Министерство образования Российской Федерации

**Комсомольский-на-Амуре государственный технический университет**

**Факультет компьютерных технологий**

**кафедра «Математического обеспечения и применения ЭВМ»**

**Тихомиров В.А.**

**УПРАВЛЕНИЕ ПАМЯТЬЮ В**

**ОПЕРАЦИОННОЙ СИСТЕМЕ**

**WINDOWS**

лабораторная работа по курсу

“Операционные системы»

##### Комсомольск-на-Амуре

**2016 г.**

Цель лабораторной работы: Изучить теоретические вопросы управления памятью в ОС Windows и освоить приемы практической реализации этого управления с использованием системных функций.

Задание на лабораторную работу:

* внимательно изучить ниже прилагаемый теоретический материал;
* набрать и проверить работоспособность программных модулей, представленных на листингах 1-3. Тексты программ вместе с результатами их тестирования вставить в отчет;
* выполнить индивидуальное задание в соответствии с выданным преподавателем вариантом:

|  |  |
| --- | --- |
| № вар. | Содержание задания |
| 1 | Выделить память под массив 1000х1000 элементов типа double и заполнить его случайными числами в интервале от 1 до 10, защитить страницы памяти с массивом от записи, выдать сумму и среднее арифметическое элементов массива, дать команду на обнуление элементов и получить системное предупреждение о невозможности записи в массив. |
| 2 | Зарезервировать память в размере 10 Мбт. Произвести физическое выделение памяти 10 раз сегментами по 1 Mбт. Вывести на экран адреса выделенных сегментов. По полученным данным построить карту проведенного выделения памяти. |
| 3 | Выделить память под два массива 1000х500 и 500х1000 элементов типа int и заполнить их случайными числами в интервале от 0 до 100.  Защитить от записи страницы первого массива.  Перемножить эти два массива и выдать результат на экран. Перехватить обработчик исключений.  Дать команду на обнуление массивов. На исключение о защите памяти выдать сообщение с указанием названия массива, который не удалось обнулить. |
| 4 | Выделить 300 Кбт памяти и скопировать в нее содержимое ПЗУ машины. Повторить это действие еще пять раз. Распечатать на экране адреса всех выделенных блоков памяти. По полученным данным построить карту проведенного выделения памяти |
| 5 | Зарезервировать память 30 Мбт. Выделяя порциями необходимую память считать в нее последовательно от 1 до 20 дискет. Найти контрольную сумму считанных байтов. Записать эти дискеты на винчестер в файлы в виде отдельных образов. |
| 6 | Составить программу, которая бы читала и выводила на экран содержимое своего кодового сегмента. |
| 7 | Выделить память 10 Мбт. Все ячейки заполнить единицами. Генератором случайных чисел создать и случайным образом «разбросать» по байтам выделенной памяти десять целых чисел в интервале от 2 до 200. Просканировать память и все страницы, содержащие только единицы пометить как запрещенные к доступу. Произвести суммирование всех байтов выделенной памяти. При возникновении исключения при обращении к запрещенной странице программа должна «понимать», что на данной странице содержатся одни единицы. Провести исследование загруженности памяти на всех этапах выполнения программы с помощью системного монитора. |
| 8 | Необходимо выделить участки памяти размером 32 Кбт по адресам: 900 000, 1 000 000 и 3 000 000. Заполнить эти участки единицами. Защитить от записи и попробывать очистить. При возникновении ошибки – перехватить ее и выдать об этом сообщение. |
| 9 | Составить программу, которая во время своей работы модифицировала бы собственный кодовый сегмент. |
| 10 | Составить программу для исследования наличия и содержания у запущенного модуля области PSP. |
| 11 | Составить программу для определения размера страницы памяти, выделяемых MMU процессам на данном компьютере, размера свободной памяти в текущий момент и процент загрузки процессора в текущий момент. |

* тщательно откомментированный текст составленной программы вместе с результатами ее тестирования представить в отчете. Там же привести теоретический материал, на основе которого выполнялось задание.
* отчет оформить в электронной форме согласно СТП КнАГТУ.

Введение

Зачем напрямую работать с механизмами управления памятью, встроенными в операционную систему? Ведь существуют столь удобные средства, как функция **malloc** языка С и оператор **new** языка C++. И действительно, подавляющее число задач, связанных с распределением памяти, можно решить, применяя эти про­стые в использовании механизмы. Но в некоторых случаях может потребоваться более гибкий контроль над использованием памяти.

Предположим, что вы разрабатываете программу, осуществляющую оцифровку видеосигнала. Чтобы сохранить в памяти пятиминутный видеофильм, может по­требоваться достаточно большой (иногда до 50 Мбайт) непрерывный участок оперативной памяти. Однако заранее неизвестно, будет ли пользователь оциф­ровывать именно пять минут, или он ограничится всего лишь 30 секундами. За­чем тратить время и ресурсы, чтобы выделять 50 Мбайт для хранения 30-секундного видеоклипа?

Можно попробовать выделять память небольшими участками по мере не­обходимости. Например, сначала выделить участок памяти для 30 секунд ви­део, затем, используя функцию **realloc**, добавить к этому участку еще один уча­сток для следующих 30 секунд и т. д. Однако при этом нельзя быть уверенным в том, что выделяемая память будет непрерывна. Если между обращениями к **realloc** другая функция или другой программный поток получат в индивиду­альное пользование фрагмент памяти, физически расположенный поблизости от места, где хранится записываемый в память видеоклип, дальнейшее рас­ширение области памяти для хранения видео будет осложнено. В этой ситуа­ции, чтобы обеспечить непрерывность участка памяти, функция **realloc** будет вынуждена переместить уже записанную часть видеоклипа в другую область ад­ресного пространства для того, чтобы продолжить ее расширение. Представьте, что к этому моменту вы уже успели записать 25 Мбайт видео. Перемещение такого объема данных в процессе оцифровки может оказаться крайне нежела­тельным.

Механизмы управления памятью, встроенные в Windows, позволяют решить эту проблему. Помимо этого Windows позволяет выделять память только для чтения, а также использовать так называемые *«разреженные» массивы* (sparse arrays).

Программист может прожить всю жизнь до самой старости и никогда не использовать для управления памятью ничего, кроме **mallос** и **new** (или других подобных механиз­мов, которые поддерживаются в используемом вами языке). Однако в случае возникновения специальных системных задач, он хотя бы должен знать, что в Windows существуют специальные возможности по управлению памятью, необходимо только научиться использовать предназначенные для этого системные функции.

Страничная организация памяти

В наше время в основе многих компьютерных архитектур располагается специ­альный механизм управления памятью, который называют MMU (Memory Management Unit — устройство управления памятью). На компьютерах, осна­щенных процессором Pentium, механизм MMU является неотъемлемой частью CPU. MMU осуществляет преобразование линейных адресов виртуального ад­ресного пространства каждого из работающих на компьютере процессов в адреса физической оперативной памяти, реально установленной на компьютере. Как мы знаем, в Windows 2000 каждый процесс обладает виртуальным адресным про­странством размером 4 Гбайт, однако, как правило, только часть этого простран­ства отображается на физическую оперативную память, реально установленную на компьютере.

