БЛОГ GUNSMOKER-A

...WHEN ALTERING ONE'S MIND BECOMES AS EASY AS PROGRAMMING A COMPUTER, WHAT DOES IT MEAN TO BE HUMAN?..

ОБО МНЕ

АЛЕКСАНДР АЛЕКСЕЕВ
ТВЕРЬ, ТВЕРСКАЯ ОБЛАСТЬ, RUSSIA

Tech geek, have social skills of a thermonuclear device.

ПРОСМОТРЕТЬ ВЕСЬ ПРОФИЛЬ
Написать письмо (e-mail)

ПОИСК ПО ЭТОМУ БЛОГУ (GOOGLE)

Google™ Пользовательский пои
Поиск

ПОИСК ПО ЭТОМУ БЛОГУ (ЯНДЕКС)

Яндекс Найти

СЛУЧАЙНЫЕ ПОСТЫ

Расширяем HelpInsight...
Разработка системы плагинов в Delphi
Несколько слов о UAC в Vista
Настройки проектов в Delphi с точки зрения поиска ошибок

25 АПРЕЛЯ 2011 Г.

Архитектура памяти в Windows: мифы и легенды (spin-off)

Этот пост - несколько необычное ответвление (spin-off) предыдущего.

Слова "звучит как сюжет для разрушителей легенд" из него, сказанные случайно, красного словца ради, сильно запали мне в голову, и я решил, что было бы неплохо самому "разрушить" несколько таких мифов. И вот появился этот пост - возможно, недостаточно точный технически, но рассказывающий различные факты о памяти в, как я надеюсь, занимательной форме.

Мне кажется, что будет особенно интересно, если вы попробуете при чтении угадывать результаты экспериментов (и статус мифов) до того, как они будут изложены.

Содержание

пространство для всех программ
6. Режим `/3GB` позволит мне выделить `1` гигантский блок памяти в `3` Гб
7. 32-х разрядная программа не может выделить более `3` Гб в своём адресном пространстве
8. 32-х разрядная операционная система не может использовать все `4` Гб оперативной памяти
9. Вам нужно включать режим `/3GB`, если у вас есть больше `2` Гб физической памяти
10. Большой .exe файл - это плохо, потому что он тратит память
11. Delphi приложение занимает много памяти
12. Доступ к невыделенной памяти приводит к возбуждению Access Violation
13. Освобождение памяти уменьшает показатели использования памяти программы
14. `Obj.Free` не приводит к `Obj = nil`
15. Если программа не освободит память, то в системе останется мусор и она замедлится

Миф №1: программа не может выделить больше памяти, чем установлено ОЗУ

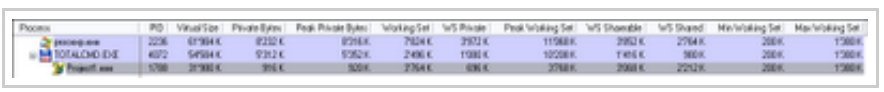
Миф происходит от того, что люди не понимают, что адресное пространство программы теперь виртуально. Оно более не связано с оперативной памятью (вот уже более пятнадцати лет).

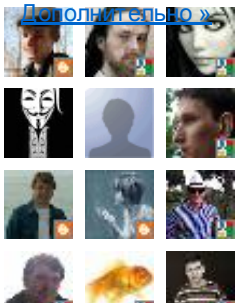
Этот миф легко разрушить непосредственным экспериментом. Я установил количество ОЗУ для виртуальной машины в `256` Мб, запустил её и выполнил такой код:

```
1  procedure TForm1.Button1Click(Sender: TObject);
2  begin
3    AllocMem(512 * 1024 * 1024); // выделить 512 Мб памя
4  end;
```

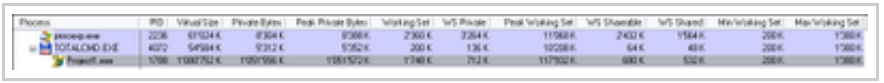
Эта операция будет успешна (хотя и достаточно медленна). Операция была бы мгновенной, если бы мы использовали только резервирование (RESERVE), вместо полноценного выделения (COMMIT), но, возможно, тогда эксперимент не был бы таким зрелищным.

Итак, вот снимок экрана с запущенной программой до выделения:

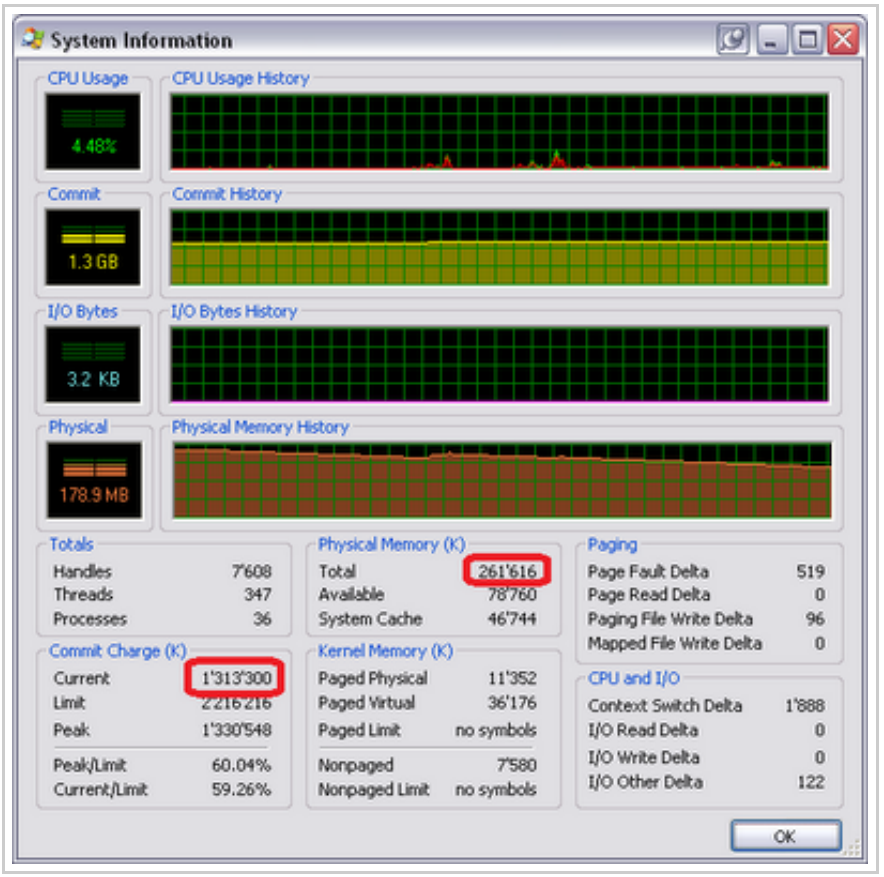




и после (я нажал на кнопку аж два раза):



А вот и общая статистика системы:



Как вы видите, на машине установлено `261'616` Кб оперативной памяти. До выделения памяти наша программа занимала `31'980` Кб виртуальной памяти и `3'764` Кб оперативной. После выделения памяти программа стала занимать `1'080'752` Кб виртуальной памяти и `1'748` Кб физической. Вы также можете увидеть, что суммарное количество выделенной памяти в системе равно `1'313'300` Кб.

Итак, легенда разрушена прямым экспериментом.

Как известно, классический сюжет Mythbusters состоит из двух частей. Когда легенда разрушена, разрушители легенд подбирают такие условия, при которых происходило бы событие, упоминающееся в легенде.

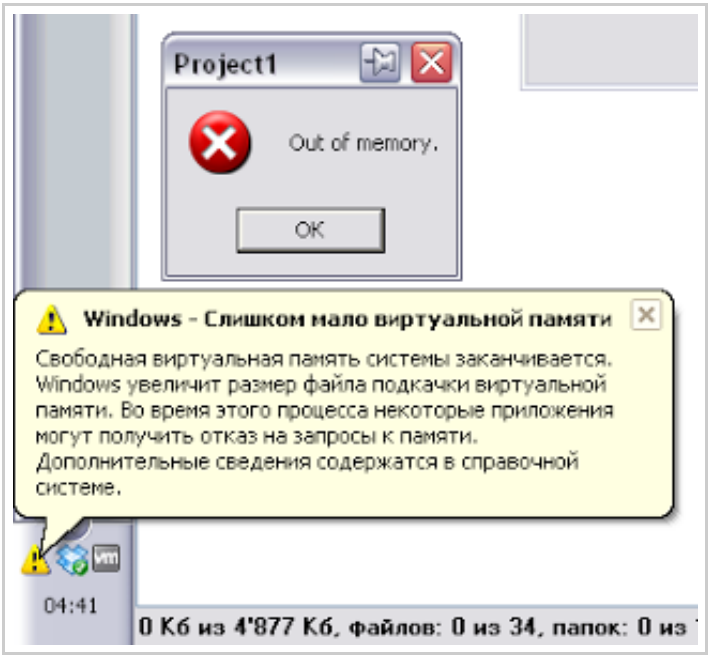
Этим мы сейчас и займёмся: мы заходим в свойства системы и уменьшаем размер файла подкачки до `128` Мб. Таким образом, суммарный объём памяти, доступный системе и всем

ТЭГИ

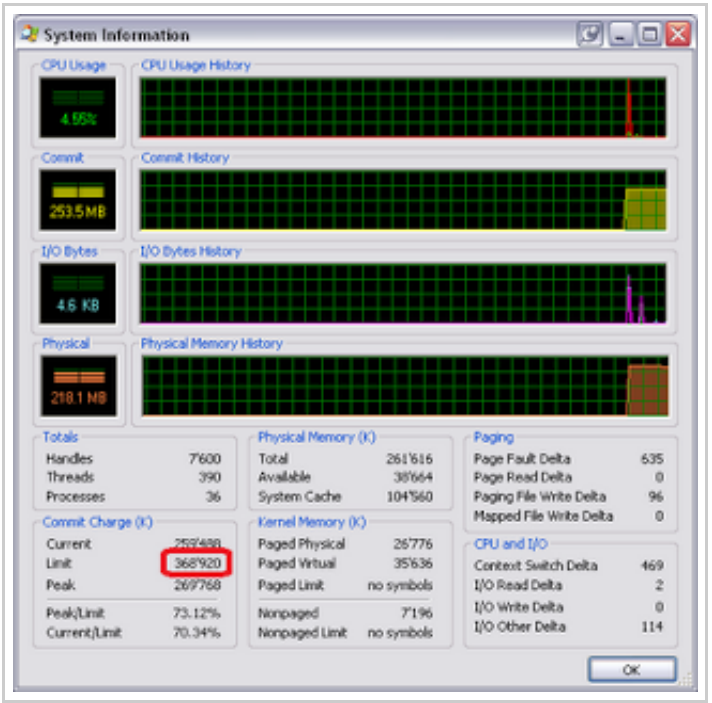
7 (4)

программам, будет равен `256 + 128 = 384` Мб.

