Министерство образования Республики Беларусь
Учреждение образования
«Белорусский государственный университет
информатики и радиоэлектроники»
Кафедра информационных технологий автоматизированных систем

А. А. Навроцкий

ИСПОЛЬЗОВАНИЕ WINDOWS API

Лабораторный практикум по курсу
«Системное программное обеспечение»
для студентов 2 – 3-го курсов специальности АСОИ

Минск БГУИР 2016

УДК 681.3.06 (075.8)
ББК 32.973.26-018.1 я7
Н 00

Навроцкий, А.А.
Н 00 Использование.

ISBN 978-985-488-374-8

Приведены 8 лабораторных работ на языке C++ в среде Microsoft Visual Studio с примерами выполнения; представлены индивидуальные задания; дана необходимая справочная информация.

УДК 681.3.06 (075.8)
ББК 32.973-018 я 73

ISBN 978-985-488-374-8

СОДЕРЖАНИЕ

ЛАБОРАТОРНАЯ РАБОТА № 1.
РАБОТА С ФАЙЛАМИ

1.1. Некоторые функции Windows для работы с файлами

CreateFile()

Функция CreateFile служит для создания или открытия нового устройства (файла). Разрешено выполнение операций над файлами, каналами передачи данных, дисковыми устройствами, консолями, директориями (их открытие). При обращении к каналам функция позволяет создавать клиентское подключение к именованным каналам, находящимся в режиме ожидания подключения.

Прототип функции:

```
HANDLE CreateFile (
  LPCTSTR lpFileName,  // Указатель на имя файла (устройства)
  DWORD dwDesiredAccess, // Параметры доступа
  DWORD dwShareMode,      // Режим совместного доступа
  LPSECURITY_ATTRIBUTES   lpSecurityAttributes,      // Защита
  DWORD dwCreationDistribution, // Режим открытия
  DWORD  dwFlagsAndAttributes, // Атрибуты файла
  HANDLE hTemplateFile // Файл шаблона
);
```

Параметры:

lpFileName передает указатель на нуль-терминальную строку, содержащую имя файла или устройства.

dwDesiredAccess определяет параметры доступа к открываемому файлу. Используются следующие константы (или их комбинация): 0 – доступ запрещен, но возможно изменение атрибутов; GENERIC_READ – разрешено чтение; GENERIC_WRITE – разрешена запись. При доступе к каналам следует учитывать режим создания канала сервером. Если сервер создал канал для записи, то клиент открывает его для чтения и наоборот. Если сервер создал канал для чтения и записи, то клиент может открыть его как для чтения, так и для записи.

dwShareMode задает режим совместного использования файла. Используются следующие константы (или их комбинация): 0 – совместное использование файла запрещено; FILE_SHARE_READ – другие приложения могут открывать файл для чтения; FILE_SHARE_WRITE – другие приложения могут открывать файл для записи.

lpSecurityAttributes указатель на дескриптор защиты (структура SECURITY_ATTRIBUTES). Если дескриптор защиты не используется, то указывается значение NULL.

dwCreationDistribution определяет режим открытия файла. Допустимы следующие режимы (комбинация не допускается): CREATE_NEW – создание нового файла (если файл существует, то генерируется ошибка); CREATE_ALWAYS – создание нового файла (если файл существует, то он перезаписывается); OPEN_EXISTING − открытие существующего файла (если

файл отсутствует, то генерируется ошибка); OPEN_ALWAYS − открытие существующего файла (если файл отсутствует, то создается новый файл); TRUNCATE_EXISTING – открывает файл и устанавливает размер равным нулю (если файл отсутствует, то генерируется ошибка);

dwFlagsAndAttributes задает атрибуты и флаги для файла (или их комбинацию). Атрибуты файла: FILE_ATTRIBUTE_NORMAL (обычный); FILE_ATTRIBUTE_ARCHIVE (архивный); FILE_ATTRIBUTE_COMPRESSED (сжатый); FILE_ATTRIBUTE_HIDDEN (скрытый); FILE_ATTRIBUTE_NORMAL (нормальный); ATTRIBUTE_OFFLINE (данные файла недоступны); FILE_ATTRIBUTE_READONLY (только для чтения); _ATTRIBUTE_SYSTEM (системный); FILE_ATTRIBUTE_TEMPORARY (временный). Флаги: FILE_FLAG_WRITE_THROUGH (возможность записи в файл через кэш); FILE_FLAG_NO_BUFFERING (нельзя использовать буферы или кэш); FILE_FLAG_RANDOM_ACCESS (случайный доступ. Используется для оптимизации кэша); FILE_FLAG_SEQUENTIAL_SCAN (доступ к файлу может быть последовательный от начала до конца); FILE_FLAG_DELETE_ON_CLOSE (операционная система должна удалить файл, когда все указатели на файл будут закрыты); FILE_FLAG_BACKUP_SEMANTICS (Backup файл, т.е. файл резервного копирования); FILE_FLAG_POSIX_SEMANTICS (доступ осуществляется в POSIX стандарте); SECURITY_ANONYMOUS (анонимный доступ); SECURITY_IDENTIFICATION (идентификационный доступ); SECURITY_IMPERSONATION (персональный доступ); SECURITY_DELEGATION (коллективный доступ); SECURITY_CONTEXT_TRACKING (динамический режим доступа); SECURITY_EFFECTIVE_ONLY (ограничение групп и привилегий).

hTemplateFile указатель на открытый файл, значения атрибутов и флагов которого будут использованы при создании нового файла (при открытии файла параметр игнорируется). Если параметр равен NULL, то используются флаги и атрибуты, заданные параметром dwFlagsAndAttributes.

При успешном завершении функция CreateFile указатель на созданное или открытое устройство, иначе возвращается сообщение об ошибке INVALID_HANDLE_VALUE.

! При работе с удаленными файлами, необходимо задавать сетевой путь к файлу: "\\\\имя_удаленного_компьютера\\путь_к_файлу\\имя_файла".

CloseHandle

Функция CloseHandle(HANDLE hObject) закрывает дескриптор открытого объекта (hObject). Если необходимо закрыть файл, то в качестве параметра передается дескриптор открытого файла.

Пример 1.1. Создать новый файл в текущей директории с именем File_New.dat, с доступом по чтению и записи для разных процессов, с обычными атрибутами.

```
#include <iostream>
#include <windows.h>
  using namespace std;
int main()
{
LPCTSTR fn="File_New.dat";
HANDLE myFile = CreateFile(
fn,     // Имя файла
GENERIC_READ | GENERIC_WRITE,
FILE_SHARE_READ | FILE_SHARE_WRITE,
NULL,
CREATE_NEW,
FILE_ATTRIBUTE_NORMAL,
NULL);
   if (myFile == INVALID_HANDLE_VALUE)
   cout << "Error";    // Файл не создан
   else
   cout << " File is created";    // Файл создан
CloseHandle (myFile);
cin.get();
return 0;
}
```

ReadFile и ***WriteFile***

Функция ReadFile используется для чтения данных из файла, а функция WriteFile для записи данных в файл. Записывается/читается указанное количество байт информации в/из текущей позиции указанного открытого файла. После чтения/записи данных указатель перемещается в следующую после последнего прочитанного/записанного данного позицию.

Прототип функций:

```
bool  <Имя_функции>(
  HANDLE  hFile,     // Идентификатор файла
  LPVOID  lpBuffer,      // Адрес буфера данных
  DWORD nNumberOfBytes,  // Количество байт
  LPDWORD lpNumberOfBytes,// Адрес слова, в котором
                                        // хранится количество байт
  LPOVERLAPPED lpOverlapped // Адрес структуры OVERLAPPED
);
```

Параметры:

hFile − идентификатор файла.

lpBuffer содержит адрес буфера, в котором хранятся прочитанные или предназначенные для записи данные.

nNumberOfBytes задает количество байт данных, которые должны быть прочитаны в буфер или записаны из буфера lpBuffer. Параметр используется для контроля правильности чтения или записи данных.

lpNumberOfBytes адрес слова в котором хранится количество действительно прочитанных или записанных данных. Параметр используется для контроля правильности чтения или записи данных.

lpOverlapped используется для организации асинхронных режимов чтения или записи. Если запись или чтение выполняются синхронно, в качестве параметра следует указать значение NULL.

Если функции были выполнены успешно, они возвращают значение true (иначе false).

FlushFileBuffers

Функция служит для принудительной записи на диск всех изменений для файла.

Прототип функции:

```
bool FlushFileBuffers(HANDLE hFile);
```

Функция возвращает значение true при успешном выполнении и false в противном случае.

SetFilePointer

Перемещает указатель текущей позиции на заданное число байт.

