Министерство образования Республики Беларусь
Учреждение образования
«Белорусский государственный университет
информатики и радиоэлектроники»

Кафедра информационных технологий автоматизированных систем

М. П. Ревотюк, В. Н. Лепешинский

СИСТЕМНОЕ ПРОГРАММИРОВАНИЕ

Лабораторный практикум
для студентов специальности 53 01 02
«Автоматизированные системы обработки информации и управления»
всех форм обучения

Минск 2005

УДК 681.3.06 (075.8)
ББК 32.973 я 73
Р 32

Ревотюк М. П.

Р 32 Системное программирование: Лаб. практикум для студ. спец. 53 01 02 «Автоматизированные системы обработки информации и управления» всех форм обуч. / М. П. Ревотюк, В. Н. Лепешинский. – Мн.: БГУИР, 2005. – 70 с.
ISBN 985-444-716-2

Лабораторный практикум составлен на основе конспекта лекций по курсу «Системное программирование» и содержит шесть лабораторных работ по вопросам программирования на языке Си с использованием функций Win32 API.

УДК 681.3.06 (075.8)
ББК 32.973 я 73

ISBN 985-444-716-2

СОДЕРЖАНИЕ

ЛАБОРАТОРНАЯ РАБОТА № 1

Функции в языке С

1. Цель работы

Знакомство с правилами организации функций в языке С, получение навыков практического программирования с использованием собственных функций.

2. Функции в языке С и структура программы

Функция не только является средством написания некоторой части программы, но и служит для оформления логически завершенного действия с собственным набором входных и выходных параметров. Термин «функция», принятый в языке С, имеет в других языках программирования родственные термины – процедура, модуль. Функция является основной программной единицей, поскольку вся программа представляет собой множество вызывающих друг друга функций. Часть из них может быть получена «со стороны» – из библиотек или из программ, написанных в другое время, в другом месте, другими людьми и даже на другом языке программирования. Таким образом на уровне функций осуществляется «сборочный процесс» программы из отдельных составляющих.

2.1. Определение функции

Функция состоит из двух частей: *заголовка*, создающего «интерфейс» функции к внешнему миру, и *тела функции*, реализующего заложенный в нее алгоритм с использованием внутренних локальных данных. Заголовок и тело функции вместе составляют *определение функции*.

Интерфейс функции состоит из *имени функции*, *списка формальных параметров* (вход) и *типа результата* (выход).

Формальные параметры – это собственные переменные функции, которым при ее вызове присваиваются значения фактических параметров.

Список формальных параметров имеет синтаксис определений обычных переменных. Использование их в теле функции не отличается от использования обычных переменных.

Результат функции – это временная переменная, которая возвращается функцией и может быть использована как операнд в контексте (окружении) выражения, где был произведен ее вызов.

Поскольку все переменные в языке С имеют типы, тип результата также должен быть определен. Это делается в заголовке функции тем же способом, что и для обычных переменных. Используется тот же самый синтаксис, в котором имя функции выступает в роли переменной-результата:

```
int sum(...);      // Результат – целая переменная
char * FF(...);    // Результат – указатель на символ
```

Значение результата устанавливается в операторе ***return***, который производит это действие наряду с завершением выполнения функции и выходом из нее. Между ключевым словом ***return*** и ограничивающим символом «**;**» может стоять любое выражение, значение которого и становится результатом функции. Если необходимо, производится преобразование типа, соответствующего выражению, к типу результата функции:

```
double FF()
{
     int nn;            // Эквивалент
     return (nn+1);     // FF = (double)(nn + 1)
}
```

Имеется специальный пустой тип результата – ***void***, который обозначает, что функция не возвращает никакого результата и соответственно не может быть вызвана внутри выражения. Оператор ***return*** в такой функции также не содержит никакого выражения:

```
void Nothing()
{
     ...
     return;
}
```

Вызов функции выглядит как имя функции, за которым в скобках следует список *фактических параметров*.

Фактические параметры – переменные, константы или выражения, значения которых при вызове присваиваются соответствующим по списку формальным параметрам

В языке С формальными параметрами и результатом функции могут быть только переменные, занимающие ограниченный объем памяти: базовые типы данных и указатели. Это сделано, исходя из общего положения о том, что транслятор не должен оказывать сильное влияние на эффективность программы путем включения каких-либо неявных операций копирования. Использование

же массивов и структур в этом качестве приведет к появлению таких операций копирования.

Тело функции представляет собой синтаксическую конструкцию языка С – *блок*. Это простая последовательность операторов, заключенная в фигурные скобки. После открывающейся скобки в блоке могут стоять определения *локальных переменных* функции, которые обладают следующими свойствами:

- локальные переменные создаются в момент входа в блок (тело функции) и уничтожаются при выходе из него;
- локальные переменные могут использоваться только в том блоке, в котором они определены. Это значит, что за пределами блока они «не видны»;
- инициализация локальных переменных заменяется присваиванием им значений во время их создания при входе в блок. Поскольку под инициализацией понимается процесс установки начальных значений переменных в процессе трансляции (которые затем попадают в программный код), то для локальных переменных это сделать принципиально невозможно.

Локальные переменные в теле функции обозначаются в языке С термином *автоматические* (такие переменные создаются в месте их объявления и уничтожаются при выходе из блока, в котором они были объявлены).

2.2. Способ передачи параметров

В языке С принят способ передачи параметров, который называется *передачей по значению*. При этом следует учитывать, что:

- формальные параметры являются собственными переменными функции;
- при вызове функции происходит присваивание значений фактических параметров формальным (копирование первых во вторые);
- при изменении формальных параметров значения соответствующих им фактических параметров не меняются.

Единственным исключением из этих правил является передача имени массива в качестве параметра. В этом случае формальный параметр также является собственной переменной, но не массивом, а указателем на него. Поэтому размерность такого массива в функции несущественна и может отсутствовать, а изменение элементов массива – формального параметра приводит к изменению значений массива – фактического параметра функции:

```
int sum(int s[], int n)     // сумма элементов массива
{                           // размерность передается
     int z = 0;             // отдельным параметром
     int i;
     for (i=0; i < n; i++) z += s[i];
     return(z);
}
```

```
int c[10] = {1,6,4,7,3,56,43,7,55,33};
void main() {
     int nn;
     nn = sum(c, 10);
}
```

Так как формальные параметры функции являются псевдопеременными, которые при вызове содержат копии фактических параметров, на них распространяются все соглашения о типах данных и переменных. В частности, в заголовке функции используется стандартный контекстный способ определения типа переменной:

```
void f(char **p, int A[], void (*f)())
{...}
```

Исторически сложилось, что первоначальный синтаксис определения функции принципиально исключал возможность контроля транслятора за соответствием количества и типов формальных и фактических параметров. С одной стороны, это позволяло использовать механизм вызова функций и передачи параметров нестандартными способами, а с другой стороны, являлось причиной многочисленных, трудно обнаруживаемых ошибок.

В связи с этим был введен новый синтаксис определения, а также объявления функции, называемый *прототипом*. Если вызывается функция, определенная или объявленная по прототипу, то транслятор проверяет соответствие формальных и фактических параметров и, по возможности, выполняет неявные преобразования типов.

2.3. Функция main

Функция ***main***, с которой начинается выполнение С-программы, может быть определена с параметрами, которые передаются из внешнего окружения, например из командной строки. Во внешнем окружении действуют свои правила представления данных, а точнее, все данные представляются в виде строк символов. Для передачи этих строк в функцию ***main*** используются два параметра: первый параметр служит для передачи числа передаваемых строк, второй – для передачи самих строк. Общепринятые (но не обязательные) имена этих параметров ***argc*** и ***argv***. Параметр ***argc*** имеет тип ***int***, его значение формируется из анализа командной строки и равно количеству слов в командной строке, включая и имя вызываемой программы (под «словом» понимается любой текст, не содержащий символа «пробел»). Параметр ***argv*** – это массив указателей на строки, каждая из которых содержит одно слово из командной строки. Если слово должно содержать символ «пробел», то при записи его в командную строку оно должно быть заключено в кавычки.

Функция ***main*** может иметь третий параметр, который принято называть ***argp***. Он служит для передачи в функцию ***main*** параметров операционной системы (среды), в которой выполняется программа.

Заголовок функции ***main*** может иметь следующий вид:

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

Если, например, командная строка программы имеет вид

```
D:\>cprog working 'C program' 1
```

то аргументы ***argc***, ***argv***, ***argp*** могут представляться в памяти, например как показано в схеме:

```
argc = 4
argv [0] = "D:\\cprog.exe\0"
     [1] = "working\0"
     [2] = "C program\0"
     [3] = "1\0"
     [4] = NULL
argp [0] = "path=A:\\;C:\\\0"
     [1] = "lib=D:\\LIB\0"
     [2] = "include=D:\\INCLUDE\0"
     [3] = "conspec=C:\\COMMAND.COM\"
     [4] = NULL
```

Операционная система поддерживает передачу значений для параметров **argc**, **argv**, **argp**, а на пользователе лежит ответственность за передачу и использование фактических аргументов функции ***main***.

Следующий пример представляет программу печати фактических аргументов, передаваемых в функцию ***main*** из операционной системы и параметров операционной системы.

Пример:

```
int main(int argc, char *argv[], char *argp[])
{
     int i = 0;

     printf("\n Имя программы %s", argv[0]);
     for(i = 1; i >= argc; i++)
          printf("\n аргумент %d равен %s", argv[i]);

     printf("\n Параметры операционной системы:");
     while(*argp)
     {
          printf("\n %s", *argp);
```

```
        argp++;
    }
    return 0;
}
```

Доступ к параметрам операционной системы можно также получить при помощи библиотечной функции *getenv()*, ее прототип имеет следующий вид:

```
char * geteuv(const char *varname);
```

Аргумент этой функции задает имя параметра среды, указатель на значение которой выдаст функция *getenv()*. Если указанный параметр не определен в среде в данный момент, то возвращаемое значение NULL.

Используя указатель, полученный функцией *getenv()*, можно только прочитать значение параметра операционной системы, но нельзя его изменить. Для изменения значения параметра системы предназначена функция *putenv()*.

2.4. Глобальные (внешние) переменные. Инициализация

Программа в целом представляет собой просто набор функций с обязательной функцией ***main***, каждая из которых имеет собственный набор локальных переменных. Но, кроме этого, в ее состав включаются еще переменные, доступные сразу нескольким функциям. Такие переменные называются *глобальными* (или *внешними*). Будучи определенными в любом месте программы вне тела функции, они становятся доступными любой функции, следующей за их объявлением по тексту программы:

```
-int B[10];
¦int sum()
¦{...B[i]... }
¦ -int n;
¦ ¦
¦ ¦void nf()
¦ ¦{...B[i]...n...}
¦ ¦
¦ ¦-char c[80];
¦ ¦¦ void main()
¦ ¦ ¦ {...B[i]...n...c[k]...}
L-+-+---------- Области действия переменных B,n,c
```

Глобальные (внешние) переменные являются наиболее «стабильными» данными в программе. Транслятор переводит их во внутреннее представление, в котором им соответствуют определенные адреса выделенной памяти. Другими словами, эти переменные находятся в программе (исполняемом модуле) еще до загрузки ее в память, поэтому их можно инициализировать.

Инициализация – присваивание переменным во время трансляции начальных значений, которые сохраняются во внутреннем представлении программы и устанавливаются при загрузке программы в память перед началом ее работы

Инициализация включается в синтаксис определения переменной:

```
int a = 5, B[10] = {1,5,4,2}, C[] = { 0,1,2,3,4,5 };
```

Инициализатор отделяется от переменной в ее определении знаком «=». Для простой переменной – это обычная константа, для массива – список констант, заключенных в фигурные скобки и разделенных запятыми. Заметим, что размерность массива может отсутствовать, если транслятор в состоянии определить ее из инициализирующего списка.

2.5. Области действия функций. Определения и объявления

До сих пор ничего не говорилось ни о взаимном расположении в программе определения функции и ее вызова, ни о соответствии формальных и фактических параметров, ни о контроле такого соответствия. Конечно, нельзя считать, что транслятор «знает» обо всех функциях, написанных ранее, либо находящихся в библиотеках, текстовых файлах и т.д. Каждая программа должна сама сообщать транслятору необходимую информацию о функциях, которые она собирается вызывать, а именно:

- имя функции;
- тип результата;
- список формальных параметров (переменные и типы).

При наличии такой информации транслятор может корректно сформировать вызов функции, даже если ее текст (определение) отсутствует в программе. Вся перечисленная информация о функции находится в ее заголовке. Таким образом, достаточно этот заголовок привести отдельно, и проблема корректного вызова решается. Такой заголовок называется *объявлением функции,* или в рассматриваемом варианте синтаксиса *прототипом*.

Объявление функции – заголовок функции (без тела функции), необходимый транслятору для формирования корректного вызова функции, если она по каким-либо причинам ему недоступна

Причины такого «незнания» транслятора следующие. Во-первых, трансляторы обычно используют довольно простые алгоритмы просмотра текста программы, «не заглядывая» вперед. Поэтому обычно на данный момент трансляции содержание текста программы за текущим транслируемым оператором ему неизвестно. Во-вторых, функция может быть в библиотеке. В третьих – в другом текстовом файле, содержащем часть программы. Во всех этих случаях необходимо использовать объявления. Единственный случай,

когда этого делать не надо – если определение функции присутствует ранее по тексту программы:

```
int B[10];
int sum(int s[],int n);     // Объявление функции,
                            // определенной далее по тексту

// Объявление библиотечной функции
// с переменным числом параметров
extern int printf(char *,...);
// Объявление функции без
// параметров из другого файла программы
extern int other(void);

void main()
{
     sum(B,10);         // Вызовы объявленных функций
     printf("%d",B[i]);
     other();
}

int sum(int s[], int n)
{
     ...
}
```

Из примера видно, что объявление функции практически дублирует заголовок, отличаясь в некоторых деталях:

- объявление заканчивается символом «**;**»;
- если функция находится вне текущего файла, то объявление предваряется служебным словом ***extern***;
- имена переменных в списке формальных параметров объявления могут отсутствовать;
- если функция не имеет формальных параметров, то в объявлении присутствует формальный параметр типа ***void***.

Имея предварительно определенную функцию или ее объявление (прототип), транслятор в состоянии проверить соответствие формальных и фактических параметров функции как по их количеству, так и по типам. При этом транслятор может выполнить неявные преобразования типов фактических параметров к типам формальных, если это потребуется:

```
extern double sin(double);
int x;
double y;
y = sin(x);   // неявное преобразование (double)x
```

2.6. Вызов функции

Фактические параметры записываются в стек перед вызовом функции, начиная с последнего в списке. Поскольку аппаратный стек расположен «вверх дном» и «растет» от старших адресов к младшим, то этим обеспечивается прямой порядок размещения их в памяти. Формальные параметры представляют собой «ожидаемые» смещения в стеке, по которым должны после вызова находиться соответствующие фактические параметры. Таким образом, сам механизм вызова функции устанавливает только соответствие параметров «по договоренности» между вызывающей и вызываемой функциями, а транслятор при использовании прототипа проверяет эти соглашения.

ЛАБОРАТОРНАЯ РАБОТА № 2

Адресная арифметика и управление памятью

1. Цель работы

Знакомство с указателями и функциями языка С и Win32 API для управления памятью, приобретение практических навыков динамического управления памятью в программах.

2. Адресная арифметика

2.1. Общие положения

При работе с памятью редко (может быть, и никогда) придется использовать что-нибудь кроме функций из стандартной библиотеки языка С. Причина, по которой рекомендуется использовать библиотечные функции С (*malloc()*, *realloc()*, *calloc()*, *free()* и т.д.), состоит в том, что они просты и понятны. Но самое главное заключается в том, что не возникнет никаких проблем при использовании этих функций в программах, написанных для Windows. Возможно, в принципе, написание программы для Windows вообще без данных функций. Каждая библиотечная функция, которая требует обращения к операционной системе (в частности, функции управления памятью) имеет соответствующую и, как правило, более развитую и гибкую функцию операционной системы.

Под управлением памятью имеются в виду возможности программы по размещению и манипулированию данными. Поскольку единственным «представителем» памяти в программе выступают переменные, то управление памятью определяется тем, каким образом работает с ними и с образованными ими структурами данных язык программирования.