Перезагрузка, запускаем тестовый пример снова и вот результат:



На этот раз наш вызов `AllocMem` проваливается с выбросом исключения `EOutOfMemory`. И Process Explorer показывает нам причину:



Статус мифа: **busted**.

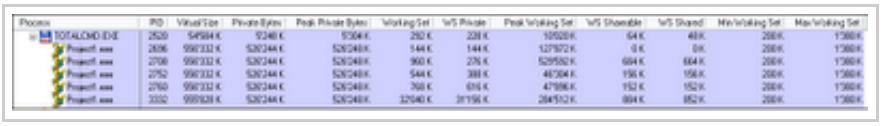
Миф №2: суммарный размер памяти для всех программ не может превышать 2 Гб

Выше мы увидели, что программа может выделить сколько угодно памяти, пока у неё есть место в виртуальном адресном пространстве. Т.е. 32-х разрядная программа может выделить 512 Мб, но не 2 Гб - потому что это размер пользовательской части адресного пространства по умолчанию. Некоторые люди считают, что все запущенные программы в системе не могут выделить более двух гигабайт памяти.

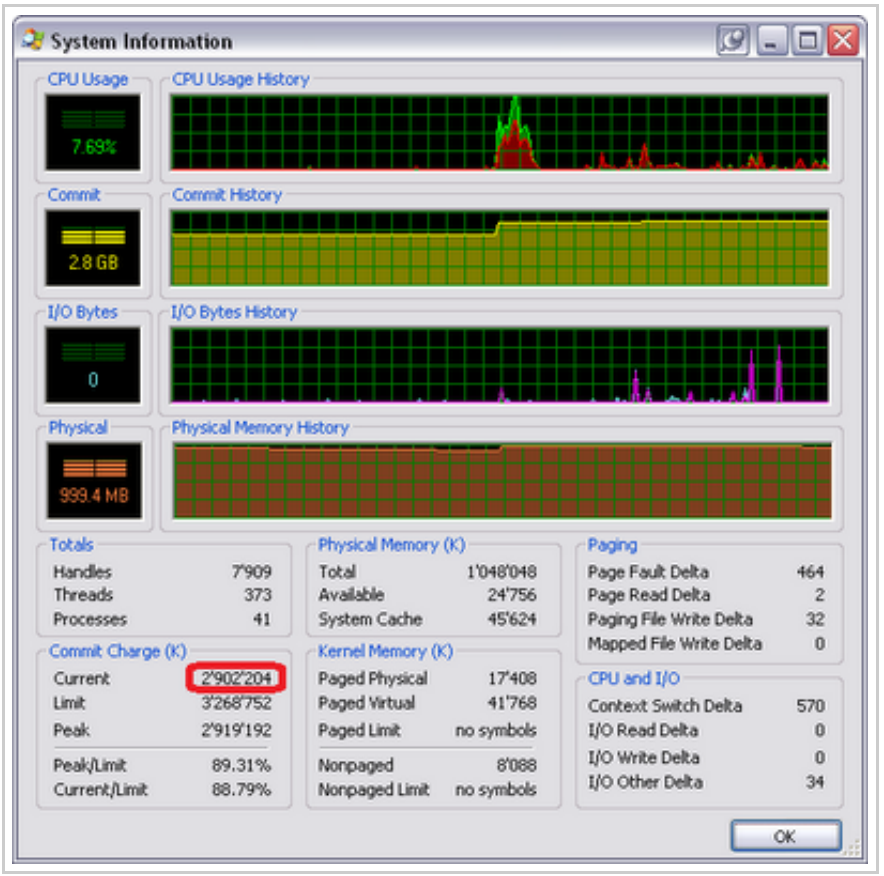
Этот миф происходит от того, что люди не понимают, что адресное пространство программы теперь своё у каждой программы (вот уже более пятнадцати лет).

Посмотрим, так ли это.

В этот раз я запустил пять копий программы-примера из предыдущего пункта и вот что получилось:



А вот и статус системы в целом:



Как видите, никаких проблем нет: все запущенные приложения в системе смогли выделить 2'902'204 Кб памяти (и да, я поднял кол-во ОЗУ до 1 Гб, чтобы система поменьше тормозила).

Что касается части два, то она выглядит так же, как ранее: нельзя выделить памяти больше, чем у вас есть оперативной памяти + файла подкачки.

Статус мифа: **busted**.

Миф №3: 32-х разрядное приложение не может выделить 1.5 Гб памяти за раз

Несмотря на то, что приложению доступно по умолчанию около 2 Гб виртуального адресного пространства, утверждается, что приложение не может выделить 1.5 Гб памяти одним куском.

Давайте проверим. Изменим код с `AllocMem` в нашем тестовом приложении на выделение 1.5 Гб и запустим программу. Получаем:

Легенда разрушена?

Не так быстро. Попробуем сделать это на другой машине:

(сообщения "Мало виртуальной памяти" нет)

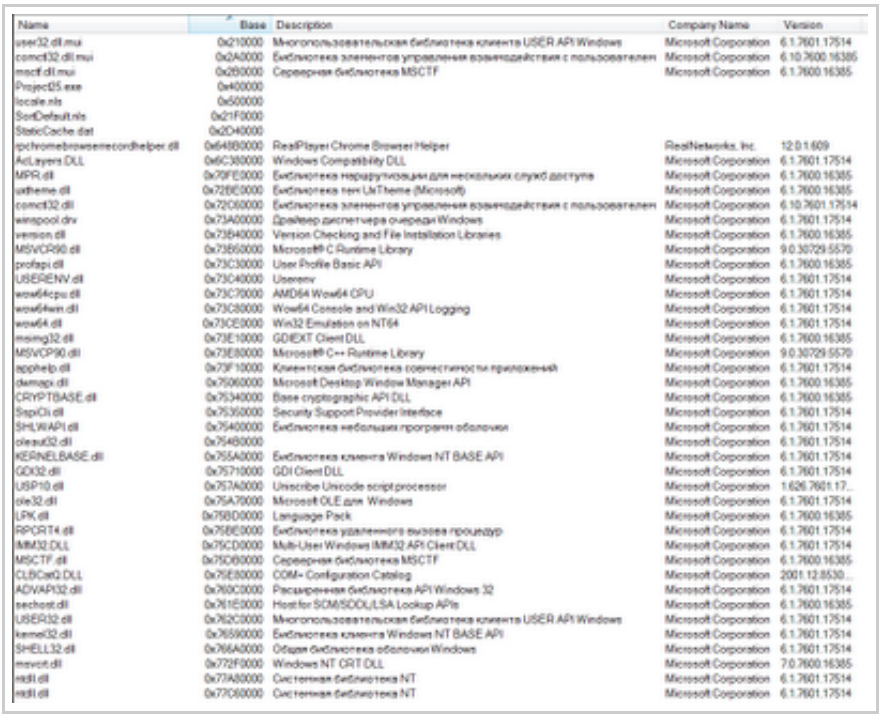
Гм, в этот раз нам не удаётся выделить 1.5 Гб памяти.

Мы получили противоречивые результаты. В чём же дело?

Хотя нам действительно доступно около 2 Гб одним куском (только в самом начале и в самом конце этого региона откушено по 64 Кб на спец. области), но нужно вспомнить, что в этом адресном пространстве лежат не только ваши данные, но и ваш код, библиотеки (DLL), их код и так далее. Даже если вы не загружали библиотек явно в вашем коде - они всё равно будут загружены. Как минимум это `kernel32.dll` и `user32.dll`. И дальше всё зависит от того, как именно они загружены. Обычно

системные библиотеки загружаются одним большим компактным регионом, расположенном по старшим адресам - поскольку они загружаются с краю адресного пространства, то в центре у вас получается большой кусок для вашей работы. Но если какая-то DLL загружается в середину адресного пространства, то оно оказывается разбито пополам, и вы уже не сможете выделить память одним куском (но всё ещё можете выделить её в два или три куска).