Благодаря MMU осуществляется поддержка многих чрезвычайно эффективных и удобных технологий. Например, если программа обращается к области памяти с адресом 0,механизм MMU может перенаправить это обращение по ад­ресу, к примеру, 16384. Таким образом, программа будет думать, что осуществлено чтение (или запись) байта по нулевому адресу, а на самом деле этот байт располагается по адресу 16384. Другая программа может также обратиться по адресу 0, однако MMU осуществляет преобразование адресов для каждой из программ по-разно­му. При обращении к ячейке с нулевым адресом второй программы он перена­правляет это обращение по совершенно другому адресу (например, 32768). Иными словами, благодаря MMU каждый процесс, работающий в системе, думает, что обладает собственным индивидуальным адресным пространством от 0 до 4 Гбайт, хотя на самом деле данные, принадлежащие этим процессам, хранятся в разных областях одного и того же физического адресного пространства.

Если бы MMU мог осуществить перенаправление каждого байта ОЗУ по дру­гому адресу, его таблицы соответствия программных адресов физическим были бы чрезмерно большими. Вместо этого MMU рассматривает всю оперативную память как набор небольших блоков, которые называют *страницами,* и осуще­ствляет трансляцию адресов в соответствии с границами этих страниц. MMU, встроенный в процессоры типа Pentium, использует страницы размером 4 Кбайт (другие процессоры могут использовать другой размер страниц; узнать этот размер можно при помощи функции **GetSystemInfo**). Таким образом, если ячейке памяти с виртуальным адресом 0 соответствует ячейка памяти с физическим адресом 16384, то ячейке памяти с виртуальным адресом 1 всегда будет соответствовать ячейка памяти с физическим адресом 16385 и т.д. Однако ячейке памяти с вир­туальным адресом 4096 может соответствовать совершенно иной, отличный от 20480 (16384+4096) физический адрес, ведь эта ячейка принадлежит другой странице памяти.

MMU может пометить страницу виртуальной памяти как *отсутствующую* (absent). Этот механизм лежит в основе организации *виртуальной памяти.* MMU сообщает операционной системе, что произошло обращение к отсутствующей в физической оперативной памяти странице. ОС загружает отсутствующую стра­ницу в физическую память и разрешает программе вновь попробовать обратить­ся к ней. Конечно же, чтобы освободить место для загрузки отсутствующей стра­ницы, ОС может записать на диск какую-либо другую страницу виртуальной памяти и пометить ее как отсутствующую.

Механизм отсутствующих страниц используется также для организации *раз­реженных массивов (sparse arrays).* Представьте, что вы работаете с очень боль­шой матрицей вещественных чисел, для хранения которой необходимо выдели­ть 10 Мбайт оперативной памяти. Допустим также, что большая часть элементов матрицы равна нулю (или любому другому константному значению). Матрицы такого рода очень часто используются при решении разного сорта инженерных задач.

Неужели нет никакого другого способа хранения такой матрицы, кроме как выделить память для того, чтобы записать в нее 10 Мбайт нулей? Windows пред­лагает следующее решение проблемы. Вся матрица разделяется на страницы по 4 Кбайта. Если страница содержит нули, она помечается как отсутствующая, Когда программа, работающая с матрицей, обращается к такой странице, MMU'сооб­щает ей, что страница отсутствует, и программа понимает, что все ячейки стра­ницы на самом деле равны нулю, Таким образом, на хранение страниц, целиком заполненных нулями, драгоценная физическая память не расходуется.

Управлять работой MMU можно при помощи вызова **VirtualAlloc**. В зависимос­ти от того, как вы обращаетесь к этой функции, она ведет себя по-разному,

В качестве аргументов функция принимает адрес области памяти, ее размер, режим выделения и уровень защиты, Если в качестве первого аргумента функ­ции передать значение NULL, операционная система самостоятельно определит подходящий адрес выделяемой области. Вместо этого вы можете попросить ее выделить память с конкретным начальным адресом (этот адрес должен соответ­ствовать границе страницы). Второй аргумент — количество байт, которое необ­ходимо выделить. Windows автоматически округлит это количество таким обра­зом, чтобы оно равнялось числу, кратному размеру страницы. Если вы хотите просто выделить память, в качестве третьего аргумента необходимо указать зна­чение МЕМ\_СОММ1Т, Последний аргумент — уровень защиты, который будет соответ­ствовать выделяемой памяти (табл.1). Очевидно, что использовать некоторые из перечисленных здесь атрибутов (например, PAGE\_READONLY) не имеет смысла, так как память не инициализирована начальными значениями,

***Константы уровня защиты* Таблица 7.1.**

|  |  |
| --- | --- |
| **Флаг** | **Значение** |
| PAGE\_READONLY | Выделяется память только для чтения. |
| PAGE\_READWRITE | Выделяется память, как для чтения, так и для записи. |
| PAGE\_WRITECOPY | После осуществления записи страница заменяется ее новой копией (используется совместно с общей памятью или отраженными на память файлами) |
| PAGE\_EXECUTE | Разрешается исполнение программного кода, расположенного в выделяемой памяти. |
| PAGE\_EXECUTEREAD | Разрешается исполнение программного кода и чтение данных, расположенных в выделяемой памяти. |
| PAGE\_EXECUTEREADW | Разрешается исполнение программного кода, а также чтение и запись данных, расположенных в выделяемой памяти. |
| PAGE\_PAGENOACCESS | Запрещает любой доступ к выделенной памяти. В случае любого доступа возникает ошибка General Protection Fault (GPF). |
| PAGE\_GUARD | Страницы в выделяемой памяти становятся «сторожевыми». При попытке осуществить чтение или запись данных в сторожевой странице операционная система инициирует исключение STATUS\_GUARD\_PAGE и сбрасывает статус сторожевой страницы. Таким образом, сторожевые страницы работают в режиме «тревога-при-первом-обращении». Обычно такой режим используется совместно с другими флагами (за исключением флага PAGE\_NOACCESS) |
| PAGE\_NOCACHE | Запрещает кэширование страниц (не рекомендуется). Используется совместно с другими флагами (за исключением флага PAGE\_NOACCESS) |

Если использовать **VirtualAlloc** описанным образом, эта функция выделит участок памяти указанного объема точно так же, как если бы вы использовали для этой цели стандартный вызов **malloc**. Зачем это может потребоваться? В от­личие от **malloc** функция **VirtualAlloc** позволяет выделить участок памяти, выров­ненный по границе страницы, который будет обладать указанными вами атрибу­тами. Кроме того, в случае необходимости вы сможете корректно освободить этот участок памяти и передать его в ведение операционной системы. Дело в том, что некоторые библиотеки не возвращают освобождаемую память операционной си­стеме.