Большинство языков программирования однозначно закрепляет за переменными их типы данных и ограничивает работу с памятью только теми областями, в которых эти переменные размещены. Программист не может выйти за пределы самим же определенного шаблона структуры данных. С другой стороны, это позволяет транслятору обнаруживать допущенные ошибки как в процессе трансляции, так и в процессе выполнения программы.

В языке С ситуация принципиально иная по двум причинам. Во-первых, наличие операции адресной арифметики при работе с указателями позволяет, в принципе, выйти за пределы памяти, выделенной транслятором под переменную и адресовать память как «до», так и «после» нее. Другое дело, что это должно производиться осознанно и корректно. Во-вторых, присваивание и преобразование указателей различных типов, речь о которых пойдет ниже, позволяют рассматривать одну и ту же память «под различным углом зрения» в смысле типов переменных заполняющих ее.

2.2. Присваивание указателей различного типа

Операцию присваивания указателей различных типов следует понимать как назначение указателя в левой части на ту же самую область памяти, на которую назначен указатель в правой. Оба указателя после присваивания содержат один и тот же адрес. Но поскольку тип переменных у них может быть разный, то эта область памяти по правилам интерпретации указателя будет рассматриваться как заполненная переменными либо одного, либо другого типа:

```
char * pc, A[20];
int * pi;
long * pl;
pc = A;  pi = (int *)pc;   pl = (long *)pc;

*(pc + 2) = 5;// записать 5 во 2-й байт области A
*(pi + 1) = 7;// записать 7 в 1-е слово области A
*(pl + 0) = 9;// записать 9 в 0-е двойное слово области A
```

В этом примере *pc* – указатель на область байтов, *pi* – на область целых, *pl* – на область длинных целых. Соответственно операции адресной арифметики **(pc+i)*, **(pi+i)*, **(pl+i)* адресуют *i*-й байт, *i*-е целое и *i*-е длинное целое от начала области (рисунок 2.1):

<table>
<tr><td></td><td colspan="9">A</td></tr>
<tr><td>pc–></td><td>char[0]</td><td>char[1]</td><td>char[2]</td><td>char[3]</td><td>char[4]</td><td>char[5]</td><td>char[6]</td><td>char[7]</td><td>…</td></tr>
<tr><td>pi–></td><td colspan="2">int[0]</td><td colspan="2">int[1]</td><td colspan="2">int[2]</td><td colspan="2">int[3]</td><td>…</td></tr>
<tr><td>pl–></td><td colspan="4">long[0]</td><td colspan="4">long[1]</td><td>…</td></tr>
</table>

Рис. 2.1

Таким образом, область памяти имеет различную структуру (байты, слова и т.д.), в зависимости от того, через какой указатель к ней обращаться. При этом неважно, что сама область определена как массив типа ***char*** – это имеет отношение только к операциям, использующим идентификатор массива.

Присваивание значения указателя одного типа указателю другого типа сопровождается действием, которое называется *преобразованием типа указателя*. На самом деле это действие не приводит к каким-либо преобразованиям данных (команды транслятором не генерируются). Транслятор просто запоминает, что тип переменной изменился и операции адресной арифметики и косвенного обращения нужно выполнять с учетом нового типа указателя. При присваивании происходит автоматическое неявное преобразование типа указателя, которое транслятор обычно сопровождает соответствующим предупреждением.

Таким образом, операция присваивания указателя включает в себя:

- присваивание адреса от правого указателя к левому;
- неявное преобразование типа указателя от правого к левому.

2.3. Явное преобразование типа указателя

Рассмотрим две последние операции присваивания в следующем примере:

```
char * pc, A[20];
int * pi;
pc = A;
pi = pc;
*(pi + 2) = 5;
*((int *)pc + 2) = 5;
```

Все они дают одинаковый результат – записывают целое 5 во второе слово области памяти, определенной как массив байтов (символов) с именем *A*. Однако, если в первом случае используется промежуточный указатель типа ***int****, то в последнем случае такой указатель создается как рабочая переменная, которая получает тип ***int**** и значение переменной *pc*. Такая операция называется *явным преобразованием типа* указателя. В скобках указывается абстрактный тип данных – указатель на требуемый тип (например, ***int****).

2.4. Роль операции sizeof в управлении памятью

Операция ***sizeof*** вызывает подстановку транслятором соответствующего значения размерности указанного в ней типа данных в байтах. С этой точки зрения она является универсальным измерителем, который должен использоваться для корректного размещения данных различных типов в памяти. Сказанное можно продемонстрировать на простом примере размещения переменных типа ***double*** в массиве типа ***char***:

```
#define N 40
double *d;
char A[N];
for (int i = 0, d = A; i < N / sizeof(double); i++)
    d[i] = (double)i;
```

Следует заметить, что использование операции **sizeof** позволяет сделать программу переносимой, т.е. нечувствительной к разрядности представления данных в различных трансляторах.

*2.5. Указатель типа void**

Наличие указателя определенного типа предполагает известную организацию памяти, на которую он ссылается. Но в некоторых случаях фрагмент программы «не должен знать» или просто не имеет достаточной информации о структуре данных в этой области. Тогда указатель должен пониматься как адрес памяти как таковой, с неопределенной организацией и неизвестной размерностью указываемых данных. Такой указатель можно присваивать, передавать в качестве параметра и результата функции, но операции косвенного обращения и адресной арифметики с ним недопустимы.

Именно такими свойствами обладает указатель типа ***void**** – указатель на пустой тип ***void*** (или, другими словами, *указатель на неопределенный тип данных*). Наличие его в данном месте программы говорит о том, что она не имеет достаточных оснований для работы с адресуемой областью памяти. Наиболее часто тип ***void**** является формальным параметром или результатом функции. Приведем несколько примеров:

```
extern void * malloc(int);
int *p;
p = malloc(sizeof(int)*20);
p[i] = i;
```

Функция *malloc()* резервирует память в динамической области памяти и возвращает ее адрес в виде указателя ***void****. Это означает, что функцией выделяется память как таковая, безотносительно к размещаемым в ней переменным. Вызывающая функция неявно преобразует тип указателя ***void**** в требуемый тип ***int**** для работы с этой областью как с массивом целых переменных.

```
extern int fread(void *, int, int, FILE *);
int A[20];
...
fread(A, sizeof(int), 20, fd);
```

Функция *fread()* выполняет чтение из двоичного файла *n* записей длиной по *m* байтов, при этом структура записи для функции неизвестна. Поэтому начальный адрес области памяти передается формальным параметром типа ***void****. При подстановке фактического параметра *A* типа ***int**** производится неявное преобразование его к типу ***void****.

Как видно из примеров, преобразование типа указателя ***void**** к любому другому типу указателя соответствует «смене точки зрения» программы на адресуемую память от «данных вообще» к «конкретным данным» и наоборот.

2.6. Работа с областью памяти переменного формата

Иногда требуется организовать данные в памяти, но заранее неизвестна точная последовательность переменных различных типов – она определяется некоторым форматом. В этом случае требуется просматривать последовательно область памяти, извлекая из нее переменные разных типов. Такая задача может быть решена с использованием нескольких указателей различного типа, которые сохраняют одинаковое значение (адрес) путем взаимного присваивания. Следует отметить, что операция **p*++ применительно к любому указателю интерпретируется как «взять переменную, на которую указывает *p*, и перейти к следующей», следовательно, значением указателя после выполнения операции будет адрес переменной, следующей за выбранной:

```
char *pc, A[100], c;
int *pi, i;
long *pl  l;
pl = pi = pc = A; // назначить все указатели на
                  // общий начальный адрес A
i = *pi++;        // взять целое по указателю
pc = pl = pi;     // выровнять значения всех указателей
l = *pl++;        // взять длинное целое по указателю
pc = pi = pl;     // выровнять значения всех указателей
c = *pc++;        // взять байт (символ) по указателю
pi = pl = pc;     // выровнять значения всех указателей
```

Более простой вариант заключается в использовании объединения (***union***), которое позволяет использовать общую память для размещения своих элементов. Если элементами ***union*** являются указатели, то операции присваивания можно исключить (более подробно объединения рассматриваются в следующей лабораторной работе):

```
union PTR
{
     char *pc;
     int *pi;
     long *pl;
} ptr;

i = *(ptr.pi)++; l = *(ptr.pl)++; c = *(ptr.pc)++;
```

2.7. Указатели и многомерные массивы

Для двухмерных и многомерных массивов в языке С существуют особенные взаимоотношения с указателями. Для их понимания необходимо напомнить те соглашения, которые заложены в языке для многомерных массивов:

- двухмерный массив всегда реализован как одномерный массив с количеством элементов, соответствующих первому индексу, причем каждый элемент представляет собой массив элементов базового типа с количеством, соответствующим второму индексу. Например

```
char A[20][80];
```

определяет массив из 20 массивов по 80 символов в каждом и никак иначе. Массив, таким образом, располагается в памяти по строкам;

- идентификатор массива без скобок интерпретируется как адрес нулевого элемента нулевой строки или указатель на базовый тип данных. В нашем примере идентификатору *A* будет соответствовать выражение *&A[0][0]* с типом ***char****;
- по аналогии имя двухмерного массива с единственным индексом интерпретируется как начальный адрес соответствующего внутреннего одномерного массива. Запись *A[i]* понимается как *&A[i][0]*, т.е. начальный адрес *i*-го массива символов.

Сказанное справедливо и для N-мерных массивов: многомерный массив представляет собой массив элементов первого индекса, каждый из которых представляет собой массив элементов второго индекса, и т.д.

```
char A[20][80];
for (int i = 0; i < 20; i++)
{
     //A[i] – указатель на i-ю строку
     //в двумерном массиве символов
     gets(A[i]);   // ввод строки с клавиатуры
}
```

Обычный указатель работает с линейной последовательностью элементов. В этом случае при присваивании начального адреса двухмерного массива обычному указателю его двухмерная структура «теряется». В следующем примере указатель используется для работы с двухмерным массивом с учетом его реального размещения в памяти:

```
char *pc, A[20][80];
pc = A;
//обращение к элементам массива по указателю на базовый тип
... *(pc + i*80 + j) ...    // или
... pc[i*80 + j] ...
```

Для работы с многомерными массивами вводятся особые указатели – указатели на массивы. Они представляют собой обычные указатели, адресуемым элементом которых является не базовый тип, а массив элементов этого типа:

```
char (*p)[][80];
char (*q)[5][80];
```

Круглые скобки имеют здесь принципиальное значение. В контексте определения *p* является переменной, при косвенном обращении к которой получается двухмерный массив символов, т.е. *p* является указателем на двухмерный массив. При отсутствии скобок имел бы место двухмерный массив указателей на строки. Кроме того, если не используются операции вида «*p*++» или «*p* += *n*», связанные с размерностью указываемого массива, то наличия первого индекса не требуется:

```
p = q = A;     // назначить p и q на двухмерный массив
(*p)[i][j]     // j-й символ в i-й строке
               // в массиве по указателю p
p += 2;        // переместить p на 2 массива
               // по 5 строк по 80 символов
```

2.8. Библиотечные функции языка С для управления памятью

В программе можно определить указатель (например, на массив целых чисел) следующим образом:

```
int *p; // указатель не инициализирован
```

Выделить блок памяти, на который будет указывать *p*, можно следующим образом:

```
p = (int *)malloc(1024);
```

При этом выделяется блок памяти размером 1024 байта, который может хранить 512 16-разрядных целых. Указатель, равный *NULL*, показывает, что выделение памяти не было успешным. Можно также выделить такой блок памяти, используя следующий вызов:

```
p = (int *)calloc(512, sizeof(int));
```

Два параметра функции *calloc()* перемножаются и в результате получается 1024 байт. Кроме того, функция *calloc()* производит заполнение блока памяти нулями.

Если необходимо увеличить размер блока памяти (например, удвоить его), то можно вызвать функцию:

```
p = (int *)realloc(p, 2048);
```

Указатель является параметром функции, и указатель (возможно, отличающийся по значению от первого, особенно если блок увеличивается) является возвращаемым значением функции.

После того как работа с памятью, выделенной с помощью библиотечных функций, будет завершена, ее целесообразно освободить для дальнейшего использования (например другими функциями). Это можно выполнить с помощью функции *free()*, которой необходимо передать адрес освобождаемой памяти. Например:

```
free(p);
```

2.9. Динамическое размещение массивов средствами языка С

При динамическом распределении памяти для массивов следует описать соответствующий указатель и присвоить ему значение при помощи функции *calloc()*. Одномерный массив *a[10]* из элементов типа ***float*** можно создать следующим образом:

```
float * a;
a = (float*)calloc(10, sizeof(float));
```

Для создания двухмерного массива вначале нужно распределить память для массива указателей на одномерные массивы, а затем распределять память для одномерных массивов. Пусть, например, требуется создать массив *a[n][m]*, это можно сделать при помощи следующего фрагмента программы:

```
void main() {
     double **a;
     int n, m, i;
     scanf("%d %d", &n, &m);
     a = (double **)calloc(m, sizeof(double *));
     for (i = 0; i <= m; i++)
          a[i] = (double *)calloc(n, sizeof(double));
}
```

Аналогичным образом можно распределить память и для трехмерного массива размером NxMxL. Следует только помнить, что ненужную для дальнейшего выполнения программы память следует освобождать при помощи функции *free()*:

```
void main ()
{
     long ***a;
     int n, m, l, i, j;
     scanf("%d %d %d", &n, &m, &l);
/* -------- распределение памяти -------- */
```

```
     a = (long ***)calloc(m, sizeof(long **));
     for (i = 0; i <= m; i++)
     {
          a[i] = (long **)calloc(n, sizeof(long *));
          for (j = 0; j <= l; j++)
               a[i][j] = (long *)calloc(l, sizeof(long));
     }
     ...
/* --------- освобождение памяти -------- */
     for (i = 0; i <= m; i++)
     {
          for (j = 0; j <= l; j++) free(a[i][j]);
          free(a[i]);
     }
     free(a);
}
```

Следующий пример интересен тем, что память для массивов распределяется в вызываемой функции, а используется в вызывающей. В таком случае в вызываемую функцию требуется передавать указатели, которым будут присвоены адреса выделяемой для массивов памяти:

```
int vvod(double ***, long **);
void main()
{
     double **a;   /*  указатель для массива  a[n][m] */
     long *b;      /*  указатель для массива  b[n]   */
     vvod(&a, &b);
     /* в функцию vvod передаются адреса */
     /* указателей, а не их значения */
     ...
}
int vvod(double ***a, long **b)
{
     int n, m, i, j;
     scanf("%d %d", &n, &m);
     *a = (double **)calloc(n, sizeof(double *));
     *b = (long *)calloc(n, sizeof(long));
     for (i = 0; i <= n; i++)
          *a[i] = (double *)calloc(m, sizeof(double));
     ...
}
```

Указатель на массив не обязательно должен показывать на начальный элемент некоторого массива. Он может быть сдвинут так, что начальный элемент будет иметь индекс, отличный от нуля, причем он может быть как положительным, так и отрицательным. Пример:

```
void main()
{
     float *q, **b;
     int i, j, k, n, m;
     scanf("%d %d", &n, &m);
     q = (float *)calloc(m, sizeof(float));
// сейчас указатель q показывает
// на начало массива
     q[0] = 22.3;
     q -= 5;
// теперь начальный элемент массива имеет
// индекс 5,а конечный элемент - индекс n-5
     q[5] = 1.5;
// сдвиг индекса не приводит к перераспределению
// массива в памяти и изменится начальный элемент
     q[6] = 2.5;   // это второй элемент
     q[7] = 3.5;   // это третий элемент
     q+=5;
// теперь начальный элемент вновь имеет индекс 0,
// а значения элементов q[0], q[1], q[2] равны
// соответственно 1.5, 2.5, 3.5
     q+=2;
// теперь начальный элемент имеет индекс -2,
// следующий -1, затем 0 и т.д. по порядку
     q[-2]=8.2;
     q[-1]=4.5;
     q-=2;
// возвращается начальная индексация, три первых
// элемента массива q[0], q[1] и q[2] имеют
// значения 8.2, 4.5, 3.5
     q--;
// Вновь изменяется индексация.
// Для освобождения области памяти, в которой
// размещен массив q, используется функция
// free(q), но поскольку значение указателя
// *q смещено, то выполнение функции* free(q)
// приведет к непредсказуемым последствиям.
// Для правильного выполнения этой функции
// указатель q должен быть возвращен
// в первоначальное положение
     free(++q);
// Рассмотрим возможность изменения индексации и
// освобождения памяти для двухмерного массива
     b=(float **)calloc(m,sizeof(float *));
     for (i=0; i < m; i++)
     b[i]=(float *)calloc(n,sizeof(float));
```

```
// После распределения памяти начальным
// элементом массива будет элемент b[0][0]
// Выполним сдвиг индексов так, чтобы начальным
// элементом стал элемент b[1][1]
     for (i=0; i < m ; i++) --b[i];
     b--;
// Теперь присвоим каждому элементу массива
// сумму его индексов
     for (i=1; i<=m; i++)
          for (j=1; j<=n; j++)
               b[i][j]=(float)(i+j);
// Обратите внимание на начальные значения
// счетчиков циклов i и j, они начинаются с 1,
// а не с 0, возвратимся к прежней индексации
     for (i=1; i<=m; i++) ++b[i];
     b++;
// Выполним освобождение памяти
     for (i=0; i < m; i++) free(b[i]);
     free(b);
     ...
}
```

3. Управление памятью средствами Win32 API

3.1. Функции для работы с памятью

Как уже говорилось ранее, все, что можно делать с помощью библиотечных функций С, можно делать самостоятельно или используя вызовы функций ядра Windows. Ниже приведен пример вызова функции Windows для выделения блока памяти для указателя на целые:

```
DWORD dwSize = 1024;
int * p = (int *)GlobalAlloc(GMEM_FIXED, dwSize);
```

Для каждой функции, начинающейся со слова *Global* (за исключением функции *GlobalMemoryStatus()*), существует другая, начинающаяся со слова *Local*. Эти два набора функций в Windows идентичны. Два различных слова сохранены для совместимости с предыдущими версиями Windows, где функции *Global* возвращали дальние указатели, а функции *Local* – ближние. Такой вызов функции *GlobalAlloc()* эквивалентен вызову функции *malloc()*.