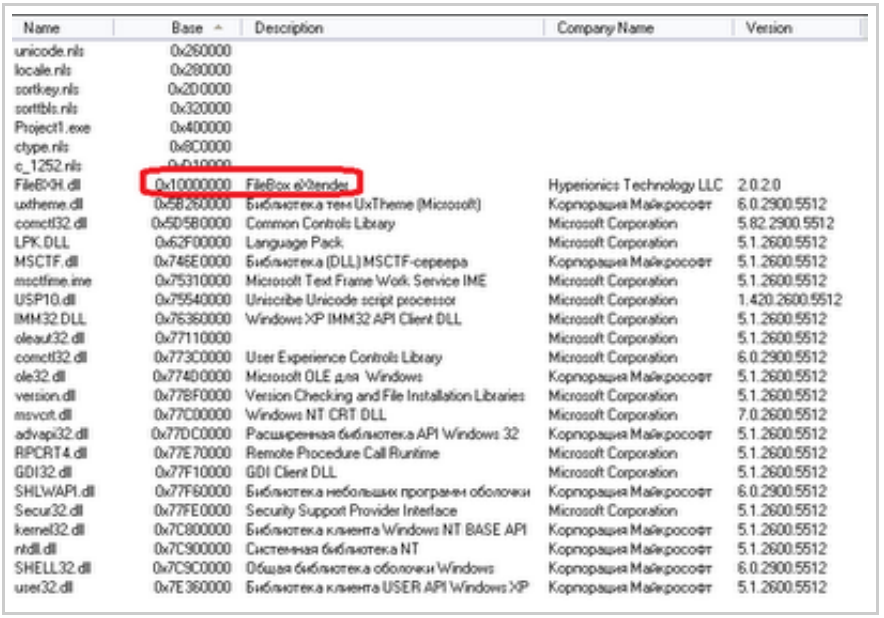
К примеру, вот снимок загруженных DLL в адресном пространстве первого примера (который успешно выделил `1.5` Гб памяти) до выделения памяти:



Name	Base	Description	Company Name	Version
user32.dll.mui	0x210000	Многопользовательская библиотека клиента USER API Windows	Microsoft Corporation	6.1.7601.17514
comctl32.dll.mui	0x2A0000	Библиотека элементов управления взаимодействия с пользователем	Microsoft Corporation	6.10.7600.16385
msctf.dll.mui	0x2B0000	Серверная библиотека MSCTF	Microsoft Corporation	6.1.7600.16385
Project25.exe	0x400000			
locale.nls	0x500000			
SortDefault.nls	0x21F0000			
StaticCache.dat	0x2D40000			
rpchromebrowserecordhelper.dll	0x648B0000	RealPlayer Chrome Browser Helper	RealNetworks, Inc.	12.0.1.609
AcLayers.DLL	0x6C380000	Windows Compatibility DLL	Microsoft Corporation	6.1.7601.17514
MPR.dll	0x70FE0000	Библиотека маршрутизации для нескольких служб доступа	Microsoft Corporation	6.1.7600.16385
uxtheme.dll	0x72BE0000	Библиотека тем UxTheme (Microsoft)	Microsoft Corporation	6.1.7600.16385
comctl32.dll	0x72C60000	Библиотека элементов управления взаимодействия с пользователем	Microsoft Corporation	6.10.7601.17514
winspool.drv	0x73A00000	Драйвер диспетчера очереди Windows	Microsoft Corporation	6.1.7601.17514
version.dll	0x73B40000	Version Checking and File Installation Libraries	Microsoft Corporation	6.1.7600.16385
MSVCR90.dll	0x73B60000	Microsoft® C Runtime Library	Microsoft Corporation	9.0.30729.5570
profapi.dll	0x73C30000	User Profile Basic API	Microsoft Corporation	6.1.7600.16385
USERENV.dll	0x73C40000	Userenv	Microsoft Corporation	6.1.7601.17514
wow64cpu.dll	0x73C70000	AMD64 Wow64 CPU	Microsoft Corporation	6.1.7601.17514
wow64win.dll	0x73C80000	Wow64 Console and Win32 API Logging	Microsoft Corporation	6.1.7601.17514
wow64.dll	0x73CE0000	Win32 Emulation on NT64	Microsoft Corporation	6.1.7601.17514
msimg32.dll	0x73E10000	GDIEXT Client DLL	Microsoft Corporation	6.1.7600.16385
MSVCP90.dll	0x73E80000	Microsoft® C++ Runtime Library	Microsoft Corporation	9.0.30729.5570
apphelp.dll	0x73F10000	Клиентская библиотека совместимости приложений	Microsoft Corporation	6.1.7601.17514
dwmapi.dll	0x75060000	Microsoft Desktop Window Manager API	Microsoft Corporation	6.1.7600.16385
CRYPTBASE.dll	0x75340000	Base cryptographic API DLL	Microsoft Corporation	6.1.7600.16385
SspiCli.dll	0x75350000	Security Support Provider Interface	Microsoft Corporation	6.1.7601.17514
SHLWAPI.dll	0x75400000	Библиотека небольших программ оболочки	Microsoft Corporation	6.1.7601.17514
oleaut32.dll	0x754B0000		Microsoft Corporation	6.1.7601.17514
KERNELBASE.dll	0x755A0000	Библиотека клиента Windows NT BASE API	Microsoft Corporation	6.1.7601.17514
GDI32.dll	0x75710000	GDI Client DLL	Microsoft Corporation	6.1.7601.17514
USP10.dll	0x757A0000	Unicode Unicode script processor	Microsoft Corporation	1.626.7601.17...
ole32.dll	0x75A70000	Microsoft OLE для Windows	Microsoft Corporation	6.1.7601.17514
LPK.dll	0x75BD0000	Language Pack	Microsoft Corporation	6.1.7600.16385
RPCRT4.dll	0x75BE0000	Библиотека удаленного вызова процедур	Microsoft Corporation	6.1.7601.17514
IMM32.DLL	0x75CD0000	Multi-User Windows IMM32 API Client DLL	Microsoft Corporation	6.1.7601.17514
MSCTF.dll	0x75D90000	Серверная библиотека MSCTF	Microsoft Corporation	6.1.7600.16385
CLBCatQ.DLL	0x75E80000	COM+ Configuration Catalog	Microsoft Corporation	2001.12.8530...
ADVAPI32.dll	0x760C0000	Расширенная библиотека API Windows 32	Microsoft Corporation	6.1.7601.17514
sechost.dll	0x761E0000	Host for SCM/SDDL/LSA Lookup APIs	Microsoft Corporation	6.1.7600.16385
USER32.dll	0x762C0000	Многопользовательская библиотека клиента USER API Windows	Microsoft Corporation	6.1.7601.17514
kernel32.dll	0x76590000	Библиотека клиента Windows NT BASE API	Microsoft Corporation	6.1.7601.17514
SHELL32.dll	0x766A0000	Общая библиотека оболочки Windows	Microsoft Corporation	6.1.7601.17514
msvcrt.dll	0x772F0000	Windows NT CRT DLL	Microsoft Corporation	7.0.7600.16385
ntdll.dll	0x77A80000	Системная библиотека NT	Microsoft Corporation	6.1.7601.17514
ntdll.dll	0x77C60000	Системная библиотека NT	Microsoft Corporation	6.1.7601.17514

Как видим, в центре у нас есть большой свободный кусок - от `$2D40000` до `$648B0000`, т.е. `$648B0000 - $2D40000 = 1'563` Мб (примечание: это не значит, что в этом промежутке нет вообще ничего - там могут быть не DLL, а данные). Т.е. у нас есть свободное место.

А вот этот же снимок DLL на машине, где выделить память не удалось:



Name	Base	Description	Company Name	Version
unicode.nls	0x260000			
locale.nls	0x280000			
sortkey.nls	0x2D0000			
sorttbls.nls	0x320000			
Project1.exe	0x400000			
ctype.nls	0x6C0000			
c_1252.nls	0x710000			
FileBXH.dll	0x10000000	FileBox eXtender	Hyperionics Technology LLC	2.0.2.0
uxtheme.dll	0x5B260000	Библиотека тем UxTheme (Microsoft)	Корпорация Майкрософт	6.0.2900.5512
comctl32.dll	0x5D5B0000	Common Controls Library	Microsoft Corporation	5.82.2900.5512
LPK.DLL	0x62F00000	Language Pack	Microsoft Corporation	5.1.2600.5512
MSCTF.dll	0x746E0000	Библиотека (DLL) MSCTF-сервера	Корпорация Майкрософт	5.1.2600.5512
msctfime.ime	0x75310000	Microsoft Text Frame Work Service IME	Microsoft Corporation	5.1.2600.5512
USP10.dll	0x75540000	Uniscribe Unicode script processor	Microsoft Corporation	1.420.2600.5512
IMM32.DLL	0x76360000	Windows XP IMM32 API Client DLL	Microsoft Corporation	5.1.2600.5512
oleaut32.dll	0x77110000		Microsoft Corporation	5.1.2600.5512
comctl32.dll	0x773C0000	User Experience Controls Library	Microsoft Corporation	6.0.2900.5512
ole32.dll	0x774D0000	Microsoft OLE для Windows	Корпорация Майкрософт	5.1.2600.5512
version.dll	0x77BF0000	Version Checking and File Installation Libraries	Microsoft Corporation	5.1.2600.5512
msvcrt.dll	0x77C00000	Windows NT CRT DLL	Microsoft Corporation	7.0.2600.5512
advapi32.dll	0x77DC0000	Расширенная библиотека API Windows 32	Корпорация Майкрософт	5.1.2600.5512
RPCRT4.dll	0x77E70000	Remote Procedure Call Runtime	Microsoft Corporation	5.1.2600.5512
GDI32.dll	0x77F10000	GDI Client DLL	Microsoft Corporation	5.1.2600.5512
SHLWAPI.dll	0x77F60000	Библиотека небольших программ оболочки	Корпорация Майкрософт	6.0.2900.5512
Secur32.dll	0x77FE0000	Security Support Provider Interface	Microsoft Corporation	5.1.2600.5512
kernel32.dll	0x7C800000	Библиотека клиента Windows NT BASE API	Корпорация Майкрософт	5.1.2600.5512
ntdll.dll	0x7C900000	Системная библиотека NT	Корпорация Майкрософт	5.1.2600.5512
SHELL32.dll	0x7C9C0000	Общая библиотека оболочки Windows	Корпорация Майкрософт	6.0.2900.5512
user32.dll	0x7E360000	Библиотека клиента USER API Windows XP	Корпорация Майкрософт	5.1.2600.5512

Как видите, в этом случае в середине большого свободного промежутка у нас разместилась DLL от FileBox Extender - это небольшая утилита, которая добавляет полезные кнопки в заголовки окон. Поскольку она меняет поведение каждого окна, то она должна быть загружена в каждую программу. Но из-за того, что она оказалась неграмотно спроектированной, её базовый адрес оказался в неудачном месте. Такая ситуация называется **фрагментацией адресного пространства**.