Если вы хотите изменить уровень защиты для какой-либо одной (или всех) страницы выделенного участка, вы можете воспользоваться вызовом **VirtualProtect**. При этом участку памяти можно присвоить любой из атрибутов, перечисленных в табл.1. Функция **VirtualProtectEx** позволяет выполнить ту же процедуру в отношении памяти, которой обладает некоторый процесс. Конечно, в этом случае, программа, обращающаяся к **VirtualProtectEx**, должна обладать необходимыми привилегиями в отношении целевого процесса. Чтобы освободить память, выде­ленную при помощи **VirtualAlloc**, необходимо обратиться к функции **VirtualFree.**

Но помимо перечисленных функция **Virtua1Alloc** обладает некоторыми дру­гими весьма полезными возможностями. Если в качестве третьего аргумента функции **Virtua1Alloc** вместо флага МЕМ\_СОММIТ использовать флаг MEM\_RESERVE, функ­ция сработает иначе. В этом случае функция **Virtua1Alloc** резервирует указанный объем линейного адресного пространства, однако реальная физическая или виртуальная память при этом не выделяется. Речь идет лишь о резервировании диапазона адресов.

Когда вы захотите выделить реальную память, соответствующую этим адре­сам, вы должны обратиться к вызову **Virtua1Alloc** и передать ему адрес участка памяти и флаг МЕМ\_СОММIТ. Выделять память сразу для всего зарезервированного диапазона адресов вовсе не обязательно. Например, можно зарезервировать 10 Мбайт адресного пространства, а затем выделять память из этого диапазона адресов фрагментами по 100 Кбайт.

VirtualAlloc на практике

Чтобы сравнить скорость выполнения функции **Virtua1Alloc** в различных режи­мах, запустите программу, исходный код которой приведен в листинге 1. Эта программа получает в индивидуальное пользование относительно большой уча­сток памяти, при этом используются три разных метода. По умолчанию про­грамма работает с блоком памяти объемом 75 Мбайт. Это значение приемлемо для тестирования механизмов выделения/освобождения памяти. Если вы захо­тите работать с блоком памяти другого размера, его размер (количество мега­байт) можно указать в командной строке при запуске программы.

**Листинг 1.**

#include <windows.h>

#include <iostream.h>

char \*p=NULL;

#define MEGABYTE 1048576

#define MSIZE 75

void main(int argc, char \*argv[])

{

char choice;

int msize=0;

if (argc>1) msize=atoi(argv[1]);

if (!msize) msize=MSIZE;

do

{

int valid=1;

if (p)

{

cout<<"\nОсвобождение памяти\n";

VirtualFree(p,0,MEM\_RELEASE);

p=NULL;

cout<<"Завершено\n";

}

cout<<"1 - Выделить " <<msize<< "MB памяти\n";

cout<<"2 - Зарезервировать " <<msize<<"MB памяти\n";

cout<<"3 - Зарезервировать " <<msize<<"MB памяти и выделить в ней 1MB\n";

cout<<"4 - Выход\n";

cout<<"Ваш выбор: ";

cin>>choice;

switch (choice)

{

case '1':

cout<<"\nВыделение памяти\n";

p=(char \*)VirtualAlloc(NULL, msize\*MEGABYTE,

MEM\_COMMIT,PAGE\_READWRITE);

break;

case '2':

cout<<"\nРезервирование памяти\n";

p=(char \*)VirtualAlloc(NULL, msize\*MEGABYTE,

MEM\_RESERVE, PAGE\_READWRITE);

break;

case '3':

cout<<"\nРезервирование и частичное выделение памяти\n";

p=(char \*)VirtualAlloc(NULL, msize\*MEGABYTE,

MEM\_RESERVE, PAGE\_READWRITE);

cout<<"\nВыделение\n";

VirtualAlloc(p, MEGABYTE, MEM\_COMMIT, PAGE\_READWRITE);

break;

case '4':

exit(0);

default:

cout<<"\nЧего-чего?\n";

valid=0;

}

if (p&&valid)

cout<<"Блок памяти @ "<<hex<<(unsigned int)p<<"\n";

if (!p&&valid) cout<<"Ошибка выделения памяти!\n";

} while (1);

}

В появившемся меню выберите пункт 1. Программа выделит блок памяти ус­тановленного размера, а затем освободит его. На большинстве современных ком­пьютеров эта процедура будет выполнена в течение всего нескольких секунд. За это время Windows должна настроить все необходимые записи в таблице соот­ветствия страниц, определить доступный объем физической памяти и создать необходимое количество страниц виртуальной памяти. Если объем физической памяти вашего компьютера превышает 75 Мбайт, для наглядности попробуйте использовать блок памяти большего размера. Выберите пункт 1 несколько раз, чтобы получить представление о том, насколь­ко быстро выполняется процедура выделения и освобождения достаточно боль­ших участков памяти.

Теперь попробуйте выбрать пункт 2. При этом программа не будет выделять физическую или виртуальную память объемом 64 Мбайт, а лишь зарезервирует диапазон адресов необходимого размера. Это происходит настолько быстро, что вы даже не сможете заметить, что что-либо произошло. Конечно, если вы зарезервировали диапазон ад­ресов, это не значит, что вы сможете немедленно приступить к использованию соответствующей памяти.

Чтобы проверить, как выполняются процедуры резервирования адресов и выделения конкретной памяти в комплексе, выберите пункт 3. При этом програм­ма резервирует адресное пространство указанного объема, а затем выделяет всего один мегабайт памяти из этого адресного пространства. Таким образом, модели­руется ситуация, в которой вы на всякий случай резервируете достаточно большой участок непрерывной памяти, однако используете лишь небольшой фрагмент этого участка. Такое поведение, в частности, напоминает работу программы, оцифро­вывающей видеосигнал, о которой мы говорили ранее. Выбрав пункт 3 тестовой программы, вы заметите, что резервирование адресного пространства и последу­ющее выделение памяти в этом пространстве выполняются фактически мгновенно. Естественно, ведь выделяется лишь один мегабайт из шестидесяти четырех. В боль­шинстве ситуаций, используя подобный подход, вы сможете добиться более глад­кой работы ваших программ. Другими словами, если вы зарезервируете 10 Мбайт, а затем выделите всю эту память последовательно блоками по одному мегабайту, у пользователя возникнет ощущение, что ваша программа работает быстрее, чем если бы было выполнено выделение всех 10 Мбайт за одну операцию.

Работа с атрибутами страниц

Определенно одним из основных преимуществ использования **VirtualAlloc** яв­ляется возможность управления уровнем доступа к страницам оперативной па­мяти как во время ее выделения, так и позже (при помощи функции **VirtualProtect**). Разумное использование этой возможности может существенно упрос­тить разработку многих приложений.

Для примера рассмотрим код в листинге 2. Это библиотечная функция, ко­торая возвращает указатель на статическую строку. В реальной жизни эта функ­ция может располагаться в библиотеке DLL, однако для целей этой программы мы включим ее в состав основного исполняемого файла наряду с функцией **main**.