Прототип функции:

```
dword SetFilePointer (
  HANDLE hFile,          // Идентификатор файла
  LONG lDistanceToMove,     // количество байт, на которое
                // будет передвинута текущая позиция
  PLONG lpDistanceToMoveHigh, // Адрес старшего слова,
                // содержащего расстояние для
                // перемещения позиции
  DWORD  dwMoveMethod    // Способ перемещения позиции
);
```

Параметры:

hFile − идентификатор файла.

lDistanceToMove – количество байт, которое будет передвинута текущая позиция (может отрицательным для перемещения к началу файла). Допускается перемещение на 2^{32} – 2 байта.

lpDistanceToMoveHigh – используется при работе с файлами большого размера в качестве второго слова для указания смещения. Позволяет сместить позицию на 2^{64} байт. Если второе слово не используется, то устанавливают значение NULL.

dwMoveMethod указывает точку отсчета смещения. Может принимать значения: FILE_BEGIN − начало файла, FILE_CURRENT − текущая позиция, FILE_END − конец файла.

В случае удачного завершения работы функция возвращает новую позицию в файле, а в при ошибке − значение 0xFFFFFFFF.

Пример 1.2. Записать в файл числа от 0 до 9, затем прочитать записанные числа и вывести их на экран.

```
  DWORD n;
// Запись данных в файл
for (int i=0; i<10; i++)  WriteFile(myFile, &i, sizeof(i), &n, NULL);
FlushFileBuffers(myFile);
// Перемещение в начало файла
SetFilePointer(myFile,0,0, FILE_BEGIN);
  int k;
// Чтение данных из файла
while (ReadFile(myFile, &k, sizeof(k), &n, NULL) && n!=0) cout << k;
```

SetEndOfFile

Устанавливает конец файла в текущую позицию.

Прототип функции:

```
bool SetEndOfFile(HANDLE hFile)
```

Функция возвращает true при удачном завершении, или false в случае ошибки.

LockFile и ***UnlockFile***

Блокировка и разблокировка области файла. Как правило, используется в случае совместного использования файла (режимы FILE_SHARE_READ или FILE_SHARE_WRITE) для исключения ошибок при одновременном использовании файла несколькими процессами. Блокирование предоставляет процессу монопольный доступ указанной области файла. Остальные процессы не могут ни читать, ни писать в заблокированную область файла. Заблокированная область может выходить за текущий конец файла.

Прототип функций:

```
bool <Имя_функции> (
  HANDLE hFile,      // Идентификатор файла
  DWORD  dwFileOffsetLow,  // Младшее слово смещения области
  DWORD  dwFileOffsetHigh, // Старшее слово смещения области
  DWORD  nNumberOfBytesToLockLow, // Младшее слово длины
                // области
  DWORD  nNumberOfBytesToLockHigh // Старшее слово длины
                // области
);
```

hFile − идентификатор файла, для которого выполняется блокировка области.

dwFileOffsetLow и *dwFileOffsetHigh* − младшее и старшее слова смещения блокируемой области.

nNumberOfBytesToLockLow и *nNumberOfBytesToLockHigh* − младшее и старшее слова для задания размера области. Если в файле блокируется несколько областей, они не должны перекрывать друг друга.

После использования заблокированной области, а также перед завершением своей работы процессы должны разблокировать все заблокированные ранее области с помощью функции UnlockFile(). Разблокируемая область файла должна в точности соответствовать ранее заблокированной области. Процесс не должен завершаться, пока имеются заблокированные области файла.

В случае успешного завершения функции возвращают значение true, иначе – false.

DeleteFile

Удаление файла. Перед удалением файл должен быть закрыт.

Прототип функций:

```
BOOL WINAPI DeleteFile(
  LPCTSTR lpFileName;  // Указатель на имя удаляемого файла
);
```

В случае успешного выполнения функция возвращает значение true, иначе − false.

SetFileAttributes

Устанавливает атрибуты файла.

Прототип функций:

```
bool SetFileAttributes(
LPCTSTR lpFileName,    // Указатель на имя файла
DWORD dwFileAttributes // Атрибуты
);
```

dwFileAttributes − устанавливаемые атрибуты файла (одно или несколько значений). Возможно использовать следующие атрибуты: FILE_ATTRIBUTE_ARCHIVE – архивный файл (подготовленный для резервирования или перемещения); FILE_ATTRIBUTE_HIDDEN – скрытый файл; FILE_ATTRIBUTE_NORMAL – нет других атрибутов файл; FILE_ATTRIBUTE_OFFLINE –данные файла не доступны и находятся на отключённом устройстве; FILE_ATTRIBUTE_READONLY –файл только для чтения; FILE_ATTRIBUTE_SYSTEM – системный файл; FILE_ATTRIBUTE_TEMPORARY – временный файл.

В случае успешного выполнения функция возвращает значение true, иначе − false.

CopyFile

Копирование файла.

Прототип функции:

```
bool CopyFile(
LPCTSTR lpExistingFileName,
```

LPCTSTR *lpNewFileName*, // Указатель на имя копии файла
bool *bFailIfExists*
);

lpExistingFileName – указатель на имя копируемого файла.

lpNewFileName – указатель на имя нового файла.

bFailIfExists – если имеет значение true, то в случае существования нового файла, он будет перезаписан, иначе произойдет ошибка.

GetFileSize

Определяет размер файла.

Прототип функции:

DWORD **GetFileSize**(
HANDLE *hFile*,
LPDWORD *lpFileSizeHigh*
);

lpFileSizeHigh − указатель на переменную, в которую возвращается старшее слово размера файла (если размер файла меньше 4 Гб, можно указать значение NULL).

1.2. Индивидуальные задания

В программе предусмотреть сохранение вводимых данных в файл и возможность чтения из ранее сохраненного файла. Результаты выводить на экран и в текстовой файл.

1. Список товаров, имеющихся на складе, включает в себя наименование товара, количество единиц товара, цену единицы и дату поступления товара на склад. Вывести список товаров, хранящихся больше месяца и стоимость которых превышает 1 000 000 р.

2. Для получения места в общежитии формируется список студентов, который включает ФИО студента, группу, средний балл, доход на члена семьи. Вывести информацию о студентах, у которых доход на члена семьи менее двух минимальных зарплат.

3. В справочной автовокзала хранится расписание движения автобусов. Для каждого рейса указаны его номер, пункт назначения, время отправления и прибытия. Вывести информацию о рейсах, которыми можно воспользоваться для прибытия в пункт назначения раньше заданного времени.

4. Информация о сотрудниках фирмы включает ФИО, количество проработанных часов за месяц, почасовой тариф. Рабочее время свыше 144 часов считается сверхурочным и оплачивается в двойном размере. Вывести размер заработной платы каждого сотрудника фирмы за вычетом подоходного налога, который составляет 12 % от суммы заработка.

5. Информация об участниках спортивных соревнований содержит название команды, ФИО игрока, возраст. Вывести информацию о спортсменах, возраст которых не достиг 18 лет.

6. Для книг, хранящихся в библиотеке, задаются автор, название, год издания, количество страниц. Вывести список книг, изданных после заданного года.

7. На заводе выпускается несколько наименований деталей. Сведения о деталях включают код детали, количество выпущенных деталей, номер месяца выпуска. Вывести информацию о продукции, выпущенной заданным цехом за последний месяц.

8. Информация о сотрудниках предприятия содержит ФИО, номер отдела, должность, дату начала работы. Вывести список сотрудников заданного отдела, проработавших на предприятии более 20 лет.

9. Ведомость абитуриентов содержит ФИО, город проживания, суммарный балл. Вывести информацию об абитуриентах, проживающих в г. Минске и имеющих балл больше 220.

10. В справочной аэропорта хранится расписание вылета самолетов на следующие сутки. Для каждого рейса указаны номер рейса, пункт назначения, время вылета. Вывести все номера рейсов и время вылета самолета для заданного пункта назначения.

11. У администратора железнодорожных касс хранится информация о свободных местах в поездах. Информация представлена в следующем виде: номер поезда, пункт назначения, время отправления, число свободных мест. Вывести информацию о поездах, в которых имеются свободные места до заданного пункта назначения.

12. Ведомость студентов, сдававших сессию, содержит ФИО и оценки по четырем предметам. Вывести список студентов, сдавших сессию со средним баллом больше 7.

13. В радиоателье хранятся квитанции о сданных в ремонт телевизорах. Каждая квитанция содержит следующую информацию: марка телевизора, дата приемки в ремонт, состояние готовности заказа (выполнен, не выполнен). Вывести информацию о заказах, которые на текущий момент не выполнены.

14. На АТС информация о разговорах содержит номер телефона абонента, время разговора и тариф. Вывести для заданного абонента сумму, которую ему следует оплатить за разговоры.