Мораль истории: либо ставьте поменьше "расширителей оболочки", либо следите, чтобы они были грамотно спроектированы.

Статус мифа: **plausible**.

Миф №4: 32-х разрядное приложение не может использовать более 2 Гб памяти

Постойте-ка, разве мы только что не подтвердили эту легенду? Не совсем. Ведь есть разница: "выделить за раз" и "использовать". Да, вы не можете выделить 2 Гб памяти (или более) - что за раз, что за несколько вызовов: ведь обычно размер пользовательской части виртуального адресного пространства равен 2 Гб, но это не ограничивает вас 2 Гб виртуальной памяти. Вы можете выделять память, без проецирования её в ваше виртуальное адресное пространство. Как мы увидели в мифах 1 и 2: виртуальное адресное пространство программы не равно виртуальной памяти в системе. Второе - больше, чем первое.

Обычно размер пользовательской части виртуального адресного

пространства равен 2 Гб, но это не ограничивает вас 2 Гб виртуальной памяти. Вы можете выделять память, без проецирования её в ваше виртуальное адресное пространство:

```
h := CreateFileMapping(INVALID_HANDLE_VALUE, 0,
                       PAGE_READWRITE, 1, 0, nil);
```

При условии, что у вас достаточно физической памяти и/или файла подкачки, этот запрос на выделение 4 Гб памяти будет успешен.

Конечно же, вы не сможете спроецировать всю эту память **сразу**, но вы можете делать это по частям.

Другим вариантом использования большего объёма памяти является AWE.

В общем, мораль в том, что виртуальное адресное пространство - это не то же самое, что виртуальная память. Как мы видели ранее, вы можете проецировать одну и ту же память по нескольким адресам, так что соответствие один-к-одному для виртуальной памяти и виртуального адресного пространства никогда и не выполнялось. Здесь мы продемонстрировали, что выделение памяти вовсе не означает, что она вообще занимает какое-то место в виртуальном адресном пространстве.

Более того: если вы укажете при загрузке системы ключ /3GB, то вы сможете использовать более 2 Гб виртуального *адресного пространства* (и снова: и ещё больше - виртуальной *памяти*). Ключ /3GB изменяет способ разбиения полных 4 Гб виртуального адресного пространства. Вместо разбиения на 2 Гб пользовательского виртуального адресного пространства и 2 Гб режима ядра, разделение будет сделано на 3 Гб пользовательского и 1 Гб адресного пространства режима ядра (это граница по умолчанию, а вообще она варьируется).

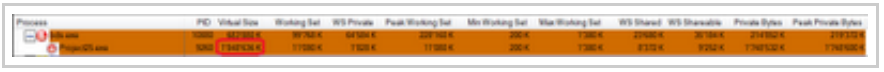
Так что это неверно даже если вы говорили про адресное пространство, а не про память вообще. В этом мифе есть лишь часть правды про адресное пространство.

Статус мифа: **(totally) busted**.

Миф №5: ключ /3GB расширяет пользовательское адресное пространство для всех программ

Ну, давайте включим режим /3GB и запустим нашу программу

пример, где `AllocMem` выделяет `100` Мб. Будем нажимать на кнопку, пока не возникнет сообщение о нехватке памяти и посмотрим, сколько же памяти нам удалось выделить:



Как видим, это существенно меньше ожидаемых `3` Гб памяти.

На самом деле, режим `/3GB` влияет только на программы с флагом `IMAGE_FILE_LARGE_ADDRESS_AWARE`.

По соображениям совместимости, только программы, которые явно пометили себя, что они умеют обрабатывать виртуальное адресное пространство больше `2` Гб, получат большее адресное пространство. Не помеченные программы получат свои обычные `2` Гб, а адресное пространство между `2` Гб и `3` Гб не будет использоваться вовсе.

Почему?

Потому что слишком много программ предполагают, что старший бит адреса в пользовательском режиме всегда очищен (т.е. равен `0`), часто делая это невольно. В MSDN есть страничка, на которой перечисленны несколько способов использования такого предположения. Например, вы можете захотеть найти средний адрес между двумя другими - используя для этого формулу `(a + b) / 2`. Но если `a` и `b` будут больше `2` Гб, то их сумма не влезет в `4`-х байтное целое - следовательно, вы получите неверный результат (для верного вычисления надо использовать выражение `a + (b - a) / 2`). Соответственно, вы не можете просто взять программу, которую вы не писали, пометить её флагом `IMAGE_FILE_LARGE_ADDRESS_AWARE` и объявить, что дело сделано. Вам вместе с авторами программы надо проверить, что код не делает никаких предположений насчёт этих `2` Гб (а тот факт, что программа не была помечена, как совместимая с `3` Гб, означает, что никаких проверок не было сделано. В самом деле - в противном случае она была бы уже помечена флагом `IMAGE_FILE_LARGE_ADDRESS_AWARE`!).

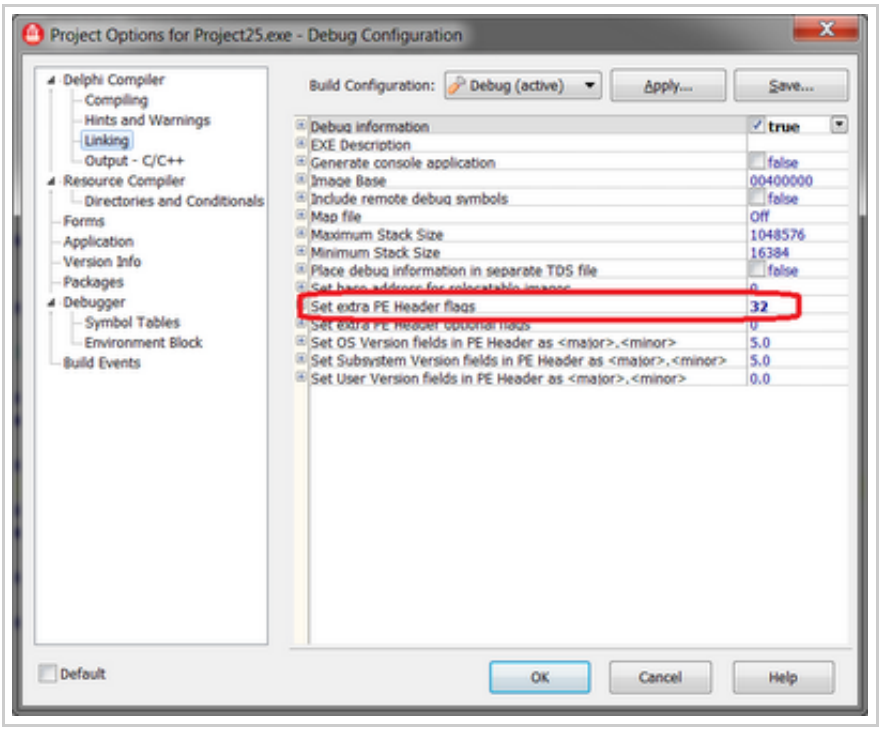
Пометка вашей программы флагом `IMAGE_FILE_LARGE_ADDRESS_AWARE` указывает операционной системе: "давай, дай мне доступ к этой дополнительной памяти пользовательского адресного пространства", в результате адреса выше `2`-х Гб становятся возможными возвращаемыми значениями в функциях выделения памяти. Если вы установите флаг "Top down" в предпочтениях менеджера памяти, вы можете указать менеджеру памяти выделять память сначала по старшим

адресам, таким образом, вы заставите свою программу работать на высоких адресах сразу же, а не когда заполнится остальное место. Это очень удобный режим для проверки вашей программы в конфигурации `/3GB`, посольку он заставляет скорее, чем в обычном режиме, использовать проблемные адреса.

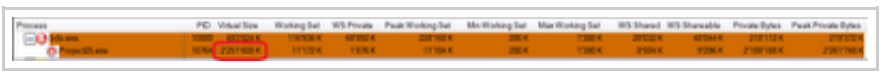
Итак, давайте включим `IMAGE_FILE_LARGE_ADDRESS_AWARE` для нашей программы:

```
1  project Project1;
2
3  uses
4    Windows; // для определения IMAGE_FILE_LARGE_ADDRESS
5
6  {$SetPEFlags IMAGE_FILE_LARGE_ADDRESS_AWARE}
7
8  ...
9
10 end.
```

Или (`IMAGE_FILE_LARGE_ADDRESS_AWARE = $20` или `32`):



...и посмотрим, как это изменит ситуацию:



Больше 2 Гб - что и требовалось показать (кстати, это же является примером и к предыдущему мифу).

Статус мифа: **busted**.

Миф №6: режим /3GB позволит мне выделить 1 гигантский блок памяти в 3 Гб

Просто то, что у вас есть аж 3 Гб виртуального адресного пространства, ещё не означает, что вы можете выделить один гигантский блок памяти размером 3 Гб. Мы уже видели (в мифе №3), что в виртуальное адресное пространство может быть фрагментировано, и вы не сможете выделить большой кусок за раз.