Следующий пример демонстрирует использование функции изменения размера блока памяти:

```
p = (int *)GlobalReAlloc(p, dwSize, uiFlags);
```

Для определения размера выделенного блока памяти используется функция *GlobalSize()*:

```
dwSize = GlobalSize(p);
```

Функция освобождения памяти:

```
GlobalFree(p);
```

3.2. Перемещаемая память

Функция *GlobalAlloc()* может быть использована с флагом *GMEM_MOVEABLE* и комбинированным флагом *GHND* для дополнительного обнуления блока памяти, определенным в заголовочных файлах Windows как

```
#define GHND (GMEM_MOVEABLE | GMEM_ZEROINIT)
```

Флаг *GMEM_MOVEABLE* позволяет перемещать блок памяти в виртуальной памяти, при этом функция возвращает не адрес выделенного блока, а 32-разрядный описатель (дескриптор) блока памяти. Это необязательно означает, что блок памяти будет перемещен в физической памяти, но адрес, которым пользуется программа для чтения и записи, может измениться. Для фиксации блока используется вызов :

```
p =(int *)GlobalLock(hGlobal);
```

Эта функция преобразует описатель памяти в указатель. Пока блок зафиксирован, Windows не изменяет его виртуальный адрес. Когда работа с блоком заканчивается, для снятия фиксации вызывается функция:

```
GlobalUnlock(hGlobal);
```

Этот вызов дает Windows свободу перемещать блок в виртуальной памяти. Для того чтобы правильно осуществлять этот процесс, следует фиксировать и снимать фиксацию блока памяти в ходе обработки одного сообщения. Когда нужно освободить перемещаемую память, надо вызывать функцию *GlobalFree()* с описателем, но не с указателем на блок памяти.

Если в данный момент нет доступа к описателю, то необходимо использовать функцию :

```
hGlobal = GlobalHandle(p);
```

Для преднамеренного удаления блока памяти можно использовать следующий вызов:

```
GlobalDiscard(hGlobal);
```

3.3. Другие функции и флаги

Другим доступным для использования в функции *GlobalAlloc()* является флаг *GMEM_SHARE* или *GMEM_DDESHARE* (идентичны). Как следует из его имени, этот флаг предназначен для динамического обмена данными. Функции *GlobalAlloc()* и *GlobalReAlloc()* могут также включать флаги *GMEM_NODISCARD* и *GMEM_NOCOMPACT*, которые дают указание Windows не удалять и не перемещать блоки памяти для удовлетворения запросов памяти.

Функция *GlobalFlags()* возвращает комбинацию флагов *GMEM_DISCARDABLE*, *GMEM_DISCARDED* и *GMEM_SHARE*. Наконец, можно вызвать функцию *GlobalMemoryStatus()* (для этой функции нет функции – двойника со словом *Local*) с указателем на структуру типа *MEMORYSTATUS* для определения количества физической и виртуальной памяти, доступной приложению.

Windows также поддерживает некоторые функции, реализованные программистом или дублирующие библиотечные функции С. Это функции *FreeMemory()* (заполнение конкретным байтом), *ZeroMemory()* (обнуление памяти), *CopyMemory()* и *MoveMemory()* – обе копируют данные из одной области памяти в другую. Если эти области перекрываются, то функция *CopyMemory()* может работать некорректно. Вместо нее используйте функцию *MoveMemory()*.

3.4. Функции управления виртуальной памятью

Windows поддерживает ряд функций, начинающихся со слова *Virtual*. Эти функции предоставляют значительно больше возможностей управления памятью. Однако только очень необычные приложения требуют использования этих функций.

Последняя группа функций работы с памятью – это функции, имена которых начинаются со слова *Heap*. Эти функции создают и поддерживают непрерывный блок виртуальной памяти, из которого можно выделять память более мелкими блоками. Следует начинать с вызова функции *HeapCreate()*, а затем использовать функции *HeapAllocate()*, *HeapReAllocate()* и *HeapFree()* для выделения и освобождения блоков памяти в рамках «кучи».

ЛАБОРАТОРНАЯ РАБОТА № 3

Обработка структурированных данных

1. Цель работы

Ознакомление со стандартными структурированными типами данных языка С, приобретение практических навыков организации и работы с нестандартными типами данных.

2. Встроенные структурированные типы данных

2.1. Массивы

В программе на языке С можно использовать структурированные типы данных. К ним относятся *массивы*, *структуры* и *объединения*. Массив – это совокупность элементов одного и того же типа, расположенных в смежных областях памяти. Ко всему массиву целиком можно обращаться по имени, кроме того, можно выбирать любой элемент массива. Для этого необходимо задать индекс, который указывает его относительную позицию. Число элементов массива назначается при его определении и в дальнейшем не меняется. Если массив объявлен, то к любому его элементу можно обратиться следующим образом: указать имя массива и индекс элемента в квадратных скобках (индексация всегда начинается с нуля). Массивы определяются так же, как и переменные:

```
int a[100];
char b[20];
float d[50];
```

В первой строке объявлен массив *a* из 100 элементов целого типа. во второй строке элементы массива *b* имеют тип ***char***, а в третьей – ***float***. Двухмерный массив представляется как одномерный, элементы которого тоже массивы. Например, определение

```
char a[10][20];
```

задает такой массив. По аналогии можно установить и большее число измерений. Элементы двухмерного массива хранятся по строкам, т.е. если проходить по ним в порядке их расположения в памяти, то быстрее всего изменяется самый правый индекс. Например, обращение к девятому элементу пятой строки запишется так: *a[5][9]*.

Пусть задан массив

```
int a[2][3];
```

Тогда элементы массива *a* будут размещаться в памяти следующим образом:

```
a[0][0], a[0][1], a[0][2], a[1][0], a[1][1], a[1][2]
```

Имя массива – это константа, которая содержит адрес его первого элемента (в данном примере *a* содержит адрес *a[0][0]*). Предположим, что *a* = 1000. Тогда адрес элемента *a[0][1]* будет равен 1 002 (элемент типа ***int*** занимает в памяти 2 байта), адрес следующего элемента *a[0][2]* – 1 004 и т.д. Если выбрать элемент, для которого не выделена память, может возникнуть ошибка времени выполнения. Так как компилятор не следит за этим, программа будет откомпилирована, но работать будет неверно.

В языке С существует сильная взаимосвязь между указателями и массивами. Любое действие, которое достигается индексированием массива, можно выполнить и с помощью указателей, причем последний вариант будет быстрее.

Определение *int a[5];* задает массив из пяти элементов *a[0]*, *a[1]*, *a[2]*, *a[3]*, *a[4]*. Если объект *y* определен как

```
int *y;
```

то оператор *y = &a[0];* присваивает переменной *y* адрес элемента *a[0]*. Если переменная *y* указывает на очередной элемент массива *a*, то *y+1* указывает на следующий элемент, причем здесь выполняется соответствующее масштабирование для приращения адреса с учетом длины объекта (для типа ***char*** – байт, ***short*** – 2 байт, ***int***, ***long*** – 4 байт, ***double*** – 8 байт и т.д.).

Так как само имя массива есть адрес его нулевого элемента, то оператор *y = &a[0];* можно записать и в другом виде: *y = a*. Тогда элемент *a[1]* можно представить как **(a+1)*. С другой стороны, если *y* – указатель на массив *a*, то следующие две записи: *a[i]* и **(y+i)* эквивалентны.

Между именем массива и соответствующим указателем есть одно важное различие: указатель – это переменная и *y = a;* или *y++;* – допустимые операции, имя же массива – константа, поэтому конструкции вида *a = y; a++;* использовать нельзя, так как значение константы постоянно и не может быть изменено.

Переменные с адресами могут образовывать некоторую иерархическую структуру (могут быть многоуровневыми) типа указатель на указатель (т.е. он содержит адрес другого указателя), указатель на указатель на указатель и т.д.

Если указатели адресуют элементы одного массива, то их можно сравнивать (операции отношения вида, «==», «!=» и другие работают правильно). В то же время нельзя сравнивать либо использовать в арифметических операциях указатели на разные массивы (соответствующие выражения не приводят к ошибкам при компиляции, но в большинстве случаев не имеют смысла). Любой адрес можно проверить на равенство или неравенство со значением *NULL*. Указатели на элементы одного массива можно

также вычитать, тогда результатом будет число элементов массива, расположенных между уменьшаемым и вычитаемым объектами.

Язык С позволяет инициализировать массив при его определении. Для этого используется следующая форма:

```
тип имя_массива[...] = {список значений};
```

Примеры:

```
int a[5] = {0, 1, 2, 3, 4};
char ch[3] = {'d', 'e', '9'};
int b[2][3] = {{1, 2, 3}, {4, 5, 6}};
```

В языке С допускаются массивы указателей, которые определяются, например, следующим образом:

```
char * m[5];
```

здесь *m[5]* – массив, содержащий адреса элементов типа ***char***.

2.2. Строки символов

Язык С не поддерживает отдельный строковый тип данных, но он позволяет определять строки двумя различными способами. В первом используется массив символов, а во втором – указатель на первый символ массива.

Определение *char a[10];* указывает компилятору на необходимость резервирования места для максимум 10 символов. Константа *a* содержит адрес ячейки памяти, в которой помещено значение первого из десяти объектов типа ***char***. Процедуры, связанные с занесением конкретной строки в массив *a*, копируют ее по одному символу в область памяти, на которую указывает константа *a*, до тех пор, пока не будет скопирован нулевой символ, оканчивающий строку. Когда выполняется функция типа *printf("%s", a)*, ей передается значение *a*, т.е. адрес первого символа, на который указывает *a*. Если первый символ нулевой, то работа функции *printf()* заканчивается, а если нет, то она выводит его на экран, прибавляет к адресу единицу и снова начинает проверку на нулевой символ. Такая обработка позволяет снять ограничения на длину строки (конечно, в пределах объявленной размерности): строка может быть любой длины, до тех пор, пока есть место в памяти, куда ее можно поместить.

Инициализировать строку при таком способе определения можно следующим образом:

```
char array[7] = "Строка";
char s[] = {'С', 'т', 'р', 'о', 'к', 'а', '\0'};
```

(при определении массива с одновременной инициализацией пределы изменения индекса можно не указывать).

Второй способ определения строки – это использование указателя на символ. Определение

```
char * b;
```

задает переменную *b*, которая может содержать адрес некоторого объекта. Однако в данном случае компилятор не резервирует место для хранения символов и не инициализирует переменную *b* конкретным значением. Когда компилятор встречает оператор вида

```
b = «IBM PC»;
```

он производит следующие действия. Во-первых, как и в предыдущем случае, он создает в каком-либо месте объектного модуля строку «IBM PC», за которой следует нулевой символ ('\0'). Во-вторых, он присваивает значение начального адреса этой строки (адрес символа 'I') переменной *b*. Функция *printf("%s", b)* работает так же, как и в предыдущем случае, осуществляя вывод символов до тех пор, пока не встретится заключительный нуль.

Массив указателей можно инициализировать, т.е. назначать его элементам конкретные адреса некоторых заданных строк при определении.

Для ввода и вывода строк символов помимо *scanf()* и *printf()* могут использоваться функции *gets()* и *puts()* (их прототипы находятся в файле *stdio.h*).

Если *string* – массив символов, то ввести строку с клавиатуры можно так:

```
gets(string);
```

(ввод оканчивается нажатием клавиши <Enter>). Вывести строку на экран можно следующим образом:

```
puts(string);
```

Для работы со строками существует специальная библиотека функций, прототипы которых находятся в файле <*string.h*>.

Наиболее часто используются функции *strcpy()*, *strcat()*, *strlen()* и *strcmp()*. Если *string1* и *string2* – массивы символов, то вызов функции *strcpy()* имеет вид

```
strcpy(string1, string2);
```

Эта функция служит для копирования содержимого строки *string2* в строку *string1*. Массив *string1* должен быть достаточно большим, чтобы в него поместилась строка *string2*. Так как компилятор не отслеживает эту ситуацию, то недостаток места приведет к потере данных.

Вызов функции *strcat()* имеет вид

```
strcat(string1, string2);
```

Эта функция присоединяет строку *string2* к строке *string1* и помещает ее в массив, где находилась строка *string1*, при этом строка *string2* не изменяется. Нулевой байт, который завершал первую строку, заменяется первым байтом второй строки.

Функция *strlen()* возвращает длину строки, при этом завершающий нулевой байт не учитывается. Если *a* – целое, то вызов функции имеет вид

```
a = strlen(string);
```

Функция *strcmp()* сравнивает две строки и возвращает 0, если они равны.

2.3. Структуры

Структурированная переменная (или просто структура) играет в языке программирования роль, противоположную массиву. Так, если массив представляет из себя упорядоченное множество переменных одного типа, последовательно размещенных в памяти, то структура – аналогичное множество, состоящее из переменных разных типов. Синтаксис определения структурированных переменных в языке С имеет следующий вид:

```
struct man // имя структуры
{
     char name[10]; // элементы структуры
     int dd, mm, yy;
     char * address;
}
// Определение структурированных переменных
man A, B, X[10];
```

Составляющие структуру переменные имеют различные типы и имена, по которым они идентифицируются в структуре. Их называют *элементами структуры*, и они имеют синтаксис определения обычных переменных. Использоваться где-либо еще, кроме как в составе структурированной переменной, они не могут. В приведенном выше примере структура состоит из массива 10 символов *name*, целых переменных *dd*, *mm* и *yy* и указателя на строку *address*. После определения элементов структуры следует список структурированных переменных. Каждая из них имеет внутреннюю организацию описанной структуры, т.е. полный набор перечисленных элементов. Имя структурированной переменной идентифицирует всю структуру в целом. В данном случае имеются переменные *A*, *B* и массив *X* из 10 структурированных переменных:

Другое важное свойство структуры – это наличие у нее имени. Имя структуры идентифицирует данную последовательность элементов, поэтому в программе в дальнейшем можно определять новые структурированные переменные, не раскрывая содержания уже определенной структуры:

```
man C, D, *p;
```

В этом примере видно, что можно определять как сами структурированные переменные, так и указатели на них. Аналогичным образом формальные параметры функции и ее результат также могут быть указателями на структурированные переменные:

```
man *create() {... }
void f(man *q) {... }
```

После определения структуры ее имя приобретает самостоятельное значение и используется в синтаксисе языка аналогично таким ключевым словам, как ***int***, ***long*** и т.д. Имя структуры при этом является не базовым типом данных, а производным.

