnd, &wnd1);
      //GetWindowPlacement(hMainWnd2, &wnd2);
      if (wnd1.showCmd != 0 && check == 0) {
         hWnd = CreateWindow(lpzClass, TEXT("Dialog Window"),
            WS_OVERLAPPEDWINDOW | WS_VISIBLE, x, y, 300, 300, NULL, NULL,
            hInstance, NULL);
         check++;
      }
   }
   return msg.wParam;  // возвращаем код завершения программы
```

```
}
// функция регистрации класса окон
ATOM RegMyWindowClass(HINSTANCE hInst, LPCTSTR lpzClassName)
{
   WNDCLASS wcWindowClass = { 0 };
   // адрес ф-ции обработки сообщений
   wcWindowClass.lpfnWndProc = (WNDPROC)WndProc;
   // стиль окна
   wcWindowClass.style = CS_HREDRAW | CS_VREDRAW;
   // дискриптор экземпляра приложения
   wcWindowClass.hInstance = hInst;
   // название класса
   wcWindowClass.lpszClassName = lpzClassName;
   // загрузка курсора
   wcWindowClass.hCursor = LoadCursor(NULL, IDC_ARROW);
   // загрузка цвета окон
   wcWindowClass.style = CS_DBLCLKS;    // Добавил стиль
   return RegisterClass(&wcWindowClass); // регистрация класса
}
// функция обработки сообщений
LRESULT CALLBACK WndProc(
   HWND hWnd, UINT message, WPARAM wParam, LPARAM lParam)
{
   // выборка и обработка сообщений
   switch (message)
   {
   case WM_LBUTTONDBLCLK: {
      PostQuitMessage(0);
   }
  /* case WM_LBUTTONUP:
      // вот тут программа завершает только если кликать в области окна
      if (check != 0)
      {
         PostQuitMessage(0);
      }
      check = 0;
      break;*/
   case WM_CLOSE:
      check = 0;
      DestroyWindow(hWnd);
      return 0;
   case WM_NCLBUTTONDBLCLK:
      if (wParam == HTCLOSE)
         PostQuitMessage(0);
   default:
      // все сообщения не обработанные Вами обработает сама Windows
      return DefWindowProc(hWnd, message, wParam, lParam);
   }
   return 0;
}
```

6.4.4 Реализация программного кода

Отдельные фрагменты реализации программного кода представлены на рисунках 6.2 – 6.4.

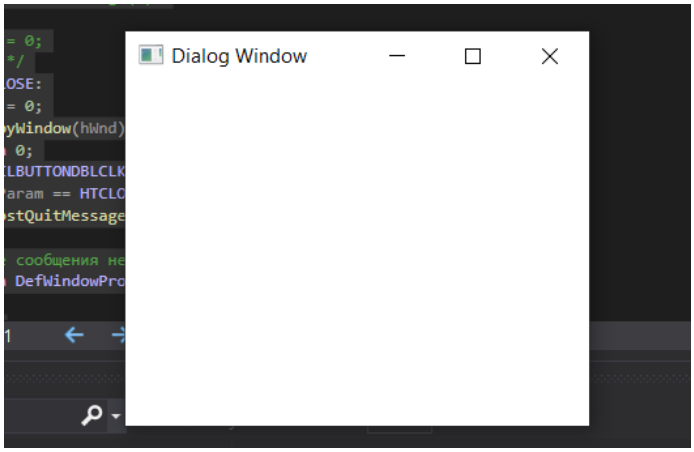

Рисунок 6.2 – Создание диалогового окна

Далее при простом щелчке по значку закрытия окна (рисунок 6.3), закроется изначальное окно и откроется окно с таким же стилем в другом месте (рисунок 6.4).