**Листинг 2.** Передача статической строки в режиме только для чтения

#include <stdio.h>

#include <string.h>

#include <windows.h>

char \*getmsg()

{

static char \*msg;

if (msg==NULL)

{

char tmp[1024];

DWORD oldprot;

FILE \*f;

f=fopen("strings.txt", "r");

if (!f) return NULL;

fgets(tmp, sizeof(tmp), f);

// Выделяет целую страницу памяти, что приемлемо для данного примера

msg=(char \*)VirtualAlloc(NULL, strlen(tmp)+1, MEM\_COMMIT, PAGE\_READWRITE);

strcpy(msg, tmp);

fclose(f);

// Назначает уровень доступа к странице памяти только для чтения

VirtualProtect(msg, strlen(msg)+1, PAGE\_READONLY, &oldprot);

}

return msg;

}

void main()

{

char \*m=getmsg();

try {

printf("Вся строка: %s\n",m);

m=strtok(m," \t");

printf("token1=%s\n",m);

m=getmsg();

printf("Вся строка: %s\n",m);

}

catch (...) // следить за любыми исключениями

{

printf("Произошло исключение!\n");

}

getchar(); // ожидание для отладочных целей

}

Основная программа получает статическую строку, а затем обращается к **strtok**, чтобы разбить ее на части. При этом происходит исключение. Почему? Функция **strtok** модифицирует строку, которую вы ей передали. Конечно, примерно того же самого можно добиться, указав, что строка является константой (const), но настойчивый программист может обойти подобное ограничение, обратившись к памяти напрямую.

Чтобы надежно защитить участок памяти от модификаций, лучше поместить строку на страницу, которая в дальнейшем переводится в режим только для чте­ния. Конечно, страницу нельзя перевести в этот режим до того, как вы размести­те в ней начальные данные, Каждый раз, когда ваша программа желает изменить содержимое страницы, она обязана обращаться к **VirtualProtect** для того, чтобы временно перевести страницу в режим как для чтения, так и для записи.

Код, приведенный в листинге 2, следит за возникновением исключений (он реагирует на любое возникающее исключение), таким образом, в случае некор­ректного обращения к защищенной странице программа сможет должным обра­зом отреагировать на это. В нашем случае на экран выводится сообщение об ошибке, текст которого определяется программистом. Если убрать из текста про­граммы оператор **try**, система прервет работу программы и выведет на экран соб­ственное системное сообщение об ошибке.

Что мешает основной программе обратиться к **VirtualProtect**? Вообще-то ни­чего. Однако обращение к этому вызову — более продуманное и взвешенное дей­ствие, чем простая передача указателя из функции в функцию. Код, использующий **VirtualProtect**, несомненно, обладает большей степенью надежности. Обратиться к этому вызову по ошибке фактически невозможно, ведь для этого необходимо знать начальный адрес защищенной области памяти, который далеко не всегда совпадает с переданным вам указателем (как это случилось в нашем случае).

Способы работы с разреженной памятью

При попытке обратиться к странице, которая зарезервирована, но не выделена, происходит исключение. Исключение также возникает в случае, если странице присвоен атрибут PAGE\_GUARD (это исключение возникает только один раз). Ис­пользуя одну из этих технологий, вы можете определить момент, когда програм­ма обращается к тем или иным страницам. Зачем это нужно?

Используя этот механизм, можно зарезервировать достаточно большой учас­ток памяти, а затем, по мере заполнения его данными, выделять дополнительную память из этого участка последовательно страница за страницей в автоматичес­ком режиме. При этом программа будет записывать данные в зарезервированную область в точности так же, как если бы вся эта память была бы уже выделена. При попытке записи в зарезервированную, но не выделенную страницу проис­ходит исключение. Можно написать специальный обработчик этого исключения, который будет выделять страницу, в которую основная программа пытается за­писать данные. Таким образом, основная программа и не подозревает о том, что память только зарезервирована, но не выделена. Выделением памяти занимается обработчик исключения.

Еще одна область применения этих механизмов — реализация эффективных *разреженных* таблиц. Представьте, что вы работаете с массивом, в котором фак­тически все элементы равны одному и тому же значению и лишь некоторые из них отличаются от этого значения. Такие массивы часто используются, например, в инженерных вычислениях. Вы можете разбить массив на страницы по 4 Кбайт и хранить в памяти только те страницы, содержимое которых отличается от об­щего значения. Например, предположим, что для хранения массива требуется выделить 100 страниц, 96 из них содержит нули, а в четырех оставшихся стра­ницах значения некоторых ячеек памяти отличаются от нуля. Нет смысла тра­тить память на хранение нулей. Все нулевые страницы помечаются как страницы с запрещенным доступом или остаются зарезервированными, но не выделенны­ми. Когда программа, осуществляющая вычисления, обращается к такой страни­це, Windows генерирует исключение. Программа перехватывает это исключение и делает вывод, что запрашиваемое значение равно нулю.

Следить за возникновением исключений можно стандартными средствами С или C++. Операторы **\_\_try** и **\_\_except**, поддерживаемые С, очень просты в исполь­зовании, однако они плохо согласуются с конструкциями C++. В C++ удобнее использовать стандартные ключевые слова **try** и **catch**.

Существуют некоторые особенности использования этих операторов. Дело в том, что по умолчанию стандартные исключения Windows не передаются в обра­ботчик исключений, использующий **catch**. Чтобы решить проблему, необходимо при помощи функции **\_set\_se\_translator** определить вашу собственную функцию, которая осуществляет преобразование исключений в стиле С (именно такие ис­ключения генерирует Windows) в исключения в стиле C++. Это может быть очень простая подпрограмма, которая просто-напросто генерирует код исключения, являющийся целым числом.

Как только вы обнаружили возникновение исключения, вы можете выделить соответствующую страницу памяти, вернуть значение элемента массива по умолчанию или выполнить любую другую операцию, которая уместна в данном случае. При обращении к зарезервированной, но не выделенной странице возникает исключение 0х000000З. Рассмотрим код в листинге 3.

**Листинг 3. Бесконечный буфер**

#include <windows.h>

#include <stdio.h>

#include <eh.h> // преобразование исключений C в исключения C++

static int maxindex;

char \*getbuffer()

{

SYSTEM\_INFO info;

// определение размера страницы

GetSystemInfo(&info);

char \*rv;

rv=(char \*)VirtualAlloc(NULL, maxindex=info.dwPageSize\*1010,

MEM\_RESERVE,PAGE\_READWRITE);

return rv;

}

void cppexcept( unsigned int u, \_EXCEPTION\_POINTERS\* pExp )

{

throw u;

}

void putbuffer(char \*p,int index, char value)

{

\_se\_translator\_function oldhandler=\_set\_se\_translator(cppexcept);

try {

p[index]=value;

}

catch (unsigned int code)

{

printf("Code=%x @%d\n",code,index);

if (code!=0xC0000005)

{

printf("Неизвестное исключение\n");

exit(1);

}

// проверка: находится ли индекс в нужном диапазоне

if (index>=maxindex)

{

printf("Попытка доступа к ячейку за границами буфера\n");

exit(2);

}

// Выделяем один байт

// на самом деле система выделяет целую страницу

VirtualAlloc(p+index, 1, MEM\_COMMIT, PAGE\_READWRITE);

\_set\_se\_translator(oldhandler);

// еще одна попытка записать в буфер

putbuffer(p,index,value);

}

\_set\_se\_translator(oldhandler);

}

void freebuffer(char \*p)

{

VirtualFree(p,0,MEM\_RELEASE);

}

//Основная программа (не может располагаться отдельно от библиотеки)

void main()

{

char \*p=getbuffer();

for (int i=0;i<4000000;i++)

{

putbuffer(p,i,i&0xFF);

}

freebuffer(p);

printf("Pass #2\n");

p=getbuffer();

// now backwards for variety

for (int j=4000000-1;j>=0;j--)

{

putbuffer(p,j,j&0xFF);

}

freebuffer(p);

}

Это исходный код простой библиотеки, которая обеспечивает для основной программы простые средства работы с буфером. Программа построена для обработки системных исключений при компилировании в Visual C++. Опять же для простоты все функции библиотеки располагаются в том же файле, что и основная программа. Функция **getbuffer** резервирует 1010 страниц памяти, но не выделяет их (на машинах архитектуры Intel такое количество страниц соответствует приблизительно 4 Мбайт памяти). Таким образом, ни одной страницы физической или виртуальной памяти не расходуется. Вместо этого лишь резервируется адресное пространство.

