Министерство образования Республики Беларусь

Учреждение образования
«Белорусский государственный университет
информатики и радиоэлектроники»

Кафедра электронных вычислительных средств

Д. С. Лихачёв

ВИРТУАЛЬНАЯ ПАМЯТЬ

Лабораторный практикум
по курсу
«Системное программирование»
для студентов специальности I-40 02 02
«Электронные вычислительные средства»
дневной формы обучения

Минск 2007

УДК 004.255 (075)
ББК 92.73 я 7
Л 65

Р е ц е н з е н т
доцент кафедры ЭВМ БГУИР, канд. техн. наук А. А. Петровский

Лихачев, Д. С.

Л 65 Виртуальная память : лаб. практикум по курсу «Системное программирование» для студ. спец. I-40 02 02 «Электронные вычислительные средства» дневн. формы обуч. / Д. С. Лихачёв. – Минск : БГУИР, 2007. – 48 с. : ил.

ISBN 978-985-488-082-2

Данный лабораторный практикум содержит описание подсистемы управления памятью Windows. Приводятся примеры использования API-функций Win32 для работы с виртальной памятью и описание лабораторных работ.

УДК 004.255 (075)
ББК 92.73 я 7

ISBN 978-985-488-082-2

1 Механизм виртуальной памяти

Оперативная память является важнейшим ресурсом любой компьютерной системы. Процессор исполняет инструкции программы только в том случае, если они загружены в память. В силу физической ограниченности оперативной памяти ограничивается и число одновременно выполняющихся программ. Решением данной проблемы стала подмена (***виртуализация***) оперативной памяти дисковой памятью, суть которой заключается в том, что программа загружается в оперативную память не целиком, а по частям, по мере необходимости, а остальная её часть хранится во внешней памяти (на жестком диске) в специально отведенном месте. Это решение позволило увеличить число одновременно выполняющихся программ за счет рационального распределения основной памяти между ними.

Одним из наиболее популярных способов управления памятью в современных операционных системах является метод ***виртуализации памяти (ВП)***. Вообще ***виртуальным*** называется ресурс, который пользователю или пользовательской программе представляется обладающим свойствами, которыми он в действительности не обладает. В данном случае наличие в операционных системах механизма виртуальной памяти позволяет программисту писать программы так, как будто в его распоряжении имеется однородная оперативная память большого объема, часто существенно превышающая объем имеющейся физической памяти. А как известно, объем оперативной памяти, который имеется в компьютере, существенно сказывается на характере протекания вычислительного процесса, а именно: он ограничивает число одновременно выполняющихся программ. Поэтому использование механизма виртуальной памяти позволяет повысить уровень ***мультипрограммирования***, так как реальный объем оперативной памяти компьютера не столь жестко ограничивает количество одновременно выполняемых программ. ***Мультипрограммирование,*** или ***многозадачность***,– это способ организации вычислительного процесса, при котором на одном процессоре попеременно выполняются сразу несколько программ.

Прежде чем продолжить дальнейшее рассмотрение механизма виртуальной памяти, введем следующие определения. ***Программа*** – это статический объект, представляющий собой файл с кодом и данными. ***Процесс*** – это динамический объект, который возникает в операционной системе после того, как пользователь или сама операционная система запускает программу на выполнение, т.е. создает новую единицу вычислительной работы. Другими словами, любая работа вычислительной системы заключается в выполнении некоторой ***программы***. Поэтому с ***процессом*** связывается определенный программный код, который для этих целей оформляется в виде исполняемого модуля. Чтобы этот программный код мог быть выполнен, его необходимо загрузить в оперативную память, выделить некоторое место на диске для хранения данных, предоставить доступ к устройствам ввода-вывода и т.д. В

ходе выполнения программе может также понадобиться доступ к информационным ресурсам, например файлам. Поэтому ***процесс*** можно рассматривать как «заявку» на потребление ресурсов компьютера (оперативная память, внешняя память и т.д.). ***Процесс*** − это ***программа*** в стадии выполнения.

Ресурсы компьютера ограничены. Поэтому чтобы процессы не могли вмешиваться в распределение ресурсов, а также не могли повредить код и данные друг друга, важнейшей задачей операционной системы является «изоляция» одного процесса от другого, которая осуществляется путем обеспечения каждого процесса отдельным ***виртуальным адресным пространством***, так что ни один процесс не может получить прямого доступа к командам и данным другого процесса. ***Виртуальное адресное пространство*** процесса – это совокупность адресов, которыми может манипулировать программа, находящаяся в стадии выполнения. Необходимо различать ***максимально возможное*** виртуальное адресное пространство процесса и ***назначенное*** **(*выделенное*)** процессу виртуальное адресное пространство. В первом случае речь идет о максимальном размере виртуального адресного пространства, определяемом архитектурой компьютера и, в частности, разрядностью его схем адресации. Например, при работе на компьютерах с 32-разрядными процессорами Intel Pentium операционная система может предоставить каждому процессу виртуальное адресное пространство до 4 Гб (2^{32} байт). Однако это значение представляет собой только потенциально возможный размер виртуального адресного пространства, который редко на практике бывает необходим процессу. Процесс использует только часть доступного ему виртуального адресного пространства. ***Назначенное*** виртуальное адресное пространство представляет собой набор виртуальных адресов, необходимых процессу для работы (т.е. эти адреса назначаются процессу в соответствии с его потребностями). В ходе своего выполнения процесс может увеличить размер первоначального назначенного ему виртуального адресного пространства (если, например, потребуется дополнительная память для хранения данных процесса). В любом случае операционная система обычно следит за корректностью использования процесса виртуальных адресов – процессу не разрешается оперировать с виртуальным адресом, выходящим за пределы ему назначенных. Максимальный размер виртуального адресного пространства ограничивается только разрядностью адреса, присущей данной архитектуре компьютера, и, как правило, не совпадает с объемом физической памяти, имеющимся в системе. Сегодня для компьютеров универсального назначения типична ситуация, когда объем виртуального адресного пространства превышает доступный объем оперативной памяти. В таком случае операционная система для хранения данных виртуального адресного пространства процесса, не помещающихся в оперативную память, использует внешнюю память, которая в современных компьютерах представлена жесткими дисками, рисунок 1.

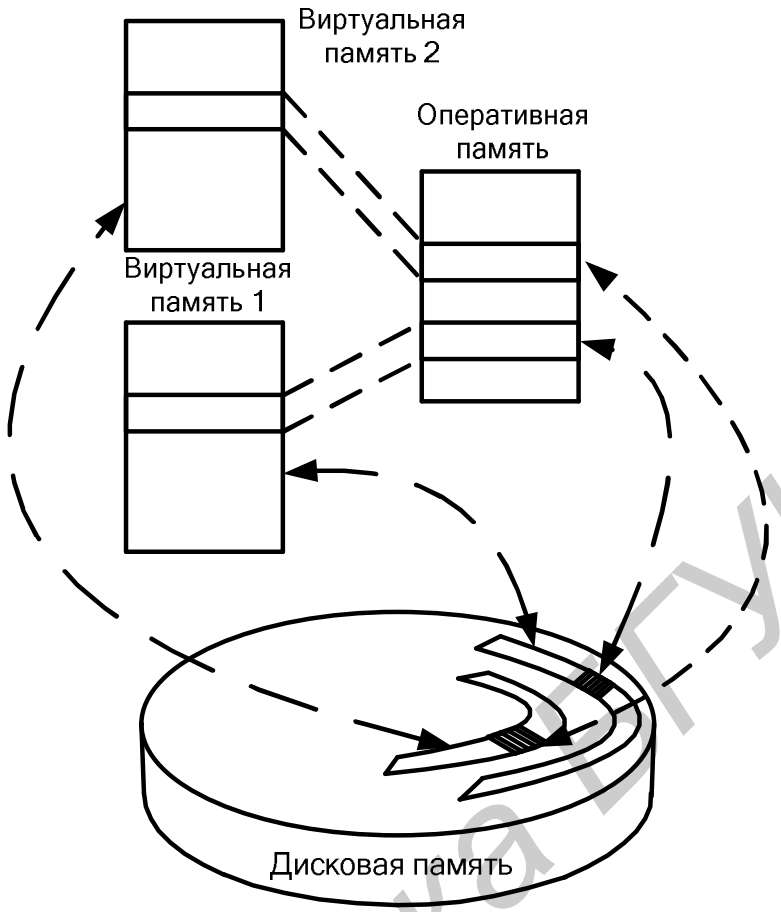

Рисунок 1 – Использование механизма виртуальной памяти (виртуальное адресное пространство превосходит объем физической памяти)

Именно на этом принципе основана ***виртуальная память*** – механизм, используемый в современных операционных системах для управления памятью.

Необходимо подчеркнуть, что виртуальное адресное пространство и виртуальная память – это различные механизмы, и они необязательно реализуются в операционной системе одновременно. Можно представить себе операционную систему, в которой поддерживаются виртуальные адресные пространства для процессов, но отсутствует механизм виртуальной памяти. Это возможно только в том случае, если размер виртуального адресного пространства каждого процесса меньше объема физической памяти (т.е. весь процесс можно загрузить в основную память, поэтому отпадает необходимость в механизме виртуальной памяти).

Содержимое назначенного процессу виртуального адресного пространства, т.е. коды команд, исходные и промежуточные данные, а также результаты вычислений, представляет собой ***образ процесса***.

Работу механизма ***виртуальной памяти*** можно кратко описать следующим образом. В мультипрограммном режиме помимо активного процесса, т.е. процесса, программный код которого в настоящий момент

обрабатывается процессором, имеются приостановленные процессы, находящиеся в ожидании завершения ввода-вывода или освобождения ресурсов, а также процессы в состоянии готовности, стоящие в очереди. Образы таких неактивных процессов могут быть одновременно до следующего цикла активности выгружены на диск. Несмотря на то, что коды и данные процесса отсутствуют в оперативной памяти, операционная система «знает» о его существовании и в полной мере учитывает это при распределении процессорного времени и других системных ресурсов. К моменту, когда подходит очередь выполнения выгруженного процесса, его образ возвращается с диска в оперативную память. Если при этом обнаруживается, что свободного места в оперативной памяти не хватает, то на диск выгружается другой процесс. Иными словами, создание нового процесса включает загрузку кодов и данных исполняемой программы данного процесса с диска в оперативную память. Для этого операционная система должна обнаружить местоположение такой программы на диске, перераспределить оперативную память и выделить память исполняемой программе нового процесса. Затем необходимо считать программу в выделенные для нее участки памяти и, возможно, изменить параметры программы в зависимости от особенностей её размещения в памяти. В системах с виртуальной памятью в начальный момент может загружаться только часть кодов и данных процесса, остальное «подкачивается» по мере необходимости.

Таким образом, функции управления памятью принадлежат операционной системе, действующей в тесной взаимосвязи с процессором.

Ключевой проблемой виртуальной памяти, возникающей в результате многократного изменения местоположения в оперативной памяти образов процессов или их частей, является преобразование виртуальных адресов в физические. Решение этой проблемы, в свою очередь, зависит от того, какой способ структуризации виртуального адресного пространства принят в данной системе управления памятью. Как уже отмечалось, ***виртуальная память*** – это механизм, при котором между оперативной памятью и диском перемещаются части (***страницы, сегменты*** и т.п.) образов процессов. В настоящее время все множество реализаций виртуальной памяти может быть представлено тремя классами.

- ***Страничная виртуальная память*** организует перемещение данных между памятью и диском страницами – частями виртуального адресного пространства, фиксированного и сравнительно небольшого размера.
- ***Сегментная виртуальная память*** предусматривает перемещение данных сегментами – частями виртуального адресного пространства произвольного размера, полученными с учетом смыслового значения данных.
- ***Сегментно-страничная виртуальная память*** использует двух-уровневое деление: виртуальное адресное пространство делится на сегменты, а затем сегменты делятся на страницы. Единицей перемещения данных здесь является страница. Этот способ управления памятью объединяет в себе элементы обоих предыдущих подходов.

Для временного хранения сегментов и страниц на диске отводится либо специальная область, либо специальный файл, которые во многих операционных системах по традиции продолжают называть областью, или файлом свопинга, хотя перемещение информации между оперативной памятью и диском осуществляется уже не в форме полного замещения одного процесса другим, а частями. Другое популярное название этой области – ***страничный файл*** (***page file***, или ***paging file***). Текущий размер страничного файла является важным параметром, оказывающим влияние на возможности операционной системы: чем больше страничный файл, тем больше приложений может одновременно выполнять операционная система (при фиксированном размере оперативной памяти). Однако увеличение числа одновременно работающих приложений за счет увеличения размера страничного файла замедляет их работу, так как значительная часть времени при этом тратится на перемещение кодов и данных из оперативной памяти на диск и обратно. Размер страничного файла в современных операционных системах является настраиваемым параметром, который выбирается администратором системы для достижения компромисса между уровнем мультипрограммирования и быстродействием системы.

Во время работы процесса постоянно выполняются переходы от прикладных кодов к кодам операционной системы, которые явно вызываются из прикладных процессов как системные функции, либо вызываются как реакция на внешние события или на исключительные ситуации, возникающие при некорректном поведении прикладных кодов (возникают ошибки работы программы). Для того чтобы упростить передачу управления от прикладного кода к коду операционной системы, а также для легкого доступа модулей операционной системы к прикладным данным (например для вывода их на внешнее устройство), в большинстве операционных систем ее сегменты разделяют виртуальное адресное пространство с прикладными сегментами активного процесса. Сегменты операционной системы и сегменты активного процесса образуют единое виртуальное адресное пространство. Поэтому обычно виртуальное адресное пространство процесса делится на две непрерывные части: системную и пользовательскую.

Механизм страничной памяти в большинстве универсальных операционных систем применяется ко всем сегментам пользовательской части виртуального адресного пространства процесса. Поэтому для общего понимания механизма виртуальной памяти рассмотрим ***схему страничного распределения*** памяти, рисунок 2.