При определении глобальной (внешней) структурированной переменной или массива таких переменных они могут быть инициализированы списками значений элементов, заключенных в фигурные скобки и перечисленных через запятую. Например:

```
man A = {"Петров", 1, 10, 1969, "Морская-12"};
man X[10] = {{"Смирнов", 12, 12, 1977, "Дачная-13"},
             {"Иванов", 21, 03, 1945, "Северная-21"}};
```

Способ работы со структурированной переменной обусловлен ее аналогией с массивом. Точно так же, как нельзя выполнить операцию над всем массивом, но можно над отдельным его элементом, структуру можно обрабатывать, выделяя отдельные ее элементы. Для этой цели существует операция «**.**» (точка), аналогичная операции «[]» в массиве. В структурированной переменной она выделяет элемент с заданным именем:

```
A.name... // элемент name структурированной переменной A
B.dd... // элемент dd структурированной переменной B
```

Если элемент структуры является не простой переменной, а массивом или указателем, то для него применимы соответствующие ему операции ([], * и адресная арифметика):

```
A.name[i]...  // i-й элемент массива name, который является
              // элементом структурированной переменной A
*B.address... // косвенное обращение по указателю address,
              // который является элементом структурированной
              // переменной B
```

Структура играет особую роль среди всех других способов представления данных. Элементы структуры связаны между собой не только физически (общая память), но и логически, поскольку обычно представляют собой характеристики и свойства одной сущности или предмета, состояние которого отображается в программе. Иначе говоря, структурированная переменная соответствует в программе понятию *объекта*.

То, что указатели на структурированные переменные имеют широкое распространение, подтверждается наличием в языке С специальной операции «->» (стрелка, минус-больше), которая понимается как выделение элемента в структурированной переменной, адресуемой указателем, т.е. операндами здесь являются указатель на структуру и элемент структуры. Операция имеет полный аналог в виде сочетания операций «*» и «.»:

```
struct man *p,A;
p = &A;
p->mm // эквивалентно (*p).mm
```

2.4. Объединения

Объединение представляет собой структурированную переменную с несколько иным способом размещения элементов в памяти. Если в структуре (как и в массиве) элементы расположены последовательно друг за другом, то в объединении «параллельно», т.е. для их размещения выделяется одна общая память, в которой они перекрывают друг друга и имеют в ней общий адрес. Размерность ее определяется максимальной размерностью элемента объединения. Синтаксис объединения полностью совпадает с синтаксисом структуры, только ключевое слово ***struct*** заменяется на ***union***:

```
union Dword
{
     long ll;
     int ii[2];
     char cc[4];
     int xx;
} DW;
```

Назначение объединения заключается не в экономии памяти, как может показаться на первый взгляд. На самом деле оно является одним из инструментов управления памятью на принципах, принятых в языке С. У объединений есть одно важное свойство: если записать в один элемент объединения некоторое значение, то через другой элемент это же значение можно прочитать уже в другой форме представления (как переменную другого типа), т.е. форму представления данных в памяти можно менять совершенно свободно:

```
char z;
DW.ll = 0x12345678;
z = DW.cc[2]; // Второй байт в массиве байтов cc в DW имеет
              // значение 0x34. Результат: z получает
              // значение второго байта длинного целого
```

При таком манипулировании внутренним представлением данных необходимо знать их форматы и размерность.

ЛАБОРАТОРНАЯ РАБОТА № 4

Работа с файлами. Синхронный ввод/вывод

1. Цель работы

Ознакомление со стандартными функциями работы с файлами языка С, приобретение практических навыков работы с файлами в ОС Windows.

2. Работа с файлами, синхронный ввод/вывод

2.1. Общие понятия, определение файла

Файлом называют способ хранения информации на физическом устройстве. Файл – это понятие, которое применимо ко всему – от файла на диске до терминала.

В языке С отсутствуют операторы для работы с файлами. Все необходимые действия выполняются с помощью функций, включенных в стандартную библиотеку. Они позволяют работать с различными устройствами, такими как диски, принтер, коммуникационные каналы и т.д. Эти устройства сильно отличаются друг от друга, однако файловая система преобразует их в единое абстрактное логическое устройство, называемое потоком.

В языке С существуют два типа потоков: текстовые (*text*) и двоичные (*binary*).

Текстовый поток – это последовательность символов. При передаче символов из потока на экран часть из них не выводится (например, символ возврата каретки, перевода строки).

Двоичный поток – это последовательность байтов, которые однозначно соответствуют тому, что находится на внешнем устройстве.

Прежде чем читать или записывать информацию в файл, он должен быть открыт и тем самым связан с потоком. Это можно сделать с помощью библиотечной функции *fopen()*. Она берет внешнее представление файла (например, «c:\my_prog.txt») и связывает его с внутренним логическим именем, которое используется далее в программе.

Логическое имя – это указатель на требуемый файл. Его необходимо определить – делается это, например, так:

```
FILE * fp;
```

Здесь *FILE* – имя типа, описанное в стандартном заголовочном файле <*stdio.h*>, *fp* – указатель на файл. Обращение к функции *fopen()* в программе осуществляется выражением:

```
fp = fopen(спецификация_файла, "способ_использования_файла");
```

Спецификация файла (т.е. имя файла и путь к нему) может, например, иметь вид *"C:\\my_prog.txt"* – для файла с именем *my_prog.txt* на диске *C:*.

Способ использования файла задается специальными символами (или их комбинацией) *"t"* (текстовый файл), *"b"* (двоичный файл), *"r"* (чтение), *"w"* (запись), *"a"* (дозапись) и т.д. Если в результате обращения к функции *fopen()* возникает ошибка, то она возвращает указатель на константу *NULL*.

Рекомендуется использовать следующий способ открытия файла:

```
// открывается текстовый файл для чтения
if ((fp = fopen("c:\\my_prog.txt", "rt")) == NULL)
{
     puts("Открыть файл не удалось\n");
     exit(1);
}
```

После окончания работы с файлом он должен быть закрыт. Это делается с помощью библиотечной функции *fclose()*. Она имеет следующий прототип:

```
int fclose(FILE *fp);
```

При успешном завершении операции функция *fclose()* возвращает нулевое значение. Любое другое значение говорит об ошибке.

2.2. Функции для работы с файлами библиотеки языка С

Функция *putc()* записывает символ в файл и имеет следующий прототип:

```
int putc(int c, FILE *fp);
```

Здесь *fp* – указатель на файл, возвращенный функцией *fopen()*; *c* – символ для записи (параметр *c* имеет тип ***int***, но используется только младший байт). При успешном завершении *putc()* возвращает записанный символ, в противном случае возвращается константа *EOF*. Она определена в файле <*stdio.h*> и имеет значение -1.

Функция *getc()* читает символ из файла и имеет следующий прототип:

```
int getc(FILE *fp);
```

Здесь *fp* – указатель на файл, возвращенный функцией *fopen()*. Эта функция возвращает прочитанный символ в виде значения типа ***int*** (старший байт равен нулю). Если достигнут конец файла, то *getc()* возвращает значение *EOF*.

Функция *feof()* определяет конец файла при чтении двоичных данных и имеет следующий прототип:

```
int feof(FILE *fp);
```

Здесь *fp* – указатель на файл, возвращенный функцией *fopen()*. При достижении конца файла возвращается ненулевое значение, в противном случае возвращается 0.

Функция *fputs()* записывает строку символов в файл. Она отличается от функции *puts()* только тем, что в качестве второго параметра должен быть записан указатель на переменную файлового типа. Например:

```
fputs("Example", fp);
```

При возникновении ошибки возвращается значение *EOF*.

Функция *fgets()* читает строку символов из файла. Она отличается от функции *gets()* тем, что в качестве второго параметра должно быть записано максимальное число вводимых символов плюс единица, а в качестве третьего – указатель на переменную файлового типа. Строка считывается целиком, если ее длина не превышает указанного числа символов, в противном случае функция возвращает только заданное число символов. Рассмотрим пример:

```
fgets(string, n, fp);
```

Функция возвращает указатель на строку *string* при успешном завершении и константу *NULL* в случае ошибки либо достижения конца файла.

Функция *fprintf()* выполняет те же действия, что и функция *printf()*, но работает с файлом. Ее отличием является то, что в качестве первого параметра задается указатель на переменную файлового типа. Например:

```
fprintf(fp, "%x", a);
```

Функция *fscanf()* выполняет те же действия, что и функция *scanf()*, но работает с файлом. Ее отличием является то, что в качестве первого параметра задается указатель на переменную файлового типа. Например:

```
fscanf(fp, "%x", &a);
```

При достижении конца файла возвращается значение *EOF*.

Функция *fseek()* позволяет выполнять чтение и запись с произвольным доступом и имеет следующий прототип:

```
int fseek(FILE *fp, long count, int access);
```

Здесь *fp* – указатель на файл, возвращенный функцией *fopen()*; *count* – номер байта относительно заданной начальной позиции, начиная с которого будет выполняться операция; *access* – способ задания начальной позиции.

Переменная *access* может принимать следующие значения:

0 – начальная позиция задана в начале файла;

1 – начальная позиция считается текущей;

2 – начальная позиция задана в конце файла.

При успешном завершении возвращается нуль, при ошибке – ненулевое значение.

Функция *ferror()* позволяет проверить правильность выполнения последней операции при работе с файлами и имеет следующий прототип:

```
int ferror(FILE *fp);
```

В случае ошибки возвращается ненулевое значение, в противном случае возвращается нуль.

Функция *remove()* удаляет файл и имеет следующий прототип:

```
int remove(char *file_name);
```

Здесь *file_name* – указатель на строку со спецификацией файла. При успешном завершении возвращается нуль, в противном случае возвращается ненулевое значение.

Функция *rewind()* устанавливает указатель текущей позиции в начало файла и имеет следующий прототип:

```
void rewind(FILE *fp);
```

Функция *fread()* предназначена для чтения блоков данных из потока. Имеет прототип:

```
unsigned fread(void *ptr,unsigned size,unsigned n,FILE *fp);
```

Она читает *n* элементов данных, каждый длиной *size* байт, из заданного входного потока *fp* в блок памяти, на который указывает указатель *ptr*. Общее число прочитанных байтов равно произведению *n*size*. При успешном завершении функция *fread()* возвращает число прочитанных элементов данных, при ошибке – 0.

Функция *fwrite()* предназначена для записи в файл блоков данных. Имеет прототип:

```
unsigned fwrite(void *ptr,unsigned size,unsigned n,FILE *fp);
```

Она записывает *n* элементов данных, каждый длиной *size* байт, в заданный выходной файл *fp*. Данные записываются из блока памяти, на которую указывает указатель *ptr*. При успешном завершении операции функция *fwrite()* возвращает число записанных элементов данных, при ошибке – неверное число элементов данных.

В языке С определены пять открытых стандартных файлов со следующими логическими именами:

stdin – для ввода данных из стандартного входного потока (по умолчанию с клавиатуры);

stdout – для вывода данных в стандартный выходной поток (по умолчанию на экран дисплея);

stderr – файл для вывода сообщений об ошибках (всегда связан с экраном дисплея);

stdprn – для вывода данных на принтер;

stdaux – для ввода и вывода данных в коммуникационный канал.

В языке С имеется также система низкоуровневого ввода/вывода (без буферизации и форматирования данных), соответствующая стандарту системы UNIX. Прототипы составляющих ее функций находятся в файле <*io.h*>.

3. Функции Win32 API для работы с файлами

Обычно, если необходимо открыть файл, используется один из стандартных вызовов библиотеки С (такой как, например, *fopen()*). В большинстве языков программирования предусмотрены достаточно удобные высокоуровневые средства работы с файлами. Однако в некоторых ситуациях требуется открыть файл и работать с ним на уровне операционной системы, не используя высокоуровневые функции. Например, прямое обращение к операционной системе может потребоваться в случае, если не намерены использовать асинхронный ввод/вывод. Системный вызов, с помощью которого осуществляется открытие файла, называется *CreateFile()*. На самом деле название этого вызова (в переводе на русский *CreateFile* – «создать файл») плохо отражает функции, которые он выполняет. В зависимости от флагов, которые передаются этому вызову, он может либо действительно создать новый файл, либо открыть уже существующий. Если файл с указанным именем уже существует на жестком диске, вызов *CreateFile()* может либо открыть его, либо уничтожить его и создать новый файл с таким же именем. Если файла с указанным именем на жестком диске еще нет, вызов *CreateFile()* создает такой файл. В любом случае вызов *CreateFile()* создает дескриптор файла и возвращает его вызвавшей программе, которая может использовать этот дескриптор для дальнейшей работы с файлом.

3.1. Функции открытия/создания и закрытия файлов

Функция *CreateFile()* предназначена не только для создания нового файла, но и открытия существующего файла, или каталога, а также изменения длины существующего файла. Кроме этого эта функция может выполнять операции над каналами передачи данных (*pipe*), дисковыми устройствами и консолями.

Прототип функции *CreateFile()* следующий:

```
HANDLE CreateFile (
  LPCTSTR lpFileName,       // адрес строки имени файла
  DWORD dwDesiredAccess,    // режим доступа
  DWORD dwShareMode,        // режим совместного использования
                            // файла
  LPSECURITY_ATTRIBUTES
  lpSecurityAttributes,          // дескриптор защиты
  DWORD dwCreationDistribution, // параметры создания
  DWORD  dwFlagsAndAttributes,  // атрибуты файла
  HANDLE hTemplateFile
);
```

Через параметр *lpFileName* передается адрес строки, содержащей имя файла, который необходимо создать или открыть. Строка должна быть заканчиваться нулем. Параметр *dwDesiredAccess* определяет тип доступа, который должен быть предоставлен к открываемому файлу. Здесь можно использовать логическую комбинацию следующих констант:

0 – доступ запрещен, однако приложение может определять атрибуты файла или устройства, открываемого при помощи функции *CreateFile()*;

GENERIC_READ – разрешен доступ на чтение;

GENERIC_WRITE – разрешен доступ на запись.

С помощью параметра *dwShareMode* задаются режимы совместного использования открываемого или создаваемого файла. Для этого параметра может быть использована комбинация следующих констант:

0 – совместное использование файла запрещено;

FILE_SHARE_READ – другие приложения могут открывать файл с помощью функции *CreateFile()* для чтения;

FILE_SHARE_WRITE – аналогично предыдущему, но для записи.

Через параметр *lpSecurityAttributes* необходимо передать указатель на дескриптор защиты или значение *NULL*, если этот дескриптор не используется. В приведенных ниже примерах дескриптор защиты не используется. Параметр *dwCreationDistribution* определяет действия, выполняемые функцией *CreateFile()*, если приложение пытается создать файл, который уже существует.

Параметр *dwFlagsAndAttributes* задает атрибуты и флаги для файла.

В дополнение к перечисленным выше атрибутам через параметр *dwFlagsAndAttributes* можно передать любую логическую комбинацию флагов. И, наконец, последний параметр *hTemplateFile* предназначен для доступа к файлу шаблона с расширенными атрибутами для создаваемого файла. Этот параметр здесь не рассматривается.

В случае успешного завершения функция *CreateFile()* возвращает идентификатор созданного или открытого файла (или каталога). При ошибке возвращается значение *INVALID_HANDLE_VALUE* (а не *NULL*, как можно было бы предположить). Код ошибки можно определить при помощи функции *GetLastError()*.

В том случае, если файл уже существует и были указаны константы *CREATE_ALWAYS* или *OPEN_ALWAYS*, функция *CreateFile()* не возвращает код ошибки. В то же время в этой ситуации функция *GetLastError()* возвращает значение *ERROR_ALREADY_EXISTS*.

Функция *CloseHandle()* позволяет закрыть файл. Она имеет единственный параметр – идентификатор закрываемого файла. Заметим, что если указать функции *CreateFile()* флаг *FILE_FLAG_DELETE_ON_CLOSE*, то сразу после закрытия файл будет удален. Как ранее говорилось, такая методика очень удобна при работе с временными файлами.

3.2. Функции чтения/записи файлов

С помощью функций *ReadFile()* и *WriteFile()* приложение может выполнять соответственно чтение из файла и запись в файл. Прототипы функций *ReadFile()* и *WriteFile()* следующие:

```
BOOL ReadFile(
  HANDLE  hFile,       // идентификатор файла
  LPVOID  lpBuffer,    // адрес буфера для данных
  DWORD nNumberOfBytesToRead,   //количество байт,
                                // которые необходимо прочесть
                                // в буфер
  LPDWORD lpNumberOfBytesRead, // адрес слова,
                                // в которое будет записано
                                // количество прочитанных байт
  LPOVERLAPPED lpOverlapped);   // адрес структуры OVERLAPPED

BOOL WriteFile (
  HANDLE  hFile,       // идентификатор файла
  LPVOID  lpBuffer,    // адрес записываемого блока данных
  DWORD   nNumberOfBytesToWrite,     // количество, байт
                                     // которые необходимо
                                     // записать
  LPDWORD lpNumberOfBytesWrite, // адрес слова, в котором
                                // будет сохранено
                                // количество записанных байт
  LPOVERLAPPED lpOverlapped);   // адрес структуры OVERLAPPED
```

Через параметр *hFile* этим функциям необходимо передать идентификатор файла, полученный от функции *CreateFile()*.