Когда основная программа хочет записать в буфер байт данных, она обращается к функции **putbuffer**. Эта функция чрезвычайно проста. Основное действие выполняется в строке, в которой элементу буфера с индексом Index присваивается значение value. Если по указанному адресу существует выделенная память, то эта строчка является единственной, которая выполняется в рамках данной функции.

Ситуация меняется в случае, если программа пытается обратиться к странице, которая зарезервирована, но не выделена (такое происходит, в частности, при| самом первом обращении к буферу). Возникает исключение 0хС0000005, на которое реагирует специальный код функции **putbuffer**. Этот код выделяет соответствующую страницу для того, чтобы программа смогла продолжить работу. После того как страница выделена, происходит рекурсивное обращение к функции **putbuffer**. He стоит забывать, что обработчик исключения реагирует на любые, возникающие исключения, поэтому обязательно следует проверить, не превышает ли значение Index допустимого предела и не вызвано ли исключение какими-либо иными причинами,

Возникающее исключение невидимо для нормального обработчика исключе­ний C++. Чтобы осуществить преобразование исключений С в исключения **unsigned int** в стиле C++, используется простая функция, указатель на которую передается функции **\_set\_se\_translator**. Функция **\_set\_se\_translator** возвращает адрес функции преобразования исключений по умолчанию, поэтому подпрограмма **putbuffer** имеет возможность восстановить адрес этой функции перед тем, как управление будет передано вызвавшей программе. Для хранения указателей на подобные функции служит специальный тип **\_se\_translator\_function** (определен­ный в заголовочном файле **eh.h**).

Если вы работаете с библиотекой подобного рода, разработка основной про­граммы существенно упрощается. Основная программа получает буфер, осуще­ствляет запись данных в любое место этого буфера в любом удобном порядке, и все работает. Основная программа и понятия не имеет о том, что подобная функ­циональность достигается при помощи внутренних механизмов управления па­мятью, поддерживаемых Windows.

Чтобы реализовать это же самое, но с использованием инструментария Builder C++ надо использовать программный код наподобие следующего:

#pragma hdrstop

#pragma argsused

#include <stdio.h>

#include <windows.h>

#include <conio.h>

//#include <winnt.h>

static int maxindex;

// процедура – фильтр системных исключений. Использует ряд системных

// структур, описанных в файле winnt.h

static int xfilter(EXCEPTION\_POINTERS \*xp)

{ int rc;

EXCEPTION\_RECORD \*xr = xp->ExceptionRecord;

CONTEXT \*xc = xp->ContextRecord;

// распечатаем код системной ошибки

printf("Code %ul ", (unsigned long)xr->ExceptionCode);

//в зависимости от кода исключения пошлем обработчику исключения

//соответствующие системные константы:

switch (xr->ExceptionCode) {

case EXCEPTION\_BREAKPOINT:

// кто-то оставил внедренную контрольную точку отладки

// just step over it (1 byte on x86)

++xc->Eip;

rc = EXCEPTION\_CONTINUE\_EXECUTION;

break;

case EXCEPTION\_ACCESS\_VIOLATION:

rc = EXCEPTION\_EXECUTE\_HANDLER;

break;

default:

// give up

rc = EXCEPTION\_CONTINUE\_SEARCH;

break;

};

return rc;

}

char \*getbuffer()

{SYSTEM\_INFO info;

// определение размера страницы

GetSystemInfo(&info);

char \*rv;

rv=(char \*)VirtualAlloc(NULL, maxindex=info.dwPageSize\*1010,

MEM\_RESERVE,PAGE\_READWRITE);

return rv;

}

void putbuffer(char \*p,int index, char value)

{ int code;

**try**

{

p[index]=value;

}

//обработчик системного исключения

**\_\_except(code = xfilter(GetExceptionInformation()))**

{

printf("Index %d\n", index);

if (code!=EXCEPTION\_EXECUTE\_HANDLER)

{ printf("Неизвестное исключение\n");

exit(1);

}

// проверка: находится ли индекс в нужном диапазоне

if (index>=maxindex)

{ printf("Попытка доступа к ячейку за границами буфера\n");

exit(2);

}

// Выделяем один байт

// на самом деле система выделяет целую страницу

VirtualAlloc(p+index, 1, MEM\_COMMIT, PAGE\_READWRITE);

// еще одна попытка записать в буфер

putbuffer(p,index,value);

}

}

void freebuffer(char \*p)

{

VirtualFree(p,0,MEM\_RELEASE);

}

void main()

{ char \*p=getbuffer();

for (int i=0;i<4000000;i++) putbuffer(p, i, i&0xFF);

freebuffer(p);

printf("Pass #2\n");

p=getbuffer();

// now backwards for variety

for (int j=4000000-1;j>=0;j--) putbuffer(p, j, j&0xFF);

freebuffer(p);

}

Использование нескольких пулов свободной памяти

Функция **VirtualAlloc** безусловно является мощным средством управления использования памяти, однако во многих случаях применять ее неудобно, так как она является слишком низкоуровневой. Для программистов, не желающих использовать **VirtualAlloc**, операционная система Windows предлагает использовать традиционные средства работы с памятью, такие как **mallос** и **new**. Эти два оператора являются стандартными механизмами выделения памяти в С и C++.

Однако в некоторых случаях использование **mallос** и **new** оказывается недоста­точно эффективным. Дело в том, что оба этих оператора выделяют память из общего *пула свободной памяти,* который называют *«кучей>* (heap). Если ваша программа активно использует динамически выделяемую память, куча подвергается фрагментации. Эта проблема возникает в случае, если программа выделяет намять небольшими участками, а затем освобождает некоторые из них. В результате вся свободная память оказывается разбитой на множество несвязанных кусков. Если требуется выделить один большой непрерывный блок памяти, система либо выполняет реорганизацию используемых участков памяти (что требует дополнительного времени), либо увеличивает размер кучи (что приводит к нежелательному расходу ресурсов).

В подобной ситуации Windows предлагает использовать несколько пулов свободной памяти вместо одного. Допустим, один из пулов служит для хранения множества структур, каждая из которых занимает 128 байт памяти. Тогда при освобождении памяти, занимаемой одной структурой, на это же место можно за­писать другую структуру. Таким образом, фрагментация не возникает. Согласитесь, что подобный подход позволяет достичь эффективности, которая фактически недостижима при использовании традиционной кучи.