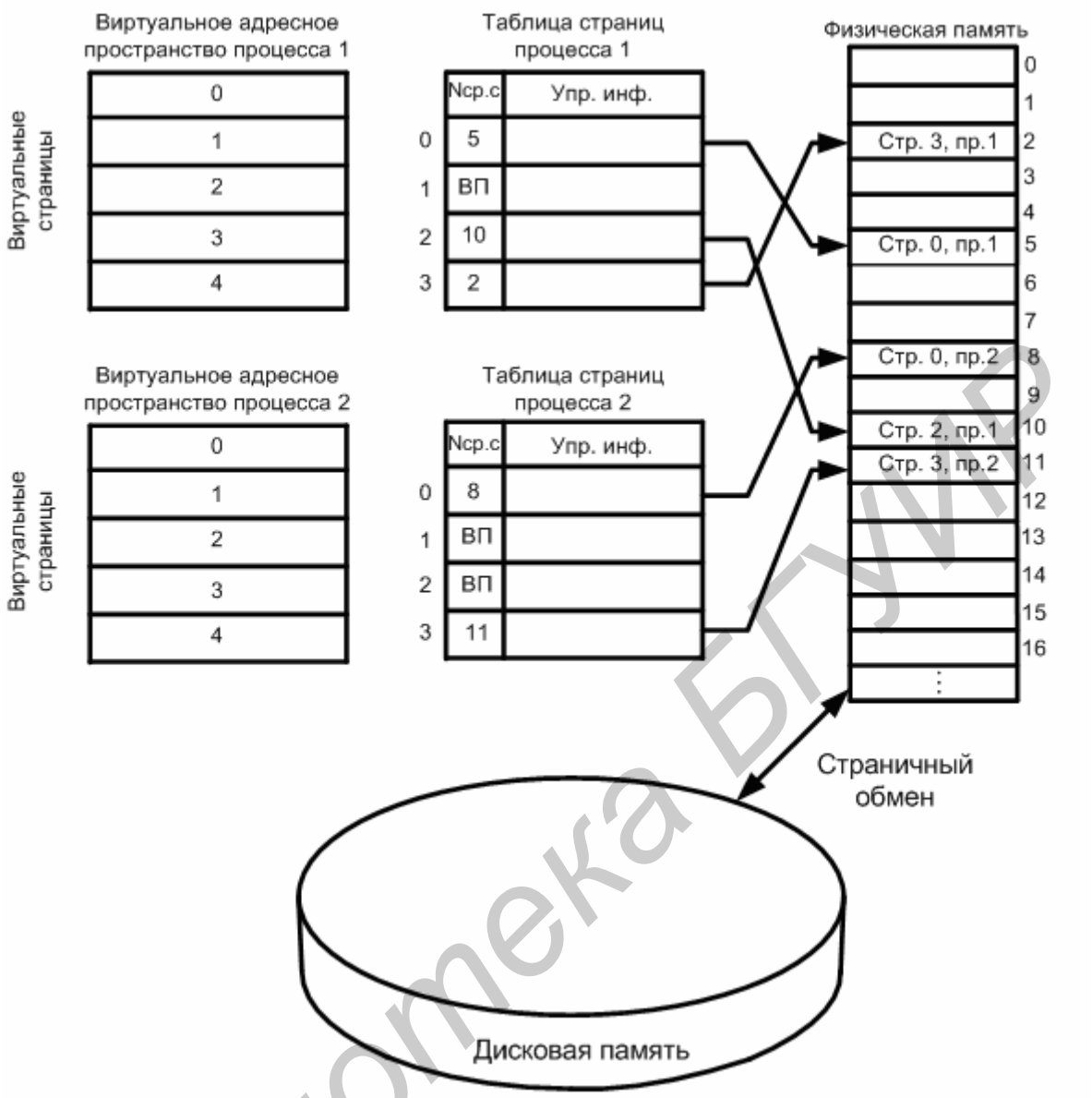

Рисунок 2 – Механизм страничного распределения памяти

Виртуальное адресное пространство каждого процесса делится на одинаковые части фиксированного для данной системы размера, называемые ***виртуальными страницами*** **(*virtual pages*)**. В общем случае размер виртуального адресного пространства процесса не кратен размеру страницы, поэтому последняя страница каждого процесса дополняется фиктивной областью.

Вся оперативная память вычислительной системы также делится на части такого же размера, называемые ***физическими страницами*** (блоками или кадрами).

Размер страницы выбирается равным степени двойки (в байтах): 512, 1024, 4096 и т.д. Это позволяет упростить механизм преобразования адресов.

При создании процесса операционная система загружает в оперативную память несколько его виртуальных страниц (начальные страницы кодового сегмента и сегмента данных). Копия всего виртуального адресного про-

странства процесса находится на диске. Смежные виртуальные страницы необязательно располагаются в смежных физических страницах. Для каждого процесса операционная система создает ***таблицы страниц*** – информационную структуру, содержащую записи обо всех виртуальных страницах процесса.

Запись таблицы, называемая ***дескриптором страницы,*** включает следующую информацию:

- ***номер физической страницы,*** в которую загружена данная виртуальная страница;
- ***признак присутствия,*** устанавливаемый в единицу, если виртуальная страница находится в оперативной памяти;
- ***признак модификации*** страницы, который устанавливается в единицу всякий раз, когда производится запись по адресу, относящемуся к данной странице;
- ***признак обращения*** к странице, называемый также ***битом доступа,*** который устанавливается в единицу при каждом обращении по адресу, относящемуся к данной странице.

Признаки присутствия, модификации и обращения в большинстве моделей современных процессоров устанавливаются аппаратно схемами процессора при выполнении операции с памятью. Информация из таблиц страниц используется для решения вопроса о необходимости перемещения той или иной страницы между памятью и диском, а также для преобразования виртуального адреса в физический. Сами таблицы страниц, так же как и описываемые ими страницы, размещаются в оперативной памяти. Адрес таблицы страниц включается в контекст соответствующего процесса. При активизации очередного процесса операционная система загружает адрес его таблицы страниц в специальный регистр процессора.

При каждом обращении к памяти выполняется поиск номера виртуальной страницы, содержащей требуемый адрес. Затем по этому номеру определяется нужный элемент таблицы страниц и из него извлекается описывающая страницу информация. Далее анализируется признак присутствия, и если данная виртуальная страница находится в оперативной памяти, то выполняется преобразование виртуального адреса в физический, т.е. виртуальный адрес заменяется указанным в записи таблицы физическим адресом. Если же нужная виртуальная страница в данный момент выгружена на диск, то происходит так называемое ***страничное прерывание.*** Выполняющийся процесс переводится в состояние ожидания, и активизируется другой процесс из очереди процессов, находящихся в состоянии готовности. Параллельно программа обработки страничного прерывания находит на диске требуемую виртуальную страницу (для этого операционная система должна помнить положение вытесненной страницы в страничном файле диска) и пытается загрузить ее в оперативную память. Если в памяти имеется свободная физическая страница, то загрузка выполняется немедленно, если же свободных страниц нет, то на основании

принятой в данной системе стратегии замещения страниц решается вопрос о том, какую страницу следует выгрузить из оперативной памяти.

После того как выбрана страница, которая должна покинуть оперативную память, обнуляется ее бит присутствия и анализируется ее признак модификации. Если выталкиваемая страница за время последнего пребывания в оперативной памяти была модифицирована, то ее новая версия должна быть переписана на диск. Если нет, то принимается во внимание, что на диске уже имеется предыдущая копия этой виртуальной страницы и никакой записи на диск не производится. Физическая страница объявляется свободной. Из соображений безопасности в некоторых системах освобождаемая страница обнуляется, с тем чтобы невозможно было использовать содержимое выгруженной страницы.

Необходимо отметить несколько наиболее важных моментов, на которые следует обращать внимание при работе с виртуальной памятью. Каждому процессу выделяется собственное виртуальное адресное пространство, размер которого ограничивается только разрядностью адреса, присущей данной архитектуре компьютера. Но это пространство ***виртуальное***, а не физическое, или, другими словами, виртуальное адресное пространство – всего лишь диапазон адресов физически не существующей памяти. Поэтому адресное пространство, выделяемое процессу в момент создания, и все свободное (не зарезервированное) можно рассматривать как простой набор ни на что не указывающих чисел. Чтобы воспользоваться какой-нибудь его частью (например для хранения данных), нужно выделить в нем определенные ***регионы*** – области, начало которых система выравнивает с учетом так называемой ***гранулярности выделения памяти*** и обычно составляет 64 Кб. Операция выделения региона называется ***резервированием*** (reserving). Резервируя регион в адресном пространстве, система обеспечивает еще и кратность размера региона размеру ***страницы***. Как и гранулярность выделения ресурсов, размер страницы зависит от типа процессора. В частности, для процессоров *x86* он равен 4 Кб, а для *Alpha* (под управлением как 32-разрядной, так и 64-разрядной Windows 2000) – 8 Кб. Если попытаться зарезервировать регион размером 10 Кб, система автоматически округлит заданное значение до большей кратной величины. А это значит, что на процессоре *x86* будет выделен регион размером 12 Кб, а на *Alpha* – 16 Кб.

Чтобы зарезервированный регион адресного пространства можно было использовать, необходимо выделить физическую память и спроецировать ее на этот регион, т.е. для зарезервированного региона выделить и указать место в физической памяти, где в дальнейшем будут храниться используемые данные. Такая операция называется ***передачей физической памяти*** (committing physical storage). В результате этой операции конкретным адресам виртуального адресного пространства для данного процесса, по которым располагается зарезервированный регион, сопоставляются реальные адреса, указывающие на физическую память. После этого можно использовать зарезервированный регион в своих целях.

Когда физическая память, переданная зарезервированному региону, больше не нужна, её необходимо освободить. Эта операция называется ***возвратом физической памяти*** (decommitting physical storage). Когда зарезервированный регион адресного пространства становится не нужен, его следует вернуть в общие ресурсы системы, т.е. произвести ***освобождение*** (releasing) региона.

Все вышеописанные операции **(*резервирование региона, передача физической памяти региону, освобождение региона* и т.д.)** осуществляются путем вызова специальных функций, входящих в состав ядра операционной системы. Эти функции могут вызываться самими приложениями (например, для открытия и чтения файла, вывода графической информации на дисплей и т.д.) и образуют интерфейс прикладного программирования – API.

Ниже описаны функции API для работы с виртуальной памятью и приведены примеры программ для Borland C++ Builder.

2 Функции для работы с виртуальной памятью

2.1 Функция GetSystemInfo

Многие параметры операционной системы (размер страницы, гранулярность выделения памяти и др.) зависят от используемого в компьютере процессора. Поэтому нельзя жестко «зашивать» их значения в исходный код программ. Эту информацию необходимо считывать в момент инициализации процесса с помощью функции **GetSystemInfo**, объявление которой выглядит следующим образом:

VOID **GetSystemInfo** (LPSYSTEM_INFO psinf);

В **GetSystemInfo** передаётся адрес структуры **SYSTEM_INFO**, а функция инициализирует (заполняет определенными значениями) элементы этой структуры:

```
typedef struct _SYSTEM_INFO {
union {
DWORD dwOemId; //не используйте этот элемент, он устарел
struct {
WORD wProcessorArchitecture;
WORD wReserved;
                };
                };
DWORD dwPageSize;
LPVOID lpMinimumApplicationAddress;
LPVOID lpMaximumApplicationAddress;
```

```
DWORD_PTR dwActiveProcessorMask;
DWORD dwNumberOf Processors;
DWORD dwProcessorType;
DWORD dwAllocationGranularity;
WORD wProcessor Level;
WORD wProcessorRevision;
} SYSTEM_INFO, *LPSYSTEM_INFO;
```

При загрузке система определяет значения элементов этой структуры, для конкретной системы эти значения постоянны. Функция **GetSystemInfo** предусмотрена специально для того, чтобы приложения могли получать эту информацию. Из всех элементов структуры лишь четыре имеют отношение к памяти. Они описаны в таблице 1.

Таблица 1– Элементы структуры SYSTEM_INFO

Элемент	**Описание**
dwPageSize	Размер страницы памяти. На процессорах *x86* это значение равно 4096, а на процессорах Alpha – 8192 байтам
lpMinimumApplicationAddress	Минимальный адрес памяти доступного адресного пространства для каждого процесса. В Windows 98 это значение равно 4 194 304, или 0x00400000, поскольку нижние 4 Мб адресного пространства каждого процесса недоступны. В Windows 2000 это значение равно 65 536, или 0x00010000, так как в этой системе резервируются лишь первые 64 Кб адресного пространства каждого процесса
lpMaximumApplicationAddress	Максимальный адрес памяти доступного адресного пространства, отведенного в «личное пользование» каждому процессу. В Windows 98 этот адрес равен 2 147 483 647, или Ox7FFFFFFF, так как верхние 2 Гб занимают общие файлы, проецируемые в память, и разделяемый код операционной системы. В Windows 2000 этот адрес соответствует началу раздела для кода и данных режима ядра за вычетом 64 Кб
dwAllocationGranularity	Гранулярность резервирования регионов адресного пространства. На момент написания пособия это значение составляет 64 Кб для всех платформ Windows

При запуске приложения система открывает его исполняемый файл и определяет объем кода и данных. Затем резервирует регион адресного пространства и помечает, что физическая память, связанная с этим регионом, это сам EXE-файл. Вместо выделения какого-то пространства из страничного файла система использует истинное содержимое или ***образ*** (image) EXE-файла как зарезервированный регион адресного пространства программы. Благодаря

этому приложение загружается очень быстро, а размер страничного файла заметно уменьшается.

Образ исполняемого файла (EXE- или DLL-файл), размещенный на жестком диске и применяемый как физическая память для того или иного региона адресного пространства, называется ***проецируемым в память файлом*** (memory-mapped file). При загрузке EXE- или DLL-файла система автоматически резервирует регион адресного пространства и проецирует на него образ файла.

Остальные элементы структуры **SYSTEM_INFO** показаны в таблице 2.

Таблица 2 − Специальные элементы структуры SYSTEM_INFO

Элемент	**Описание**
dwOemld	Устарел; больше не используется
wReserved	Зарезервирован на будущее; пока не используется
dwNumberOfProcessors	Число процессоров в компьютере
dwActiveProcessorMask	Битовая маска, которая сообщает, какие процессоры активны
dwProcessorType	Используется только в Windows 98; сообщает тип процессора, например, Intel 386, 486 или Pentium
dwProcessorArchitecture	Используется только в Windows 2000; сообщает тип архитектуры процессора, например, Intel, Alpha, 64-разрядный Intel или 64-разрядный Alpha
wProcessorLevel	Используется только в Windows 2000 и XP; сообщает дополнительные подробности об архитектуре процессора, например Intel Pentium Pro или Pentium II
wProcessorRevision	Используется только в Windows 2000 и XP; сообщает дополнительные подробности об уровне данной архитектуры процессора

Пример 1. Работа с функцией GetSystemInfo

В данном примере при использовании функции **GetSystemInfo** выводятся только значения полей ***dwPageSize, dwAllocationGranularity, dwNumber OfProcessors, dwProcessorArchitecture, lpMinimumApplicationAddress, lpMaximum ApplicationAddress*** структуры **SYSTEM_INFO**. При разработке программ можно использовать и другие поля структуры **SYSTEM_INFO**.