Стандартные дыры в виртуальном адресном пространстве не изменились: это 64 Кб внизу и 64 Кб около границы в 2 Гб.

Более того, системные DLL продолжают загружаться по их предпочтительным базовым адресам, которые лежат ниже границы 2 Гб. Куча процесса и другие типичные данные также откусывают понемногу от вашего виртуального адресного пространства.

В результате то, что пользовательское виртуальное адресное пространство практически равно 3 Гб, ещё не значит, что всё свободное пространство представлено одним блоком. Дыры около границы 2 Гб не дают вам получить непрерывного участка даже в 2 Гб.

> Примечание: некоторые люди могут захотеть попробовать переместить системные DLL по другим адресам, чтобы освободить побольше места, но это не сработает по нескольким причинам. Во-первых, конечно же, этим вы не избавитесь от пробела в 64 Кб около 2 Гб-ной границы. Во-вторых, система выделяет и другие данные, такие как блоки с информацией о потоках (thread information blocks) и переменные окружения, до того, как ваша программа получит шанс на выполнение; так что к тому времени, как ваша программа сможет выделять память, адресное пространство уже будет занято.
>
> Кроме того, системе действительно нужно, чтобы некоторые ключевые системные DLL были загружены по одним и тем же адресам во всех процессах. Например, ловушка `syscall` должна находиться в фиксированном месте, чтобы обработчик ловушки режима ядра опознал её как допустимую ловушку `syscall`, а не как недопустимую инструкцию. Также этого требует отладчик, чтобы он мог использовать функцию `CreateRemoteThread` для внедрения точки останова в процесс.

Статус мифа: **busted**.

Миф №7: 32-х разрядная программа не может выделить более 3 Гб в своём адресном пространстве

Как мы увидели выше (в мифе №5), включение режима `/3GB` позволяет вам выделить память больше `2` Гб, но в том эксперименте вы могли столкнуться об ограничение в `3` Гб. Утверждается, что `32`-х разрядная программа, скомпилированная с `IMAGE_FILE_LARGE_ADDRESS_AWARE`, не может выделить более `3` Гб памяти.

Кажется, что легенда подтверждена в мифе №5? Но не так быстро!

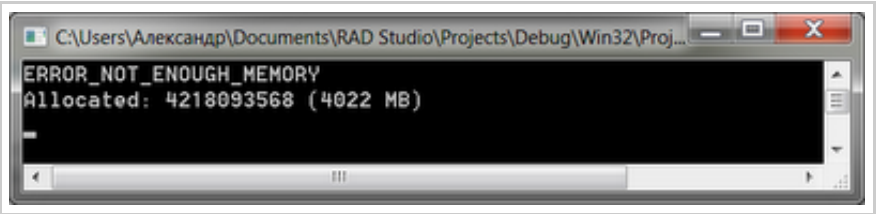
Создадим такую программу:

```
1  program Project1;
2
3  {$APPTYPE CONSOLE}
4
5  uses
6    Windows;
7
8  {$SetPEFlags IMAGE_FILE_LARGE_ADDRESS_AWARE}
9
10 const
11   ReserveSize = 1024;            // 1024 * 64 Kb - рез
12   IncSize: Cardinal = 64 * 1024; // выделения по 64 Kb
13 var
14   Sz: Cardinal;
15   LasrErr: Cardinal;
16   Reserve: Pointer;
17 begin
18   // Сохранили резерв
19   Reserve := VirtualAlloc(nil, ReserveSize * IncSize,
20
21   // Цикл по определению максимума
22   Sz := ReserveSize * IncSize;
23   while Assigned(VirtualAlloc(nil, IncSize, MEM_RESERV
24     Inc(Sz, IncSize);
25   LasrErr := GetLastError;
26
27   // Отпустили резерв, чтобы у нас была память для обр
28   VirtualFree(Reserve, 0, MEM_RELEASE);
29
30   // Смотрим, что получилось
31   if LasrErr = ERROR_NOT_ENOUGH_MEMORY then
32     WriteLn('ERROR_NOT_ENOUGH_MEMORY')
33   else
34     WriteLn(LasrErr);
35   WriteLn('Allocated: ', Sz, ' (', Sz div (1024 * 1024
36   ReadLn;
37 end.
```

Эта программа пытается исчерпать память кусками по `64` Кб. Кроме того, она держит резерв памяти, чтобы выполнить `WriteLn` и работу со строками в конце (в самом деле, если вы исчерпаете всю память, то не сможете вывести результат). Программа

также помечена флагом `IMAGE_FILE_LARGE_ADDRESS_AWARE`, что даёт ей доступ к памяти больше 2 Гб.

Теперь запустим эту 32-х разрядную программу на **64-х разрядной системе** (никаких дополнительных действий вроде включения спец. режимов не требуется):



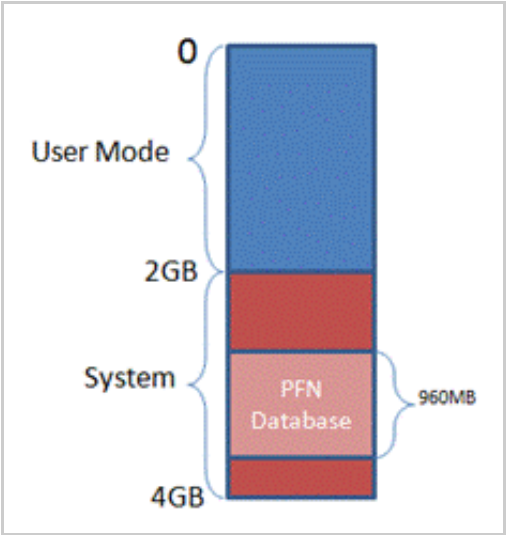
Это ж без малого аж 4 Гб для 32-х разрядной программы! Т.е. почти двукратное увеличение по сравнению с обычными 2 Гб. Круто.

Статус мифа: **plausible**.

Миф №8: 32-х разрядная операционная система не может использовать все 4 Гб оперативной памяти

Максимальный для 32-разрядных систем объем памяти – это 128 Гб, как указано в спецификации на Windows Server 2003 Datacenter Edition.

> Такое ограничение связано с тем, что в более мощных системах структуры, применяемые диспетчером памяти для отслеживания физической памяти, потребляли бы слишком большую часть пространства виртуальных адресов. Диспетчер памяти отслеживает страницы памяти при помощи массива, называемого базой данных PFN, и в целях оптимизации производительности отображает все содержимое этой базы в виртуальную память. Так как каждая страница памяти представлена структурой данных объемом 28 байт, в системе с физической памятью емкостью 128 Гб для размещения базы данных PFN потребуется 930 Мб. В 32-разрядных ОС Windows предусмотрено пространство виртуальных адресов объемом 4 Гб, зависящее от оборудования и по умолчанию распределяемое между текущим процессом пользовательского режима (например, блокнотом) и системой. В таких условиях база данных PFN объемом 980 Мб занимает почти половину из доступных 2 Гб системной части пространства виртуальных адресов, а значит, на отображение ядра, драйверов устройств, системного кэша и других структур данных системы остается всего 1 Гб:



> По той же причине в таблице ограничений объема памяти указаны пониженные лимиты при загрузке в режиме `/3GB`. Дело в том, что для этого режима характерна такая схема разделения физической памяти, при которой процессам пользовательского режима достается `3` Гб, а системе – всего `1` Гб. В целях повышения производительности в ОС Windows Server 2008 для системных нужд резервируется более значимая доля адресного пространства. Для этого максимальный объем физической памяти, поддерживаемый в `32`-разрядных версиях ОС, сокращается до `64` Гб.

Но разрушители легенд не были бы разрушителями легенд, если бы они верили на слово. Поэтому они должны это проверить.

Берём виртуальную машину, устанавливаем ей количество ОЗУ в `4` Гб и запускаем. Что же мы видим?

Что-то не очень похоже на обещанные `128` Гб. В чём же дело?

Дело в том, что ограничение в `128` Гб - это ограничение ***серверных*** ОС. ***Клиентские*** ОС (а Windows XP и Windows 7 - это *клиентские* ОС) имеют ограничения в `4` Гб.

Ну, это ничего не объясняет. Во-первых, почему такая разница? Это маркетинговый ход? Во-вторых: где же наши обещанные `4` Гб? Мы видим всего `3.5` Гб.

Во-первых, в ходе тестирования Windows выяснилось, что если разрешить использование памяти более 4 Гб, то многие системы аварийно завершают работу, зависают и отказываются загружаться. Происходит это из-за того, что некоторые

драйверы устройств (в особенности аудио- и видеоустройств) запрограммированы на работу с физическими адресами в пределах 4 Гб. Эти драйверы, оказывается, обрубают адреса свыше 4 Гб, что приводит к повреждению содержимого памяти со всеми вытекающими последствиями. В серверных же системах, которые, как правило, оснащаются менее специфичными устройствами с относительно простыми и надежными драйверами, подобные проблемы обнаружены не были. Выявленные недостатки экосистемы драйверов заставили применительно к клиентским версиям ОС отказаться от работы с памятью в объеме свыше 4 Гб, несмотря на то, что теоретически её адресация возможна (обращаю внимание, что речь идёт о физической памяти, а не о виртуальном адресном пространстве, которое даже теоретически не может быть больше 4 Гб в 32-х разрядной системе).