15. В магазине составлен список людей, которым выдана карта постоянного покупателя. Каждая запись этого списка содержит номер карточки, ФИО, предоставляемую скидку. Вывести информацию о покупателях, имеющих 10 %-ную скидку в магазине.

ЛАБОРАТОРНАЯ РАБОТА № 2.
ПРОЦЕССЫ

2.1. Процессы в Windows

Процессом (process) называется объект, имеющий собственное независимое виртуальное адресное пространство, в котором размещается код и данные, защищенные от других процессов. Выполнение каждого процесса начинается с первичного потока (threads). При выполнении процесс может создавать и использовать новые потоки и независимые процессы, а так же управлять их взаимодействием и синхронизацией.

CreateProcess.

Создает процесс с главным потоком.

Прототип функции:

```
bool CreateProcess(
  LPCTSTR lpApplicationName,
  LPTSTR lpCommandLine,
  LPSECURITY_ATTRIBUTES lpProcessAttributes,
  LPSECURITY_ATTRIBUTES lpThreadAttributes,
  BOOL bInheritHandles
  DWORD dwCreationFlags,
  LPVOID lpEnvironment,
  LPCTSTR lpCurrentDirectory,
  LPSTARTUPINFO lpStartupInfo,
  LPPROCESS_INFORMATION lpProcessInformation  );
```

Параметры:

lpApplicationName – указатель на строку с символов, содержащую полный путь и имя файла исполняемого модуля. Если значение равно NULL, то имя модуля берётся из командной строки lpCommandLine.

lpCommandLine – аргументы командной строки, используемые при создании нового процесса. Первый компонент строки должен содержать имя исполняемого файла (с расширением exe по умолчанию).

lpProcessAttributes и *lpThreadAttributes* − указатели на структуры атрибутов защиты процесса и потока SECURITY_ATTRIBlITES. Значениям NULL соответствует использование атрибутов защиты, заданных по умолчанию.

bInheritHandles − определяет наследование дескрипторов вызывающего процесса. Если этот параметр равен true, то каждый открытый дескриптор вызывающего процесса будет наследоваться новым процессом.

dwCreationFlags − устанавливает флаги, управляющие классом приоритета и созданием нового процесса (0 – нет флагов создания процесса).

Значение	Предназначение
CREATE_BREAKAWAY_FROM_JOB	Позволяет процессу, включенному в задание, создать процесс, отделенный от этого задания
CREATE_DEFAULT_ERROR_MODE	Новый процесс не наследует режимы обработки ошибок, установленные в родительском процессе.
CREATE_NEW_CONSOLE	Новый процесс создает новую консоль, вместо наследования консоли родителя (по умолчанию). Не может использоваться с DETACHED_PROCESS.
CREATE_NEW_PROCESS_GROUP	Новый процесс будет корневым процессом группы, включающей всех потомков этого процесса.
CREATE_NO_WINDOW	Запуск приложения без консольного окна
CREATE_SEPARATE_WOW_VDM	Используется при запуске 16-разрядных приложений Windows для использования Виртуальной Машины DOS (VDM).
CREATE_SHARED_WOW_VDM	Используется при запуске 16-разрядных приложений Windows.
CREATE_SUSPENDED	Позволяет создать процесс и одновременно приостанавливать его первичный поток. Это позволяет внести необходимые изменения в дочерний процесс до запуска его на выполнение функцией ResumeThread.
CREATE_UNICODE_ENVIRONMENT	Сообщает системе, что блок переменных окружения дочернего процесса должен содержать Unicode-символы. По умолчанию блок формируется на основе ANSI-символов
DEBUG_ONLY_THIS_PROCESS	В отличие от DEBUG_PROCESS может проводить отладка только в одном дочернем процессе — его прямом потомке.
DEBUG_PROCESS	Позволяет родительскому процессу проводить отладку всех порожденных процессов. Система уведомляет родительский процесс о возникновении событий в любом из дочерних процессов.

DETACHED_PROCESS	Для консольных приложений позволяет дочернему процессу перенаправлять вывод в новое консольное окно в более позднее время. Не может использоваться с CREATE_NEW_CONSOLE.

Параметр так же позволяет задавать и класс приоритета процесса (не рекомендуется). Обычно система присваивает новому процессу класс приоритета по умолчанию. Возможные следующие классы приоритета:

Приоритет	Класс приоритета
ABOVE_NORMAL_PRIORITY_CLASS	Выше обычного
BELOW_NORMAL_PRIORITY_CLASS	Ниже обычного
HIGH_PRIORITY_CLASS	Высокий приоритет. Обычно обозначает процесс, выполняющий критические по времени задачи. Используется с осторожностью, т.к. процесс может приостановить выполнение других процессов.
IDLE_PRIORITY_CLASS	Запускается если система неактивна и выгружается потоками любого процесса, запущенного с более высоким классом приоритета.
NORMAL_PRIORITY_CLASS	Обычный приоритет.
REALTIME_PRIORITY_CLASS	Самый высокий приоритет. Выгружает потоки всех других процессов, включая процессы операционной системы, выполняющие важные задачи..

lpEnvironment – указывает на блок параметров настройки окружения нового процесса. Блок конфигурации состоит из блока нуль-терминальных строк, представленных в форме: "*имя=значение*". Если параметр равен NULL, то новый процесс использует среду родительского процесса.

lpCurrentDirectory − указатель на строку символов, содержащую текущее устройство и каталог для нового процесса. Если параметр равен NULL, то новый процесс создаётся с тем же текущим устройством и каталогом, что и у вызывающего процесса.

lpStartupInfo – указатель на структуру STARTUPINFO, которая устанавливает оконный режим терминала, рабочий стол, стандартные дескрипторы и внешний вид главного окна для нового процесса.

Структура содержит следующие поля:

```
typedef struct _STARTUPINFO { i
DWORD cb;
LPTSTR lpReserved;
LPTSTR lpDesktop;
LPTSTR lpTitle;
DWORD dwX;
DWORD dwY;
DWORD dwXSize;
DWORD dwYSize;
DWORD dwXCountChars;
DWORD dwYCountChars;
DWORD dwFillAttribute;
DWORD dwFlags;
WORD wShowWindow;
WORD cbReserved2;
LPBYTE lpReserved2;
HANDLE hStdInput;
HANDLE hStdOutput;
HANDLE hStdError;
} STARTUPINFO, *LPSTARTUPINFO;
```

Элементы структуры:

Элемент	Описание
cb	Устанавливает размер структуры в байтах
lpReserved	Зарезервировано. Установить NULL
lpDesktop	Указатель на строку символов с нулем в конце, которой устанавливает или имя рабочего стола или, и имя оконной станции и рабочего стола для этого процесса. Наклонная черта влево (обратный слеш) в строке, обозначает, что строка включает в себя, и имя оконной станции и рабочего стола. Если имеет значение NULL, то новый процесс наследует рабочий стол и оконный терминал своего родительского процесса. Если имеет значение «пустая строка», то для нового процесса система выясняет, надо ли создавать новый рабочий стол и оконный терминал. Если пользователь уже имеет рабочий стол, система использует существующий рабочий стол.
lpTitle	Для консольных процессов – указатель на заголовок консольного окна. Если параметр имеет значение NULL, то в качестве заголовка используется имя исполняемого файла. Для графического интерфейса пользователя (GUI) параметр должен иметь значение NULL.
dwX *dwY*	Смещением по X или Y верхнего левого угла окна в пикселях (если dwFlags установлен в STARTF_USEPOSITION, иначе элементы игнорируются).

dwXSize *dwYSize*	Определяют ширину и высоту (в пикселах) окна приложения (если dwFlags установлен в STARTF_USESIZE, иначе элементы игнорируются).
dwXCountChars *dwYCountChars*	При создании нового консольного окна в консольном процессе, определяет ширину и высоту экранного буфера дисплея в столбцах и строках символов (если dwFlags установлен в STARTF_USECOUNTCHARS, иначе элементы игнорируются).
dwFillAttribute	Задает цвет текста и фона в консольных окнах дочернего процесса (если dwFlags установлен в STARTF_USEFILLATTRIBUTE, иначе элементы игнорируются). Определены любые комбинации значений: FOREGROUND_BLUE, FOREGROUND_GREEN, FOREGROUND_RED, FOREGROUND_INTENSITY, BACKGROUND_BLUE, BACKGROUND_GREEN, BACKGROUND_RED и BACKGROUND_INTENSITY. Белый устанавливается комбинацией RED, GREEN и BLUE.
dwFlags	Определяет, какие члены структуры STARTUPINFO будут использоваться в процессе создания окна (может состоять из одного или нескольких флагов). Кроме уже рассмотренных, могут использоваться следующие флаги: STARTF_RUN_FULLSCREEN − процесс запускается в полноэкранном режиме; STARTF_FORCEONFEEDBACK и STARTF_FORCEOFF FEEDBACK − позволяют контролировать форму курсора мыши в момент запуска нового процесса.
wShowWindow	Определяет вид первого перекрываемого окна дочернего процесса (если dwFlags установлен в STARTF_USESHOWWINDOW, иначе элементы игнорируются).
cbReserved2	Зарезервировано. Установить NULL
lpReserved2	Зарезервировано. Установить NULL
hStdInput *hStdOutput* *hStdError*	Стандартные дескрипторы ввода, вывода и ошибки для процесса (если dwFlags установлен в STARTF_USESTDHANDLES, иначе элементы игнорируются).