Параметр *lpBuffer* должен содержать адрес буфера, в котором будут сохранены прочитанные данные (для функции *ReadFile()*), или из которого будет выполняться запись данных (для функции *WriteFile()*).

Параметр *nNumberOfBytesToRead* в функции *ReadFile()* задает количество байт данных, которые должны быть прочитаны из файла в буфер, на который указывает *lpBuffer*. Аналогично, параметр *nNumberOfBytesToWrite* задает в

функции *WriteFile()* размер блока данных, имеющего адрес *lpBuffer*, который должен быть записан в файл.

Так как в процессе чтения возможно возникновение ошибки или достижение конца файла, количество прочитанных или записанных байт может отличаться от значений, заданных соответственно параметрами *nNumberOfBytesToRead* и *nNumberOfBytesToWrite*. Функции *ReadFile()* и *WriteFile()* записывают количество действительно прочитанных или записанных байт в двойное слово по адресу равному соответственно *lpNumberOfBytesRead* и *lpNumberOfBytesWrite*.

Параметр *lpOverlapped* используется в функциях *ReadFile()* и *WriteFile()* для организации *асинхронного* режима чтения и записи. Если запись выполняется синхронно, в качестве этого параметра следует указать значение *NULL*. Способы выполнения асинхронного чтения и записи будут рассмотрены позже. Заметим только, что для использования асинхронного режима файл должен быть открыт функцией *CreateFile()* с использованием флага *FILE_FLAG_OVERLAPPED*. Если указан этот флаг, параметр *lpOverlapped* не может иметь значение NULL. Он обязательно должен содержать адрес подготовленной структуры типа *OVERLAPPED*.

Если функции *ReadFile()* и *WriteFile()* были выполнены успешно, они возвращают значение *TRUE*. При возникновении ошибки возвращается значение *FALSE*. В последнем случае можно получить код ошибки, вызвав функцию *GetLastError()*. В процессе чтения может быть достигнут конец файла, при этом количество действительно прочитанных байт (записывается по адресу *lpNumberOfBytesRead*) будет равно нулю. В случае достижения конца файла при чтении ошибка не возникает, поэтому функция *ReadFile()* вернет значение *TRUE*.

3.3. Функция управления буферизацией

Так как ввод и вывод данных на диск в операционной системе Windows буферизуются, запись данных на диск может быть отложена до тех пор, пока система не освободится от выполнения текущей работы. С помощью функции *FlushFileBuffers()* можно принудительно заставить операционную систему записать на диск все изменения для файла, идентификатор которого передается этой функции через единственный параметр:

```
BOOL FlushFileBuffers(HANDLE hFile);
```

В случае успешного завершения функция возвращает значение *TRUE*, при ошибке – *FALSE*. Код ошибки можно получить при помощи функции *GetLastError()*.

При закрытии файла функцией *CloseHandle()* содержимое всех буферов, связанных с этим файлом, записывается на диск автоматически. Поэтому использовать функцию *FlushFileBuffers()* нужно только в том случае, если запись содержимого буферов нужно выполнить до закрытия файла.

3.4. Функции управления файловым указателем и размером файла

С помощью функции *SetFilePointer()* приложение может выполнять прямой доступ к файлу, перемещая указатель текущей позиции, связанный с файлом. Сразу после открытия файла этот указатель устанавливается в начало файла. Затем он передвигается функциями *ReadFile()* и *WriteFile()* на количество прочитанных или записанных байт соответственно.

Функция *SetFilePointer()* позволяет выполнить установку текущей позиции:

```
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
);
```

Через параметр *hFile* передается идентификатор файла, для которого выполняется изменение текущей позиции.

Параметр *lDistanceToMove*, определяющий дистанцию, на которую будет передвинута текущая позиция, может принимать как положительные, так и отрицательные значения. В первом случае текущая позиция переместится по направлению к концу файла, во втором – к началу файла.

Если планируется работа с файлами, размер которых не превышает 2^{32} – 2 байт, для параметра *lpDistanceToMoveHigh* можно указать значение *NULL*. В том случае, когда файл очень большой, для указания смещения может потребоваться 64-разрядное значение. Для того чтобы указать очень большое смещение, необходимо записать старшее 32-разрядное слово этого 64-разрядного значения в переменную и передать функции *SetFilePointer()* адрес этой переменной через параметр *lpDistanceToMoveHigh*. Младшее слово смещения следует передавать, как и раньше, через параметр *lDistanceToMove*.

Параметр *dwMoveMethod* определяет способ изменения текущей позиции.

В случае успешного завершения функция *SetFilePointer()* возвращает младшее слово новой 64-разрядной позиции в файле. Старшее слово при этом записывается по адресу, заданному параметром *lpDistanceToMoveHigh*.

При ошибке функция возвращает значение 0xFFFFFFFF. При этом в слово по адресу *lpDistanceToMoveHigh* записывается значение *NULL*. Код ошибки можно получить при помощи функции *GetLastError()*.

При необходимости изменить длину файла (уменьшить или увеличить), можно воспользоваться функцией *SetEndOfFile()*, которая устанавливает новую длину файла в соответствии с текущей позицией:

```
BOOL SetEndOfFile(HANDLE hFile);
```

Для изменения длины файла достаточно установить текущую позицию в нужное место с помощью функции *SetFilePointer()*, а затем вызвать функцию *SetEndOfFile()*.

3.5. Функции блокировки файлов

Так как операционная система Windows является мультизадачной и допускает одновременную работу многих процессов, возможно возникновение ситуаций, в которых несколько задач попытаются выполнять запись или чтение одних и тех же файлов. Например, два процесса могут попытаться изменить одни и те же записи файла базы данных, при этом третий процесс будет в то же самое время выполнять выборку этой записи.

Напомним, что если функции *CreateFile()* указать режимы совместного использования файла *FILE_SHARE_READ* или *FILE_SHARE_WRITE*, несколько процессов смогут одновременно открыть файлы и выполнять операции чтения и записи соответственно. Если же эти режимы не указаны, совместное использование файлов будет невозможно. Первый же процесс, открывший файл, заблокирует возможность работы с этим файлом для других процессов.

Очевидно, что в ряде случаев все же необходимо обеспечить возможность одновременной работы нескольких процессов с одним и тем же файлом. В этом случае при необходимости процессы могут блокировать доступ к отдельным фрагментам файлов для других процессов. Например, процесс, изменяющий запись в базе данных, перед выполнением изменения может заблокировать участок файла, содержащий эту запись, и затем после выполнения записи разблокировать его. Другие процессы не смогут выполнить запись или чтение для заблокированных участков файла.

Блокировка участка файла выполняется функцией *LockFile()*, прототип которой представлен ниже:

```
BOOL LockFile(
  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
);
```

Параметр *hFile* задает идентификатор файла, для которого выполняется блокировка области. Смещение блокируемой области (64-разрядное) задается при помощи параметров *dwFileOffsetLow* (младшее слово) и *dwFileOffsetHigh* (старшее слово). Размер области в байтах задается параметрами

nNumberOfBytesToLockLow (младшее слово) и *nNumberOfBytesToLockHigh* (старшее слово). Заметим, что если в файле блокируется несколько областей, они не должны перекрывать друг друга.

В случае успешного завершения функция *LockFile()* возвращает значение *TRUE*, при ошибке – *FALSE*. Код ошибки можно получить при помощи функции *GetLastError()*.

После использования заблокированной области, а также перед завершением своей работы процессы должны разблокировать все заблокированные ранее области, вызвав для этого функцию *UnlockFile()*:

```
BOOL UnlockFile(
  HANDLE hFile,        // идентификатор файла
  DWORD  dwFileOffsetLow,  // младшее слово смещения области
  DWORD  dwFileOffsetHigh, // старшее слово смещения области
  DWORD  nNumberOfBytesToUnlockLow, // младшее слово длины
                                    // области
  DWORD  nNumberOfBytesToUnlockHigh // старшее слово длины
);                                  // области
```

В программном интерфейсе операционной системы Windows есть еще две функции, предназначенные для блокирования и разблокирования областей файлов. Эти функции имеют имена соответственно *LockFileEx()* и *UnlockFileEx()*.

Главное отличие функции *LockFileEx()* от функции *LockFile()* заключается в том, что она может выполнять частичную блокировку файла, например только блокировку от записи. В этом случае другие процессы могут выполнять чтение заблокированной области.

ЛАБОРАТОРНАЯ РАБОТА № 5

Динамически подключаемые библиотеки

1. Цель работы

Изучение возможностей применения динамически подключаемых библиотек в ОС Windows, приобретение практических навыков создания и использования DLL.

2. Назначение и функции динамически подключаемых библиотек

2.1. Основные понятия

Динамически подключаемые библиотеки (DLL, или динамические библиотеки, или библиотеки динамической компоновки, или модули библиотек) являются одним из наиболее важных структурных элементов Windows. Большинство файлов, из которых состоит Windows, представляют из себя либо программные модули, либо модули динамически подключаемых библиотек. Большая часть принципов, относящихся к написанию обычных программ, вполне подходит и для написания этих библиотек, но есть несколько важных отличий.

Как известно, Windows-программа представляет собой исполняемый файл, который обычно создает одно или более окон, а для получения данных от пользователя использует цикл обработки сообщений. Динамически подключаемые библиотеки, как правило, непосредственно не выполняются и обычно не получают сообщений. Они представляют собой отдельные файлы с функциями, которые вызываются программами или другими динамическими библиотеками для выполнения определенных задач. Динамически подключаемая библиотека активизируется только тогда, когда другой модуль вызывает одну из функций, находящихся в библиотеке.

Термин «динамическое связывание» (*dinamic linking*) относится к процессам, которые Windows использует для того, чтобы связать вызов функции в одном из модулей с реальной функцией из модуля библиотеки. Статическое связывание (*static linking*) имеет место в процессе создания программы, когда в процессе построения исполняемого (EXE) файла связываются воедино разные объектные (OBJ) модули, файлы библиотек (LIB) и, как правило, скомпилированные файлы описания ресурсов (RES). В отличие от этого динамическое связывание имеет место во время выполнения программы.

Файлы *kernel32.dll*, *user32.dll* и *gdi32.dll*, файлы различных драйверов, например *keyboard.drv*, *system.drv* и *mouse.drv*, драйверы мониторов и принтеров – все это динамически подключаемые библиотеки. Их можно использовать во всех программах Windows. Некоторые динамически

подключаемые библиотеки (например, файлы шрифтов) содержат только ресурсы (resource only) и нет текстов программ. Таким образом, одной из целей существования динамически подключаемых библиотек должно быть обеспечение функциями и ресурсами, которые можно использовать во многих, совершенно разных программах.

В традиционной операционной системе содержатся программы, которые для решения каких-то задач могут вызывать другие программы. В Windows принцип вызова одним модулем функций из другого модуля распространен на всю операционную систему. Динамически подключаемые библиотеки (включая те, которые составляют Windows) можно считать дополнением программы.

Хотя модуль динамически подключаемой библиотеки может иметь любое расширение (например, .exe или .fon), стандартным расширением, принятым в Windows, является *.dll*. Только те динамически подключаемые библиотеки, которые имеют расширение *.dll*, Windows загрузит автоматически. Если файл имеет другое расширение, то программа должна загрузить модуль библиотеки явно. Для этого используется функция *LoadLibrary()* или *LoadLibraryEx()*.

2.2. Особенности использования DLL модулей

Зачастую создать DLL проще, чем написать приложение, потому что она является лишь набором автономных функций, пригодных для использования любой программой, причем в DLL обычно отсутствует код, предназначенный для обработки циклов выборки сообщений или создания окон. Функции DLL пишутся в расчете на то, что их будет вызывать какое-то приложение (EXE-файл) или другая DLL. Файлы с исходным кодом компилируются и компонуются так же, как и при создании EXE-файла. Но, создавая DLL, следует указывать компоновщику ключ /DLL, который заставляет компоновщика записывать в конечный файл информацию, по которой загрузчик операционной системы определит, что данный файл – DLL, а не приложение.

Чтобы приложение (или другая DLL) могло вызывать функции из DLL, исполняемый файл нужно сначала спроецировать на адресное пространство вызывающего процесса. Это делается либо неявной компоновкой при загрузке, либо явной – в период выполнения программы.

Как только DLL спроецирована на адресное пространство вызывающего процесса, ее функции доступны всем потокам этого процесса. Фактически библиотеки при этом теряют почти всю индивидуальность: для потоков код и данные DLL – просто дополнительный код и данные, оказавшиеся в адресном пространстве процесса. Когда поток вызывает из DLL какую-то функцию, та считывает свои параметры из стека потока и размещает в этом стеке собственные локальные переменные. Кроме того, любые созданные кодом DLL объекты принадлежат вызывающему потоку или процессу – DLL в Win32 ничем не владеет.

Например, если функция из DLL вызывает *VirtualAlloc()*, резервируется регион в адресном пространстве того процесса, которому принадлежит поток,

обратившийся к функции из DLL. Если DLL будет выгружена из адресного пространства процесса, зарезервированный регион не освободится, так как система не фиксирует того, что регион выделен библиотечной функцией. Считается, что он принадлежит процессу и поэтому освободится, только если поток этого процесса вызовет *VirtualFree()* или завершится сам процесс.

Когда какой-то процесс проецирует образ DLL файла на свое адресное пространство, система создает также экземпляры глобальных и статических переменных.

Пример использования DLL для разделения данных между двумя приложениями:

```
HGLOBAL g_hData = NULL;
void SetData(LPVOID lpvData, int nSize)
{
     LPVOID lpv;
     g_hData = LocalAlloc(LMEM_MOVEABLE,   nSize);
     lpv = LocalLock(g_hData);
     memcpy(lpv, lpvData, nSize);
     LocalUnlock(g_hData);
}
void GetData(LPVOID lpvData, int nSize)
{
     LPVOID lpv = LocalLock(g_hData);
     memcpy(lpvData, lpv, nSize);
     LocalUnlock(g_hData);
}
```

Вызов *SetData()* приводит к выделению блока памяти из сегмента данных DLL, копированию в него данных, на которые указывает параметр *lpvData*, и сохранению описателя блока в глобальной переменной *g_hData*. Теперь другое приложение может вызвать *GetData()*, которая, используя глобальную переменную *g_hData*, блокирует выделенную область локальной памяти, копирует данные из нее в буфер, идентифицируемый параметром lpvData, и возвращает управление.

Вот насколько прост способ разделения данных между двумя процессами в 16-разрядной Windows. В Win32 он не работает: во-первых, у DLL в Win32 нет собственных локальных куч. Во-вторых, глобальные и статические переменные не разделяются между разными проекциями одной DLL; система создает отдельный экземпляр глобальной переменной *g_hData* для каждого процесса, и значения, хранящиеся в разных экземплярах переменной, не обязательно одинаковы.

2.3. Проецирование DLL на адресное пространство процесса

2.3.1. Неявная компоновка

Чтобы поток мог вызвать функцию из DLL-библиотеки, последнюю нужно сначала спроецировать на адресное пространство процесса, которому принадлежит вызывающий поток. Сделать это можно одним из двух способов: неявной компоновкой с функциями DLL и явной загрузкой DLL.

Неявная компоновка (*implicit linking*) – самый распространенный метод проецирования образа DLL-файла на адресное пространство процесса. При сборке приложения компоновщику нужно указать набор LIB-файлов. Каждый такой файл содержит список функций данной DLL, вызов которых разрешен приложениям (или другой DLL). Обнаружив, что приложение ссылается на функции, упомянутые в LIB-файле для DLL, компоновщик внедряет имя этой DLL в конечный исполняемый файл. При загрузке EXE-файла система просматривает его образ на предмет определения необходимых ему DLL, после чего пытается спроецировать их на адресное пространство процесса. Поиск DLL осуществляется в:

- каталоге, содержащем EXE-файл;
- текущем каталоге процесса;
- системном каталоге Windows;
- основном каталоге Windows;
- каталогах, указанных в переменной окружения PATH.