Во-вторых, фактический лимит поддержки объема памяти ниже. Кроме того, он зависит от набора микросхем и характеристик подключенных устройств. Дело в том, что в таблицу физических адресов включается не только оперативная память, но и память устройств. При этом, для совместимости с 32-разрядными операционными системами, которые не способны обрабатывать адреса свыше 4 Гб, в системах x86 и x64 память устройств отображается ниже границы адресации 4 Гб. Предположим, что в системе установлено 4 Гб оперативной памяти, а окна в память сетевых адаптеров, аудио- и видеоустройств в сумме составляют 500 Мб, тогда 500 Мб из 4 Гб оперативной памяти окажутся за границей адресации - и мы получим доступные только 3.5 Гб физической памяти.

Даже если система оснащена всего 2 Гб физической памяти, может случиться так, что часть её окажется недоступной под управлением 32-разрядной версии Windows. Причиной тому – наборы микросхем, практикующие агрессивное резервирование областей памяти для устройств. Хотя такой сценарий, конечно, достаточно редок.

Статус мифа: **plausible**.

Миф №9: вам нужно включать режим /3GB, если у вас есть больше 2 Гб физической памяти

Физическая память - это не виртуальная память.

Я не уверен, какой логический процесс привёл к рождению этого

мифа. Это не может быть из-за неверной интерпретации соответствия один-к-одному виртуальной и физической памяти, поскольку отображение явно не один-к-одному. Обычно у вас намного больше виртуальной памяти, чем физической. *Свободная физическая память не имеет соответствия ни в одном виртуальном адресном пространстве*. А разделяемая память обозначена в нескольких виртуальных адресных пространствах, хотя соответствует одним и тем же страницам физической памяти.

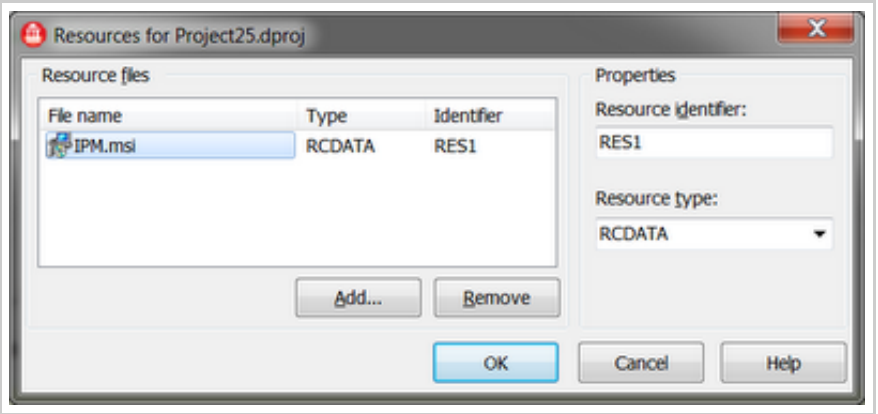
Этот миф разрушен в предыдущем расследовании, где никакого режима `/3GB` мы не включали, но получили `3.5` Гб памяти.

Статус мифа: **busted**.

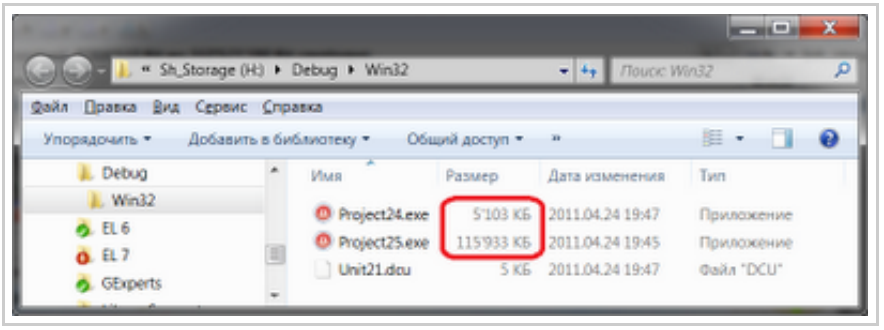
Миф №10: большой .exe файл - это плохо, потому что он тратит память

Замечу, что, конечно же, в этом мифе имеется в виду физическая память, а не виртуальная - в том смысле, что большее потребление физической памяти - это нагрузка на систему, т.е. плохо. Очевидно, что виртуальное адресное пространство включает в себя .exe файл, так что тут даже нечего обсуждать.

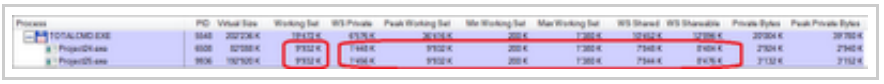
Итак, я создал два идентичных проекта (пустых VCL приложения). Но во втором приложении я сделал `Project / Resources and images` и выбрал 110-мегабайтный файл:



Компиляция и мы получаем два файла - в примерно 5 и 110 Мб (включена отладочная информация TD32):



Запускаем обе программы и...



Неожиданно оказывается, что обе программы, несмотря на двадцатикратную разницу в размере, потребляют полностью идентичное количество оперативной памяти! Сюрприз.

Мне кажется, что этот миф получается из-за того, что люди не видят разницы между оперативной памятью и виртуальным адресным пространством.

Смысл в том, что при загрузке программы (или DLL) её файл проецируется на адресное пространство процесса с помощью механизма проецируемых в память файлов. Как и виртуальная память, проецируемые файлы позволяют резервировать регион адресного пространства и передавать ему физическую память. Различие между этими механизмами состоит в том, что в последнем случае физическая память не выделяется из страничного файла (файла подкачки), а берется из файла, уже находящегося на диске. Как только файл спроецирован в память, к нему можно обращаться так, будто он целиком в нее загружен.

Проецируемые файлы применяются для:

- загрузки и выполнения EXE- и DLL-файлов. Это позволяет существенно экономить как на размере страничного файла, так и на времени, необходимом для подготовки приложения к выполнению;
- доступа к файлу данных, размещенному на диске. Это позволяет обойтись без операций файлового ввода-вывода и буферизации его содержимого;
- разделения данных между несколькими процессами, выполняемыми на одной машине (в Windows есть и другие методы для совместного доступа разных процессов к одним данным — но все они так или иначе реализованы на основе проецируемых в память файлов);

Вот почему мы видим увеличение на `+110` Мб виртуальной памяти у второго процесса - потому что туда спроецирован больший по размеру .exe файл.

При вызове из потока функции CreateProcess система действует так:

1. Отыскивает EXE-файл, указанный при вызове функции `CreateProcess`;
2. Создает новый объект ядра "процесс";
3. Создает адресное пространство нового процесса;
4. Резервирует регион адресного пространства — такой, чтобы в него поместился данный EXE-файл. Желательное расположение этого региона указывается внутри самого EXE-файла. По умолчанию базовый адрес EXE-файла — $00400000.
5. Отмечает, что физическая память, связанная с зарезервированным регионом, — EXE-файл на диске, а не страничный файл.

Спроецировав EXE-файл на адресное пространство процесса, система обращается к разделу EXE-файла со списком DLL, содержащих необходимые программе функции. После этого система, вызывая `LoadLibrary`, поочередно загружает указанные (а при необходимости и дополнительные) DLL-модули. Всякий раз, когда для загрузки DLL вызывается `LoadLibrary`, система выполняет действия, аналогичные описанным выше в пп. 4 и 5.

После увязки EXE- и DLL-файлов с адресным пространством процесса начинает исполняться стартовый код EXE-файла. Подкачку страниц, буферизацию и кэширование система берет на себя. Например, если код в EXE-файле переходит к команде, не загруженной в память, возникает ошибка. Обнаружив её, система перекачивает нужную страницу кода из образа файла на страницу оперативной памяти. Затем она отображает страницу оперативной памяти на должный участок адресного пространства процесса, тем самым позволяя потоку продолжить выполнение кода. Все эти операции скрыты от приложения и периодически повторяются при каждой попытке процесса обратиться к коду или данным, отсутствующим в оперативной памяти.

Иными словами, при загрузке процесса, не имеет значения, какого размера будет .exe файл - файл будет лишь спроецирован на адресное пространство, но в физической памяти будет только первая страница кода. Все остальные части будут загружены только по мере обращения кода к ним.

Но наша работа на этом ещё не закончена. Когда же размер файла

имеет значение?

Ответ: при упаковке или шифровании.

В Windows загрузчик читает лишь заголовок и таблицу импорта файла, а затем проецирует его на адресное пространство процесса так, будто бы файл является частью виртуальной памяти, хранящейся на диске. Подкачка с диска происходит динамически - по мере обращения к соответствующим страницам памяти, причем загружаются только те из них, которые действительно нужны.

Например, если в текстовом редакторе есть модуль работы с таблицами, он не будет загружен с диска до тех пор, пока пользователь не захочет создать (или отобразить) свою таблицу. Причем неважно - находится ли этот модуль в динамической библиотеке или в основном файле. Загрузка таких "монстров", как Delphi и Word, как бы "размазывается" во времени и к работе с приложением можно приступать практически сразу же после его запуска. А что произойдет, если файл упаковать? Правильно - он будет должен считаться с диска целиком (!) и затем - опять-таки, целиком - распаковаться в оперативную память.

Стоп! Откуда у нас столько оперативной памяти? Ее явно не хватит и распакованные страницы придется вновь скидывать на диск! Как говорится: за что боролись, на то и напоролись. Причем, если при проецировании неупакованного EXE-файла оперативная память не выделяется, (во всяком случае, до тех пор, пока в ней не возникнет необходимость), то уж распаковщику без памяти никак не обойтись. А поскольку оперативной памяти никогда не бывает в избытке, она может быть выделена лишь за счет других приложений! Отметим также, что в силу конструктивных особенностей железа и архитектуры операционной системы, операция записи на диск заметно медленнее операции чтения.