! Элементы структуры STARTUPINFO всегда необходимо инициализировать. Если приложение порождает процесс с атрибутами по умолчанию, то все поля должны быть нулевыми значениями, а в поле cb должен быть занесен размер структуры:

```
ZeroMemory( & STARTUPINFO, sizeof(STARTUPINFO));
STARTUPINFO.cb = sizeof(STARTUPINFO);
```

lpProcessInformation − указатель на структуру PROCESS_INFORMATION, в которую помещаются возвращаемые функцией значения дескрипторов и глобальных идентификаторов процесса.

Структура содержит следующие поля:

```
typedef struct _PROCESS_INFORMATION {
    HANDLE hProcess;  // Дескриптор процесса
    HANDLE hThread;   // Дескриптор первичного потока
```

```
    DWORD dwProcessId; // Идентификатор процесса
    DWORD dwThreadId;  // Идентификатор потока
};
```

GetStartupInfo

Получение копии структуры STARTUPINFO.

Прототип функции:

```
VOID GetStartupInfo(PSTARTUPINFO pStartupInfo);
```

Структура STARTUPINFO определяется процессом, который создавал вызывающий процесс.

ZeroMemory

Заполняет нулями блок памяти.

Прототип функции:

```
void ZeroMemory(PVOID Destination, SIZE_T Length)
```

Destination – адрес начала блока заполняемого нулями.

Length – длина блока заполняемого нулями.

ExitProcess

Завершение всех потоков процесса с указанным кодом возврата.

Прототип функции:

```
VOID ExitProcess(
    UINT uExitCode // Код возврата из процесса
);
```

! Необходимо всегда закрывать описатели дочернего процесса и его первичного потока, иначе, будет происходить утечка ресурсов.

TerminateProcess

Завершение одного процесса.

Прототип функции:

```
BOOL TerminateProcess(
HANDLE hProcess, // дескриптор процесса
UINT uExitCode // код возврата
);
```

Функция завершает работу процесса, но не освобождает все ресурсы, принадлежащие этому процессу. Поэтому эта функция должна вызываться только в аварийных ситуациях при зависании процесса.

GetCurrentProcess

Получение псевдодескриптора процесса. Используется в случае необходимости изменения характеристик процесса (например, приоритета).

Прототип функции:

```
HANLDE GetCurrentProcess(VOID);
```

Псевдодескриптор отличается от настоящего дескриптора процесса тем, что он может использоваться только текущим процессом и не может наследоваться другими процессами. Псевдодескриптор процесса не нужно закрывать после его использования.

SetPriorityCiass

Устанавливает класс приоритета для процесса. Это значение вместе со значением приоритета каждого потока процесса обуславливает базовый уровень приоритета каждого потока.

Прототип функции:

```
BOOL SetPriorityCiass(
HANDLE hProcess,        // Дескриптор процесса
DWORD dwPriorityClass // Приоритет
);
```

dwPriorityClass − класс приоритета для процесса. Может принимать следующие значения: ABOVE_NORMAL_PRIORITY_CLASS − приоритет выше обычного ; BELOW_NORMAL_PRIORITY_CLASS – приоритет ниже обычного; HIGH_PRIORITY_CLASS – приоритет для выполнения критических по времени задач; IDLE_PRIORITY_CLASS − запускается при отсутствии других задач; NORMAL_PRIORITY_CLASS – приоритет для большинства задач; REALTIME_PRIORITY_CLASS – самый высокий приоритет, останавливающий все другие процессы.

GetPriorityCiass

Извлечение приоритета процесса:

Прототип функции:

```
DWORD GetPriorityCiass(
HANDLE hProcess // Дескриптор процесса
);
```

При успешном завершении эта функция возвращает флаг установленного приоритета процесса, в противном случае возвращаемое значение равно нулю.

GetProcessPriorityBoost

Проверка возможности динамического повышения базовых приоритетов.

Прототип функции:

```
BOOL GetProcessPriorityBoost(
HANDLE hProcess, // дескриптор процесса
PBOOL pDisablePriorityBoost // состояние повышения приоритета
);
```

pDisablePriorityBoost − получает информацию о состоянии режима динамического повышения базовых приоритетов потоков. Если значение равно true, то режим динамического повышения базовых приоритетов потоков запрещен, иначе − разрешен.

WaitForSingleObject

Ожидание завершения порожденного процесса.

```
BOOL WaitForSingleObject(
HANDLE hHandle,
DWORD dwMilliseconds);
```

hHandle − дескриптор дочернего процесса.

dwMilliseconds − интервал ожидания в миллисекундах (значение INFINITE означает, что будет ожидаться завершение процесса).

2.2. Примеры

Пример 2.1. Создать процесс, запускающий приложение «Калькулятор» и ожидать завершения его работы.

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

int main()
{
   STARTUPINFO StartupInfo;
ZeroMemory( &StartupInfo, sizeof(StartupInfo));
StartupInfo.cb = sizeof(StartupInfo);

PROCESS_INFORMATION ProcInfo;
ZeroMemory(&ProcInfo, sizeof(ProcInfo));

   char AppName[] = "c:\\windows\\system32\\calc.exe";

if( !CreateProcess(AppName, NULL, NULL, NULL, FALSE, 0,
NULL, NULL, &StartupInfo, &ProcInfo))   return 0;

 WaitForSingleObject(ProcInfo.hProcess, INFINITE);

CloseHandle( ProcInfo.hProcess );
CloseHandle( ProcInfo.hThread );
}
```

Пример 2.2. Создать процесс, запускающий приложение «Калькулятор» и ожидать завершения его работы. Для запуска программы использовать второй параметр функции CreateProcess.

Программа отличается от примера 2.1. двумя операторами:

```
      char AppName[] = "calc.exe";
   if( !CreateProcess(NULL, AppName, NULL, NULL, FALSE, 0,
   NULL, NULL, &StartupInfo, &ProcInfo))   return 0;
```

В этой программе имя нового процесса и, при необходимости, его параметры передаются через командную строку. При использовании параметра lpCommandLine допускается указание только имени исполняемого файла. Если полный путь к файлу не указан, то система последовательно ищет необходимый файл в:

− текущем каталоге приложения;
− текущем каталоге родительского процесса;
− системном каталоге Windows;
− каталоге Windows;
− каталогах, перечисленных в переменной PATH среды окружения.

Пример 2.3. Запустить программу, реализующую бесконечный процесс увеличение значения переменной, начиная с числа 12, переданного как параметр командной строки. Завершить процесс по требованию пользователя.

Программа Endless, реализующая бесконечный процесс:

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

 int main(int n, char** s)
{
   int i=0;
     if (n>0) i=atoi(s[1]);
 for (;;Sleep(100)) cout <<  i++ << " ";
return 0;
}
```

Ниже приведена программа, которая создает этот процесс, а потом завершает при нажатии на клавишу ‘q’.

```
   #include <iostream>
   #include <windows.h>
   #include <conio.h>
     using namespace std;
   int main()
   {
      STARTUPINFO StartupInfo;
    ZeroMemory( &StartupInfo, sizeof(StartupInfo));
   StartupInfo.cb = sizeof(StartupInfo);

   PROCESS_INFORMATION ProcInfo;
       ZeroMemory(&ProcInfo, sizeof(ProcInfo));

      char AppName[] = "endless.exe 12";

   if(!CreateProcess(NULL, AppName, NULL, NULL, FALSE, 0,
```

```
    NULL, NULL, &StartupInfo, &ProcInfo))   return 0;

     cout << "Click 'q' to stop" << endl;

     char ch=NULL;
       while(ch != 'q') ch=getch();
         TerminateProcess(ProcInfo.hProcess,1);

  CloseHandle( ProcInfo.hProcess );
  CloseHandle( ProcInfo.hThread );
  return 0;
  }
```

2.3. Индивидуальные задания

Использовать файл с данными, созданный при выполнении л.р. № 1.

Написать программы для организации двух процессов MainProc и Daughter, предназначенные для корректировки данных в бинарном файле.