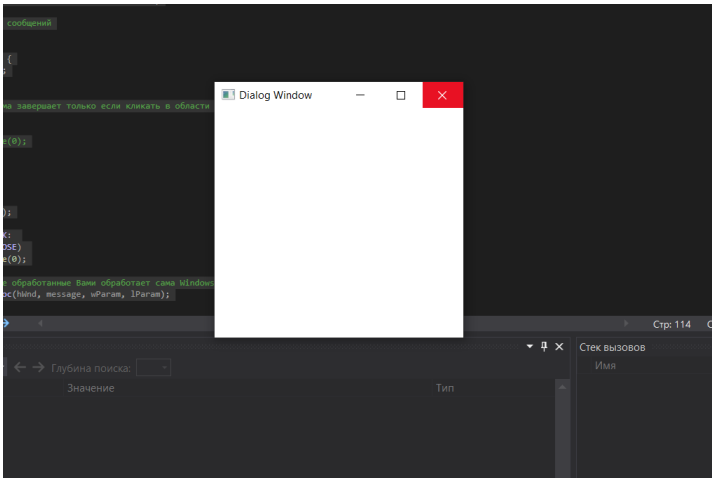

Рисунок 6.3 – Закрытие окна одинарным шелчком мыши

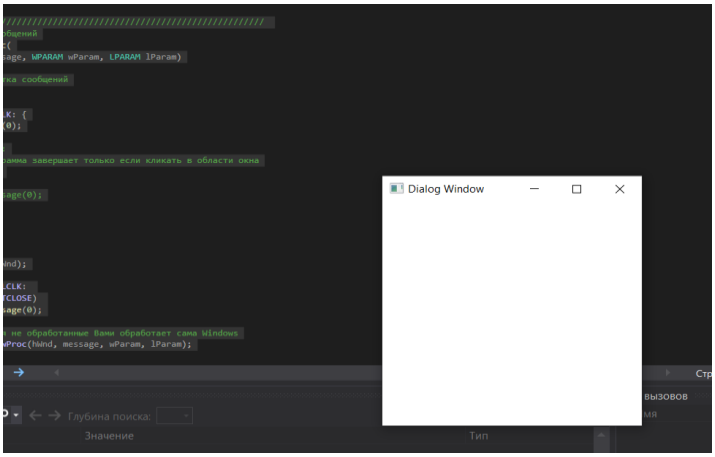

Рисунок 6.4 – Открытие окна с таким же стилем после закрытия исходного

Заключение

1. Настоящий лабораторный практикум позволяет получить практические навыки программирования основных задач, решаемых в операционных системах.

2. Задачи могут решаться с использованием различных языков программирования высокого уровня.

3. Индивидуальные варианты задач обеспечивают персональный подход к каждому обучающемуся.

4. Приведенные сведения из теории обеспечивают связь теории и практики и повышают качество решений.

5. Лабораторный практикум может быть реализован в различных формах и технологиях обучения.

6. Учебно – методический и компетентностный подходы, позволяют освоить, закрепить и развить требуемую компетенцию.

Список использованных источников

1. Джонс, Э. Программирование в сетях Microsoft Windows. Мастер-класс / Э. Джонс, Дж. Оланд. – СПб.: Питер, М.: Русская редакция, 2002. – 608 с.

2. Ковалев, И. В. Операционные системы и системное программное обеспечение: учеб. пособие / И. В. Ковалев, А. С. Кузнецов. – Красноярск: ИПЦ КГТУ, 2005. – 302 с.

3. Гордеев, А.В. Системное программное обеспечение / А.В. Гордеев, А. Ю. Молчанов. – СПб.: Питер, 2003. – 400 с.

4. Русинович, М. Внутреннее устройство Windows. 7-е изд. / Д. Соломон, А. Ионеску, П. Йосифович. – СПб.: Питер, 2018. – 944 с. ISBN 978-5-4461-0663-9

5. Танненбаум, Э. Современные операционные системы / Э. Танненбаум. – СПб.: Питер, 2015. – 1040 с.

6. Харт, Д. Системное программирование в среде Win32 / Д. Харт. – М.: Вильямс, 2001. – 464 с.