Важно понять: Windows никогда не сбрасывает на диск не модифицированные страницы проецируемого файла. Зачем ей это? Ведь в любой момент их можно вновь считать из оригинального файла. Но при распаковке модифицируются все страницы файла! Значит, система будет вынуждена "гонять" их между диском и памятью, что существенно снизит общую производительность всех приложений в целом.

Еще большие накладные расходы влечет за собой сжатие динамических библиотек. Для экономии памяти страницы, занятые динамической библиотекой совместно используются всеми процессами, загрузившими эту DLL (об этом - в следующем мифе). Но как только один из процессов пытается что-то

записать в память, занятую DLL, система мгновенно создает копию модифицируемой страницы и предоставляет ее в "монопольное" распоряжение процесса-писателя. Поскольку распаковка динамической библиотеки происходит в контексте процесса, загрузившего ее, система вынуждена многократно дублировать все страницы памяти, выделенные библиотеке, фактически предоставляя каждому процессору свой собственный экземпляр DLL. Предположим, одна DLL размером в мегабайт, была загружена десятью процессами - посчитайте: сколько памяти напрасно потеряется, если она сжата!

Таким образом, под Windows сжимать исполняемые файлы нецелесообразно - вы платите гораздо больше, чем выручаете.

Статус мифа: **busted**.

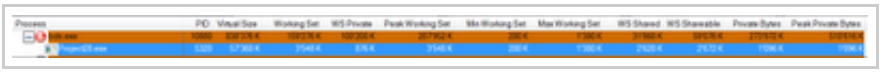
Миф №11: Delphi приложение занимает много памяти

Я уже высказывался на эту тему и даже говорил о типичных ошибках при поиске утечек памяти.

Самое время взяться за эту легенду!

Я взял последнюю версию Delphi на сегодня - Delphi XE (ведь известно, чем старше версия Delphi, тем больший размер она имеет) и создал в ней два пустых приложения - VCL Forms и консольное. Запускаем и видим - VCL Forms:

И консольное:



Вы только посмотрите на эти числа: около 80 и 57 мегабайт! И это - пустые приложения. Просто ужасно.

Кажется, что легенда подтверждена, но так ли это? Давайте посмотрим внимательнее.

Напомню, что это - **виртуальная** память. Никого не волнует, сколько её вы захапаете. Для экосистемы приложений важно, сколько оперативной (физической) памяти вы занимаете. Почему? Ну, чем плохо, что вы тратите много ресурсов? В абстрактном вакууме - ничем. Если в системе есть ресурсы -

тратьте их как угодно. Но на практике машину не покупают для запуска исключительно вашей программы, поэтому здесь важно, что на машине работают и **другие программы**. Вот почему важно, чтобы вы оставляли как можно больше совместных ресурсов свободными - чтобы ими могло воспользоваться больше программ. Но ведь виртуальное адресное пространство не является общим ресурсом! Оно своё у каждой программы. А общий ресурс - это процессор и физическая память. Вот они делятся между всеми программами, в отличие от виртуального адресного пространства, которое выделяется каждой программе в эксклюзивное пользование.

Мне кажется, что этот миф происходит из неосознавания этой связи.

Но легенда ещё не разрушена - что там у нас с физической памятью?

Каждой программе при её работе выделяется физическая память. Программа не может работать с памятью, выгруженной на диск. Если программа выполняет код, находящийся в файле подкачки, либо обращается к данным, выгруженным на диск, то система автоматически загрузит эти код и данные в оперативную память. Если там есть свободное место. Если его там нет - то часть старых данных (к которым давно не было обращения или к ним обращаются редко) будет выгружена на диск, чтобы освободить место для новых данных. Оперативная память тратится и на такие вещи как дисковый кэш.

Итак, что там с оперативной памятью в нашей программе? Если вы посмотрите на снимки экрана выше, то получите два числа: `10'072` Кб для VCL Forms и `3'548` Кб для консольного (колонка "Working Set Size", это значение также называется "песочницей" программы и показывается Диспетчером Задач в колонке "Память"). Кажется, что это огромные значения - в несколько раз больше размера .exe файлов (который равен `894` Кб для VCL Forms и `22` Кб для консольного).

Кажется, что теперь легенда подтверждена? Но не будем спешить с выводами.

Как мы узнали выше, память в адресном пространстве, хотя и выделяется с кратностью в `64` Кб, но минимальным блоком (для прочих операций с памятью) является **страница памяти** (она имеет размер в `4` Кб на текущих редакциях Windows). Кроме того, мы узнали, что одна и та же страница памяти может присутствовать (быть ассоциирована) с несколькими программами (адресными пространствами). Вспомните

проецируемые в память файлы.

К чему я это говорю?

Когда вы запускаете одну и ту же программу второй раз, система просто открывает другое проецируемое в память представление объекта "проекция файла", идентифицирующего образ исполняемого файла. С помощью проецируемых в память файлов несколько одновременно выполняемых экземпляров программы могут совместно использовать один и тот же код, загруженный в оперативную память. Т.е. система просто-напросто проецирует страницы виртуальной памяти, содержащие код и данные .exe файла, второй программы на адресное пространство первого экземпляра программы.

Если один экземпляр программы модифицирует какие-либо данные, размещенные на общей (разделяемой)странице данных, система перехватывает эту попытку, выделяет новый блок памяти, копирует в него нужную программе страницу и после этого разрешает запись в новый блок памяти. Благодаря этому механизму (называемому copy-on-write - копирование при записи), работа остальных экземпляров программы не нарушается. Аналогичная цепочка событий происходит и при отладке приложения. Например, запустив несколько экземпляров программы, вы хотите отладить только один из них. Вызвав отладчик, вы ставите в строке исходного кода точку прерывания. Отладчик модифицирует ваш код, заменяя одну из команд на языке ассемблера другой — заставляющей активизировать сам отладчик. И снова система использует копирование при записи. Обнаружив попытку отладчика изменить код, она выделяет новый блок памяти, копирует туда нужную страницу и позволяет отладчику модифицировать код на этой копии.

Иными словами, то, что вашей программе выделено `10'072` Кб оперативной памяти, - ещё не означает, что это "её вина". Т.е. эти `10'072` Кб - не лично ваша собственность, они совместно используются ещё и другими программами. Можно ли узнать, сколько в этих 10 Мб ваших данных? Да, можно. Это значения в колонке "WS Private" (private working set). Для VCL Forms мы получаем `1'604` Кб, а для консольного - `876` Кб. Это и есть те реальные значения, на которые ваша программа загружает систему. Ради сравнения - эти же программы на Delphi 3 дают `692` Кб и `332` Кб соответственно. Достаточно мало и намного меньше тех значений, о которых обычно думает тот, кто кричит: "ай как много занимает памяти Delphi приложение". И это в системе, где куча свободной ОЗУ и нет давления на память - т.е. это почти максимум. В условиях давления на память эти значения были бы

ещё ниже. Посмотрите, как в мифе №1 размер потребляемой Total Commander-ом оперативной памяти снизился с 1'080 Кб до 136 Кб в условиях нехватки памяти (выделения 2x512 Мб на системе с 256 Мб ОЗУ). И заметьте, что даже при выделении 1 Гб памяти, песочница вашей программы осталась очень компактной - менее 2 Мб: потому что к этой памяти мы не обращались. Мы её только выделили.

Статус мифа: **busted**.

Миф №12: доступ к невыделенной памяти приводит к возбуждению Access Violation

Гм, разве каждый ребёнок не знает про то, что прежде чем использовать память, её надо выделить? Попытка доступа к невыделенной памяти неизменно закончится ошибкой доступа к памяти. Звучит разумно и миф кажется правдоподобным. Но давайте посмотрим, так ли это на самом деле:

```
1  program Project1;
2
3  {$APPTYPE CONSOLE}
4
5  uses
6    Windows;
7
8  var
9    P: Pointer;
10 begin
11   P := VirtualAlloc(nil, 1024, MEM_RESERVE or MEM_COMM
12   FillChar(P^, 2 * 1024, 0);
13   ReadLn;
14 end.
```

Чтобы исключить влияние менеджера памяти Delphi, мы выделяем память не через GetMem / AllocMem, а прося её напрямую у системы - через VirtualAlloc. Суть примера в том, что мы выделяем 1 Кб памяти (1024 байт), а потом записываем в них 2 Кб. Казалось бы, это должно привести к возбуждению Access Violation, но при запуске программы мы обнаруживаем, что она успешно выполняется до конца.

В чём же дело? Как мы помним, выделение памяти происходит с гранулярностью в 64 Кб, а размер выделяемых блоков кратен размеру страницы - т.е. 4 Кб. Да, это странное поведение (почему бы не выделять память с гранулярностью в 4 Кб?), но у него есть причины. Но это означает, что если вы просите у системы 1 Кб, то будет выделено все 4 Кб, а 60 Кб, следующие за этой страницей, останутся неиспользуемыми (ведь следующий блок памяти

может начинаться лишь на границе `+64` Кб от текущего).

Вот и причина для успешного выполнения этого кода - на самом деле код программы выделяет не `1` Кб, а `4` Кб. Это легко можно подтвердить, если заменить множитель `2` в `FillChar` на `5`: `5` Кб больше `4` Кб, поэтому теперь программа вылетит.

Несмотря на это, миф нельзя назвать полностью разрушенным, ведь технически эта память выделена - даже хотя вы логически этого не просили. Поэтому я бы назвал его "правдоподобным".

Статус мифа: **plausible**.

Миф №13: освобождение памяти уменьшает показатели использования памяти программы

Многие ожидают, что освобождая память, вы возвращаете её системе. И снова это выглядит логично, но что будет на практике?