Интерфейс программы показан на рисунке 3.

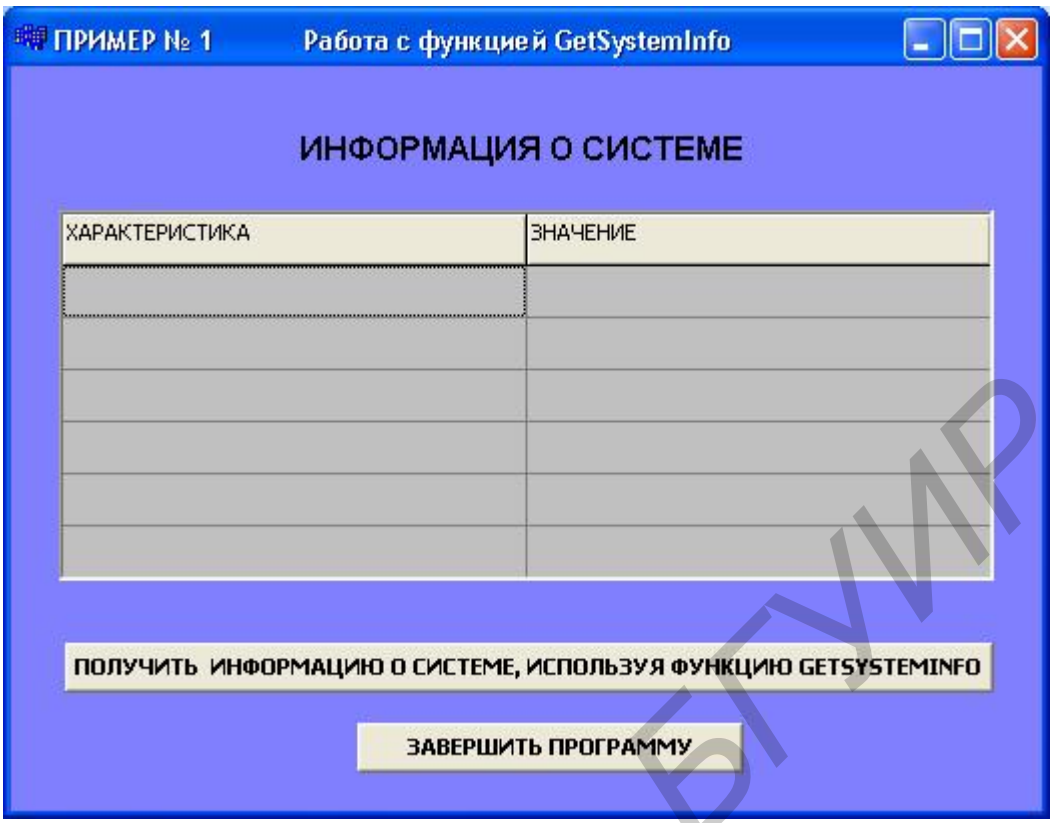

Рисунок 3 – Интерфейс программы для примера 1

Текст программы приведен ниже.

Листинг 1

```
#include <vcl.h>
#pragma hdrstop

#include "Examples.h"
//---------------------------------------------
#pragma package(smart_init)
#pragma resource "*.dfm"
TForm1 *Form1;
//---------------------------------------------
__fastcall TForm1::TForm1(TComponent* Owner):
 TForm(Owner)
{
StringGrid1->Cells[0][0]="ХАРАКТЕРИСТИКА";
StringGrid1->Cells[1][0]="ЗНАЧЕНИЕ";
}
//---------------------------------------------
void __fastcall TForm1::Button1Click(TObject *Sender)
{
SystemInfo inf;   // переменная для хранения
                     структуры SystemInfo
```

```
GetSystemInfo(&inf); // в качестве параметра функции
                        передаётся адрес структуры

StringGrid1->Cells[0][1]="Размер страницы виртуальной
 памяти";
StringGrid1->Cells[1][1]=IntToStr(inf.dwPageSize)+
"bytes";

StringGrid1->Cells[0][2]="Гранулярность резервирования
 регионов";
StringGrid1->Cells[1][2]=IntToStr(inf.dwAllocation
Granularity)+"bytes";

StringGrid1->Cells[0][3]="Число процессоров в системе";
StringGrid1->Cells[1][3]=IntToStr(inf.dwNumberOf
 Processors);

StringGrid1->Cells[0][4]="Тип архитектуры процессора";
switch(inf.wProcessorArchitecture)
{
case PROCESSOR_ARCHITECTURE_INTEL:
StringGrid1->Cells[1][4]="INTEL";break;
case PROCESSOR_ARCHITECTURE_ALPHA64:
StringGrid1->Cells[1][4]="Alpha64";break;
case PROCESSOR_ARCHITECTURE_IA64:
StringGrid1->Cells[1][4]="IA-64";break;
case PROCESSOR_ARCHITECTURE_AMD64:
StringGrid1->Cells[1][4]="AMD64";break;
case PROCESSOR_ARCHITECTURE_UNKNOWN:
StringGrid1->Cells[1][4]="OTHER";break;
}

StringGrid1->Cells[0][5]="Min доступный адрес";
StringGrid1->Cells[1][5]=(IntToHex(int
(inf.lpMinimumApplicationAddress),2))+"h";

StringGrid1->Cells[0][6]="Max доступный адрес";
StringGrid1->Cells[1][6]=(IntToHex(int
(inf.lpMaximumApplicationAddress),2))+"h";
}
//---------------------------------------------------------
void __fastcall TForm1::Button2Click(TObject *Sender)
{
Form1->Close();
}
//---------------------------------------------------------
```

Результат работы программы на тестовом компьютере показан на рисунке 4.

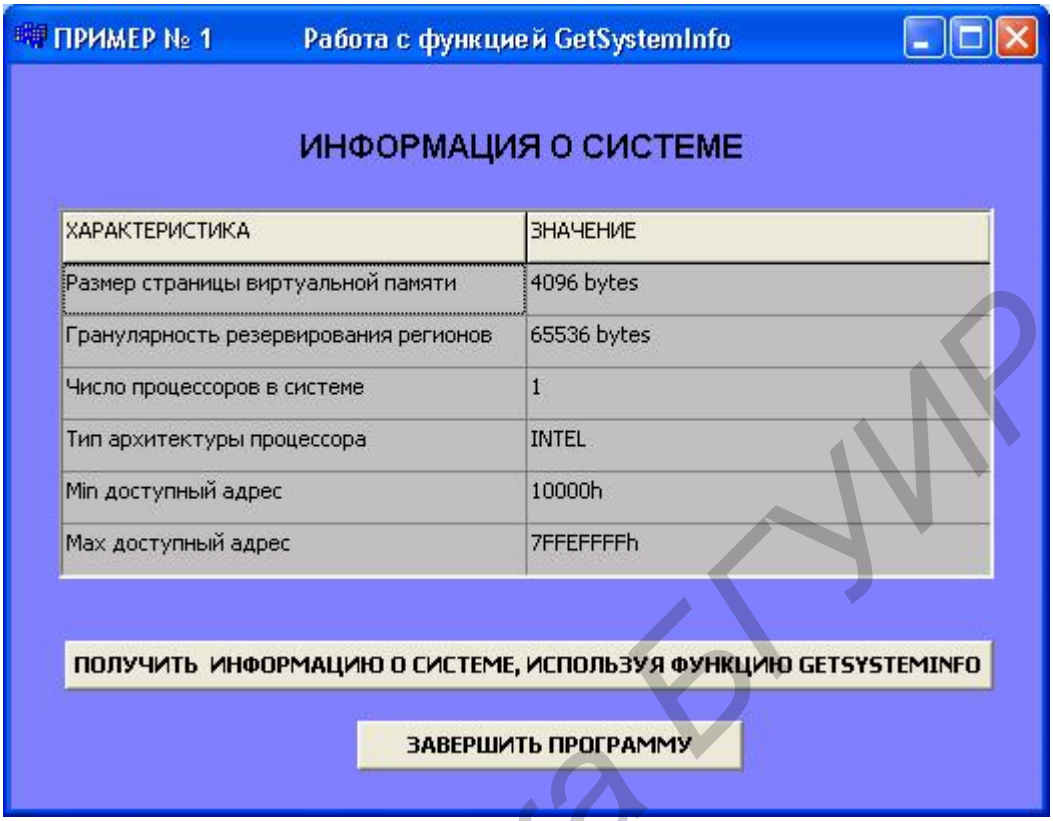

Рисунок 4 – Результат работы программы

2.2 Функция GlobalMemoryStatus

Функция **GlobalMemoryStatus** позволяет отслеживать текущее состояние памяти:

VOID **GlobalMemoryStatus**(LPMEMORYSTATUS pmst);

При вызове функции **GlobalMemoryStatus** необходимо передавать адрес структуры **MEMORYSTATUS*:***

```
typedef struct _MEMORYSTATUS {
DWORD dwLength;
DWORD dwMemoryLoad;
SIZE_T dwTotalPhys;
SIZE_T dwAvailPhys;
SIZE_T dwTotalPageFile;
SIZE_T dwAvailPageFile;
SIZE_T dwTotalVirtual;
SIZE_T dwAvailVirtual;
} MEMORYSTATUS, *LPMEMORYSTATUS;
```

Перед вызовом **GlobalMemoryStatus** надо записать в элемент ***dwLength*** размер структуры в байтах. Такой принцип вызова функции дает возможность расширять эту структуру в будущих версиях Windows, не нарушая работу существующих приложений. После своего вызова функция **GlobalMemoryStatus** инициализирует остальные элементы структуры и возвращает управление. Назначения элементов этой структуры показаны в таблице 3.

Таблица 3 − Назначение элементов структуры GlobalMemoryStatus

Элемент	Описание
dwMemoryLoad	Позволяет оценить, насколько занята подсистема управления памятью. Это число может быть любым в диапазоне от 0 до 100. В Windows 98 и Windows 2000 алгоритмы, используемые для его подсчета, различны
dwTotalPhys	Отражает общий объем физической (оперативной) памяти в байтах. Следует помнить о том, что система при загрузке резервирует небольшой участок оперативной памяти, недоступный даже ядру. Этот участок никогда не сбрасывается на диск. Поэтому элемент *dwAvailPhys* дает число байтов свободной физической памяти
dwTotalPageFile	Сообщает максимальное количество байтов, которое может содержаться в страничном файле (файлах) на жестком диске (дисках)
dwAvailPageFile	Показывает, сколько в данный момент байтов в страничном файле свободно и может быть передано любому процессу
dwTotalVirtual	Отражает общее количество байтов, отведенных под закрытое адресное пространство процесса
dwAvailVirtual	При подсчете значения *dwAvailVirtual* функция суммирует размеры всех свободных регионов в адресном пространстве вызывающего процесса. В данном случае его значение говорит о том, сколько в распоряжении вызывающей программы имеется байтов свободного адресного пространства

Отслеживать текущее состояние памяти на персональных компьютерах с объемом оперативной памяти более 4 Гб или файлом подкачки более 4 Гб необходимо при помощи функции **GlobalMemoryStatusEx**.

Пример 2 Работа с функцией GlobalMemoryStatus

В данном примере использования функции **GlobalMemoryStatus** выводятся только значения полей ***dwMemoryLoad, dwTotalPhys, dwTotalPageFile, dwTotalVirtual, dwAvailVirtual*** структуры **MEMORYSTATUS,** но при необходимости можно использовать и другие поля этой структуры.

Окно интерфейса программы показано на рисунке 5.

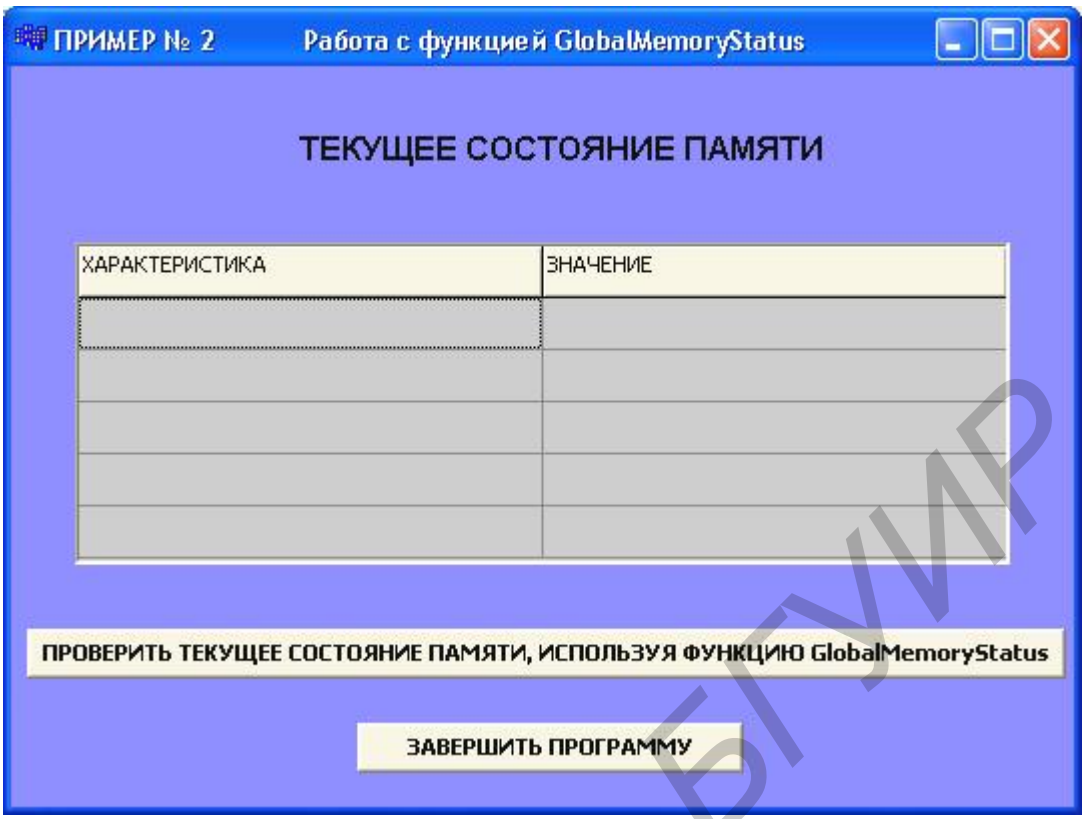

Рисунок 5 – Интерфейс программы для примера 2

Текст программы приведен ниже.