MainProc:

1. Открывает файл с данными и выводит их на экран.
2. Запрашивает номер структуры, которая должна быть изменена.
3. Запускает процесс Daughter в который через параметры командной строки передается информация о необходимой корректировке.
4. Ожидает завершения работы процесса Daughter.
5. Выводит откорректированные данные из файла на экран.
6. Завершает свою работу.

Daughter:

1. Выводит на экран информацию, полученную через командную строку.
2. Вносит необходимые изменения в файл.
3. Завершает свою работу.

ЛАБОРАТОРНАЯ РАБОТА № 3. ПОТОКИ

3.1. Потоки в Windows

Потоком (thread) в Windows называется объект ядра, которому операционная система выделяет процессорное время для выполнения приложения. В каждом процессе есть как минимум один, первичный (primary thread), поток. В адресном пространстве процесса могут одновременно выполняться несколько потоков.

CreateThread

Создание потока
Прототип функции:
HANDLE **CreateThread**(
LPSECURITY_ATTRIBUTES lpThreadAttributes,
DWORD dwStackSize,
LPTHREAD_START_ROUTINE lpStartAddress,
LPVOID lpParameter,
DWORD dwCreationFlags,
LPDWORD lpThreadld
Параметры:

lpThreadAttributes − устанавливает атрибуты защиты создаваемого потока. Если значение равно NULL, то операционная система сама установит атрибуты защиты потока, используя настройки по умолчанию.

dwStackSize определяет размер стека, который выделяется потоку при запуске. Система округляет это значение до самой близкой страницы памяти. Если этот параметр равен нулю, то потоку выделяется стек, размер которого по умолчанию равен 1 Мбайт. Каждый новый поток получает свое собственное пространство стека, состоящее, и из виртуальной памяти в файле подкачки, и зарезервированной памяти.

lpStartAddress указывает на исполняемую потоком функцию. Эта функция должна иметь следующий прототип:

DWORD WINAPI *имя_функции_потока*(LPVOID lpParameters);

Функции потока может быть передан единственный параметр lpParameter, который является указателем на пустой тип. Так как функции потоков вызываются операционной системой, то они также получили название функции обратного вызова.

dwCreationFiags определяет, в каком состоянии будет создан поток. Если значение этого параметра равно 0, то функция потока начинает выполняться сразу после создания потока. Если же значение этого параметра равно CREATE_SUSPENDED, то поток создается в режиме ожидания. Для запуска такого потока можно использовать функцию ResumeThread.

lpThreadid является указателем на переменную, в которую Windows помещает идентификатор потока. Этот идентификатор уникален для всей системы и

может в дальнейшем использоваться для ссылок на поток. Может быть установлен вы NULL.

При успешном завершении функция CreateThread возвращает дескриптор созданного потока и его идентификатор, который является уникальным для всей системы. В противном случае функция возвращает значение NULL.

Поток прекращает свое выполнение, в случае, если:

- функция потока возвращает управление вызывающей функции,
- поток самоуничтожается вызовом функции ExitThread (нежелательно);
- один из потоков текущего или стороннего процесса вызывает функцию TerminateThread (нежелательно);
- завершается процесс, содержащий данный поток (нежелательно).

ExitThread

Завершение потока.

Прототип функции:

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

Эта функция может вызываться как явно, так и неявно при возврате значения из функции потока. При выполнении этой функции система посылает динамическим библиотекам, которые загружены процессом, сообщение DLL_THREAD_DETACH, которое говорит о том, что поток завершает свою работу. Ресурсы C++ (например, объекты, созданные из C++-классов) не очищаются.

TerminateThread

Завершение потока.

Прототип функции:

```
BOOL TerminateThread(
HANDLE hThread, // дескриптор потока
DWORD dwExitThread // код завершения потока
);
```

Функция TerminateThread завершает поток, но не освобождает ресурсы, поэтому используется только в наиболее критических случаях.

SuspendThread

Приостановка выполнения потока.

Прототип функции:

```
DWORD SuspendThread(
   HANDLE hThread
);
```

Эта функция увеличивает значение счетчика приостановок на 1 и, при успешном завершении, возвращает текущее значение этого счетчика. В случае неудачи функция SuspendThread возвращает значение, равное -1.

Каждый созданный поток имеет счетчик приостановок, максимальное значение которого равно MAXIMUM_SUSPEND_COUNT. Счетчик приостановок показывает, сколько раз исполнение потока было приостановлено. Поток может исполняться только при условии, что значение счетчика приостановок равно нулю. В противном случае поток не исполняется.

Поток может приостановить и сам себя. Для этого он должен передать функции SuspendThread свой псевдодескриптор. Псевдодескриптор текущего потока отличается от настоящего дескриптора потока тем, что он может использоваться только самим текущим потоком и, следовательно, может наследоваться другими процессами.

GetCurrentThread

Получение псевдодескриптора.

Прототип функции:

HANLDE **GetCurrentThread**(VOID);

ResumeThread

Возобновления исполнения потока

Прототип функции:

```
DWORD ResumeThread(
  HANDLE hThread // дескриптор потока
);
```

Функция ResumeThread уменьшает значение счетчика приостановок на 1 при условии, что это значение было больше нуля. Если полученное значение счетчика приостановок равно 0, то исполнение потока возобновляется, в противном случае поток остается в состоянии ожидания. Если при вызове функции ResumeThread значение счетчика приостановок было равным 0, то функция не выполняет никаких действий.

При успешном завершении функция ResumeThread возвращает текущее значение счетчика приостановок, в противном случае — значение -1.

Sleep

Приостановка выполнения потока (процесса).

Прототип функции:

```
VOID Sleep(
  DWORD dwMiHiseconds // Миллисекунды
);
```

Параметр dwMiHiseconds определяет количество миллисекунд, на которые поток, вызвавший эту функцию, приостанавливает свое исполнение. Если значение этого параметра равно 0, то выполнение потока просто прерывается, а за-

тем возобновляется при условии, что нет других потоков, ждущих выделения процессорного времени. Если же значение этого параметра равно INFINITE, то поток приостанавливает свое исполнение навсегда, что приводит к блокированию работы приложения.

3.2. Примеры

Пример 3.1. Запустить поток, который подсчитывает факториал числа *n*.

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

DWORD WINAPI F(LPVOID n)
{
 cout << "Potok started." << endl;
    int m = (int) n;
int s=1;
       for (int i=1; i<=m; i++) s*=i;
 cout << "Factorial " << m << " = " << s << endl;
 cout << "Potok finished." << endl;
return 0;
}

int main ()
{
int n;
cout << " Vvedie n: " ;
cin >> n;
HANDLE hThread;
       // Запуск потока F
hThread = CreateThread (NULL, 0, F, (void*)n, 0, NULL);
if (hThread == NULL) return GetLastError();
       // Ожидание завершение работы потока F
WaitForSingleObject(hThread, INFINITE);
       // Закрытие потока F
CloseHandle(hThread);
   cin.get();
return 0;
}
```

В MVC++ 6.0 для отладки многопоточного приложения следует сделать следующие настройки: Project → Settings →ProjectSettings → C/C→ Code Generation, в списке Use run-time library выбрать Debug Multithreaded.

Пример 3.2. Создать поток увеличивающий значение переменной целого типа, из основной программы запрашивать текущее значение этой переменной. Поток прекращает свою работу при вызове TerminateThread.

```
#include <iostream>
#include <windows.h>
using namespace std;
   int n=0;

void thread()
{
  while (true)    {  n++; Sleep(100); }
return;
}

int main()
{
HANDLE hThread;
DWORD IDThread;
char c;
hThread = CreateThread(NULL, 0, (LPTHREAD_START_ROUTINE) thread,
NULL,0, &IDThread);
if (hThread == NULL) return GetLastError();
do
{
cout << "Click 'w' to display, 'q' to exit" << endl;
cin >> c;
 if (c == 'w') cout << "n= " << n << endl;
} while(c != 'q');
   TerminateThread(hThread, 0); // Прерывание выполнения потока thread
   CloseHandle(hThread); // Закрытие дескриптора потока
  return 0;
}
```

Пример 3.3. Создать поток увеличивающий значение переменной целого типа. Предусмотреть возможность приостановки и возобновления выполнения потока, с выводом на экран значения счетчика приостановок.