Для работы с несколькими кучами свободной памяти в Windows используется вызовы, перечисленные в табл. 2. Каждый процесс обладает традиционно кучей, которая используется операторами **mallос** и **new**. Дескриптор этой кучи можно узнать при помощи функции **GetProcessHeap**. Однако помимо этой тради­ционной кучи вы можете создать любое количество дополнительных пулов сво­бодной памяти (для этого служит вызов **HeapCreate**).

При создании новой кучи в качестве одного из аргументов вызова **HeapCreate** указывается максимальный размер кучи. Если этот аргумент имеет значение 0, значит, куча сможет увеличиваться без ограничений. Куча фиксированного раз­мера обладает ограничением на размер каждого из элементов (размер каждого из элементов не может превышать 512 Кбайт).

Получив дескриптор кучи, можно выделить в ней участок памяти при по­мощи функции **НеарAllос**. Освободить память в куче можно при помощи функ­ции **HeapFree**. Доступ к куче можно заблокировать для всех потоков, кроме од­ного. Обычно это осуществляется автоматически при обращении одного из потоков к куче, однако вы можете сделать это самостоятельно перед началом серии операций с кучей. Таким способом можно улучшить производительность программы.

Еще одним преимуществом использования нескольких куч является улучше­ние производительности многопоточных приложений. Обычно Windows синхро­низирует доступ к куче со стороны нескольких потоков при помощи блокирова­ния кучи перед началом выполнения той или иной относящейся к ней операции. Если каждый из потоков программы будет обладать собственной кучей, вы смо­жете увеличить быстродействие, присвоив куче атрибут HEAP\_NO\_SERIALIZE. При этом система не будет следить за корректностью обращения к куче со стороны несколь­ких потоков, подразумевая, что об этом позаботитесь вы сами. Не упускайте из виду, что доступом к куче, обладающей атрибутом HEAP\_NO\_SERIALIZE, может обла­дать только поток, которому она принадлежит.

В большинстве случаев нет надобности в предоставлении каждому потоку индивидуальной кучи, однако в некоторых случаях, используя такую мето­дику, можно получить максимальную возможную производительность прило­жения.

**Вызовы для работы с несколькими кучами свободной памяти Таблица 2.**

|  |  |
| --- | --- |
| Функция | Описание |
| HeapCreate | Создает новую кучу |
| НеарАllос | Выделяет блок памяти в куче |
| HeapCompact | Дефрагментирует кучу |
| HeapDestroy | Уничтожает кучу |
| HeapFree | Освобождает блок памяти в куче |
| GetProcessHeap | Возвращает дескриптор стандартной кучи процесса |
| GetProcessHeaps | Возвращает дескрипторы всех куч |
| HeapLock | Блокирует доступ к куче для остальных потоков |
| HeapUnlock | Разрешает нормальный доступ к куче после того, как она была заблокирована при помощи HeapLock |
| HeapReAlloc | Изменяет размер блока памяти, выделенного в куче |
| HeapSize | Возвращает размер блока, выделенного в куче |
| HeapValidate | Тестирует целостность кучи |
| HeapWalk | Последовательно перебирает блоки памяти, выделенные в куче |

**ЗАКЛЮЧЕНИЕ**

Основной способ управления памятью в Win32 — это системный вызов **VirtualAlloc**. Этот вызов может оказаться полезным в следующих ситуациях:

* вы хотите быть уверенным в том, что освобождаемая вами память возвраща­ется под контроль операционной системы;
* вы хотите зарезервировать некоторый диапазон адресов, однако до определен­ного момента не намерены выделять физическую или виртуальную память, соответствующую этим адресам. Возможно, вы хотите выделять соответству­ющую память последовательно, небольшими блоками, или вы намерены реа­лизовать разреженный массив таким образом, что некоторые из участков за­резервированного вами диапазона адресов так и останутся невыделенными;
* некоторый участок выделяемого вами блока памяти или весь блок целиком должен обладать специальным атрибутом (например, только для чтения).

В любом случае вызов **VirtualAlloc** обычно используется для выделения дос­таточно больших участков памяти. Минимальный объем памяти, с которым рабо­тает **VirtualAlloc**, называется *страницей.* Для компьютеров на базе процессоров Intel каждая станица имеет размер 4 Кбайт. Другие компьютеры могут исполь­зовать больший размер страницы (определить размер страницы можно при по­мощи вызова **GetSystemInfo**).

Использование VirtualAlloc и VirtualFree

Системный вызов **VIrtualAlloc** принимает четыре аргумента:

* *Линейный адрес выделяемого участка***.** Этот аргумент может иметь значение NULL в случае, если вы желаете получить память в любом доступном месте ад­ресного пространства,
* *Размер выделяемого блока в байтах***.** Windows округляет это значение таким образом, чтобы оно соответствовало целому количеству страниц памяти, Дру­гими словами, если вы хотите выделить 1 байт на компьютере архитектуры Intel, функция VIrtualAlloc выделит 4 Кбайт, то есть целую страницу. Если вы запрашиваете 4100 байт, будет выделено 8 Кбайт, то есть две страницы.
* *Режим работы функции***.** Этот параметр может содержать либо значение МЕМ\_СОММ1Т (память выделяется обычным способом), либо значение MEM\_RESERVE (происходит резервирование диапазона адресов, но реальная память не выде­ляется). Любое из этих значений можно использовать совместно с флагом MEM\_TOPDOWN, который приказывает Windows выделять память таким образом, чтобы указанный в качестве первого аргумента функции адрес был самым старшим адресом диапазона. Если вы резервируете адресное пространство позже, вы должны вновь обратиться к **VirtualAlloc** для того, чтобы выделить соответствующую память, иначе вы не сможете сохранить по указанным адресам каких-либо данных. За одну операцию можно выделить участок памяти соответствующий всему зарезервированному диапазону адресов, а можно ограничиться выделением участка, соответствующего только части диапазона.
* **Режим защиты памяти**. Различные уровни защиты перечислены в табл.1**.** Чаще всего при выделении памяти используется значение PAGE\_READWRITE, чтобы можно было инициализировать выделенную память начальным значением. Позже уровень защиты участка памяти можно изменить при помощи функции **Virtual Protect**.

Допускается использовать **VirtualAlloc** для повторного выделения участка памяти, который ранее уже был выделен, однако нельзя повторно зарезервировать ранее зарезервированный участок адресного пространства. Функция возвращает адрес начала выделенного участка.

При обращении к **VirtualAlloc** в качестве третьего аргумента можно исполь­зовать значение MEM\_RESET. Этим вызывающая программа сообщает операционной системе о том, что содержимое участка памяти больше не имеет значения. Таким образом, в случае если потребуется переместить содержимое участка из физичес­кой оперативной памяти в виртуальную память, Windows просто не будет этого делать — содержимое участка будет утеряно.