Листинг 2

```
  #include <vcl.h>
  #pragma hdrstop
  #include "Example_2.h"
  //---------------------------------------------------
  #pragma package(smart_init)
  #pragma resource "*.dfm"
  TForm1 *Form1;
  //TForm *Form2;
  //---------------------------------------------------
  __fastcall TForm1::TForm1(TComponent* Owner):
   TForm(Owner)
  {
  StringGrid1->Cells[0][0]="ХАРАКТЕРИСТИКА" ;
  StringGrid1->Cells[1][0]="ЗНАЧЕНИЕ";
  }
  //---------------------------------------------------void
   __fastcall TForm1::Button1Click(TObject *Sender)
              {
   //записываем размер структуры в байтах
  MEMORYSTATUS ms = { sizeof(ms) };
  // в качестве параметра функции передаем адрес структуры
  GlobalMemoryStatus(&ms);
 StringGrid1->Cells[0][1]="Memory Load" ;
```

```
StringGrid1->Cells[1][1]=IntToStr(ms.dwMemoryLoad);
StringGrid1->Cells[0][2]="Total Phys" ;
StringGrid1->Cells[1][2]=IntToStr(ms.dwTotalPhys)+
" bytes";
StringGrid1->Cells[0][3]="TotalPageFile" ;
StringGrid1->Cells[1][3]=IntToStr(ms.dwTotalPageFile)+
" bytes";
StringGrid1->Cells[0][4]="TotalVirtual";
StringGrid1->Cells[1][4]=IntToStr(ms.dwTotalVirtual)+
" bytes";
StringGrid1->Cells[0][5]="AvailVirtual";
StringGrid1->Cells[1][5]=IntToStr(ms.dwAvailVirtual)+
" bytes";
}
//---------------------------------------------
void __fastcall TForm1::Button2Click(TObject *Sender)
{
Form1->Close();
}
//---------------------------------------------
```

Результат работы программы на тестовом компьютере показан на рисунке 6.

Рисунок 6 – Результат работы программы

2.3 Функции VirtualAlloc, FillMemory, CopyMemory, ZeroMemory, VirtualFree

Как уже отмечалось, адресное пространство, выделяемое процессу в момент создания, изначально практически все ***свободно*** (незарезервировано). Поэтому, чтобы воспользоваться какой-нибудь его частью, нужно выделить в нем определенные регионы через функцию **VirtualAlloc**.

При резервировании система обязательно выравнивает начало региона с учетом так называемой ***гранулярности выделения памяти.***

Размер страницы можно узнать, воспользовавшись вышеописанной функцией **GetSystemInfo**, а точнее, полем ***dwPageSize*** структуры **SYSTEM_INFO**.

Итак, прежде чем осуществить какую-либо операцию с памятью для текущего приложения (например сохранить временные данные по определенному адресу из доступного приложению адресного пространства), необходимо выполнить следующую последовательность действий:

- зарезервировать регион(-ы) адресного пространства;
- передать зарезервированному(-ым) региону(-нам) физическую память из страничного файла.

Для этого предназначена функция **VirtualAlloc**:

PVOID **VirtualAlloc**(PVOID pvAddress, SIZE_T dwSize,
DWORD fdwAllocationType, DWORD fdwProtect);

В параметре ***pvAddress*** содержится адрес памяти, указывающий, где именно система должна зарезервировать адресное пространство. Обычно в качестве этого параметра передают значение NULL, тем самым сообщая функции **VirtualAlloc**, что ведущая учет свободных областей система должна зарезервировать регион там, где, по ее мнению, будет лучше. Поэтому нет никаких гарантий, что система станет резервировать регионы, начиная с нижних адресов или, наоборот, с верхних.

Так как регионы всегда резервируются с учетом гранулярности выделения памяти, то все адреса резервируемых регионов кратны 64 Кб.

Параметр ***dwSize*** указывает размер резервируемого региона в байтах. Поскольку система резервирует регионы только порциями, которые кратны размеру страницы, то попытка зарезервировать 62 Кб даст регион размером 64 Кб (если размер страницы составляет 4, 8 или 16 Кб).

Параметр ***fdwAllocationType*** сообщает системе, что именно необходимо сделать: зарезервировать регион или передать физическую память. Такое разграничение необходимо, поскольку **VirtualAlloc** позволяет не только резервировать регионы, но и передавать им физическую память. Чтобы зарезервировать регион адресного пространства, в этом параметре нужно передать идентификатор MEM_RESERVE.

Чтобы зарезервировать регион по самым старшим адресам, при вызове функции **VirtualAlloc** в параметре ***pvAddress*** необходимо передать NULL, а в качестве параметра ***fdwAllocationType*** – флаг MEM_RESERVE, скомбинированный с флагом MEM_TOP_DOWN.

Параметр ***fdwProtect*** указывает атрибут защиты, присваиваемый региону. Атрибут защиты можно охарактеризовать как управляющий набор флагов, позволяющий производить с памятью те или иные действия (в зависимости от его значения). Атрибут защиты, связанный с регионом, не влияет на память, отображаемую на этот регион. Но если ему не передана физическая память, то, какой бы атрибут защиты у него ни был, любая попытка обращения по одному из адресов в этом диапазоне приведет к нарушению доступа для данного потока.

Чтобы зарезервированный регион адресного пространства можно было использовать, необходимо выделить физическую память и спроецировать ее на этот регион (операция ***передачи физической памяти***). Для передачи физической памяти зарезервированному региону необходимо также обращаться к функции **VirtualAlloc**. Передавая физическую память регионам, нет нужды отводить ее целому региону. Можно зарезервировать регион размером 64 Кб и передать физическую память только его второй и четвертой страницам. Когда приложение передает физическую память какому-нибудь региону адресного пространства (вызывая **VirtualAlloc**), она на самом деле выделяется из файла, размещенного на жестком диске. Размер страничного файла в системе – главный фактор, определяющий количество физической памяти, доступное приложениям.

Для передачи физической памяти необходимо вызвать **VirtualAlloc** еще раз, указав в параметре ***fdwAllocationType*** не MEM_RESERVE, а MEM_COMMIT. Обычно указывается тот же атрибут защиты, что и при резервировании региона, хотя можно задавать и другой. Функции **VirtualAlloc** также сообщается, по какому адресу и сколько физической памяти следует передать. Для этого в параметр ***pvAddress*** записывается желательный адрес, а в параметр ***dwSize*** – размер физической памяти в байтах. Передавать физическую память сразу всему региону необязательно.

Например, необходимо зарезервировать регион для текущего приложения размером 2 страницы и затем передать 1-й странице физическую память размером в 1 страницу. Код будет выглядеть следующим образом.

Листинг 3

```
SystemInfo info; // объявляем переменную для хранения
                    структуры TSystemInfo
GetSystemInfo(&inf);//в качестве параметра функции
                      передаётся адрес структуры
int PAGE_SIZE = int(inf.dwPageSize); //размер страницы
```

```
        // резервируем  регион для текущего приложения
        // размером 2 страницы:
PVOID adr1 = VirtualAlloc(NULL,2* PAGE_SIZE,
MEM_RESERVE|MEM_TOP_DOWN,PAGE_READWRITE);
if(adr1==NULL){Exit(0);}
VirtualAlloc(adr1, PAGE_SIZE,MEM_COMMIT,PAGE_READWRITE);
 // передаем 1-й странице физическую память размером
    в 1  страницу
//-----------------------------------------------------
```

Если функция **VirtualAlloc** в состоянии удовлетворить запрос (зарезервировать регион), она возвращает базовый адрес зарезервированного региона (запоминается в переменной ***adr1***). Если параметр ***pvAddress*** содержал конкретный адрес, функция возвращает этот адрес, округленный при необходимости до меньшей величины, кратной 64 Кб. Если по этому адресу можно разместить регион требуемого размера, система зарезервирует его и вернет соответствующий адрес. Если же по этому адресу свободного пространства недостаточно или просто нет, система не удовлетворит запрос, и функция **VirtualAlloc** вернет NULL. Поэтому адрес, передаваемый в ***pvAddress***, должен укладываться в границы раздела пользовательского режима процесса.

После того, как физическая память зарезервирована и передана региону, можно обращаться к этой памяти для сохранения данных. Для этого можно воспользоваться функциями **FillMemory**, **CopyMemory**, **ZeroMemory**.

С помощью функции **FillMemory** можно заполнять участки памяти определёнными значениями:

void **FillMemory**(PVOID Destination, SIZE_T Length, BYTE Fill);

Назначения параметров этой функции показаны в таблице 4.

Таблица 4 – Назначения параметров функции FillMemory

Параметр	Описание
Destination	Указатель начального адреса блока памяти, который необходимо заполнить значениями *Fill*
Length	Размер блока в памяти (в байтах), который необходимо заполнить
Fill	Значение, которым необходимо заполнить блок памяти

Например, в качестве последней строки в ***Листинге 3*** можно написать:

FillMemory(adr1, PAGE_SIZE, 1);

В результате блок памяти размером PAGE_SIZE, начиная с адреса ***adr1,*** будет заполнен значениями ***1***.

С помощью функции **CopyMemory** можно копировать значения из одного участка памяти в другой:

void **CopyMemory**(PVOID Destination, const VOID* Source, SIZE_T Length);

Назначения параметров этой функции показаны в таблице 5.

Таблица 5 – Назначения параметров функции CopyMemory

Параметр	Описание
Destination	Указатель начального адреса блока памяти, в который будем осуществлять копирование
Source	Указатель начального адреса блока памяти, из которого будем осуществлять копирование
Length	Размер копируемого блока памяти в байтах

С помощью функции **ZeroMemory** можно заполнять участки памяти нулевыми значениями (обнулять):

void **ZeroMemory** (PVOID Destination, SIZE_T Length);

Назначения параметров этой функции показаны в таблице 6.

Таблица 6 – Назначения параметров функции ZeroMemory

Параметр	Описание
Destination	Указатель начального адреса блока памяти, который будем обнулять
Length	Размер этого блока памяти в байтах

Например, в качестве последней строки ***Листинга 3*** можно написать:

ZeroMemory(adr1, PAGE_SIZE);

В результате блок памяти размером PAGE_SIZE, начиная с адреса ***adr1,*** будет заполнен значениями ***0.***

Когда зарезервированный регион адресного пространства становится не нужен, его следует вернуть в общие ресурсы системы. Эта операция называется ***освобождение*** (releasing) региона. Для возврата физической памяти, отображенной на регион, или освобождения всего региона адресного пространства используется функция **VirtualFree**:

BOOL **VirtualFree** (LPVOID lpAddress, SIZE_T dwSize, DWORD dwFreeType);

Рассмотрим простейший случай вызова этой функции – для освобождения зарезервированного региона. Когда процессу больше не нужна физическая память, переданная региону, зарезервированный регион и всю связанную с ним

физическую память можно освободить единственным вызовом **VirtualFree**. В этом случае в параметр ***pvAddress*** надо поместить базовый адрес региона, т.е. значение, возвращенное функцией **VirtualAlloc** после резервирования данного региона. Системе известен размер региона, расположенного по указанному адресу, поэтому в параметре ***dwSize*** можно передать 0. Фактически это необходимо сделать, иначе вызов **VirtualFree** не даст результата. В параметре ***dwFreeType*** передаётся идентификатор MEM_RELEASE, что приводит к возврату системе всей физической памяти, отображенной на регион, и к освобождению самого региона. Освобождая регион, необходимо освободить и зарезервированное под него адресное пространство. Нельзя выделить регион размером, допустим, 128 Кб, а потом освободить только 64 Кб – надо освобождать все 128 Кб.

Если нужно, не освобождая регион, вернуть в систему часть физической памяти, переданной региону, для этого тоже следует вызвать **VirtualFree**. При этом ее параметр ***pvAddress*** должен содержать адрес, указывающий на первую возвращаемую страницу. Кроме того, в параметре ***dwSize*** задаётся количество освобождаемых байтов, а в параметре ***dwFreeType*** – идентификатор MEM_DECOMMIT.

Как и передача, возврат памяти осуществляется с учетом размерности страниц. Иначе говоря, задание адреса, указывающего на середину страницы, приведет к возврату всей страницы. Разумеется, то же самое произойдет, если суммарное значение параметров ***pvAddress*** и ***dwSize*** выпадет на середину страницы. Системе возвращаются все страницы, попадающие в диапазон от ***pvAddress*** до ***pvAddress + dwSize***.

Если же параметр ***dwSize*** равен 0, а ***pvAddress*** указывает на базовый адрес выделенного региона, **VirtualFree** вернет системе весь диапазон выделенных страниц. После возврата физической памяти освобожденные страницы доступны любому другому процессу, а попытка обращения к адресам, уже не связанным с физической памятью, приведет к нарушению доступа.

Пример 3 Работа с функциями VirtualAlloc, FillMemory, CopyMemory, ZeroMemory, VirtualFree

В качестве примера можно привести программу, которая с помощью функции **VirtualAlloc** резервирует 2 региона памяти (по 2 страницы каждый); затем обнуляет данные в 1-м регионе памяти с помощью функции **ZeroMemory**; после чего копирует с помощью функции **CopyMemory** данные из 1-го региона во 2-й, а 1-й регион заполняет значениями ***2Fh*** с помощью функции **FillMemory**. Демонстрация конечного содержимого используемых регионов памяти состоит в выводе значений, хранящихся по начальным значениям этих регионов.

Главное окно программы показано на рисунке 7.