```
#include <iostream>
#include <windows.h>
using namespace std;
int n=0, dwn;
void thread()
{
while(true)   {
    n++;     Sleep(100);
           }
}

int main ()
{
```

```
HANDLE hTh;
DWORD IDTh;
char ch;
hTh = CreateThread(NULL, 0, (LPTHREAD_START_ROUTINE) thread, NULL, 0, &IDTh);
if (hTh == NULL) return GetLastError();

do
{
cout << "Enter :" << endl;
cout << " 'q'  Exit" << endl;  // Выход
cout << " 'n'  Display n" << endl;    // Вывод текущего значения n
cout << " 's'  Suspend" << endl;  // Приостановка выполнение потока
cout << " 'r'  Resume" << endl;   // Возобновление выполнения потока
cin >> ch;
switch (ch)
{
  case 'n': cout << "count = " << n << endl; break;
  case 's': dwn = SuspendThread(hTh); // Увеличение считчика приостановок
            cout << "Thread suspend count = " << dwn << endl;
            break;
  case 'r': dwn = ResumeThread(hTh); // Уменьшение считчика приостановок
            cout << "Thread suspend count = " << dwn << endl;
            break;
}
} while(ch!='q');
// Прерывание выполнения потока
TerminateThread (hTh, 0);
// Закрытие потока
CloseHandle(hTh);
}
```

Пример 3.4. Создать два потока. Один поток увеличивает значение счетчика. Второй поток отслеживает значение счетчика. После того, как значение счетчика превысит 30, второй поток останавливает выполнение первого потока.

```
#include <iostream>
#include <windows.h>
using namespace std;
int n=0;
HANDLE hTh01,  hTh02;

void thread01() // Первый поток
{
while(true) {
    n++;  Sleep(1);
      }
}
```

```
void thread02()// Второй поток
{
while(true) {
Sleep(100);
if (n>30) {
   cout << "n=" << n << " - thread 1 terminated";
      TerminateThread (hTh01, 0);
   return;
         }
}
}

int main ()
{
DWORD  IDTh01;
hTh01 = CreateThread(NULL, 0, (LPTHREAD_START_ROUTINE) thread01,
NULL, 0, &IDTh01);
if (hTh01 == NULL) return GetLastError();
   DWORD  IDTh02;
hTh02 = CreateThread(NULL, 0, (LPTHREAD_START_ROUTINE) thread02,
NULL, 0, &IDTh02);
if (hTh02 == NULL) return GetLastError();
   cin.get();
// Закрытие потоков
CloseHandle(hTh01);
CloseHandle(hTh02);
}
```

3.3. Индивидуальные задания

Написать программу, запускающую три дочерних потока. Каждый поток увеличивает (начиная с 0) значение счетчика с разной скоростью (использовать функцию sleep). При нажатии на клавишу ‘q’ закрыть все потоки и завершить выполнение программы.

1. Вывести на экран номер потока, значение счетчика которого первым достигнет значения 50.
2. Приостановить выполнение первого потока, на время, пока счетчик второго потока не достигнет значения 80.
3. Закрыть два потока, счетчики которых первыми достигли значения 100.
4. При нажатии на клавишу ‘d’ удалить поток, имеющий самое малое значение счетчика.
5. При нажатии на клавишу ‘v’ вывести на экран номер потока, который имеет наибольшее значение счетчика.
6. При нажатии на клавишу ‘s’ удалить поток, имеющий наибольшее значение счетчика.

7. Приостановить выполнение первого потока, пока счетчики остальных потоков не достигнут значения 100.

8. Выводить на экран номера потоков, значения счетчиков которых кратны 30.

9. Приостановить на 3 секунды выполнение потока, счетчик которого первым достигнет значения 120.

10. Вывести, в какой очередности счетчики потоков достигнут значения 100.

11. Остановить поток, счетчик которого последним достигнет значения 120.

12. Приостановить выполнение первого потока, пока счетчики второго и третьего не достигнут значения 80.

13. Остановить второй поток в момент, когда произведение счетчиков первого и третьего потоков станет равным 5000.

14. Остановить выполнение потоков при постижении их счетчиками значения 100.

15. Приостановить выполнение первого поток при достижении счетчиком второго потока значения 100 и возобновить выполнение при достижении счетчиком третьего потока значения 300.

ЛАБОРАТОРНАЯ РАБОТА № 4.
СИНХРОНИЗАЦИЯ ПОТОКОВ
В ПОЛЬЗОВАТЕЛЬСКОМ РЕЖИМЕ

Потоки должны взаимодействовать друг с другом в случае совместного использования общих ресурсов, а так же, при необходимости, уведомлять другие потоки о завершении определенных операций.

4.1. Блокирующие функции

Для блокирования доступа к используемым данным со стороны других потоков можно использовать специальные блокирующие функции (interlocked functions). Блокирующие функции выполняют одну или несколько элементарных операций, которые объединяются в одну неделимую операцию, называемую атомарной операцией. Все блокирующие функции требуют, чтобы адреса переменных были выровнены на границу слова, т. е. были кратны 32 (типы long, unsigned long, LONG, ULONG, DWORD).

InterlockedExchange

Замена значения переменной.

Прототип функции:

```
LONG InterlockedExchange(
LPLONG lpTarget,
LONG lValue
);
```

lpTarget − адрес заменяемой переменной;

lValue − новое значение переменной;

Функция возвращает старое значение заменяемой переменной.

InterlockedCompareExchange

Условная замена значения переменной.

Прототип функции:

```
PVOID InterLockedCompareExchange(
PVOID *Destination, // адрес переменной, значение которой заменяется
PVOID Exchange, // новое значение переменной
PVOID Comperand // значение для сравнения
);
```

Destination − адрес переменной, значение которой заменяется;

Exchange − новое значение переменной;

Comperand − значение для сравнения. Замена происходит при совпадении старого значения переменной и значения, заданного параметром Comperand.

Функция возвращает старое значение заменяемой переменной.

InterLockedIncrement

Увеличение значения переменной на единицу.

Прототип функции:

```
LONG InterLockedIncrement(
LPLONG lpAddend
);
```

IpAddend – адрес переменной.

Функция возвращает новое значение переменной.

InterLockedDecrement

Уменьшение значения переменной на единицу.

Прототип функции:

```
LONG InterLockedDecrement
 (
LPLONG lpAddend
);
```

IpAddend – адрес переменной.

Функция возвращает новое значение переменной.

InterlockedExchangeAdd

Изменение значения переменной.

Прототип функции:

```
LONG InterlockedExchangeAdd (
LPLONG lpAddend, // адрес переменной, значение которой изменяется
LONG Increment // прибавляемое значение
);
```

lpAddend − адрес переменной;

Increment − прибавляемое значение.

Функция возвращает старое значение переменной.

4.3. Критические секции

Блокирующие функции предназначены для разделения доступа к одной переменной. При использовании более сложных структур данных могут быть использованы критические секции.

Критическая секция (critical section) — это участок кода, требующий монопольного доступа к общим данным.

В операционных системах Windows используется объект типа CRITICAL_SECTION, для работы которым определены следующие функции:

InitializeCriticalSection

Инициализация критической секции.

Прототип функции:

```
VOID InitializeCriticalSection(
```

```
LPCRITICAL_SECTION lpCriticalSection
);
```

EnterCriticalSection

Вход в критическую секцию.

Прототип функции:

```
VOID EnterCriticalSection(
LPCRITICAL_SECTION lpCriticalSection
);
```

TryEnterCriticalSection

Попытка входа в критическую секцию.

Прототип функции:

```
BOOL TryEnterCriticalSection(
LPCRITICAL_SECTION lpCriticalSection
);
```

Функция возвращает ненулевое значение, если поток вошел в критическую секцию или уже находится в ней и false в противном случае.

LeaveCriticalSection

Выход из критической секции.

Прототип функции:

```
VOID LeaveCriticalSection(
LPCRITICAL_SECTION lpCriticalSection
);
```

DeleteCriticalSection

Разрушение объекта критическая секция.

Прототип функции:

```
VOID DeleteCriticalSection(
LPCRITICAL_SECTION lpCriticalSection
);
```

Все функции имеют один параметр − *lpCriticalSection* (указатель на объект типа CRITICAL_SECTION).

После инициализации объекта типа CRITICAL_SECTION для входа в критическую секцию используется функция EnterCriticalSection, которая исключает одновременное использование ресурсов. После завершения работы с разделяемым ресурсом поток должен покинуть критическую секцию посредством вызова функции LeaveCriticalSection.

4.3. Примеры

Пример 4.1. Написать программу, отслеживающую количество входящих и выходящих из здания людей. Первый поток учитывает количество входящих

людей, а второй поток – количество выходящих. Данные задаются с помощью датчика случайных чисел. Результат выводить на экран один раз в секунду.