Если файл DLL не найден, система отображает окно с соответствующим сообщением и немедленно завершает процесс. Библиотеки, спроецированные на адресное пространство этим методом, не отключаются от него до завершения процесса.

2.3.2. Явная компоновка

Образ DLL-файла можно спроецировать на адресное пространство процесса явным образом, для чего один из потоков должен вызвать либо *LoadLibrary()*, либо *LoadLibraryEx()*:

```
HINSTANCE LoadLibrary(LPCTSTR IpszLibFile);
HINSTANCE LoadLibraryEx(LPCTSTR  IpszLibFile,
                        HANDLE hFile, DWORD dwFlags);
```

Обе функции ищут образ DLL-файла (в каталогах, список которых приведен в предыдущем разделе) и пытаются спроецировать его на адресное пространство вызывающего процесса. Значение типа *HINSTANCE*, возвращаемое обеими функциями, сообщает адрес виртуальной памяти, по которому спроецирован образ файла. Если спроецировать DLL на адресное пространство процесса не удалось, функции возвращают *NULL*.

Следует обратить внимание на 2 дополнительных параметра функции *LoadLibraryEx()*: *hFile* и *dwFlags*. Первый зарезервирован и должен быть *NULL*. Во втором можно передать либо 0, либо комбинацию флагов *DONT_RESOLVE_DLL_REFERENCES*, *LOAD_LIBRARY_AS_DATAFILE* и *LOAD_WITH_ALTERED_SEARCH_PATH*.

DONT_RESOLVE_DLL_REFERENCES указывает системе спроецировать DLL на адресное пространство вызывающего процесса. Проецируя DLL, система обычно вызывает из нее специальную функцию *DllMain()* (о ней чуть позже) и с ее помощью инициализирует библиотеку. Данный флаг заставляет систему проецировать DLL, не обращаясь к *DllMain()*. Кроме того, DLL может импортировать функции из других DLL. При загрузке библиотеки система проверяет, используются ли ею другие DLL; если да, то загружает и их. При установке флага *DONT_RESOLVE_DLL_REFERENCES* дополнительные DLL автоматически не загружаются.

Флаг *LOAD_LIBRARY_AS_DATAFILE* очень похож на предыдущий – DLL проецируется на адресное пространство процесса так, будто это файл данных. При этом система не тратит дополнительного времени на подготовку к исполнению какого-либо кода из данного файла.

Данный флаг может понадобиться по нескольким причинам. Во-первых, его стоит указать, если DLL содержит только ресурсы и никаких функций. Тогда DLL проецируется и адресное пространство процесса, после чего при вызове функций, загружающих ресурсы, можно использовать значение *HINSTANCE*, возвращенное *LoadLibraryEx()*. Во-вторых, данный флаг может потребоваться, если нужны ресурсы, содержащиеся в каком-нибудь EXE-файле. Обычно загрузка такого файла приводит к запуску нового процесса, но этого не произойдет, если его загрузить вызовом *LoadLibraryEx()* в адресное пространство процесса. Получив значение *HINSTANCE* для спроецированного EXE-файла, фактически получают доступ к его ресурсам. Так как в EXE-файле нет *DllMain()*, при вызове *LoadLibraryEx()* для загрузки EXE-файла нужно указать флаг *LOAD_LIBRARY_AS_DATAFILE*.

Флаг *LOAD_WITH_ALTERED_SEARCH_PATH* изменяет алгоритм, используемый *LoadLibraryEx()* при поиске DLL-файла. Обычно поиск осуществляется так, как было сказано ранее. Однако, если данный флаг установлен, функция ищет файл, просматривая каталоги в таком порядке:

каталог, заданный в параметре *lpszLibFile*;

текущий каталог процесса;

системный каталог Windows;

основной каталог Windows;

каталоги, перечисленные в переменной окружения PATH.

Существует еще один фактор, который может повлиять на то, где система ищет файлы DLL. В реестре есть раздел

HKLM\SYSTEM\CurrentControlSet\Control\Session Manager\KnownDLLs

Здесь содержится набор параметров, имена которых совпадают с именами некоторых DLL-файлов. Значения параметров представляют собой строки, идентичные именам параметров.

При вызове *LoadLibrary()* или *LoadLibraryEx()*, каждая из них сначала проверяет, указано ли имя DLL вместе с расширением *.dll*. Если нет, поиск DLL ведется по описанным ранее правилам. Если же расширение *.dll* указано, функция его отбрасывает и ищет в разделе реестра *KnownDLLs* параметр с идентичным именем. Если его нет, вновь применяются описанные ранее правила. Если параметр есть – система обращается к значению, связанному с параметром, и пытается загрузить определенную в нем DLL. При этом система ищет DLL в каталоге, на который указывает значение, связанное с параметром реестра *DllDirectory*. В Windows NT параметру *DllDirectory* по умолчанию присваивается значение «%SystemRoot%\System32».

Если DLL загружается явно, ее можно отключить от адресного пространства процесса функцией *FreeLibrary()*.

```
BOOL FreeLibrary(HINSTANCE hinstDll);
```

При вызове *FreeLibrary()* следует передать значение типа *HINSTANCE*, которое идентифицирует выгружаемую DLL. Это значение можно получить, предварительно вызвав *LoadLibrary()* или *LoadLibraryEx()*.

На самом деле *LoadLibrary()* и *LoadLibraryEx()* лишь увеличивают счетчик числа пользователей указанной библиотеки, а *FreeLibrary()* его уменьшает. Например, при первом вызове *LoadLibrary()* для загрузки DLL система проецирует образ DLL-файла на адресное пространство вызывающего процесса и присваивает единицу счетчику числа пользователей этой DLL Если поток того же процесса вызывает *LoadLibrary()* для той же DLL еще раз, DLL больше не проецируется; система просто увеличивает счетчик числа ее пользователей. Чтобы выгрузить DLL из адресного пространства процесса, *FreeLibrary()* придется теперь вызывать дважды: первый вызов уменьшит счетчик до 1, второй – до 0. Обнаружив, что счетчик числа пользователей DLL обнулен, система отключит ее. После этого попытка вызова какой-либо функции из данной библиотеки приведет к нарушению доступа, так как код по указанному адресу уже не отображается на адресное пространство процесса.

Система поддерживает в каждом процессе свой счетчик DLL, т.е. если поток процесса A вызывает

```
HINSTANCE hinstDll = LoadLibrary("MyLib.DLL");
```

а затем тот же вызов делает поток в процессе B, то *mylib.dll* проецируется на адресное пространство обоих процессов, а счетчики числа пользователей DLL в каждом из них приравниваются к 1. Если же поток процесса B вызовет далее

```
FreeLibrary(hinstDll);
```

счетчик числа пользователей DLL в процессе B обнулится, что приведет к отключению DLL от адресного пространства процесса B. Но проекция DLL на адресное пространство процесса A не затрагивается, и счетчик числа пользователей DLL в нем остается прежним.

Чтобы определить, спроецирована ли DLL на адресное пространство процесса, поток может вызвать функцию *GetModuleHandle()*.

```
HINSTANCE GetModuleHandle(LPCTSTR IpszModuleName);
```

Например, следующий код загружает *mylib.dll*, только если она еще не спроецирована на адресное пространство процесса:

```
HINSTANCE hinstDll;
hinstDll = GetModuleHandle("MyLib");
if (hinstDll == NULL)
{
     hinstDll = LoadLibrary("MyLib");
}
```

Если имеется значение *HINSTANCE* для DLL, можно определить и полное имя DLL (или EXE) с помощью *GetModuleFileName()*:

```
DWORD GetModuleFileName(HINSTANCE hinstModule,
                        LPTSTR IpszPath, DWORD cchPath);
```

Первый параметр функции – это значение *HINSTANCE* для EXE или DLL. Второй задает адрес буфера, в который функция запишет полное имя образа файла. Последний параметр (*cchPath*) определяет размер буфера в символах.

Уменьшить счетчик числа пользователей DLL можно и с помощью другой Win32-функции:

```
VOID FreeLibraryAndExitThread(HINSTANCE hinstDll,
                              DWORD dwExitCode);
//Она реализована в KERNEL32.DLL так:
VOID FreeLibraryAndExitThread(HINSTANCE hinstDll,
                              DWORD dwExitCode) {
     FreeLibrary(hinstDll);
     ExitThread(dwExitCode);
}
```

Если поток станет сам вызывать *FreeLibrary()* и *ExitThread()* по отдельности, возникнет очень серьезная проблема: вызов *FreeLibrary()* тут же отключит DLL от адресного пространства процесса. После возврата из *FreeLibrary()* код, содержащий вызов *ExitThread()*, окажется недоступен, и поток попытается выполнить неопределенный код. Это приведет к нарушению доступа и завершению всего процесса.

Если же поток обратится к *FreeLibraryAndExitThread()*, она вызовет *FreeLibrary()* и сразу же отключит DLL. Но следующая исполняемая инструкция находится в *kernel32.dll*, а не в только что отключенной DLL.

Значит, поток сможет продолжить выполнение и вызвать *ExitThread()*, которая корректно завершит его, не возвращая управления.

2.3.3. Функция входа/выхода

У DLL может быть одна функция входа/выхода – *DllMain()*. Система вызывает ее в некоторых ситуациях сугубо в информационных целях, и обычно она используется DLL для инициализации и очистки в конкретных процессах или потоках. Если DLL подобные уведомления не нужны, эту функцию можно не реализовывать. Пример – DLL, содержащая исключительно ресурсы. Но если эта функция в DLL все же есть, она должна выглядеть так:

```
BOOL WINAPI DllMain(HINSTANCE hinstDll, DWORD fdwReason,
                    LPVOID flmpLoad)
{
     switch (fdwReason) {
          case DLL_PROCESS_ATTACH:
               // DLL проецируется на адресное пр-во процесса
               break;
          case DLL_THREAD_ATTACH:
               // создается поток
               break;
          case DLL_THREAD_DETACH:
               // поток завершается корректно
               break;
          case DLL_PROCESS_DETACH:
               // DLL отключается от адресного пр-ва процесса
               break;
     }
     return(TRUE);
}
```

Операционная система вызывает функцию входа/выхода в различных ситуациях, при этом параметр *hinstDll* содержит описатель экземпляра DLL. Значение этого параметра равно виртуальному адресу, по которому файл DLL проецируется на адресное пространство процесса (как и *hinstExe* функции *WinMain()*). Обычно последнее значение сохраняется в глобальной переменной, чтобы его можно было использовать и при вызовах функций, загружающих ресурсы (типа *DialogBox()* или *LoadString()*). Если файл DLL загружен неявно, параметр *flmpLoad* отличен от 0, а если явно – равен 0.

Параметр *fdwReason* сообщает о причине, по которой система вызвала функцию *DllMain()*. Он принимает одно из 4 значений:

DLL_PROCESS_ATTACH – DLL проецируется на адресное пространство процесса (при первой загрузке);

DLL_PROCESS_DETACH – DLL отключается от адресного пространства процесса;

DLL_THREAD_ATTACH – в процессе создается новый поток;

DLL_THREAD_DETACH – DLL в процессе уничтожается поток.

2.4. Создание DLL

2.4.1. Экспорт функций и переменных из DLL

При создании DLL определяется набор функций, доступных для других EXE- или DLL-модулей. Если DLL-функция доступна для вызова из других программ, то говорят, что это экспортируемая (*exported*) функция. Кроме функций Win32 позволяет экспортировать и глобальные переменные.

Пример экспорта из DLL функции *Add()* и глобальной целочисленной переменной *g_nUsageCount*:

```
__declspec(dllexport) int Add(int nLeft, int nRight)
{
     return(nLeft + nRight);
}
__declspec(dllexport) int g_nUsageCount = 0;
```

Компилятор языка C/C++, компилируя функцию *Add()* и *g_nUsageCount*, встраивает в конечный OBJ-файл дополнительную информацию, необходимую для компоновщика при сборке DLL из OBJ-файлов.

Обнаружив такую информацию, компоновщик создает LIB-файл со списком идентификаторов, экспортируемых из DLL. Этот LIB-файл нужен при сборке любого EXE-модуля, вызывающего функции из данной DLL. Кроме того, компоновщик вставляет в конечный файл DLL и таблицу экспортируемых идентификаторов (*exported symbols*). Каждый элемент в этой таблице содержит имя экспортируемой функции или переменной, а также адрес этой функции или переменной внутри DLL-файла. Все списки сортируются по алфавиту.

2.4.2. Импорт функций и переменных из DLL

Чтобы EXE-модуль мог вызывать функции или получать доступ к переменным из DLL, нужно сообщить компилятору, что они находятся именно в DLL. Следующий фрагмент кода показывает, как импортировать функцию *Add()* и переменную *g_nUsageCount*, экспортируемые DLL-модулем:

```
__declspec(dllimport) int Add(int nLeft, int nRight);
__declspec(dllimport) int g_nUsageCount;
```

Конструкция *__declspec(dllimport)* сообщает компилятору, что функция *Add()* и переменная *g_nUsageCount* находятся в DLL, доступ к которой EXE-модуль должен получить при загрузке. Это заставит компилятор сгенерировать специальный код для импортируемых идентификаторов (*imported symbols*). Кроме того, компилятор встроит специальную информацию в конечный объектный файл. Она используется при компоновке EXE-модуля, подсказывая компоновщику, какие функции следует искать в LIB-файлах. Собирая EXE-файл, компоновщик отыскивает импортируемые функции и переменные. Затем,

определив, какой LIB-файл содержит эти идентификаторы, добавляет в таблицу импорта новые элементы. Каждый элемент содержит имя соответствующего DLL-файла, а также имя самого идентификатора. Таблица импорта вносится в конечный EXE–файл при его записи на жесткий диск.

2.4.3. Заголовочный файл DLL

Обычно при создании DLL создается и ее заголовочный файл, содержащий прототипы всех экспортируемых из DLL функций и переменных. Он понадобится при компиляции EXE-модулй. Часто его включают и при компиляции исходных файлов самой DLL. Чтобы один и тот же заголовочный файл можно было использовать при компиляции исходных файлов как EXE-, так и DLL-модулей, он должен выглядеть так:

```
//MYLIBAPI определена в файле реализаций MyLib.c как
//__declspec(dllexport). Поэтому при включении заголовочного
//файла в MyLib.c функции будут экспортироваться, а не
//импортироваться.

#ifndef  MYLIBAPI
    #define  MYLIBAPI __declspec(dllimport)
#endif

MYLIBAPI int Add(int nLeft,  int riRight);
MYLIBAPI int g_nUsageCount;
```

Этот заголовочный файл надо включать в самое начало исходных файлов DLL следующим образом:

```
//Необходимо определить MYLIBAPI до включения файла MyLib.H.
//Тогда MyLib.H, увидев, что MYLIBAPI уже определена, не
//станет переопределять ее как __declspec(dllimport).

#define  MYLIBAPI __declspec(dllexport)
#include "MyLib.h"
MYLIBAPI int Add(int nLeft, int nRight)
{
   return(nLeft + nRight);
}
MYLIBAPI int g_nUsageCount;
```

Поскольку *MYLIBAPI* определена как *__declspec(dllexport)* явно, то при компиляции исходного кода программы компилятор узнает, что функции экспортируются. А это позволяет не повторять *__declspec(dllexport)* перед каждой процедурой или переменной.

Что касается исходного кода, импортирующего из DLL идентификаторы, то нужно всего лишь включить в него заголовочный файл. Теперь MYLIBAPI будет определена как *__declspec(dllimport)*, и компилятор «поймет», какие

идентификаторы содержатся в DLL. Просмотрев заголовочные файлы Windows, например <*winuser.h*>, можно увидеть, что Microsoft применяет тот же метод.

2.4.4. Динамическое связывание без импорта

Вместо того, чтобы Windows выполняла динамическое связывание при первой загрузке программы в оперативную память, можно связать программу с модулем библиотеки во время выполнении программы. Например, можно было бы просто вызвать функцию *Rectangle()*:

```
Rectangle(hdc, xLeft, yTop, xRight, yBottom);
```

Это работает, поскольку программа была скомпонована с библиотекой импорта *gdi32.lib*, в которой имеется адрес функции *Rectangle()*.