Если участок памяти больше не нужен, необходимо вызвать **VirtualFree** для того, чтобы освободить его. Этой функции необходимо передать адрес участка памяти, его размер (в байтах), а также флаг, определяющий тип операции. Если флаг равен MEM\_RELEASE, адрес должен быть адресом начала участка памяти, ранее выделенного функцией **VirtualAlloc**, а размер должен равняться нулю. В этом случае **VirtualFree** полностью освобождает ранее выделенную память.

Значение флага может быть МЕМ\_ОЕСОММ1Т, При этом выделенная физическая память освобождается, но соответствующее адресное пространство остается в зарезервированном состоянии. Функция **VirtualFree** округляет начальный адрес вниз, до ближайшей границы страниц, а размер участка — вверх таким образом, чтобы получилось целое количество страниц. Если вы освобождаете память, ко­торая была зарезервирована, но не выделена, это не считается ошибкой.

Изменение уровня защиты страниц

Если вы получили с собственное распоряжение участок памяти и заполнили его начальными данными, у вас появляется возможность изменить атрибуты без­опасности этого участка (см. табл. 1). Например, вы можете перевести этот уча­сток в режим только для чтения или полностью запретить к нему любой доступ. Смена уровня защиты участка памяти осуществляется при помощи функции **VirtualProtect**. Этот вызов принимает четыре параметра. Первый параметр — это адрес страницы, атрибуты безопасности которой вы намерены изменить. Далее следует указать количество байт, атрибуты которых будут изменены. Вызов **VirtualProtect** работает с целым количеством страниц. Таким образом, если вы хотите изменить атрибуты одного байта, на самом деле будут изменены атрибу­ты целой страницы, к которой принадлежит этот байт.

Следующий аргумент вызова **VirtualProtect** содержит новый атрибут безопас­ности, который вы хотели бы установить. Последний аргумент — это указатель на DWORD. Эта переменная будет содержать уровень безопасности, которым обла­дал участок до обращения к **VirtualProtect**. Если вы изменяете уровень защиты для набора страниц, возвращенный этой функцией уровень будет соответствовать уровню, которым обладала первая страница диапазона. Даже если вы не нуждае­тесь в получении информации о предыдущем уровне защиты, вы обязаны пере­дать функции **VirtualProtect** корректный указатель на существующую перемен­ную типа DWORD.

Получение информации об уровне защиты страниц

Иногда может потребоваться определить существующий уровень защиты неко­торой страницы (или набора страниц). Для получения этой информации исполь­зуется вызов **VirtualQuery**. Этому вызову передается базовый адрес, указатель на структуру MEMORY\_BASIC\_INFORMATION и размер этой структуры в байтах.

Когда функция возвращает управление вызвавшей программе, структура MEMORY\_BASIC\_INFORMATION содержит базовый адрес выделенной области памяти, к которой принадлежит интересующий вас адрес (AllocationBase), текущий уровень защиты (Protect), уровень защиты, которым обладала страница в момент ее вы­деления (AllocationProtect), текущее состояние страницы (поле State может иметь одно из значений: MEM\_FREE — свободная, MEM\_RESERVE — зарезервированная или MEM\_COMMIT — выделенная) и тип данных, хранящихся на странице (поле Type мо­жет иметь одно из значений: MEM\_IMAGE — исполняемый код, MEM\_MAPPED — файл, отображенный на память, MEM\_PRIVATE — память принадлежит одному из процес­сов и не является памятью общего доступа). Структура также содержит адрес набора страниц (BaseAddress) и количество байт (RegionSize), которые обладают указанными атрибутами.

Использование исключений

При работе с памятью исключения — нередкая вещь. Исключения могут возни­кать при самых разных условиях, например, в случае, если произошло обраще­ние по адресу за пределами массива, или в случае, если вы обратились к страни­це, обладающей атрибутом PAGE\_GUARD. По умолчанию исключения обрабатываются операционной системой, однако вы можете включить в программу ваш собствен­ный обработчик исключения. Windows ожидает, что вы будете обрабатывать ис­ключение с использованием так называемой структурной обработки исключе­ний (Structured Exception Handling, SEH), однако использовать SEH не очень удобно. Вместо этого компилятор С поддерживает конструкции **\_try** и **\_except**, облегчающие обработку исключений.

Стандартным средством обработки исключений в C++ являются ключевые слова **try** и **catch**. Эти операторы совместимы с объектной моделью C++. К сожа­лению, обработчики исключений C++ не понимают формат, в котором иденти­фицируются исключения Windows. К счастью, вы можете без труда написать простую функцию, которая осуществляет несложное преобразование формата идентификации исключений. Эта функция (возвращающая значение void) при­нимает код исключения и указатель на \_EXCEPTION\_POINTERS. Эти аргументы содер­жат всю необходимую информацию об исключении. Ваша задача — сгенериро­вать соответствующее исключение в формате C++.

При желании вы можете создать объект исключения (который может быть производным от класса CException, входящего в состав MFC), инициализировать этот объект при помощи информации двух переданных вам аргументов, а затем сгенерировать соответствующее этому объекту исключение. В большинстве случаев достаточно сгенерировать исключение, соответствующее коду ошибки. В этом случае функция, осуществляющая преобразование, будет наиболее простой:

void cppexcept( unsigned int u, \_EXCEPTION\_ POINTERS\* pExp )

( throw u;

}

Чтобы зарегистрировать эту функцию в качестве функции-преобразователя исключений, вы должны использовать вызов **\_set\_se\_translator**. Этот вызоввозвращает указатель на предыдущую функцию-преобразователь. Этот указатель можно сохранить в переменной типа \_se\_translator\_function для того, чтобы позже можно было восстановить функцию-преобразователь по умолчанию. Пример использования исключений приведен в листинге 3.

Создание дополнительного пула свободной памяти

Традиционно выделяемая программе память извлекается из *пула свободной памяти,* который называют *кучей (heap).* Обычно программа обладает собственной единственной кучей, которая используется при обращении к **malloc** или **new**. Однако в Windows можно создавать дополнительные кучи, которые будут использоваться одной и той же программой для получения новых участков памяти.) Для этого служит вызов **HeapCreate**. Этот вызов возвращает дескриптор новой кучи и принимает в качестве аргументов три значения: набор параметров, а также начальный и максимальный допустимый размеры кучи, 1

В качестве первого аргумента можно указать комбинацию из двух флагов: HEAP\_GENERATE\_EXCEPTIONS, который предписывает Windows генерировать исключение в случае, если вызов окончился неудачей (если этот флаг сброшен, в случае неудачи вызов **HeapCreate** возвращает значение NULL), и HEAP\_NO\_SERIALIZE, который предписывает Windows не осуществлять слежение за одновременным доступом к куче нескольких потоков (если этот флаг сброшен, Windows автоматически блокирует одновременный доступ к куче нескольких потоков, благодаря чему потоки обращаются к куче по очереди). Значения этих флагов можно установить как при обращении к **HeapCreate**, так и при обращении к другим вызовам, обслуживающим кучу.

Второй аргумент содержит начальный размер кучи. Система округляет это значение таким образом, чтобы оно соответствовало целому количеству страниц! На компьютерах архитектуры Infel размер страницы составляет 4 Кбайт, таким образом, если вы захотите создать кучу объемом 4000 байт, реально будет создана куча объемом 4096 байт. Вы можете узнать размер страницы при помощи вызова **GetSystemInfo**.