```
#include <iostream>
#include <windows.h>
#include <time.h>
  using namespace std;
LONG sum=0;

void thread01()
{
srand(time_t(NULL));
  while(true)
{
    Sleep(500);
    LONG k=rand()*10/(RAND_MAX);
   InterlockedExchangeAdd (&sum,k);
 }
}

void thread02()
{
srand(time_t(NULL)+10);
  while(true)
{
    Sleep(500);
    LONG k=rand()*10/(RAND_MAX);
       InterlockedExchangeAdd (&sum,-k);
 }
}

void thread03()
{
  while(true)
{
    Sleep(1000);
    cout << "Sum = "<< sum << endl;
 }
}

int main ()
{
HANDLE hTh01;  DWORD  IDTh01;
hTh01 = CreateThread(NULL, 0,
(LPTHREAD_START_ROUTINE) thread01, NULL, 0, &IDTh01);
```

```
   if (hTh01 == NULL) return GetLastError();
HANDLE hTh02;   DWORD  IDTh02;
hTh02 = CreateThread(NULL, 0,
(LPTHREAD_START_ROUTINE) thread02, NULL, 0, &IDTh02);
    if (hTh02 == NULL) return GetLastError();
HANDLE hTh03;   DWORD  IDTh03;
hTh03 = CreateThread(NULL, 0,
(LPTHREAD_START_ROUTINE) thread03, NULL, 0, &IDTh03);
    if (hTh03 == NULL) return GetLastError();
cin.get();
// Закрытие потоков
TerminateThread(hTh01, 0); CloseHandle(hTh01);
TerminateThread(hTh02, 0); CloseHandle(hTh02);
TerminateThread(hTh03, 0); CloseHandle(hTh03);
}
```

Пример 4.2. Работа несинхронизированных потоков.

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

     CRITICAL_SECTION CrS;

   void thread01()
   {
      for (int i = 0; i < 3; i++) {
         EnterCriticalSection(&CrS);
            for (int j = 0; j < 7; j++)  {
               cout << "0 " ;
               Sleep(100);
                                        }
               cout << endl;
         LeaveCriticalSection(&CrS);
     Sleep(10);
                                    }
   }

   void thread02()
   {
      for (int i = 0; i < 3; i++) {
         EnterCriticalSection(&CrS);
            for (int j = 0; j < 7; j++)  {
               cout << "1 " ;
               Sleep(100);
```

```
                                    }
            cout << endl;
        LeaveCriticalSection(&CrS);
  Sleep(10);
                                  }
}


int main ()
{
   InitializeCriticalSection(&CrS);

   HANDLE hTh01;  DWORD  IDTh01;
hTh01 = CreateThread(NULL, 0,
      (LPTHREAD_START_ROUTINE) thread01, NULL, 0, &IDTh01);
  if (hTh01 == NULL) return GetLastError();
HANDLE hTh02;  DWORD  IDTh02;
hTh02 = CreateThread(NULL, 0,
      (LPTHREAD_START_ROUTINE) thread02, NULL, 0, &IDTh02);
  if (hTh02 == NULL) return GetLastError();

WaitForSingleObject(thread01, INFINITE);
WaitForSingleObject(thread02, INFINITE);

cin.get();
DeleteCriticalSection(&CrS);
   // Закрытие потока
CloseHandle(hTh01);
CloseHandle(hTh02);
return 0;
}
```

4.4. Индивидуальные задания

Написать программу, запускающую два дочерних потока. Первый поток последовательно, с помощью датчика случайных чисел, изменяет элементы массива, числами из диапазона от -150 до 150. На экран, с интервалом 1 секунда, выводить текущее состояние элементов массива. Второй поток проверяет элементы массива и, при необходимости, их изменяет в соответствии условием соответствующего варианта.

1. Заменить отрицательные элементы нулями.
2. Элементы, имеющие нулевое значение, заменить любым ненулевым элементом массива.
3. Элементы, имеющие значение >100, заменить нулями.
4. Все положительные элементы, заменить на нулевые.

5. Обнулить все четные (по значению) элементы массива.
6. Обнулить все четные (по номеру) элементы массива.
7. Если первый элемент массива больше последнего, то поменять их местами.
8. На место нечетных (по значению элементов) установить значение 100.
9. Возвести все отрицательные элементы во вторую степень.
10. Вместо последнего элемента поставить значение суммы всех предыдущих элементов.
11. Если первый элемент равен нулю, то вместо него поставить ближайший положительный элемент.
12. Заменить на ноль значение элементов (кроме первого и последнего), у которых оба соседних элемента имеют отрицательные значения.
13. Заменить все двузначные числа нулями.
14. Заменить все трехзначные числа их остатком от целочисленного деления на 100.
15. Поместить в нулевую позицию среднее арифметическое всех остальных элементов массива.

ЛАБОРАТОРНАЯ РАБОТА № 5.
СИНХРОНИЗАЦИЯ ПОТОКОВ
ПРИ ПОМОЩИ МЬЮТЕКСОВ

5.1. Мьютексы

Для решения проблемы взаимного исключения между параллельными потоками, выполняющимися в контекстах разных процессов, в операционных системах Windows используется объект ядра ***мьютекс*** (взаимное исключение). Мьютекс может принадлежать только одному потоку (несигнальное состояние). В сигнальном (*свободном*) состоянии мьютекс находится, если не принадлежит ни одному потоку.

Потоки, ждущие сигнального состояния мьютекса, обслуживаются в порядке очереди "первый пришел — первый вышел".

Для работы со мьютексами используются следующие функции:

CreateMutex

Создание мьютекса.

Прототип функции:

```
HANDLE CreateMutex(
  LPSECURITY_ATTRIBUTES lpMutexAttributes,
  BOOL bInitialOwner,
  LPCTSTR lpName
);
```

lpMutexAttributes − атрибуты защиты. Если равно NULL (атрибуты защиты будут заданы по умолчанию), то доступ к мьютексу будет открыт для всех пользователей.

bInitialOwner – первый владелец мьютекса. Если значение равно *true,* то мьютекс переходит во владение потоку, который его создал, иначе мьютекс свободен.

lpName − имя мьютекса. Позволяет обращаться к мьютексу из других процессов. Если значением параметра lpName равно NULL, то система создает безымянный мьютекс.

В случае удачного завершения функция CreateMutex возвращает дескриптор созданного мьютекса, иначе − NULL.

! Мьютекс захватывается потоком посредством любой функции ожидания.

ReleaseMutex

Освобождения мьютекса.

Прототип функции:

```
BOOL ReleaseMutex (
  HANDLE hMutex // Дескриптор мьютекса
);
```

В случае успешного завершения функция ReleaseMutex возвращает ненулевое значение, иначе − false.

OpenMutex

Получение доступа к мьютексу.

Прототип функции:

```
HANDLE OpenMutex(
  DWORD dwDesiredAccess,
  BOOL bInheritHandle
  LPCTSTR lpName // Имя мьютекса
);
```

dwDesiredAccess − Доступ к мьютексу. Может принимать значения:

- MUTEX_ALL_ACCESS — полный доступ;
- SYNCHRONIZE — синхронизация. Поток может использовать мьютекс только в функциях ожидания, чтобы захватить мьютекс, или в функции ReieaseMutex для его освобождения

bInheritHandle − свойство наследования мьютекса. Если значение этого параметра равно true, то дескриптор открываемого мьютекса является наследуемым. В противном случае дескриптор не наследуется.

В случае успешного завершения функция OpenMutex возвращает дескриптор открытого мьютекса, иначе − NULL.

5.2. Пример

Пример 5.1. Имеется два потока, которые работают с разной скоростью. Первый поток выводит на экран три группы по десять единиц, а второй – три группы по десять нулей. Потоки, с помощью мьютекса, синхронизированы таким образом, что бы в одну строку выводились или все единицы или все нули.