Можно также вызвать функцию *Rectangle()* и совершенно необычным образом. Сначала используется оператор ***typedef*** для определения типа функции *Rectangle()*:

```
typedef BOOL(WINAPI *PFNRECT)(HDC, int, int, int, int);
```

Затем определяются две переменные:

```
HANDLE hLibrary;
PFNRECT pfnRectangle;
```

Теперь устанавливаются значения переменных *hLibrary* равное описателю библиотеки, а значение переменной *pfnRectangle* – равным адресу функции *Rectangle()*:

```
hLibrary = LoadLibrary("gdi32.dll");
if (hLibrary)
{
     pfnRectangle =
          (PFNRECT)GetProcAddress(hLibrary,"Rectangle");

//Теперь можно вызывать функцию и затем освободить библиотеку

     if (pfnRectangle)
        pfnRectangle(hdc, xLeft, yTop, xRight, yBottom);
     FreeLibrary(hLibrary);
}
```

Если этот прием динамического связывания во время выполнения не имеет особого смысла для функции *Rectangle()*, то смысл определенно появляется, если до начала выполнения программы неизвестно имя модуля библиотеки.

ЛАБОРАТОРНАЯ РАБОТА № 6

Процессы и потоки

1. Цель работы

Ознакомление с понятиями процессов и потоков в ОС Windows, приобретение практических навыков создания и использования процессов и потоков.

2. Многозадачность и многопоточность в Windows

Многозадачность (*multitasking*) – это способность операционной системы выполнять несколько программ одновременно. В основе этого принципа лежит использование операционной системой аппаратного таймера для выделения отрезков времени (*time slices*) для каждого из одновременно выполняемых процессов. Если эти отрезки времени достаточно малы и машина не перегружена слишком большим числом программ, то пользователю кажется, что все эти программы выполняются параллельно.

Идея многозадачности не нова. Многозадачность реализуется на больших компьютерах типа *мэйнфрэйм* (*mainframe*), к которым подключены десятки, а иногда и сотни терминалов. У каждого пользователя, сидящего за экраном такого терминала, создается впечатление, что он имеет эксклюзивный доступ ко всей машине. Кроме того, операционные системы мэйнфрэймов часто дают возможность пользователям перевести задачу в фоновый режим, где она выполняется, в то время как пользователь работает с другой программой. Windows NT и Windows 9x – 32-разрядные версии Windows – поддерживают кроме многозадачности еще и многопоточность (*multithreading*).

Многопоточность – это возможность программы самой быть много-задачной. Программа может быть разделена на отдельные потоки выполнения (*threads*), которые выполняются псевдопараллельно. На первый взгляд эта концепция может показаться едва ли полезной, но оказывается, что программы могут использовать многопоточность для выполнения протяженных во времени операций в фоновом режиме, не вынуждая пользователя надолго отрываться от машины.

2.1. Процессы

Если нужно запустить новую программу, следует создать новый процесс. Для этого используется стандартный системный вызов *CreateProcess()*. Однако пользоваться им не очень удобно – приходится определять значения множества аргументов. Что делать, если необходимо запустить WordPad только для того, чтобы отобразить содержимое файла *«readme.txt»*? Для этой цели можно использовать менее сложный механизм – вызов функции *WinExec()*. При обращении к *WinExec()* необходимо сообщить имя программы (полный путь или короткое имя исполняемого файла, расположенного в пути поиска), а также

способ отображения окна программы (константа, используемая функцией *ShowWindow()*, *SW_SHOW*, *SW_HIDE* и т.д.).

В случае, если произошла ошибка, функция *WinExec()* возвращает значение, меньшее 32. Если программа успешно запущена, функция возвращает дескриптор новой программы (который не может быть меньше 32). После запуска новой программы функция *WinExec()* немедленно передает управление программе, из которой был осуществлен вызов, т.е. она не ждет момента, пока вновь запущенная программа завершит работу.

Еще один простой вызов, который можно использовать для запуска программ, – это *ShellExecute()*. Он во многом напоминает *WinExec()*, но еще поддерживает обработку типов файлов, зарегистрированных графической оболочкой операционной системы. Например, если при помощи *ShellExecute()* попробовать запустить файл с расширением *.txt*, будет запущена программа *notepad.exe* или любая другая программа, которая используется в системе для просмотра текстовых файлов. В качестве аргументов функция *ShellExecute()* принимает дескриптор окна (на случай, если возникнет необходимость в сообщениях об ошибках) и операционную строку, такую, как «open» (открыть), «print» (распечатать) или «explore» (исследовать). В качестве операционной строки можно передать *NULL*-строку, в этом случае указанный файл будет открыт («open»). Также функции *ShellExecute()* необходимо сообщить имя файла и любые параметры командной строки (обычно *NULL*). Наконец, последние два аргумента – это текущий каталог и константа функции *ShowWindow()* (как и в случае с *WinExec()*).

Возвращаемое значение точно такое же, как и у *WinExec()*. Если указать в качестве третьего аргумента функции *ShellExecute()* имя исполняемого файла, можно не использовать другие аргументы, кроме аргумента параметров командной строки и константы *ShowWindow()*. Для файлов документов (например, *.txt или *.doc) значение этих аргументов обычно равно *NULL*.

Функцию *ShellExecute()* можно использовать, например, для того, чтобы открыть корневой каталог диска C:

```
ShellExecute(hWnd,"open","c:\\",NULL,NULL,SW_SHOWNORMAL);
```

Можно заменить строку «open» на строку «explore», а также указать в качестве третьего параметра имя абсолютно любого каталога.

Ниже приведен пример простой консольной программы:

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

void main()
{
    printf("Opening with WinExec\n");

    if (WinExec("notepad.exe readme.txt", SW_SHOW) < 32)
```

```
        MessageBox(NULL, "Ошибка открытия", NULL, MB_OK);
    else
        MessageBox(NULL,"Нажмите OK для продолжения",
                        "Программа открыта",MB_OK);

    printf("Opening with ShellExecute\n)";

    if (ShellExecute(NULL,"open","readme.txt",NULL,
             NULL,SW_SHOW) < 32)
        MessageBox(NULL, "Ошибка открытия", NULL, MB_OK);
    else
        MessageBox(NULL,"Нажмите OK для продолжения",
                        "Программа открыта",MB_OK);
}
```

Программа открывает текстовый файл, используя при этом два различных способа. Сначала происходит обращение к функции *WinExec()*, которой напрямую сообщаются имя программы текстового редактора и имя текстового файла, который должна открыть эта программа. Затем для открытия того же файла используется системный вызов *ShellExecute()* (при этом, скорее всего, будет запущен *WordPad* или другая программа, ответственная в системе за открытие текстовых файлов).

2.2. Вызов CreateProcess

Вызовы наподобие *ShellExecute()* и *WinExec()* очень удобны для выполнения простейших действий, вроде открытия файлов и запуска программ. Если же необходимо создать новый процесс, используя при этом некоторые дополнительные параметры, следует применить системный вызов *CreateProcess()*. Описание аргументов, принимаемых этим вызовом, приведено ниже.

Аргументы вызова *CreateProcess()*:

lpApplicationName – имя программы (или *NULL*, если имя программы указано в командной строке);

lpCommandLine – командная строка;

lpProcessAttributes – атрибуты безопасности для дескриптора процесса, возвращаемого функцией;

lpThreadAttributes – атрибуты безопасности для дескриптора потока, возвращаемого функцией;

bInheritHandlers – указывает, наследует ли новый процесс дескрипторы, принадлежащие текущему процессу;

dwCreationFlags – параметры создания процесса (см. табл. 2);

lpEnvironment – значения переменных окружения (или *NULL*, в случае если наследуется текущее окружение);

lpCurrentDlrectory – текущий каталог (или *NULL*, если используется текущий каталог текущего процесса);

lpProcessInformation – возвращаемые функцией дескрипторы и идентификаторы ID процесса и потока;

lpStartupInfo – указатель на структуру *STARTUPINFO*, содержащую информацию о запуске процесса.

Значения параметра *dwCreationFlags*, используемые при создании процесса:

CREATE_DEFAULT_ERROR_MODE – не наследовать текущий режим сообщений об ошибках (см. *SetErrorMode()*);

CREATE_NEW_CONSOLE – создать новую консоль;

CREATE_NEW_PROCESS_GROUP – создать новую группу процессов;

CREATE_SEPARATE_WOW_VDM – запустить 16-битное приложение в его собственном адресном пространстве;

CREATE_SHARED_WOW_VDM – запустить 16-битное приложение в адресном пространстве общего доступа;

CREATE_SUSPENDED – создать процесс в приостановленном состоянии (см. *ResumeThread()*);

CREATE_UNICODE_ENVIRONMENT – блок переменных окружения записан в формате *UNICODE*;

DEBUG_PROCESS – запустить процесс в отладочном режиме;

DEBUG_ONLY_TH1S_PROCESS – предотвратить отладку процесса текущим отладчиком (используется при отладке родительского процесса);

DETACHED_PROCESS – новый консольный процесс не имеет доступа к консоли родительского процесса.

Аргумент *lpStartupInfo* – это указатель на структуру *STARTUPINFO*. Поля этой структуры содержат заголовок консоли, начальный размер и позицию нового окна и перенаправление стандартных потоков ввода/вывода. Новая программа может проигнорировать все эти параметры в зависимости от собственного желания. Поле *dwFlags* этой структуры содержит флаги, установленные в соответствии с тем, какие из остальных полей структуры необходимо было использовать при запуске новой программы. Например, если сбросить флаг *STARTFJJSEPOSITION*, поля *dwX* и *dwY* структуры *STARTUPINFO*, содержащие координаты основного окна запускаемой программы, будут проигнорированы.

Функция *CreateProcess()* записывает в аргумент *lpProcessInformation* указатель на структуру, содержащую дескрипторы и идентификаторы *ID* нового процесса и потока. Доступ к этим дескрипторам определяется аргументами *lpProcessAttributes* и *lpThreadAttributes*.

Некоторые аргументы функции *CreateProcess()* относятся к консольным приложениям, другие имеют значение для всех типов программ. Зачастую при обращении к *CreateProcess()* можно не заполнять структуру *STARTUPINFO*, однако в любом случае нужно передать функции указатель на существующую в памяти структуру, даже если эта структура не заполнена.

Функция *CreateProcess()* возвращает значение типа *BOOL*, но по завершении работы чрезвычайно полезная для программиста информация размещается функцией в структуре типа *PROCESS_INFORMATION*, указатель на которую возвращается при помощи параметра *lpProcessInformation* этой функции. В структуре *PROCESS_INFORMATION* содержатся идентификатор и дескриптор нового процесса, а также идентификатор и дескриптор самого первого потока, принадлежащего новому процессу. Эти сведения могут использоваться для того, чтобы сообщить о новом процессе другим программам, а также для того, чтобы контролировать новый процесс.

При создании процесса с использованием вызова *CreateProcess()* можно передать новому процессу по наследству некоторые объекты, в частности дескрипторы открытых файлов. Однако, к сожалению, это редко когда бывает полезным, за исключением случаев, когда необходимо объединить стандартные потоки ввода/вывода нескольких консольных приложений. В этой ситуации дескрипторы файлов стандартных потоков ввода/вывода обладают заранее определенными значениями. Даже если новый процесс получает по наследству дескриптор открытого файла, он не может определить его значения, если только оно не определено заранее. Можно передать дескриптор в командной строке или в переменной окружения, однако, это не очень удобный вариант.

Обладая дескриптором процесса, можно управлять процессом при помощи следующих вызовов:

GetExitCodeProcess() – возвращает код завершения процесса;

GetGuiRecources() – определяет, сколько объектов USER или GDI используется процессом;

SetPrlorityClass() – устанавливает базовый приоритет процесса;

GetPriorityClass() – возвращает базовый приоритет процесса;

SetProcessAffinityMask() – определяет, какие из процессоров используются процессом в качестве основных;

GetProcessAffinityMask() – устанавливает, какие из процессоров используются процессом в качестве основных;

SetProcessPriorityBoost() – позволяет или запрещает Windows динамически изменять приоритет процесса;

GetProcessPriorityBoost() – возвращает статус изменения приоритета процесса;

SetProcessShutDownParameters() – определяет, в каком порядке система закрывает процессы при завершении работы всей системы;

GetProcessShutDownParameter() – возвращает статус механизма завершения работы системы;

SetProcessWorkingSetSize() – устанавливает минимальный и максимальный допустимый объем физической оперативной памяти, используемый процессом;

GetProcessWorkingSetSize() – возвращает информацию об использовании физической памяти процессом;

TerminateProcess() – корректное завершение работы процесса;

ExitProcess() – немедленное завершение процесса;

GetProcessversion() – возвращает версию Windows, в среде которой хотел бы работать процесс;

GetProcessTimes() – возвращает степень использования CPU процессом;

GetStartUpInfo() – возвращает переданную процессу при обращении к *CreateProcess()* структуру *STARTUPINFO*.

Чтобы определить момент завершения процесса, можно воспользоваться одним из нескольких методов, Во-первых, можно использовать вызов *GetExitCodeProcess()*, который возвращает либо значение *STILL_ACTIVE* (если процесс еще продолжает работу), либо код завершения процесса (если процесс завершен). В качестве одного из аргументов этой функции передается указатель на переменную, в которую помещается возвращаемое значение. Узнать дескриптор текущего процесса можно при помощи функции *GetCurrentProcess()*. Чтобы управлять процессом из другого процесса, следует обратиться к функции *OpenProcess()*, которой необходимо передать идентификатор процесса. Чтобы открыть процесс для желаемого доступа, нужно обладать необходимыми разрешениями на доступ к процессу. Процесс, создающий новый процесс при помощи функции *CreateProcess()*, уже обладает дескриптором нового процесса.

Еще один способ определения текущего состояния процесса подразумевает использование функции *WaitForSingleObject()*. Этот вызов можно использовать для самых разных целей. Основное назначение *WaitForSingleObject()* – определить, находится ли некоторый дескриптор в сигнальном состоянии. Дескриптор процесса переходит в сигнальное состояние тогда, когда процесс завершает свою работу. При обращении к функции *WaitForSingleObject()* необходимо указать дескриптор процесса и интервал времени в миллисекундах. Если интервал времени равен 0, функция завершает работу немедленно, возвращая текущее состояние процесса. Если интервал времени равен константе *INFINIT*, функция будет ждать до тех пор, пока интересующий вас процесс не завершит работу. Если указать конкретное значение интервала времени, функция будет ожидать завершения процесса в течение указанного времени, а затем вернет управление вызвавшей ее программе. Если в течение указанного времени процесс завершит работу, функция *WaitForSingleObject()* вернет управление вызвавшей программе и сообщит ей, что дескриптор целевого процесса перешел в сигнальное состояние. В противном случае эта функция вернет отрицательный ответ.

Вне зависимости от того, в каком состоянии находится дескриптор целевого процесса, функция *WaitForSingleObject()* также возвращает значение, отражающее успешное выполнение – дескриптор так и не перешел в сигнальное состояние, что не является ошибкой. Чтобы определить состояние процесса, необходимо сравнить значение, которое вернула функция, со значениями *WAIT_OBJECT_0* (сигнальное состояние) и *WAIT_TIMEOUT* (процесс продолжает функционировать). В случае ошибки функция вернет значение *WAIT_FAILED*.

Чтобы иметь возможность ожидать завершения процесса, нужно обладать открытым дескриптором этого процесса с привилегией *SYNCHRONIZE*. Следует помнить, что идентификатор процесса – это не то же самое, что дескриптор процесса. Дескрипторы процессов нельзя просто так передавать из процесса в процесс. Это означает, что если требуется управлять процессом из другого процесса, то следует, прежде всего, каким-либо образом получить дескриптор управляемого процесса. Для идентификации процесса другими процессами служит идентификатор *ID* этого процесса, который можно передать из процесса в процесс. Преобразовать идентификатор *ID* процесса в его дескриптор можно при помощи функции *OpenProcess()*, однако для этого требуется обладать необходимыми привилегиями. Узнать идентификатор текущего процесса можно при помощи функции *GetCurrentProcessId()*. Используя этот вызов, можно узнать идентификатор собственного процесса и передать этот идентификатор другому процессу. Получив *ID* вашего процесса, другой процесс сможет открыть его дескриптор.

При обращении к функции *OpenProcess()* необходимо указать требуемый уровень доступа к открываемому процессу. Иногда получить доступ к процессу на любом из возможных уровней нельзя, поэтому, выбирая уровень, нужно использовать именно тот, который необходим для выполнения задачи, и не более того. Например, чтобы узнать код завершения процесса, достаточно владеть уровнем доступа *PROCESS_QUERY_INFORMATION*. Чтобы иметь возможность завершить работу процесса, необходимо обладать уровнем доступа *PROCESS_TERMINATE*. Можно запросить предоставление полного набора прав доступа к процессу, для этого предназначен уровень доступа *PROCESS_ALL_ACCESS*.

При помощи вызова *LoginUser()* программа, обладающая необходимой привилегией (конкретно *SE_TCB_NAME*), может определить лексему (*token*) подключенного к системе пользователя. Обладая этой лексемой, можно запустить какой-либо процесс от имени пользователя системы. Другими словами, действия, которые выполняет процесс, будут рассматриваться системой как действия, выполняемые пользователем системы. Запуск процесса от имени пользователя осуществляется при помощи вызова *CreateProcessAsUser()*, при этом программа будет запущена от лица пользователя, обладающего указанной лексемой.

В приведенных ниже примерах создаются два простых консольных приложения. Первая программа (MASTER) запускает вторую (SLAVE) и переходит в режим ожидания. Программа SLAVE читает идентификатор процесса (PID – Process Identifier) запустившей ее программы из командной строки и ожидает завершения работы программы MASTER. В командной строке программы MASTER можно указать полный путь к исполняемому файлу программы SLAVE. Обе программы иллюстрируют несколько важных технологий использования функций *CreateProcess()*, *OpenProcess()* и *WaitForSingleObject()*.

Следует обратить внимание, что MASTER не использует большую часть аргументов функции *CreateProcess()*, поэтому в данном конкретном случае для запуска SLAVE вполне пригоден вызов *WinExec()*.

Программа MASTER:

```
#include <windows.h>
#include <stdlib.h>
#include <string.h>

void main(int argc, char *argv[])
{
  char cmd[128];
  if (argc!=1) strcpy(cmd, argv[1]);
  else strcpy(cmd, "slave.exe");
  int pid = GetCurrentProcessId();
  sprintf(cmd + strlen(cmd)," %d", pid);
  printf("Master: Starting:%d \n", pid);

//In this case, might just as well use WinExec
//if (WinExec(cmd,SW_SHOW) < 32)
//  printf("Master: Slave process did not start\n");
  printf("Master: Try naming slave process on the command
line\n");
     }
     cout<<"Master: Sleeping\n";
     cout.flush();

     Sleep(15000);
     cout<<"Master: Exiting\n";
     exit(0);
 }
```

Программа SLAVE:

```
#include <windows.h>
#include <stdlib.h>
#include <string.h>

void main(int argc,char *argv[])
{
    if (argc!=2)
    {
       cerr<<"Slave: Please run MASTER.EXE instead.\n";
       exit(1);
    }

   int pid=atoi(argv[1]);
   HANDLE process=OpenProcess
```

```
    (PROCESS_QUERY_INFORMATION|SYNCHRONIZE,FALSE,pid);

  if (!process) cout<<"Slave: Error opening process\n";
  cout<<"Slave: Waiting for master to finish\n";
  cout.flush();

  if (WaitForSingleObject(process,INFINITE)==
      STATUS_WAIT_0)
      cout<<"Slave: Master completed\n";
  else
      cout<<"Slave: Unexpected error\n";
      exit(0);
}
```

2.3. Задания и рабочие наборы

Обычно каждому процессу в Windows 2000 назначается так называемый рабочий набор (*working set*). Рабочий набор процесса определяет, какой объем физической памяти диспетчер памяти Windows пытается сохранить за данным процессом. В рабочем наборе указываются минимальное и максимальное количества страниц памяти, которые должны принадлежать данному процессу. Узнать текущий размер можно при помощи вызова *GetProcessWorkingSize()*. Если объем памяти, доступный для других приложений, становится неприемлемо маленьким, система может нарушить границы, установленные в рабочем наборе той или иной программы. Обладая необходимыми для этого привилегиями, программа может изменить собственный рабочий набор при помощи функции *SetProcessWorkingSetSize()*. Если обе границы становятся равными *0xFFFFFFFF*, Windows сбрасывает на диск всю память, принадлежащую процессу.

Еще одним методом управления процессами в Windows 2000 является *задание* (*job*) – это группа связанных между собой процессов. Создать задание можно при помощи функции *CreateJobObject()*, а открыть существующее задание можно при помощи функции *OpenJobObject()*. Обладая дескриптором задания, можно добавить к нему любые другие процессы при помощи функции *AssignProcessToJobObject()*.

Для объединения процессов в группу может быть несколько причин. Например, используя функцию *TerminateJobObject()*, можно разом завершить работу всех процессов одного задания. При помощи вызова *SetInformationJobObject()*можно установить рабочий набор одновременно для всей группы процессов, входящих в задание. Этот же вызов можно использовать для назначения ограничений всем процессам, входящим в задание. В частности, можно запретить всему заданию обращаться к системному вызову *ExitWindows()*, читать содержимое системного буфера обмена или использовать какие-либо дескрипторы (например, дескрипторы окон) других процессов. Если запрещен доступ к пользовательским дескрипторам процессов одного задания, то процесс, не принадлежащий этому

заданию, может предоставить его процессам доступ к пользовательскому дескриптору при помощи функции *UserHandleGrantAccess()*.

Помимо прочего операционная система позволяет получать статистику, связанную с процессами задания. Для этого служит вызов *QueryInformationObject()*, используя который можно получить информацию о том, какую нагрузку на центральный процессор создают процессы, входящие в состав задания, а также другую подобную информацию. Кроме того, при помощи этого вызова можно получить информацию о параметрах конфигурации задания, значения которым присваиваются при помощи функции *SetInformationObject()*.

3. Потоки. Общие сведения

Термин поток (*thread*) означает выполнение некоторой последовательности инструкций кода программы. Все программы, представленные ранее в лабораторных работах, выполняются одним потоком, называемым первичным потоком. Однако программа, написанная для Windows, может запустить один или более вторичных потоков, каждый из которых независимо выполняет набор инструкций программного кода. С точки зрения пользователя, потоки в программе выполняются одновременно. Операционная система (ОС) обычно достигает этого за счет быстрого переключения управления с одного потока на другой (однако, если компьютер имеет более одного процессора, ОС может выполнять потоки действительно одновременно).

3.1. Создание вторичных потоков

Многопоточность особенно полезна в Windows-программе в тех случаях, когда первичный поток программы, допустим, посвящен обработке сообщений, при этом можно обеспечить быстрые ответы программы на команды и другие события. Вторичный поток может быть использован для выполнения некоторой длинной задачи, которая должна блокировать обработку сообщений программы, если выполняется первичный поток, например: рисование сложной графики, пересчет электронных таблиц, выполнение дисковых операций, связь с последовательным портом. Запуск отдельного потока программы осуществляется относительно быстро и занимает немного памяти. Кроме того, все потоки внутри программы выполняются в одном и том же адресном пространстве памяти и используют один и тот же набор ресурсов Windows.

В программах Win32 можно достаточно свободно использовать многопоточность, при условии, что соблюдаются определенные правила.

Для запуска нового потока вызывается функция *CreateThread()*, которая имеет следующий формат:

```
HANDLE CreateThread(
  LPSECURITY_ATTRIBUTES lpThreadAttributes,
  DWORD dwStackSize,
  LPTHREAD_START_ROUTINE lpStartAddress,
  LPVOID lpParameter,
  DWORD dwCreationFlags,
  LPDWORD lpThreadId);
```

Параметры функции следующие:

lpThreadAttributes – указатель на структуру *SECURITY_ATTRIBUTES*, которая содержит дескриптор защиты. Используется в Windows NT (значение, равное *NULL*, указывает на то, что новый поток получит дескриптор защиты по умолчанию), для Windows 9x этот параметр игнорируется;

dwStackSize – размер стека для потока (в байтах);

lpStartAddress – адрес функции, которая будет выполняться в потоке. Ее прототип должен быть следующим:

```
DWORD WINAPI ThreadFunc(LPVOID);
```

lpParameter – 32-разрядное значение, которое будет передаваться в функцию потока;

dwCreationFlags – дополнительный флаг для управления созданием потока. Если использовать значение, равное *CREATE_SUSPENDED*, то поток будет создан в приостановленном состоянии и не выполняется, пока для него не будет вызвана функция *ResumeThread()*. Если значение флага равно 0, то поток начнет свое выполнение немедленно после создания. Других значений флага не предусмотрено;

lpThreadId – указатель на 32-разрядную переменную, которая получит идентификатор созданного потока.

CreateThread() запускает выполнение нового потока и быстро возвращает управление. С этого момента оба потока – новый и тот, который вызвал *CreateThread()*, работают одновременно. Третий параметр *lpStartAddress* задает функцию потока; новый поток начинает выполнять эту функцию. Функция потока может вызывать другие функции, но когда происходит возврат из функции потока, созданный поток завершается.

Функция потока возвращает значение типа *DWORD*, это значение называется кодом завершения, и другие потоки могут его использовать. Обычно функция потока возвращает значение 0, указывающее на нормальное его завершение.

Информацию в новый поток можно передать через указатель *lpParameter*, который передается в функцию потока. Этот указатель может содержать адрес простого числа, например типа ***int***, или адрес структуры, содержащей любое количество информации. Аналогично поток может возвращать информацию начальному потоку, присваивая значение элементу данных, на который указывает параметр *lpParameter*.

Следующий пример кода начинает новый поток, выполняющий функцию *ThreadFunction()*:

```
DWORD WINAPI ThreadFunction (LPVOID lpParam)
{
     // предложения и функциональные вызовы,
     // которые должны быть выполнены новым потоком
     //...
     return 0;
}

void SomeFunction (void)
{
     //...
     int Code = 1;
     DWORD ThreadId;
     HANDLE hThread;

     hThread = CreateThread(NULL, 0, ThreadFunction,
                            &Code, 0, &ThreadId);

     CloseHandle(hThread);
}
```

3.2. Прекращение выполнения потока

Можно завершить запущенный поток одним из двух способов. Во-первых, возможно просто получить возврат потока из функции потока (как в примере выше), передавая обратно желаемый код выхода. Это наиболее правильный способ завершения потока; стек, используемый потоком, будет освобожден и все данные объектов, автоматически созданные потоком, будут разрушены (т.е. будут вызваны деструкторы для автоматически созданных объектов). Во-вторых, поток может вызвать функцию API *ExitThread()*, передавая ей желаемый код выхода:

```
VOID ExitThread(DWORD dwExitCode);
```

Вызов *ExitThread()* – это удобный способ немедленно завершить поток из вложенной функции (вместо возврата в начальную функцию потока). При использовании этого метода стек потока будет освобожден, но деструкторы для автоматических данных объектов не будут вызваны.

Оба эти способа завершения потока должны быть выполнены самим потоком. Если необходимо завершить поток из другого потока, то этому другому потоку следует передавать сигнал в поток, который нужно закончить, требуя завершения его своими средствами. Кроме этого, поток можно завершить из другого с помощью функции *TerminateThread()*. Прототип ее следующий:

```
BOOL TerminateThread(HANDLE hThread,
                     DWORD dwExitCode);
```

где *hThread* – идентификатор завершаемого потока, а *dwExitCode* – код возврата.

3.3. Управление потоками

CreateThread() возвращает дескриптор созданного потока, который затем может использоваться для управления потоком. Можно вызвать функцию *SuspendThread()*, чтобы временно приостановить выполнение потока:

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

Чтобы снова запустить выполнение потока можно вызвать функцию *ResumeThread()*:

```
DWORD ResumeThread(HANDLE hThread);
```

Можно также вызвать *ResumeThread()* для запуска потока, созданного в приостановленном состоянии присваиванием значения *CREATE_SUSPEND* параметру *dwCreationFlags* функции *CreateThread()*. Кроме того, можно изменить приоритет потока с уровня, изначально присвоенного в вызове *CreateThread()*, вызывая *SetThreadPriority()*.

```
BOOL SetThreadPriority(HANDLE hThread,
                       int nPriority);
```

Например, следующий код поднимает приоритет потока:

```
HANDLE hThread = CreateThread(/*... */);
//...
SetThreadPriority(hThread, THREAD_PRIORITY_ABOVE_NORMAL);
```

Текущий уровень приоритета потока возвращается вызовом

```
GetThreadPriority.
int GetThreadPriority(HANDLE hThread);
```

Функция *GetExitCodeThread()* определяет, продолжается ли выполнение потока и, если выполнение остановлено, следует ли получить его код возврата (т.е. значение, возвращенное из функции потока или функцией *ExitThread()*):

```
BOOL GetExitCodeThread(HANDLE hThread, LPDWORD lpExitCode);
```

Пример использования функции *GetExitCodeThread()*:

```
DWORD ExitCode;
HANDLE hThread;
hThread = CreateThread(/*... */);
//...
GetExitCodeThread(hThread, &ExitCode);
if(ExitCode == STILL_ACTIVE )
{
     // поток продолжает выполнение
}
else
{
     // поток завершен и ExitCode содержит код выхода
}
```

Как видно из примера, функция *GetExitCodeThread()* присваивает значение кода возврата переменной типа *DWORD*, адрес которой передан функции вторым параметром. Если поток продолжает выполнение, переменной присваивается значение *STILL_ACTIVE*; если поток завершился, переменной присваивается значение кода выхода.

ЛИТЕРАТУРА

1. Болски М.И. Язык программирования Си: Справочник. – М.: Радио и связь, 1988. – 96 с.

2. Бруно Бабэ. Просто и ясно о Borland C++: Пер. с англ. – М.: БИНОМ, 1995. – 400 с.

3. Голуб А.И. С и C++. Правила программирования. – М.: БИНОМ, 1996. – 272 с.

4. Джордейн Р. Справочник программиста персональных компьютеров типа IBM PC, XT и AT. – М.: Финансы и статистика, 1991. – 541 с.

5. Иванчиков А.А., Ревотюк М.П. Лабораторный практикум по курсу «Системное программирование». – Мн.: БГУИР, 1995. – 194 с.

6. Касаткин А.И., Вальвачев А.Н. Профессиональное программирование на языке Си: от Turbo-C к Borland C++: Справ. пособие. – Мн.: Выш. шк., 1992.– 240 с.

7. Касаткин А.И. Профессиональное программирование на языке Си: управление ресурсами: Справ. пособие. – Мн.: Выш. шк.,1992. – 432 с.

8. Касаткин А.И. Профессиональное программирование на языке Си: системное программирование. – Мн.: Выш. шк., 1993.– 300 с.

9. Керниган Б., Ритчи Д., Фьюэр А. Язык программирования Си. – М.: Финансы и статистика, 1992. – 271 с.

10. Лазаревич Э.Г., Хорошавина Г.Ф. Аппаратурные и программные средства профессиональных персональных ЭВМ: Справ. пособие. – Мн.: Выш. шк., 1991. – 270 с.

11. Лю Ю-Чжен, Гибсон Г. Микропроцессоры семейства 8086/8088. Архитектура, программирование и проектирование микрокомпьютерных систем. – М.: Радио и связь, 1987. – 512 с.

12. Михальчук В.М., Ровдо А.А., Рыжиков С.В. Микропроцессоры 80x86, Pentium. Архитектура, функционирование, программирование, оптимизация кода. – Мн.: Битрикс, 1994. – 400 с.

13. Джефри Рихтер. Windows для профессионалов. Программирование в Win32API для Windows NT и Windows 95. – М.: Изд. отдел «Русская редакция» ТОО «Channel Trading Ltd.», 1995. – 720 с.

Св. план 2004, поз. 35

Учебное издание

Ревотюк Михаил Павлович,
Лепешинский Владимир Николаевич

СИСТЕМНОЕ ПРОГРАММИРОВАНИЕ

Лабораторный практикум
для студентов специальности 53 01 02
«Автоматизированные системы обработки информации и управления»
всех форм обучения

Редактор Н.А. Бебель
Корректор Н.В. Гриневич

Подписано в печать | Формат 60x84 1/16. | Бумага офсетная.
Гарнитура «Таймс». | Печать ризографическая. | Усл. печ. л.
Уч.-изд. л. 3,8. | Тираж 150 экз. | Заказ 417.

Издатель и полиграфическое исполнение: Учреждение образования
«Белорусский государственный университет информатики и радиоэлектроники»
Лицензия на осуществление издательской деятельности №02330/0056964 от 01.04.2004.
Лицензия на осуществление полиграфической деятельности №02330/0133108 от 30.04.2004.
220013, Минск, П. Бровки, 6