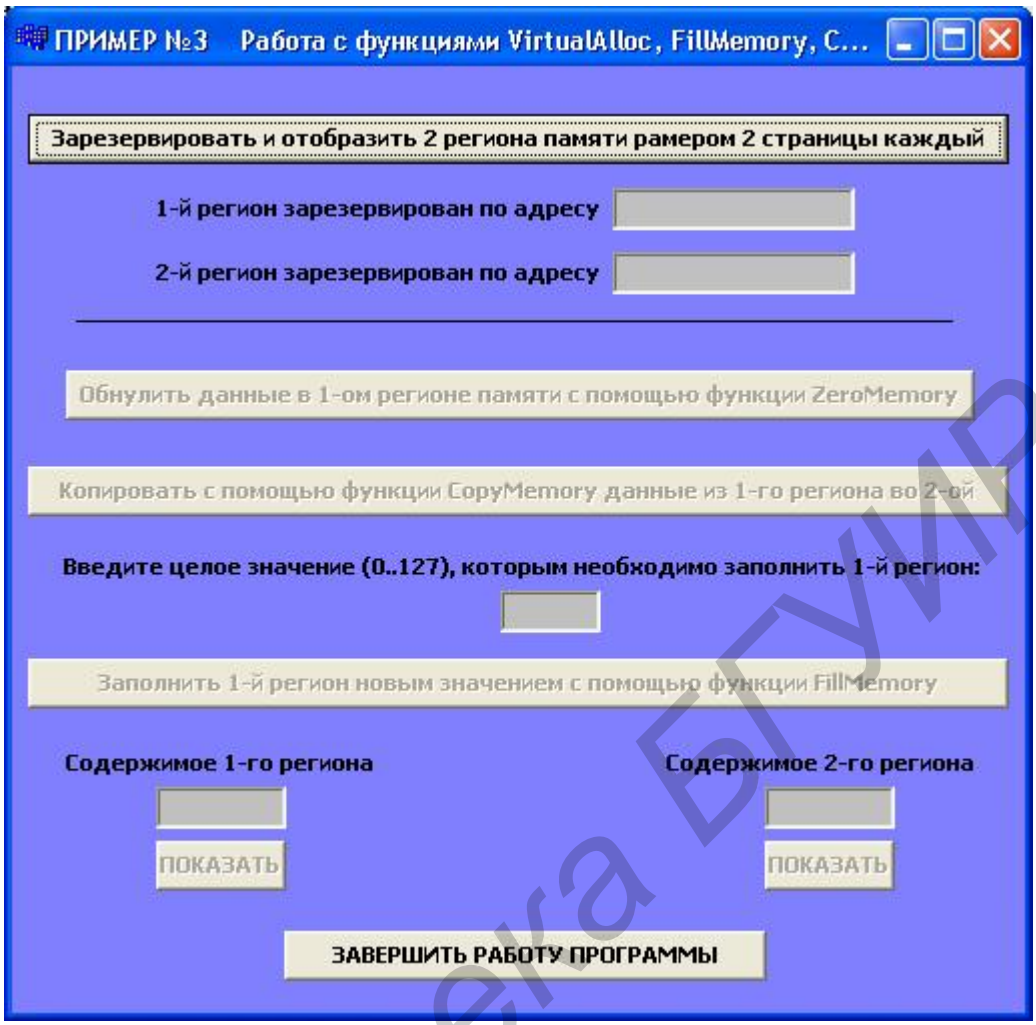

Рисунок 7 – Интерфейс программы для примера 3

Текст программы приведен ниже.

Листинг 4

```
#include <vcl.h>
#pragma hdrstop
#include "Example_3.h"
//---------------------------------------------
#pragma package(smart_init)

#pragma resource "*.dfm"
TForm1 *Form1;
PVOID adr1,adr2;
int PAGE_SIZE;

//---------------------------------------------
__fastcall TForm1::TForm1(TComponent* Owner)
  : TForm(Owner)
{
}
//---------------------------------------------
```

```
void __fastcall TForm1::Button1Click(TObject *Sender)

//Кнопка “Зарезервировать…”
{
Button1->Enabled = false; // делаем недоступной эту
 кнопку
Button2->Enabled = true; // делаем доступной кнопку
 "Обнулить..."
TSystemInfo inf;
GetSystemInfo(&inf);

PAGE_SIZE = int(inf.dwPageSize);

adr1
 =VirtualAlloc(NULL,2*PAGE_SIZE,MEM_RESERVE|MEM_TOP_DOWN,
 PAGE_READWRITE);
if(adr1==NULL){Form1->Close();}
VirtualAlloc(adr1,2*PAGE_SIZE,MEM_COMMIT,PAGE_READWRITE);
Edit1->Text=(IntToHex(int(adr1),2))+"h";

adr2 = VirtualAlloc
(NULL,2*PAGE_SIZE,MEM_RESERVE|MEM_TOP_DOWN,
PAGE_READWRITE);
if(adr2==NULL){Form1->Close();}
VirtualAlloc(adr2,2*PAGE_SIZE,MEM_COMMIT,PAGE_READWRITE);
Edit2->Text=(IntToHex(int(adr2),2))+"h";
}
//---------------------------------------------
void __fastcall TForm1::Button2Click(TObject *Sender)
 // Кнопка "Обнулить..."
{
Button2->Enabled = false;// делаем недоступной эту кнопку
Button3->Enabled = true; // делаем доступной кнопку
 "Копировать..."
Button5->Enabled = true; // делаем доступной кнопку
 "Показать" содержимое 1-го региона
 ZeroMemory(adr1,0);
 }

 //---------------------------------------------
 void __fastcall TForm1::Button3Click(TObject *Sender)
 // Кнопка "Копировать..."
 {
 Button3->Enabled = false;// делаем недоступной эту
  кнопку
 Button4->Enabled = true; // делаем доступной кнопку
  "Заполнить..."
```

```
Button6->Enabled = true; // делаем доступной кнопку
 "Показать" содержимое 2-го региона
CopyMemory(adr2,adr1,NULL);
}
//---------------------------------------------
void __fastcall TForm1::Button7Click(TObject *Sender)
{
if(adr1!=NULL)
{
 VirtualFree(adr1,NULL,MEM_RELEASE);
}
if(adr2!=NULL)
{
 VirtualFree(adr2,NULL,MEM_RELEASE);
}
Form1->Close();
}
//---------------------------------------------
void __fastcall TForm1::Button4Click(TObject *Sender)

// Кнопка "Заполнить..."
{
Button4->Enabled = false;// делаем недоступной эту
 кнопку
if(Edit3->Text=="")

{Form1->Close();}
FillMemory(adr1,2*PAGE_SIZE,__int8(StrToInt
(Edit3->Text)));
}
//---------------------------------------------
void __fastcall TForm1::Button5Click(TObject *Sender)
{
__int8 *p1= static_cast<__int8*>(adr1);
Edit4->Text = IntToHex(__int8(*(p1)),2)+"h";
    }
//---------------------------------------------
void __fastcall TForm1::Button6Click(TObject *Sender)
{
__int8 *p1= static_cast<__int8*>(adr2);
Edit5->Text = IntToHex(__int8(*(p1)),2)+"h";
}
//---------------------------------------------
```

Результат работы программы на тестовом компьютере показан на рисунке 8.

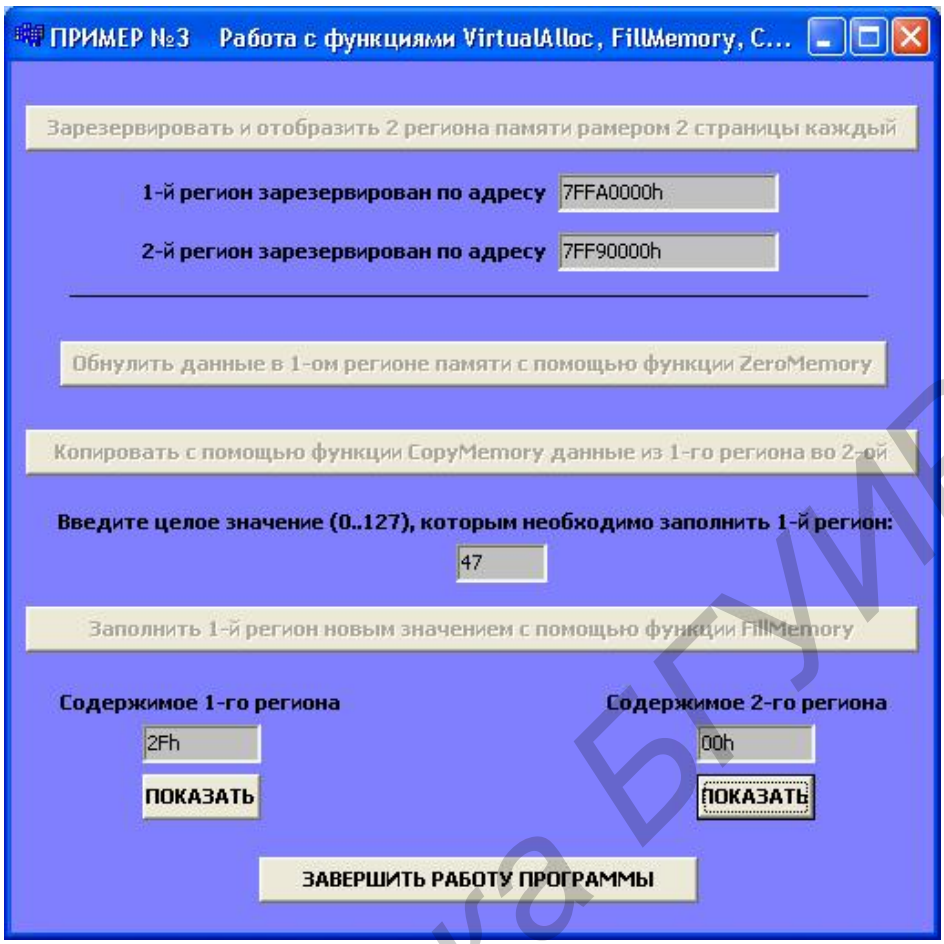

Рисунок 8 – Результат работы программы

2.4 Функция VirtualQuery

DWORD **VirtualQuery**(LPCVOID pvAddress,
PMEMORY_BASIC_INFORMATION pmbi, DWORD dwLength);

Эта функция позволяет запрашивать определенную информацию об участке памяти по заданному адресу в пределах адресного пространства вызывающего процесса: размер, тип памяти и атрибуты защиты.

При вызове **VirtualQuery** параметр ***pvAddress*** должен содержать адрес виртуальной памяти, о которой необходимо получить информацию. Параметр ***pmbi*** – это адрес структуры MEMORY_BASIC_INFORMATION, которую надо создать перед вызовом функции.

Данная структура определена в файле WinNT.h следующим образом:

```
typedef struct _MEMORY_BASIC_INFORMATION
{
    PVOID BaseAddress;
    PVOID AllocationBase;
    DWORD AllocationProtect;
    SIZE_T RegionSize;
    DWORD State;
```

```
    DWORD Protect;
    DWORD Type;
} MEMORY_BASIC_INFORMATION,
                *PMEMORY_BASIC_INFORMATION;
```

Параметр ***dwlength*** задает размер структуры MEMORY_BASIC_INFORMATION. Функция **VirtualQuery** возвращает число байтов, скопированных в буфер. Используя адрес, указанный в параметре ***pvAddress***, функция **VirtualQuery** заполняет структуру информацией о диапазоне смежных страниц, имеющих одинаковые состояние, атрибуты защиты и тип. Описание элементов структуры приведено в таблице 7.

Существует дополнительный вариант данной функции: **VirtualQueryEx**, которая принимает описатель процесса, об адресном пространстве которого возвращается информация. Чаще всего функцией **VirtualQueryEx** пользуются отладчики и системные утилиты, остальные приложения обращаются к **VirtualQuery**.

Таблица 7 – Описание элементов структуры MEMORY_BASIC_INFORMATION

Элемент	**Описание**
BaseAddress	Сообщает то же значение, что и параметр *pvAddress*, но округленное до ближайшего меньшего адреса, кратного размеру страницы
AllocationBase	Идентифицирует базовый адрес региона, включающего в себя адрес, указанный в параметре *pvAddress*
AllocationProtect	Идентифицирует атрибут защиты, присвоенный региону при его резервировании
RegionSize	Сообщает суммарный размер (в байтах) группы страниц, которые начинаются с базового адреса *BaseAddress* и имеют те же атрибуты защиты, состояние и тип, что и страница, расположенная по адресу, указанному в параметре *pvAddress*
State	Сообщает состояние (MEM_FREE, MEM_RESERVE или MEM_COMMIT) всех смежных страниц, которые имеют те же атрибуты защиты, состояние и тип, что и страница, расположенная по адресу, указанному в параметре *pvAddress*. При MEM_FREE элементы *AllocationBase*, *AllocationProtect*, *Protect* и *Type* содержат неопределенные значения, а при MEM_RESERVE неопределенное значение содержит элемент *Protect*
Protect	Идентифицирует атрибут защиты (PAGE_*) всех смежных страниц, которые имеют те же атрибуты защиты, состояние и тип, что и страница, расположенная по адресу, указанному в параметре *pvAddress*
Type	Идентифицирует тип физической памяти (MEM_IMAGE. MEM_MAPPED или MEM_PRIVATE), связанной с группой смежных страниц, которые имеют те же атрибуты защиты, состояние и тип, что и страница, расположенная по адресу, указанному в параметре *pvAddress*. В Windows 98 этот элемент всегда дает MEM_PRIVATE

Пример 4 Работа с функцией VirtualQuery

Например, необходимо зарезервировать, отобразить и заполнить значением **7*Fh*** регион памяти размером в 1 страницу и с помощью функции **VirtualQuery** собрать информацию об этом участке памяти, используя поля ***RegionSize, AllocationProtect, BaseAddress, State***.

Главное окно данной программы показано на рисунке 9.

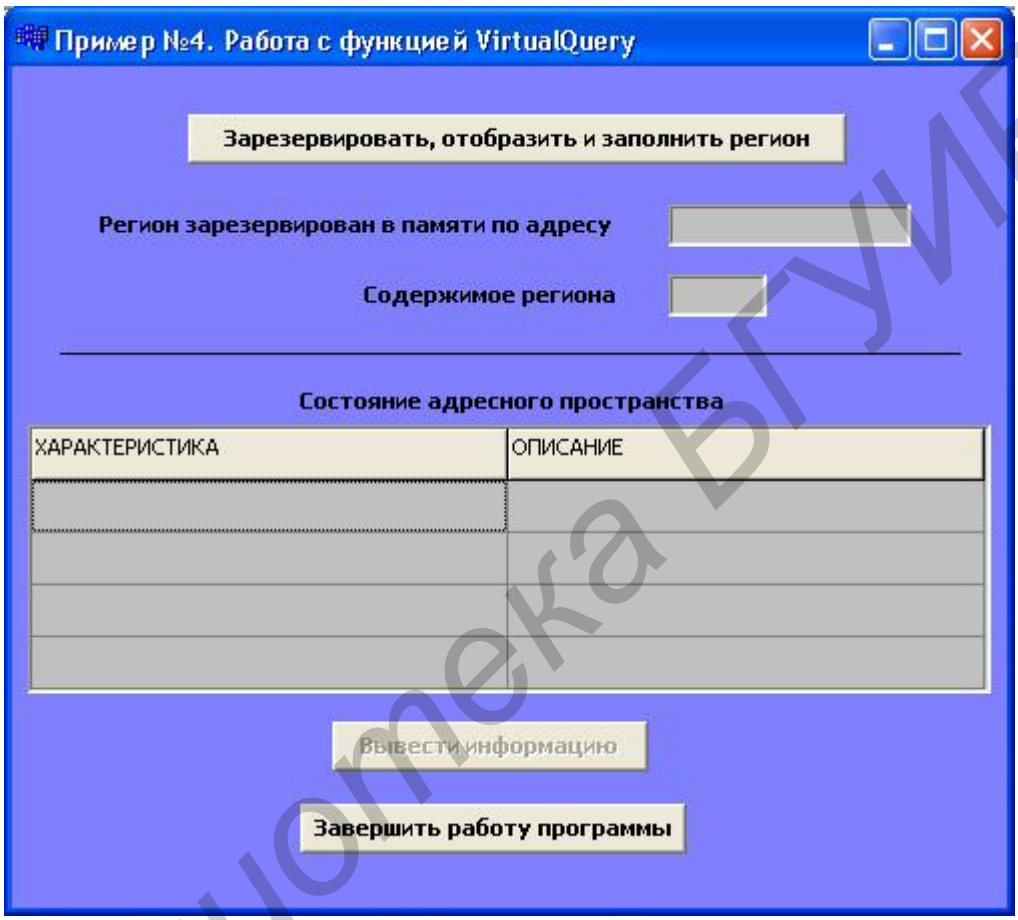

Рисунок 9 – Интерфейс программы для примера 4

Текст программы приведен ниже.