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

int thread01() // Первый поток
{
  // Открытие мьютекса
HANDLE hMutex01 = OpenMutex(SYNCHRONIZE,
FALSE, "MyMutex");
  if (hMutex01 == NULL)
cout << "Open mutex01 failed" << GetLastError() << endl;
for (int i = 0; i < 3; i++) {
 WaitForSingleObject(hMutex01, INFINITE); // Захват мьютекса
for (int j = 0; j < 10; j++) {
```

```
cout << "1";
Sleep(220);
                }
cout << endl;
   ReleaseMutex(hMutex01);  // Освобождение мьютекса
                }
CloseHandle(hMutex01);
return 0;
}

int thread02() // Первый поток
{
  // Открытие мьютекса
HANDLE hMutex02 = OpenMutex(SYNCHRONIZE,
FALSE, "MyMutex");
 if (hMutex02 == NULL)
cout << "Open mutex02 failed" << GetLastError() << endl;
for (int i = 0; i < 3; i++) {
 WaitForSingleObject(hMutex02, INFINITE);// Захват мьютекса
for (int j = 0; j < 10; j++) {
cout << "0";
Sleep(100);
                }
cout << endl;
   ReleaseMutex(hMutex02);  // Освобождение мьютекса
                }
CloseHandle(hMutex02);
return 0;
}

int main ()
{
 // Создание мьютекса
  HANDLE hMutex = CreateMutex(NULL, FALSE, "MyMutex");
if (hMutex == NULL)
  cout << "Create mutex failed" << GetLastError() << endl;

HANDLE hTh01 = CreateThread(NULL, 0,
(LPTHREAD_START_ROUTINE) thread01, NULL, 0, NULL);
if (hTh01 == NULL) return GetLastError();

HANDLE hTh02 = CreateThread(NULL, 0,
(LPTHREAD_START_ROUTINE) thread02, NULL, 0, NULL);
 if (hTh02 == NULL) return GetLastError();
```

```
 cin.get();
// Закрытие мьютекса
 CloseHandle(hMutex);
// Закрытие потоков
CloseHandle(hTh01);
CloseHandle(hTh02);
return 0;
}
```

5.3. Индивидуальные задания

Написать программу, запускающую два дочерних потока. Первый поток с помощью датчика случайных чисел заполняет элементы массива, состоящего из 10 чисел. После заполнения всех элементов, второй поток изменяет элементы массива в соответствии с заданием, указанным в соответствующем варианте. Вывести на экран исходный и результирующий массивы. Задание выполнить для трех различных массивов. Потоки синхронизировать с помощью мьютекса.

1. Обнулить все четные (по номеру) элементы массива.
2. Если первый элемент массива больше последнего, то поменять их местами.
3. На место нечетных (по значению элементов) установить значение 100.
4. Возвести все отрицательные элементы во вторую степень.
5. Вместо последнего элемента поставить значение суммы всех предыдущих элементов.
6. Если первый элемент равен нулю, то вместо него поставить ближайший положительный элемент.
7. Заменить на ноль значение элементов (кроме первого и последнего), у которых оба соседних элемента имеют отрицательные значения.
8. Заменить отрицательные элементы нулями.
9. Элементы, имеющие нулевое значение, заменить любым ненулевым элементом массива.
10. Элементы, имеющие значение >100, заменить нулями.
11. Все положительные элементы, заменить на нулевые.
12. Обнулить все четные (по значению) элементы массива.
13. Заменить все двузначные числа нулями.
14. Заменить все трехзначные числа их остатком от целочисленного деления на 100.
15. Поместить в нулевую позицию среднее арифметическое всех остальных элементов массива.

ЛАБОРАТОРНАЯ РАБОТА № 6.
СИНХРОНИЗАЦИЯ ПОТОКОВ
ПРИ ПОМОЩИ СЕМАФОРОВ

6.1. Семафоры

Семафор может находиться только в двух состояниях: *сигнальном* (свободном), если его ***значение больше нуля*** и *несигнальном* (занятом). Потоки, ждущие сигнального состояния семафора, обслуживаются в порядке очереди.

Для работы с семафорами используются следующие функции:

CreateSemaphore

Создание семафора.

Прототип функции:

```
HANDLE CreateSemaphore(
LPSECURITY_ATTRIBUTES lpSemaphoreAttribute,
LONG lInitialCount,
LONG lMaximumCount,
LPCTSTR lpName // Имя семафора
);
```

lpSemaphoreAttributes − атрибуты защиты (можно устанавливать в NULL);

lInitialCount − начальное значение семафора (от 0 до lMaximumCount);

lMaximumCount − максимальное значение семафора.

В случае успешного завершения функция CreateSemaphore возвращает дескриптор семафора, иначе − NULL.

! Значение семафора уменьшается на 1 при каждом использовании функции ожидания WaitForSingleObject.

OpenSemaphore

Получение доступа к семафору.

Прототип функции:

```
HANDLE OpenSemaphore(
DWORD dwDesiredAccess,
BOOL bInheritHandle,
LPCTSTR lpName // Имя семафора
);
```

dwDesiredAccess – доступ к семафору. Используются следующие флаги (или их комбинация):

- SEMAPHORE_ALL_ACCESS — полный доступ к семафору. Поток может выполнять над семафором любые действия;
- SEMAPHORE_MODIFY_STATE — изменение состояния семафора. Поток может использовать функцию ReieaseSemaphore для изменения значения семафора;

- SYNCHRONIZE — синхронизация. Поток может использовать семафор только в функциях ожидания;

bInheritHandle – режим наследования (см. выше);

В случае успешного завершения функция OpenSemaphore возвращает дескриптор семафора, иначе − NULL.

ReleaseSemaphore

Увеличение значения семафора. Доступ к семафору можно открыть функциями CreateSemaphore или OpenSemaphore.

Прототип функции:

```
BOOL ReleaseSemaphore(
  HANDLE hSemaphore, // Дескриптор семафора
  LONG lReleaseCount, // положительное число, на которое
// увеличивается значение семафора
  LPLONG lpPreviousCount); // предыдущее значение
                                  // семафора (может быть NULL)
```

lReleaseCount – величина, на которую увеличивается значение счетчика;

lpPreviousCount – указатель на переменную, в которую записывается предыдущее значение счетчика. Если предыдущее значение сохранять не нужно, то параметр равен NULL.

При попытке увеличения значения счетчика на величину, превышающую заданный максимум функция значение счетчика не изменяется, а функция возвращает false. При успешном завершении функция возвращает true.

6.2. Пример

Пример 6.1. Имеется два потока, которые работают с разной скоростью. Первый поток заполняет элементы массива цифрами от 10 до 30. Второй поток выводит значения тех элементов массива, в которые уже помещены данные. Потоки синхронизированы с помощью семафора.

```
#include <iostream>
#include <windows.h>
using namespace std;
 HANDLE hSemaphore;
 int mas[20];

 int thread01() // Поток, заполняющий массив
{
for (int i = 0; i < 21; i++) {
mas[i] = i+10;
Sleep(100);
ReleaseSemaphore(hSemaphore,1,NULL);
                }
return 0;
}
```

```
int thread02() // Поток, извлекающий элементы массива
{
for (int i = 0; i < 21; i++) {
WaitForSingleObject(hSemaphore, INFINITE);
cout << "mas["<<i<<"]= "<< mas[i]<<endl;
                    }
return 0;
}

int main ()
{
  // Создание семаформа
    hSemaphore=CreateSemaphore(NULL, 0, 21, "MySemaphore");
if (hSemaphore == NULL) cout << "Create semaphore failed"
<< GetLastError() << endl;

HANDLE hTh01 = CreateThread(NULL, 0,
(LPTHREAD_START_ROUTINE) thread01, NULL, 0, NULL);
if (hTh01 == NULL) return GetLastError();

HANDLE hTh02 = CreateThread(NULL, 0,
(LPTHREAD_START_ROUTINE) thread02, NULL, 0, NULL);
if (hTh02 == NULL) return GetLastError();

 cin.get();
// Закрытие семафора
 CloseHandle(hSemaphore);
// Закрытие потоков
CloseHandle(hTh01);
CloseHandle(hTh02);
return 0;
}
```

6.3. Индивидуальные задания

Написать программу, запускающую два дочерних потока, работающих с разной скоростью. Первый поток с помощью датчика случайных чисел заполняет элементы одномерного массива, состоящего из n элементов, числами из диапазона 0..500. Второй поток изменяет введенные элементы массива в соответствии с заданием, указанным в соответствующем варианте. Потоки синхронизировать с помощью семафора.

1. Найти среднее арифметическое значение нечетных (по номеру) элементов массива.

2. Найти среднее значение арифметическое значение положительных элементов массива.
3. Найти значение количества положительных элементов массива.
4. Обнулить все четные (по значению) элементы массива.
5. На место нечетных (по значению элементов) установить значение 1000.
6. Найти количество нечетных (по значению) элементов массива.
7. Обнулить все четные (по значению) элементы массива.
8. Умножить все двузначные числа на 100.
9. Заменить все трехзначные числа их остатком от целочисленного деления на 100.
10. Возвести все отрицательные элементы в третью степень.
11. Обнулить все четные (по номеру) элементы массива.
12. Найти количество неотрицательных четных элементов массива.
13. Найти значение минимального элемента массива.
14. Найти произведение отрицательных элементов массива.
15. Найти произведение четных (по значению) элементов массива.

Литература

Св. план

Учебное издание

Навроцкий Анатолий Александрович

Редактор
Корректор

Подписано в печать Формат 68x84 1/16. Бумага офсетная.
Гарнитура «Times» Печать ризографическая. Усл. печ. л.
Уч. изд. л. Тираж экз. Заказ №

Издатель и полиграфическое исполнение: Учреждение образования
«Белорусский государственный университет информатики
и радиоэлектроники»
ЛИ №02330/0056964 от 01.04.2004. ЛП № 02330/0131666 от 30.04.2004.
220013, Минск, П. Бровки, 6