Третий аргумент вызова — **HeapCreate** — это максимальный допустимый раз­мер кучи. Это значение также округляется таким образом, чтобы оно соответство­вало целому количеству страниц, Чаще всего в качестве максимального размера кучи указывают значение 0. В этом случае размер кучи не ограничивается. Если размер кучи ограничен, в этой куче за одну операцию можно выделить участок памяти, размер которого не может превышать 0x7FFF8 байтов. При этом **HeapCreate** зарезервирует (но не выделит) адресное пространство, соответствующее макси­мальному размеру кучи. Если размер кучи не ограничен, куча будет расти до тех пор, пока хватит виртуального адресного пространства. В этом случае количество байт, которые допускается выделить за одну операцию, ничем не ограничивается.

Если вы намерены ограничить максимальный размер кучи, имейте в виду, что система использует некоторое количество памяти для управления кучей. Это значит, что в куче размером 64 Кбайт для вашей программы будет доступно не­сколько меньшее количество памяти.

Если куча больше не нужна, вы должны уничтожить ее при помощи вызова **HeapDestroy**,

Куча по умолчанию

Каждая программа обладает кучей по умолчанию. Если вы хотите получить дес­криптор этой кучи, воспользуйтесь вызовом **GetProcessHeap**. Вызывать функцию **HeapDestroy** для этого дескриптора нельзя.

Выделение и освобождение памяти в куче

Обладая дескриптором кучи, вы можете использовать вызов **НеарAllос** для того, чтобы выделить память в этой куче. Этой функции необходимо передать деск­риптор кучи, аргумент с флагами и размер в байтах выделяемого участка. Флаги могут быть следующими: HEAP\_GENERATE\_EXCEPTIONS, HEAP\_NO\_SERIALIZE или НЕАР\_ ZERO\_MEMORY, Первые два флага имеют такое же значение, как и при обращении к **HeapCreate**. Флаг HEAP\_ZERO\_MEMORY предписывает Windows обнулить значение вы­деляемых ячеек памяти перед тем, как управление будет возвращено вызвавшей программе.

Помните, что, если куча имеет фиксированный размер, нельзя выделить уча­сток памяти, размер которого превышает 0x7FFF8 байт. Выделив участок памя­ти, вы можете узнать его реальный размер (который может превышать указан­ное вами значение) при помощи вызова **HeapSize**.

Изменить размер ранее выделенного участка можно при помощи вызова **Heap ReAltoc**. Помимо уже известных вам флагов HEAP\_GENERATE\_EXCEPTIONS, HEAP\_NO\_ SERIALIZE или HEAP\_ZERO\_MEMORY, можно использовать флаг HEAP\_REALLOC\_IN\_PLACE\_ONLY, который указывает на то, что процедура изменения размера участка должна за­вершиться неудачей в случае, если для изменения размера участка требуется пе­ремещение этого участка в другую область памяти. Такая ситуация может воз­никнуть в случае, если вы увеличиваете размер участка памяти. Уменьшение размера участка никогда не требует его перемещения.

Чтобы освободить ранее выделенный участок памяти, следует обратиться к функции **HeapFree**. Чтобы предотвратить сериализацию памяти, следует исполь­зовать флаг HEAP\_NO\_SERIALIZE.

Уплотнение кучи

По мере того как вы используете кучу, принадлежащая ей память становится все более фрагментированной. Это значит, что свободная память кучи разбита на множество небольших участков, перемежающихся с блоками памяти, которые используются вашей программой для хранения данных. Чтобы скомпоновать все свободное пространство кучи в один большой блок, необходимо выполнить уп­лотнение кучи при помощи вызова **HeapCompact**. Этому вызову следует передать дескриптор кучи. Кроме того, вы можете использовать флаг НЕАР\_ NO\_SERIALIZE.

Проверка корректности данных, расположенных в куче

В процессе работы программа может нарушить целостность кучи. Например, при последовательном доступе к элементам массива вы можете по ошибке зайти за его пределы и осуществить запись в память, которая этому массиву не принад­лежит. Возможно, вы не обратите внимание на то, что некоторый блок памяти уже освобожден, и попытаетесь сохранить в нем данные. Чтобы проверить цело­стность кучи, можно использовать функцию **HeapValidate**. При помощи этой функции можно проверить целостность всей кучи (для этого следует указать в каче­стве адреса проверки значение 0) или отдельного блока памяти.

Функция **HeapWalk** позволяет последовательно перебрать блоки памяти, выде­ленные в куче. Эта функция возвращает информацию о каждом блоке в структу­ре PROCESS\_HEAP\_ENTRY (табл.3). Каждое обращение к функции возвращает ин формацию о следующем блоке до тех пор, пока вы не переберете все блоки. Если при этом вы намерены использовать **HeapValidate** для того, чтобы проверить целостность блока, информацию о котором удалось получить при помощи **HeapWalk**, имейте в виду, что исполнение **HeapValidate** в отношении блока, который не вы­делен, закончится неудачей. По этой причине функцию **HeapValidate** следует ис­пользовать в отношении только тех блоков, для которых установлен флаг PROCESS\_ HEAP\_ENTRY\_BUSY.

Структура PROCESS\_HEAP\_ENTRY **Таблица 3.**

|  |  |  |
| --- | --- | --- |
| Поле | Тип | Описание |
| IpData | PVOID | Указатель на данные |
| cdData | DWORD | Размер блока |
| cbOverhead | BYTE | Количество байт, используемых для управления  блоком |
| iRegionIndex | BYTE | Индекс участка памяти |
| wFlags | WORD | Статус участка памяти |
| hMem | HANDLE | Дескриптор выделенного участка памяти |
| Region.dwCommittedSize | DWORD | Количество выделенных байт |
| Region.dwUnCommittedSize | DWORD | Количество невыделенных байт |
| RegionlpFirstBlock | LPVOID | Указатель на первый корректный блок памяти |
| Region.IpLastBlock | LPVOID | Указатель на первый некорректный блок памяти |

Увеличение производительности программы с использованием нескольких куч

Используя несколько пулов свободной памяти вместо одного, можно улучшить производительность вашей программы. Очевидным способом повышения эффек­тивности использования памяти является применение разных куч для хранения элементов и объектов разного размера. В этом случае удается избежать фрагментации памяти, так как при освобождении участка, использованного для хра­нения одного элемента, этот же участок можно использовал для хранения дру­гого элемента того же самого размера.

Еще один способ увеличения производительности — использование отдельных куч для обслуживания разных потоков программы. Если кучей пользуется толь­ко один поток, для нее можно отключить сериализацию. При этом Windows не будет осуществлять автоматическое блокирование потоков, обращающихся к куче в одно и то же время, что приведет к увеличению производительности. Сериализа­цию кучи можно отключить также в том случае, если потоки вашей программы уже используют собственный метод синхронизации доступа к куче. Отключая сериализацию, убедитесь в том, что в любой момент времени *к* куче обращается лишь один программный поток.