Листинг 5

```
#include <vcl.h>
#pragma hdrstop
#include "Example_4.h"
//---------------------------------------------
#pragma package(smart_init)
#pragma resource "*.dfm"
TForm1 *Form1;
PVOID adr1;
//---------------------------------------------
__fastcall TForm1::TForm1(TComponent* Owner):
 TForm(Owner)
```

```
{
StringGrid1->Cells[0][0]="ХАРАКТЕРИСТИКА";

StringGrid1->Cells[1][0]="ОПИСАНИЕ";
}
//---------------------------------------------
void __fastcall TForm1::Button1Click(TObject *Sender)
{
TSystemInfo inf;
GetSystemInfo(&inf);
adr1 =
 VirtualAlloc(NULL,inf.dwPageSize,MEM_RESERVE|MEM_TOP_
DOWN,PAGE_READWRITE);
if(adr1==NULL){Form1->Close();}
VirtualAlloc(adr1,inf.dwPageSize,MEM_COMMIT,PAGE_
READWRITE);
Edit1->Text=(IntToHex(int(adr1),2))+"h";
FillMemory(adr1,inf.dwPageSize,127);
__int8 *p= static_cast<__int8*>(adr1);
Edit2->Text = IntToHex(__int8(*(p)),2)+"h";
Button1->Enabled = false;
Button2->Enabled = true;
}
//---------------------------------------------
void __fastcall TForm1::Button2Click(TObject *Sender)
{
PMEMORY_BASIC_INFORMATION pmbi = new
 MEMORY_BASIC_INFORMATION;
VirtualQuery(adr1,pmbi,sizeof(*pmbi));
StringGrid1->Cells[0][1]="Суммарный размер (в байтах)
 группы страниц";
StringGrid1->Cells[1][1]=IntToStr(__int64
(pmbi->RegionSize))+" или
"+IntToStr(__int64(pmbi->RegionSize)/1024)+"Kbytes" ;
StringGrid1->Cells[0][2]="Атрибут защиты региона";
switch(pmbi->AllocationProtect)
{
case PAGE_READWRITE: StringGrid1-> Cells[1] [2]=
  "PAGE_READWRITE"; break;
case PAGE_EXECUTE:StringGrid1->Cells[1][2]=
"PAGE_EXECUTE"; break;
default:StringGrid1->Cells[1][2]="Unknown"; break;
}
StringGrid1->Cells[0][3]="Базовый адрес региона";
StringGrid1->Cells[1][3]=IntToHex(int(pmbi->Base
Address),2)+"h";
StringGrid1->Cells[0][4]="Тип региона";
switch(pmbi->State)
```

```
{
case MEM_FREE: StringGrid1->Cells[1][4]="FREE"; break;
case MEM_RESERVE: StringGrid1->Cells[1][4]="RESERVE";
 break;
case MEM_COMMIT: StringGrid1->Cells[1][4]="COMMIT";
 break;
default:StringGrid1->Cells[1][4]="Unknown";
break;
}
Button2->Enabled = false;
}
//---------------------------------------------
void __fastcall TForm1::Button3Click(TObject *Sender)
{
if(adr1!=NULL)
{
 VirtualFree(adr1,NULL,MEM_RELEASE);
}
Form1->Close();
}
//---------------------------------------------
```

Результат работы программы на тестовом компьютере показан на рисунке 10.

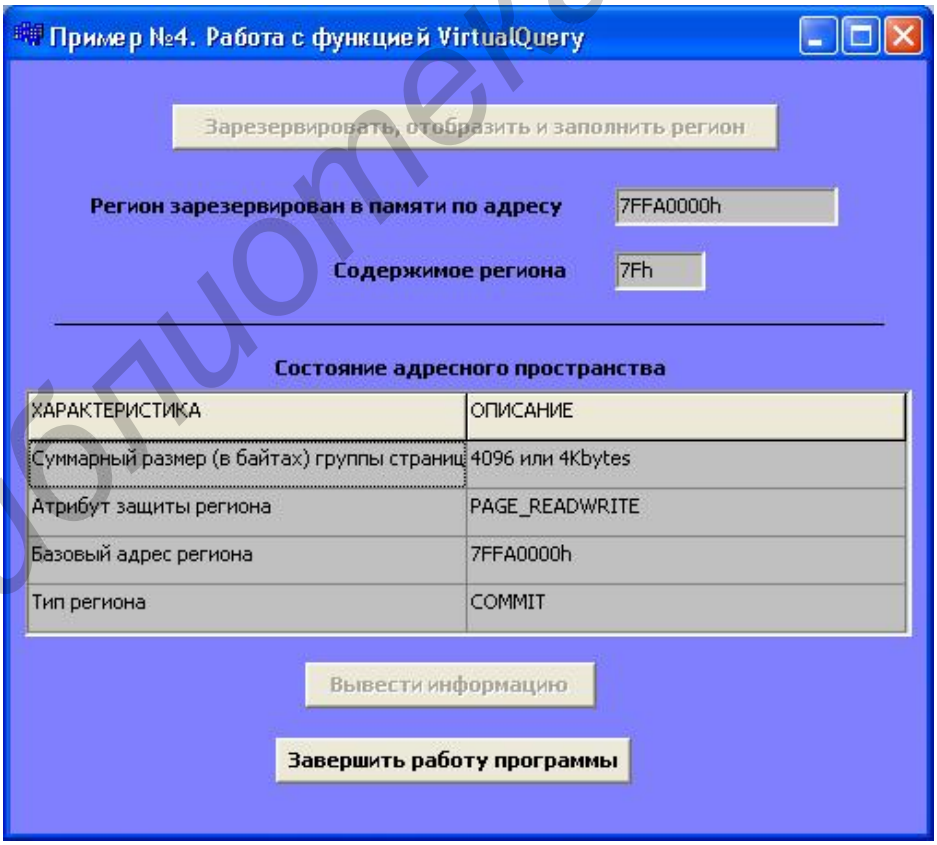

Рисунок 10 – Результат работы программы

2.5 Функция VirtualProtect

Как уже отмечалось, отдельным страницам физической памяти можно присвоить свои атрибуты защиты (таблица 8).

Хоть это и не принято, но атрибуты защиты, присвоенные странице или страницам переданной физической памяти, можно изменять. Для этого необходимо воспользоваться функцией **VirtualProtect**:

BOOL **VirtualProtect**(PVOID pvAddress, SIZE_T dwSize, DWORD flNewProtect, PDWORD pflOldProtect);

Таблица 8 – Описание атрибутов защиты

Атрибут защиты	**Описание**
PAGE_NOACCESS	Попытки чтения, записи или исполнения содержимого памяти на этой странице вызывают нарушение доступа
PAGE_READONLY	Попытки записи или исполнения содержимого памяти на этой странице вызывают нарушение доступа
PAGE_READWRITE	Попытки исполнения содержимого памяти на этой странице вызывают нарушение доступа
PAGE_EXECUTE	Попытки чтения или записи на этой странице вызывают нарушение доступа
PAGE_EXECUTE_READ	Попытки записи на этой странице вызывают нарушение доступа
PAGE_EXECUTE_READWRITE	На этой странице возможны любые операции
PAGE_WRITECOPY	Попытки исполнения содержимого памяти на этой странице вызывают нарушение доступа; попытка записи приводит к тому, что процессу предоставляется «личная» копия данной страницы
PAGE_EXECUTE_WRITECOPY	На этой странице возможны любые операции; попытка записи приводит к тому, что процессу предоставляется «личная» копия данной страницы

Здесь ***pvAddress*** указывает на базовый адрес памяти (который должен находиться в пользовательском разделе процесса), ***dwSize*** определяет число байтов, для которых изменяется атрибут защиты, ***flNewProtect*** содержит один из идентификаторов PAGE_*, кроме PAGE_WRITECOPY и PAGE_EXECUTE_WRITECOPY. Параметр ***pflOldProtect*** содержит адрес переменной типа DWORD, в которую **VirtualProtect** заносит старое значение атрибута защиты для данной области памяти. В этом параметре (даже если такая информация не интересует) нужно передать корректный адрес, иначе функция приведет к нарушению доступа. Атрибуты защиты связаны с целыми страницами памяти и не могут присваиваться отдельным байтам.

Функцию **VirtualProtect** нельзя использовать для изменения атрибутов защиты страниц, диапазон которых охватывает разные зарезервированные регионы. В таких случаях **VirtualProtect** надо вызывать для каждого региона отдельно.

Пример 5 Работа с функцией VirtualProtect

В качестве примера можно привести самый простой случай: программа резервирует и отображает регион размером в 1 страницу с атрибутом PAGE_READWRITE; если пользователь нажимает кнопку «Использовать функцию VirtualProtect», то программа изменит атрибут PAGE_READWRITE зарезервированного региона на PAGE_NOACCESS, что приведёт к запрету доступа процесса к данному участку памяти. Чтобы убедиться в корректности работы программы, используется функция **IsBadReadPtr,** позволяющая определить доступность текущему процессу блока памяти, адрес которого передается ей в качестве параметра:

BOOL **IsBadReadPtr** (const VOID* *lp*, UINT_PTR *ucb*);

Параметр ***lp*** указывает на первый байт требуемого блока памяти, а параметр ***ucb*** содержит размер этого блока памяти в байтах.

В случае если процессу доступны для чтения все байты из указанного блока памяти, то функция **IsBadReadPtr** возвращает 0, иначе 1.

Интерфейс программы показан на рисунке 11.

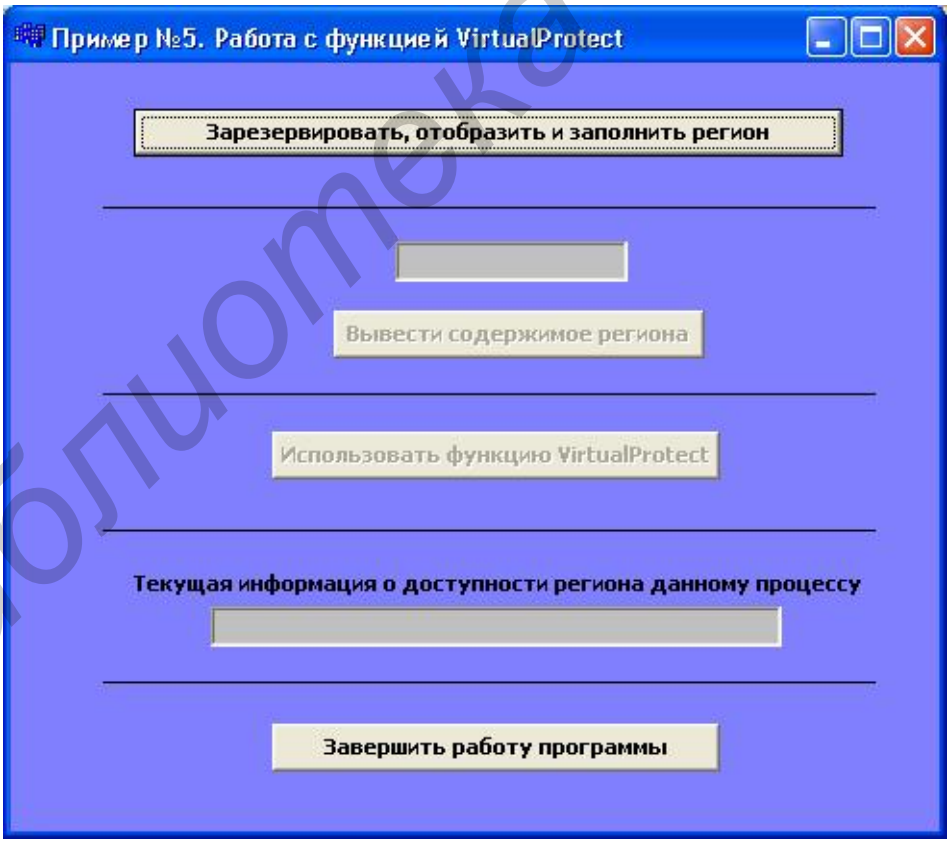

Рисунок 11 – Интерфейс программы для примера 5

Текст программы приведен ниже.

Листинг 6

```
//$$---- Form CPP ----

//---------------------------------------------------
#include <vcl.h>
#pragma hdrstop
#include "Example_5.h"

//---------------------------------------------------
#pragma package(smart_init)
#pragma resource "*.dfm"
TForm1 *Form1;
PVOID   adr1;
TSystemInfo inf;

//---------------------------------------------------
__fastcall TForm1::TForm1(TComponent* Owner):
  TForm(Owner)
{
Button2->Enabled=false;
Button3->Enabled=false;
}
//---------------------------------------------------
void __fastcall TForm1::Button1Click(TObject *Sender)

// Кнопка "Зарезервировать..."
{
Button1->Enabled=false; // Делаем недоступной эту кнопку
Button2->Enabled=true; // Делаем доступной кнопку
 “Вывести...”
GetSystemInfo(&inf);

adr1 = VirtualAlloc(NULL,inf.dwPageSize,MEM_RESERVE|
MEM_TOP_DOWN,PAGE_READWRITE);
if(adr1!=NULL){VirtualAlloc(adr1,inf.dwPageSize,MEM_COMM
 IT,PAGE_READWRITE);}
FillMemory(adr1,inf.dwPageSize,47);

BOOL b = IsBadReadPtr(adr1,inf.dwPageSize);
if(!b)
{
Edit2->Text="Регион доступен для чтения данному
 процессу!";
}else
{Edit2->Text="Регион НЕ доступен для чтения данному
 процессу!";}
}
//---------------------------------------------------
```

```
void __fastcall TForm1::Button4Click(TObject *Sender)
// Кнопка "Завершить..."
{
Form1->Close();
}

//---------------------------------------------------

void __fastcall TForm1::Button2Click(TObject *Sender)
// Кнопка “Вывести...”
{
Button2->Enabled=false; // Делаем недоступной эту кнопку
Button3->Enabled=true; // Делаем доступной кнопку
 "Использовать..."
__int8 *p1= static_cast<__int8*>(adr1);
Edit1->Text = IntToHex(__int8(*(p1)),2)+"h";
}

//---------------------------------------------

void __fastcall TForm1::Button3Click(TObject *Sender)

// Кнопка "Использовать..."
{
Button3->Enabled=false; // Делаем недоступной эту кнопку
DWORD per3;
VirtualProtect(adr1,inf.dwPageSize,PAGE_NOACCESS,&per3);
BOOL b = IsBadReadPtr(adr1,inf.dwPageSize);
if(!b)
{
Edit2->Text="Регион доступен для чтения данному
 процессу!";
}else
{Edit2->Text="Регион НЕ доступен для чтения данному
 процессу!";}
}
//---------------------------------------------
```

Результат работы программы на тестовом компьютере показан на рисунке 12.

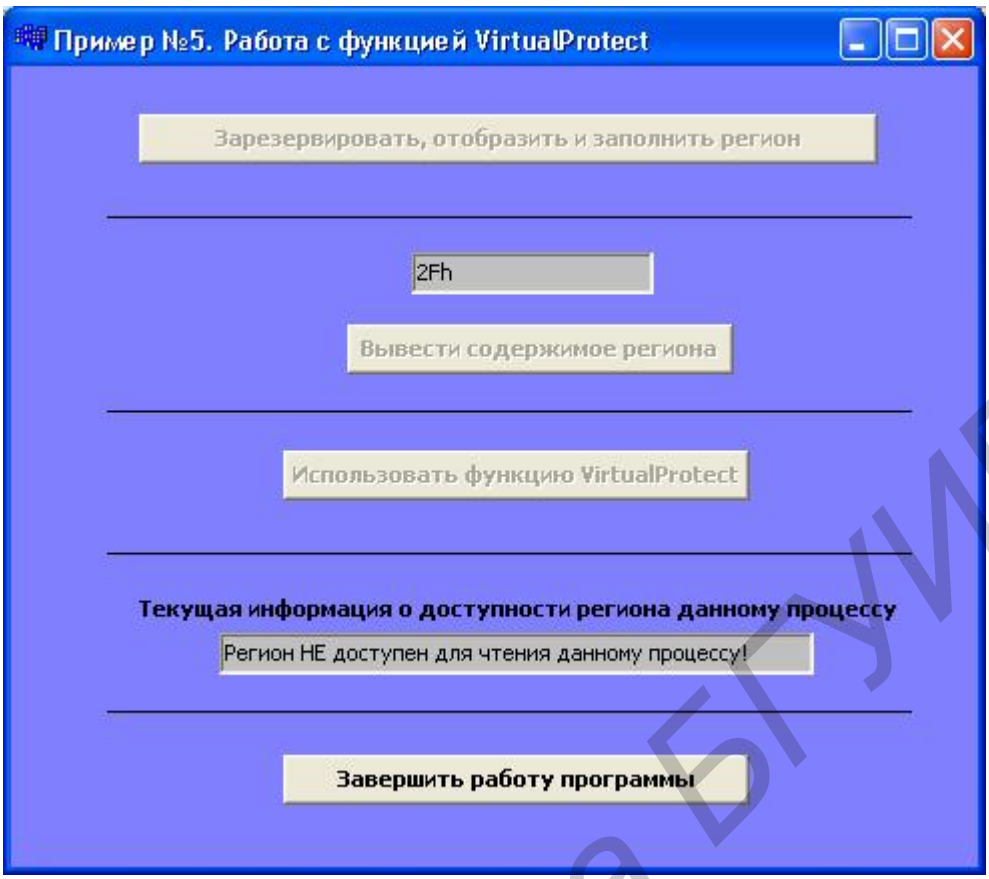

Рисунок 12 – Результат работы программы

2.6 Обработка ошибок

Как известно, любое «правильно» спроектированное приложение должно соответствующим образом реагировать на любые действия со стороны пользователя, в том числе и ошибочные. Ошибки – это очень частое явление, как в действиях пользователей, так и программистов. Поэтому важной задачей со стороны программиста является контроль и, по-возможности, информирование пользователя о правильности его действий.

Изучая описания API-функций, можно заметить, что почти каждая функция возвращает значение определенного типа. Дело в том, что когда функция вызывается, она проверяет переданные ей параметры, а затем пытается выполнить свою работу. Если был передан недопустимый параметр или если данную операцию нельзя выполнить по какой-то другой причине, функция возвращает значение, свидетельствующее об ошибке. Для примера в таблице 9 показаны типы данных для возвращаемых значений большинства функций Windows.

Таблица 9 – Стандартные типы значений, возвращаемых функциями Windows

Тип данных	Значение, свидетельствующее об ошибке
VOID	Функция всегда (или почти всегда) выполняется успешно (таких функций в Windows очень мало)
BOOL	Если вызов функции заканчивается неудачно, возвращается 0; в остальных случаях возвращаемое значение отлично от 0 (нельзя проверять его на соответствие TRUE или FALSE)
HANDLE	Если вызов функции заканчивается неудачно, то обычно возвращается NULL; в остальных случаях HANDLE идентифицирует объект, которым программист может манипулировать. Некоторые функции возвращают HANDLE со значением INVALID_HANDLE_VALUE, равным -1
PVOID	Если вызов функции заканчивается неудачно, возвращается NULL; в остальных случаях PVOID сообщает адрес блока данных в памяти
LONG или DWORD	Функции, которые сообщают значения каких-либо счетчиков, обычно возвращают LONG или DWORD. Если по какой-то причине функция не сумела сосчитать то, что требовалось, она обычно возвращает 0 или -1 (все зависит от конкретной функции)

За каждой ошибкой закреплен свой код – 32-битное число. Когда функция возвращает управление, ее возвращаемое значение будет указывать на то, что произошла какая-то ошибка. Чтобы узнать, какая именно ошибка произошла, необходимо вызвать функцию **GetLastError**:

DWORD **GetLastError**();

Эта функция возвращает 32-битный код ошибки. Список кодов ошибок, определенных Microsoft, содержится в заголовочном файле WinError.h. В этом файле с каждой ошибкой связан идентификатор сообщения. Его можно использовать в исходном коде для сравнения со значением, возвращаемым **GetLastError.** Функцию **GetLastError** нужно вызывать сразу же после неудачного вызова функции Windows, иначе код ошибки может быть потерян.

Обычно, если приложение обнаруживает какую-нибудь ошибку, то, как правило, сообщает о ней пользователю, выводя на экран ее описание. В Windows для этого есть специальная функция, которая «конвертирует» код ошибки в ее описание – **FormatMessage**:

DWORD **FormatMessage** (DWORD *dwFlags*, LPCVOID *pSource*,
DWORD *dwMessageId*, DWORD *dwLanguageId*,
PTSTR *pszBuffer*, DWORD *nSize*, va_list **Arguments*);

FormatMessage – весьма богатая по своим возможностям функция, и именно ее желательно применять при формировании всех строк, показываемых пользователю. Дело в том, что она позволяет легко работать со множеством

языков. **FormatMessage** определяет, какой язык выбран в системе в качестве основного (этот параметр задается через апплет Regional Settings в Control Panel), и возвращает текст на соответствующем языке.

Первый параметр ***dwFlags*** представляет собой набор флагов: флаг FORMAT_MESSAGE_FROM_SYSTEM сообщает функции, что нужна строка, соответствующая коду ошибки, определенному в системе; флаг FORMAT_MESSAGE_ALLOCATE_BUFFER указывает функции выделить соответствующий блок памяти для хранения текста. Описатель этого блока будет возвращен в переменной ***pszBuffer***, а минимальный размер этого блока в байтах указывается в шестом параметре – ***nSize***. При таком наборе флагов в первом параметре второй игнорируется, поэтому в качестве последнего передается NULL. Третий параметр указывает код ошибки (полученный с помощью **GetLastError**), а четвертый – язык, на котором производится ее описание. Чтобы выводилось сообщение на русском языке, необходимо также указать в качестве четвертого параметра соответствующий языковый идентификатор, который можно получить, воспользовавшись функцией **MAKELANGID**():

WORD **MAKELANGID** (USHORT *usPrimaryLanguage*,
USHORT *usSubLanguage*);

В качестве параметров этой функции необходимо передать языковые идентификаторы, описание которых можно найти в документации к среде, в которой будете разрабатывать приложение. Мы же будем использовать следующую комбинацию: в качестве первого параметра будем передавать LANG_NEUTRAL, а в качестве второго − SUBLANG_DEFAULT. В результате функция возвратит языковый идентификатор, соответствующий ***«User default language»*** (как указано в документации). На тестовом компьютере пользовательским языком по умолчанию являлся русский.

Если выполнение **FormatMessage** заканчивается успешно, описание ошибки помещается в блок памяти.

Для вывода сообщения на экран можно воспользоваться функцией **MessageBox**:

int WINAPI **MessageBox**(HWND *hwndParent*, LPCSTR *lpszText*,
LPCSTR *lpszTitle*, UINT *fuStyle*);

Эта функция создает на экране диалоговую панель с текстом, заданным параметром ***lpszText***, и заголовком, заданным параметром ***lpszTitle***. Если заголовок указан как NULL, используется заголовок по умолчанию – строка «Error». Параметр ***hwndParent*** указывает идентификатор родительского окна, создающего диалоговую панель. Этот параметр можно указывать как NULL, в этом случае у диалоговой панели не будет родительского окна. Можно вызвать

функцию **MessageBox()** из функции диалога, в этом случае первый параметр должен содержать идентификатор окна диалоговой панели.

Последний параметр ***fuStyle*** определяет стиль и внешний вид диалоговой панели. Можно использовать этот параметр для отображения кнопки «ОК» и пиктограммы (MB_ICONERROR).

Желательно также пользоваться функцией **wsprintf**(), которая позволяет выводить собщения формата: «ОШИБКА X: Y», где X и Y – сообщения об ошибке, которые будем получать от системы:

int **wsprintf**(LPTSTR *lpOut*, LPCTSTR *lpFmt*, ...).

В качестве первого параметра ***lpOut*** передаётся указатель на выходной массив, в качестве второго – указатель на строку, содержащую аргументы, которые управляют форматом этой строки.

Пример 6 Обработка ошибок

В приведённом ниже примере показан общий принцип обработки возможных ошибок при вызовах API-функций для работы с виртуальной памятью. Для использования даного примера для обработки ошибок при работе с другими функциями API необходимо просто по соответствующей документации изучить описание этих функций, выяснить возможные варианты значений, возвращаемых функциями, и далее аналогично организовать обработку возможных ошибок.

В данном примере пользователю предлагается зарезервировать и отобразить регион памяти: в первом случае он резервируется с атрибутом защиты PAGE_READWRITE, а во втором – PAGE_WRITECOPY. При резервировании адресного пространства или передаче физической памяти через **VirtualAlloc** нельзя указывать атрибут PAGE_WRITECOPY, иначе вызов **VirtuatAlloc** даст ошибку, а **GetLastError** вернет код ERROR_INVALID_PARAMETER. Этот атрибут используется операционной системой, только когда она проецирует образы EXE- или DLL-файлов. Что же касается резервирования и отображения региона с атрибутом PAGE_READWRITE, то в случае успешного выполнения данных операций **VirtuatAlloc** вернет начальный адрес региона.

Интерфейс программы показан на рисунке 13.

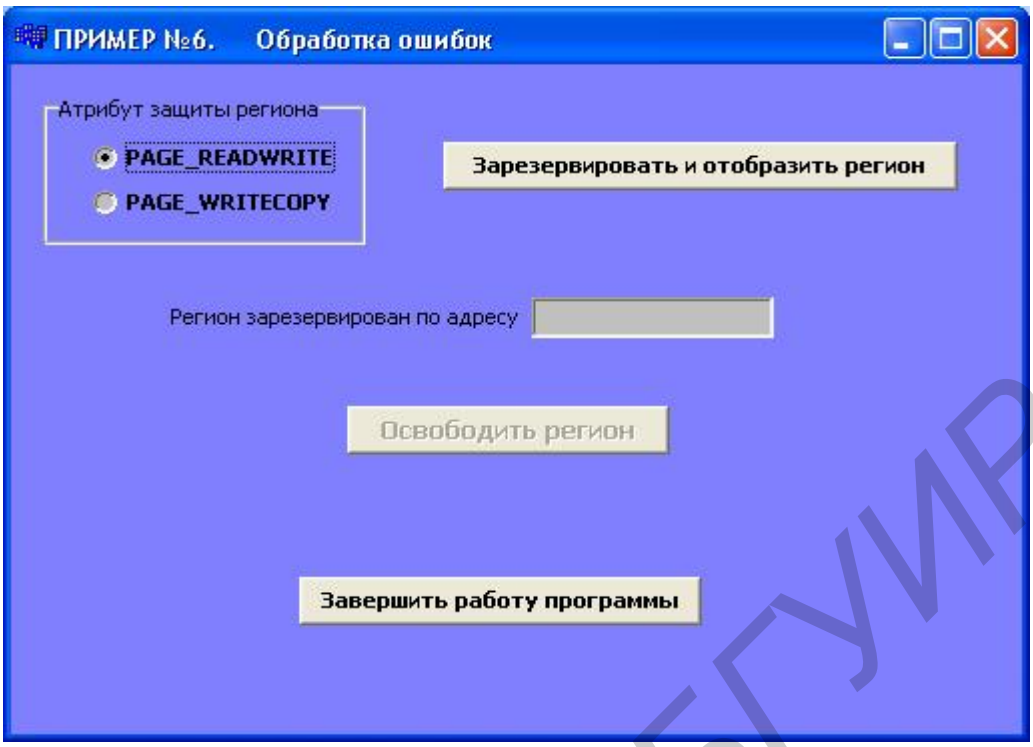

Рисунок 13 – Интерфейс программы для примера 6

Текст программы приведен ниже.

Листинг 7

```
//$$---- Form CPP ----
//---------------------------------------------------
#include <vcl.h>
#pragma hdrstop

#include "Example_6.h"

//---------------------------------------------------
#pragma package(smart_init)
#pragma resource "*.dfm"
TForm1 *Form1;
PVOID adr;

//---------------------------------------------------
__fastcall TForm1::TForm1(TComponent* Owner)
  : TForm(Owner)
{

Button2->Enabled = false;
}

//---------------------------------------------------
```

```
void __fastcall TForm1::Button1Click(TObject *Sender)

// Кнопка “Зарезервировать...”
{
TSystemInfo inf;
GetSystemInfo(&inf);

if(RadioButton1->Checked == true)
{
adr
VirtualAlloc(NULL,inf.dwPageSize,MEM_RESERVE|MEM_TOP_
DOWN,PAGE_READWRITE);
   if(adr)
   {

VirtualAlloc(adr,inf.dwPageSize,MEM_COMMIT,PAGE_READ
WRITE);
    Edit1->Text=(IntToHex(int(adr),2))+"h";
    Button1->Enabled = false;
    Button2->Enabled = true;
   } else
   {
    TCHAR error[80];
    LPVOID lpMsgBuf;
    int er = GetLastError();
    FormatMessage(FORMAT_MESSAGE_ALLOCATE_BUFFER |
 FORMAT_MESSAGE_FROM_SYSTEM,NULL,er,MAKELANGID(LANG_NEU
 TRAL, SUBLANG_DEFAULT),(LPTSTR) &lpMsgBuf,0, NULL );
 wsprintf(error,"ОШИБКА %d: %s", er, lpMsgBuf);

 MessageBox(NULL,error,"ОШИБКА!!!",MB_OK|MB_ICONERROR);
   }
} else if(RadioButton2->Checked == true)
{
adr = VirtualAlloc(NULL,inf.dwPageSize,MEM_RESERVE|MEM_
TOP_DOWN,PAGE_WRITECOPY);
 if(adr)
 {
    VirtualAlloc(adr,inf.dwPageSize,MEM_COMMIT,PAGE_
WRITECOPY);
    Edit1->Text=(IntToHex(int(adr),2))+"h";
    Button1->Enabled = false;
    Button2->Enabled = true;
 } else
 {
    TCHAR error[80];
    LPVOID lpMsgBuf;
    int er = GetLastError();
```

```
FormatMessage(FORMAT_MESSAGE_ALLOCATE_BUFFER
FORMAT_MESSAGE_FROM_SYSTEM,NULL,er,MAKELANGID(LANG_
NEUTRAL, SUBLANG_DEFAULT),(LPTSTR) &lpMsgBuf,0, NULL );
     wsprintf(error,"ОШИБКА %d: %s", er, lpMsgBuf);

 MessageBox(NULL,error,"ОШИБКА!!!",MB_OK|MB_ICONERROR);
 }
}
}

//---------------------------------------------------
void __fastcall TForm1::Button3Click(TObject *Sender)
 // Кнопка “Завершить...”
{
if(adr!=NULL)
{
 VirtualFree(adr,0,MEM_RELEASE);
}
Form1->Close();
}

//-----------------------------------------------------void
 __fastcall TForm1::Button2Click(TObject *Sender)
 // Кнопка “Освободить...”
{
if(adr!=NULL)
{
 VirtualFree(adr,0,MEM_RELEASE);
 Button2->Enabled=false;
 Button1->Enabled=true;
 Edit1->Text="";
}
}

//---------------------------------------------
```

Результат работы программы на тестовом компьютере показан на рисунке 14.

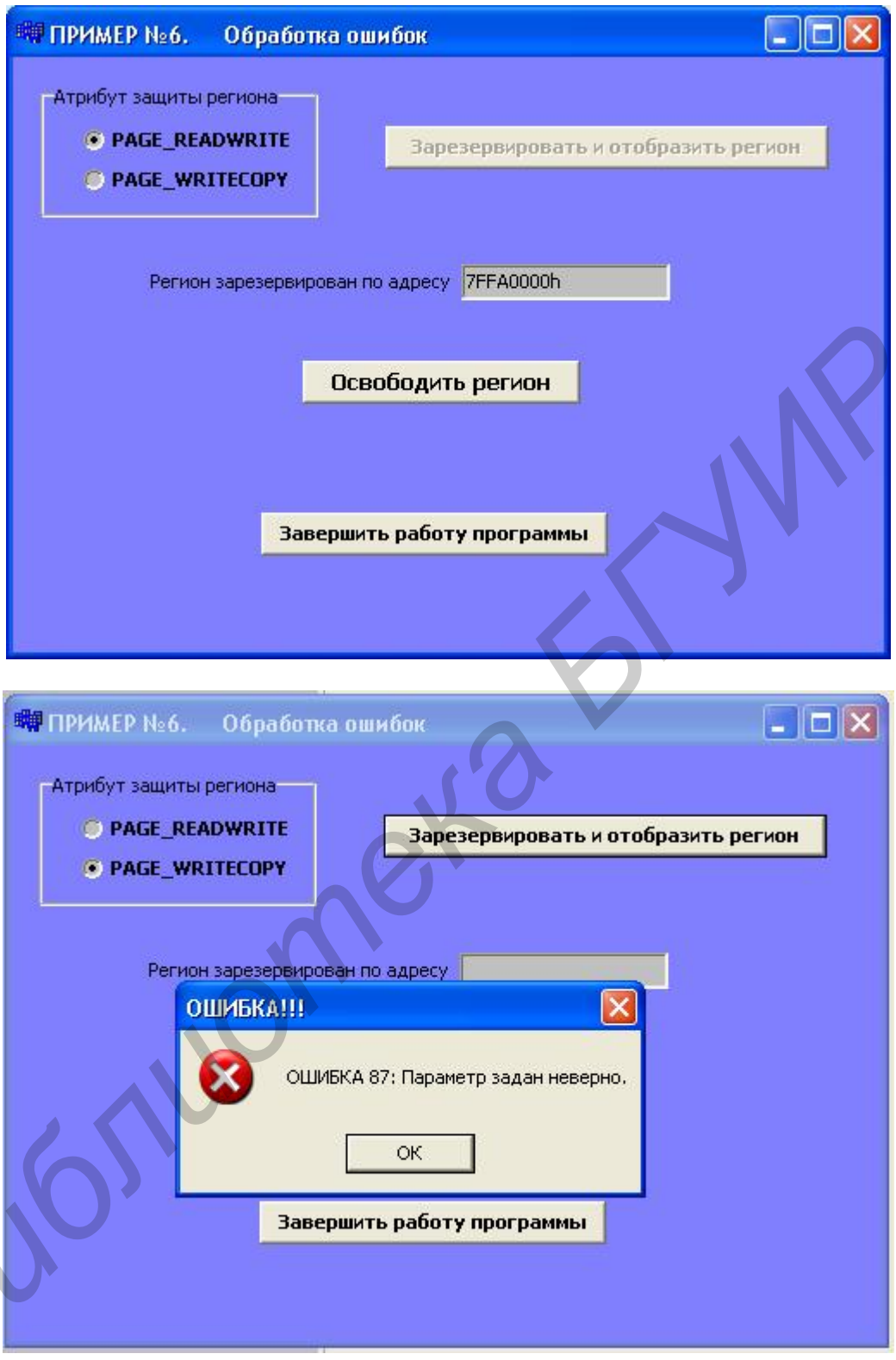

Рисунок 14 – Результат работы программы

3 Описание лабораторных работ

Лабораторная работа №1 Ознакомление с основными функциями для работы с виртуальной памятью

Цель работы

Получить практические навыки по использованию основных API-функций для работы с виртуальной памятью.

Порядок выполнения работы

1 Изучить описание и особенности применения представленных в пособии API-функций для работы с виртуальной памятью.

2 Получить у преподавателя задание для выполнения практической части работы.

3 Продемонстрировать результат работы преподавателю и защитить её.

Варианты заданий

Вариант 1 Получить размер страницы виртуальной памяти и другие характеристики, используя функцию **GetSystemInfo**. Проверить текущее состояние памяти с помощью функции **GlobalMemoryStatus**. С помощью функции **VirtualAlloc** зарезервировать 2 региона памяти (по 2 страницы каждый). Заполнить 1-й регион предварительно сгенерированными случайными числами. Скопировать с помощью функции **CopyMemory** данные из 1-го региона во 2-й. Обнулить данные в 1-м регионе памяти с помощью функции **ZeroMemory**. Используя функцию **VirtualQuery**, продемонстрировать системную информацию и содержимое используемых регионов памяти. Освободить всю зарезервированную ранее память с помощью функции **VirtualFree**.

Вариант 2 Получить размер страницы виртуальной памяти и другие характеристики, используя функцию **GetSystemInfo**. Проверить текущее состояние памяти с помощью функции **GlobalMemoryStatus**. С помощью функции **VirtualAlloc** зарезервировать 2 региона памяти (по 2 страницы каждый). С помощью функции **FillMemory** заполнить 1-й регион значениями **0Fh**. Скопировать с помощью функции **CopyMemory** данные из 1-го региона во 2-й. Обнулить данные в 1-м регионе памяти с помощью функции **ZeroMemory**. Используя функцию **VirtualQuery**, продемонстрировать системную информацию и содержимое используемых регионов памяти. Освободить всю зарезервированную ранее память с помощью функции **VirtualFree**.

Вариант 3 Получить размер страницы виртуальной памяти и другие характеристики, используя функцию **GetSystemInfo**. Проверить текущее состояние памяти с помощью функции **GlobalMemoryStatus**. С помощью

функции **VirtualAlloc** зарезервировать 2 региона памяти (по 2 страницы каждый). С помощью функции **FillMemory** заполнить 1-й регион значениями **0Fh**. Скопировать с помощью функции **CopyMemory** данные из 1-го региона во 2-й. Обнулить данные в 1-м регионе памяти с помощью функции **ZeroMemory**. Используя функции **VirtualQuery**, продемонстрировать системную информацию и содержимое используемых регионов памяти. Освободить всю зарезервированную ранее память с помощью функции **VirtualFree**.

Вариант 4 Получить размер страницы виртуальной памяти и другие характеристики, используя функцию **GetSystemInfo**. Проверить текущее состояние памяти с помощью функции **GlobalMemoryStatus**. С помощью функции **VirtualAlloc** зарезервировать 2 региона памяти (по 2 страницы каждый). Обнулить данные в 1-м регионе памяти с помощью функции **ZeroMemory**. Скопировать с помощью функции **CopyMemory** данные из 1-го региона во 2-й. 1-й регион заполнить значениями **2Fh** с помощью функции **FillMemory.** Используя функцию **VirtualQuery**, продемонстрировать системную информацию и содержимое используемых регионов памяти. Освободить всю зарезервированную ранее память с помощью функции **VirtualFree**.

Вариант 5 Получить размер страницы виртуальной памяти и другие характеристики, используя функцию **GetSystemInfo**. Проверить текущее состояние памяти с помощью функции **GlobalMemoryStatus**. С помощью функции **VirtualAlloc** зарезервировать 2 региона памяти (по 2 страницы каждый). 1-й регион заполнить значениями **7Fh** с помощью функции **FillMemory.** Скопировать с помощью функции **CopyMemory** данные из 1-го региона во 2-й. Обнулить данные в 1-м регионе памяти с помощью функции **ZeroMemory**. Используя функцию **VirtualQuery**, продемонстрировать системную информацию и содержимое используемых регионов памяти. Освободить всю зарезервированную ранее память с помощью функции **VirtualFree**.

Вариант 6 Получить размер страницы виртуальной памяти и другие характеристики, используя функцию **GetSystemInfo**. Проверить текущее состояние памяти с помощью функции **GlobalMemoryStatus**. С помощью функции **VirtualAlloc** зарезервировать 2 региона памяти (по 2 страницы каждый). Заполнить 1-й регион предварительно сгенерированными случайными числами. Скопировать с помощью функции **CopyMemory** данные из 1-го региона во 2-й. 1-й регион заполнить значениями **9Fh** с помощью функции **FillMemory.** Используя функцию **VirtualQuery**, продемонстрировать системную информацию и содержимое используемых регионов памяти. Освободить всю зарезервированную ранее память с помощью функции **VirtualFree**.

Лабораторная работа №2 Изучение особенностей использования функции VirtualProtect

Цель работы

Получить практические навыки по использованию функции **VirtualProtect**.

Порядок выполнения работы

1 Изучить описание и особенности применения функции **VirtualProtect** для управления виртуальной памятью.

2 Получить у преподавателя задание для выполнения практической части работы.

3 Продемонстрировать результат работы преподавателю и защитить её.

Задание

Модифицировать программу из **лабораторной работы №1** следующим образом:

1) добавить код для демонстрации работы функции **VirtualProtect**;

2) дополнить программу кодом для обработки возможных ошибок (выдать сообщение об ошибке) при вызовах API-функций для работы с виртуальной памятью.

При обработке ошибок там, где это возможно, использовать функции **GetLastError** и **FormatMessage**.

Литература

1 Рихтер, Дж. Windows для профессионалов. Создание эффективных Win32-приложений с учетом специфики 64-разрядной версии Windows / Дж. Рихтер; пер.с англ. – 4-е изд. – СПб. : Питер; М. : Издательско-торговый дом «Русская Редакция», 2004.

2 Олифер, В. Г. Сетевые операционные системы / В. Г. Олифер, Н. А. Олифер. – СПб. : Питер, 2002.

3 Щупак, Ю. А. Win32 API. Эффективная разработка приложений / Ю. А. Щупак.– СПб. : Питер, 2006.

4 Дейтел, Х. Как программировать на C++/ Х. Дейтел, П. Дейтел; пер. с англ. – 3-е изд. – М. : БИНОМ, 2003.

5 Лаптев, В. В. C++. Экспресс-курс / В. В. Лаптев. – СПб. : БХВ-Петербург, 2004. – 512 с. : ил.

6 Культин, Н. Б. Самоучитель C++ Builder / Н. Б. Культин. – СПб. : БХВ-Петербург, 2004.

Содержание

Св. план 2007, поз. 25

Учебное издание

Лихачев Денис Сергеевич

ВИРТУАЛЬНАЯ ПАМЯТЬ

Лабораторный практикум
по курсу
«Системное программирование»
для студентов специальности I-40 02 02
«Электронные вычислительные средства»
дневной формы обучения

Редактор Е. Н. Батурчик
Корректор М. В. Тезина

Подписано в печать 23.05.2007. Формат 60x84 1/16. Бумага офсетная.
Гарнитура «Таймс». Печать ризографическая. Усл. печ. л. 3,02.
Уч.-изд. л. 2,7. Тираж 200 экз. Заказ 31.

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