Создадим пустое приложение с двумя кнопками и Edit-ом: первая кнопка будет выделять память, указанную в Edit-е, а вторая - её освобождать:

```
1  procedure TForm1.Button1Click(Sender: TObject);
2  begin
3    Tag := Integer(AllocMem(StrToInt(Edit1.Text) * 1024)
4  end;
5
6  procedure TForm21.Button2Click(Sender: TObject);
7  begin
8    FreeMem(Pointer(Tag));
9  end;
```

Запустим программу и попробуем щёлкать на кнопках со значением `10240` (`10` Мб).

Ну, при выделении памяти потребление виртуальной памяти приложением подскакивает на `+10` Мб, а при освобождении - уменьшается.

Миф подтверждён? Мы так легко не сдаёмся: попробуйте повторить этот же эксперимент, указывая значения вроде `1` или `4`. Теперь вы можете заметить, что при освобождении памяти, занятая виртуальная память не изменяется. Более того, если вам достаточно повезёт, то вы увидите, что при выделении памяти, потребление виртуальной памяти не увеличивается!

(примечание: это плавающее поведение; возможно, вам придётся поэкспериментировать с выделением/освобождением памяти, прежде чем вы его воспроизведёте)

Неужели мы открыли неизвестный науке принцип возникновения памяти из ничего? Вовсе нет - вспомните, что работа с памятью в Delphi (да и других языках тоже) идёт через менеджер памяти - это прослойка между вами и системой, которая, грубо говоря, упаковывает ваши запросы на память в один пакет. В предыдущем мифе мы уже увидели, что при выделении всего 1 Кб памяти на деле расходуется в несколько раз больше памяти - из-за её гранулярности. Чтобы память не пропадала зря, менеджер памяти располагает в одном блоке памяти сразу несколько ваших запросов на память - вот почему потребление памяти может не изменяться при выделении/освобождении памяти: потому что память будет "выделена" в уже существующем блоке памяти, либо же при освобождении памяти менеджер памяти не сможет освободить блок памяти, потому что там есть и другие занятые регионы (либо он может просто придержать свободный блок, на случай, если вы сейчас захотите заново выделить память).

Заметьте, что это не является какой-то "плохой" вещью, как вам может показаться. Мы уже разрушили такой миф: вспомните, что потребление оперативной памяти программы крайне слабо связано с выделением памяти в ней. Вы можете выделить 1 Гб памяти, но в оперативной памяти система даст вам всего 2 Мб. Так и с этой, временно не используемой памятью: она никак не мешается и лежит в файле подкачки, пока вам не понадобится или пока её не освободят.

Конечно же, сказанное не означает утечку памяти - вся эта память будет в итоге возвращена системе. Либо с течением времени, либо при последующих освобождениях памяти. Так что в целом, на длительном участке, миф выполняется: освобождение памяти в вашем коде уменьшает потребление памяти программой, но это может быть не так локально, на достаточно малых участках кода.

Статус мифа: **plausible**.

Примечание: предыдущие два мифа приводят к неизбежному заключению: вы не можете судить о том, валиден (т.е. допустим, корректен) ли данный вам указатель (т.е. была ли под него выделена память) или нет. Более того, это нельзя сделать даже при отсутствии факторов, про которые говорится в мифах:

```
1 P := AllocMem(1024);
2 FreeMem(P);
3 N := AllocMem(1024);
```

Валиден ли `P`? Логически - нет, т.к. у нас есть явное освобождение памяти. Но любая проверка покажет вам, что он допустим: потому что память под `N` выделится ровно на том же самом месте, где были данные от `P`. Таким образом, `P` и `N` будут указывать на одно и то же место в памяти, даже хотя `P` более не считается допустимым по этому коду. Поскольку вы не можете гарантировать неосуществимость этой операции на практике (за исключением специальных отладочных сборок, конечно же), то вы и не можете судить о допустимости указателя, основываясь на любых его проверках: какую-бы проверку вы ни предложили бы, всегда найдётся ситуация, когда проверка будет давать ложно-положительный результат.

Поэтому, когда вы освобождаете память, всегда присваивайте указателю `nil`: тогда его проверка на допустимость будет тривиальной `if Assigned(P) then`.

Миф №14: `Obj.Free` не приводит к `Obj = nil`

Если вы работали с объектами, то знаете, что одним из способов удалить объект - это вызывать метод `Free`, который проверит ссылку объекта и вызовет его деструктор. Правда ли, что после этого ссылка объекта не изменяется и продолжает указывать на бывший объект?

Это очень легко проверить:

```
1  procedure TForm1.Button1Click(Sender: TObject);
2  var
3    O: TObject;
4  begin
5    O := TObject.Create;
6    O.Free;
7    if Assigned(O) then
8      ShowMessage('O <> nil');
9  end;
```

Запустите её - и вы получите сообщение.

Почему так происходит? Ну, об этом можно догадаться. `Free` - это метод объекта. Да, в него передаётся указатель на объект, как и в любой другой метод (далее, в методе, этот указатель становится `Self`) - но передаётся **по значению**. Иными словами, `Self := nil` внутри `Free` не изменит `O` - ведь любые изменения в параметре, переданном по значению, не влияют на исходное значение параметра. `Free` не может изменить `O` даже теоретически.

Если бы это было не так, то вызов конструктора для создания

объекта мог бы быть таким:

```
1 O.Create;
```

Если бы изменения в `Self` влияли бы на исходное значение, то подобный вызов мог бы создать объект и записать ссылку в `O`. Но вместо этого мы пишем:

```
1 O := TObject.Create;
```

Что означает создание объекта и запись ссылки в переменную.

Так же и с освобождением объекта: если вы хотите об-nil-ить ссылку - передавайте её по ссылке (в `FreeAndNil`):

```
1 FreeAndNil(O);
```

`FreeAndNil` освободит объект и присвоит `O` в `nil`. Я уже упоминал, что `FreeAndNil` является самым правильным вариантом освобождения объекта из трёх (вызов деструктора `Destroy`, вызов прослойки `Free` и вызов `FreeAndNil`).

Статус мифа: **confirmed**.

Миф №15: если программа не освободит память, то в системе останется мусор и она замедлится

За эти 14 мифов мы уже столько раз запускали тестовые программы на выделение огромных количеств памяти (некоторые - даже больше размера установленной памяти в системе), но после закрытия программ система продолжала работать как ни в чём не бывало, что этот миф разрушен ещё до того, как я его озвучил.

Но особо дотошные могут его проверить самостоятельно: это будет домашнее задание для начинающих разрушителей легенд. Как бы вы его проверили?

Суть в том, что при закрытии программы, все ресурсы, которые были с ней ассоциированы (виртуальная память, физическая память, открытые файлы и т.п.), автоматически удаляются/закрываются/очищаются системой. Конечно, в системе есть глобальные ресурсы/объекты, не являющиеся частью состояния программы (и которые, таким образом, не меняются при закрытии программы), но виртуальная память к ним не относится.

Мне кажется, что миф пришёл из времён, когда адресное

пространство было тождественно физической памяти: его ошибочно перенесли на современные системы, не разбираясь, как они работают.

Статус мифа: **(totally) busted**.

P.S. Я не уверен, насколько удачным получился этот пост, ведь он написан достаточно нестандартно. Возможно, стоило добавить парочку взрывов. Шучу.
P.P.S. Интересно, что много мифов говорят про различные ограничения `32`-х разрядных процессов и не существуют для `64`-х разрядных приложений с их фантастическими `16`-ю эксабайтами адресного пространства.

См. также:

- Серия про ключ `/3GB` от Реймонда Чена
- Серия "Преодолевая ограничения Windows" от Марка Руссиновича

Читать далее: Адресное пространство под микроскопом.

-= АЛЕКСАНДР АЛЕКСЕЕВ =- 0:21 ТЭГИ DELPHI, НАЧИНАЮЩИМ, СТАТЬЯ

26 КОММЕНТАРИЙ(ЕВ):

pda комментирует...

Миф №16: Разрушители легенд за одну передачу разрушают не более четырёх мифов.

Статус мифа: (totally) busted. :)

26 АПРЕЛЯ 2011 Г., 13:15

deadbitch комментирует...

Зачётный текст ;) Спасибо за информацию!

26 АПРЕЛЯ 2011 Г., 14:53

Юрий комментирует...

По поводу Миф №4 и ключ /3GB

Моё приложение загружает много граф. файлов (В среднем размер одного колеблется 150 ~ 250 Кб).

Что с к ключом /3GB (и {$SetPEFlags IMAGE_FILE_LARGE_ADDRESS_AWARE}) что без него результат одинаков: Out of memory. на 1,9Gb (2`008`453 Kb). Использую D2007, Win32 XP SP3

27 АПРЕЛЯ 2011 Г., 0:16

Анонимный комментирует...

А в мифе №7 о какой ОС идёт речь?

27 АПРЕЛЯ 2011 Г., 9:30

GunSmoker комментирует...

"Мифы" они такие: нечёткие. Поэтому в формулировке нет упоминания о точных деталях (разрядности системы).

В ходе разбирательств выясняется, что миф зависит от разрядности системы: 32-х или 64-х разрядной. Т.е. миф может как выполняться (в 32-х разрядной), так и нет (в 64-х разрядной), но от самого приложения это не зависит.

Поэтому и статус - plausible.

Такая вот логика повествования.

27 АПРЕЛЯ 2011 Г., 9:34

Bogdan комментирует...

Спасибо, было очень интересно и познавательно, всегда в Ваших постах нахожу маленькие открытия для себя.

27 АПРЕЛЯ 2011 Г., 10:13

Анонимный комментирует...

Шикарная подборка мифов и фактов! Огромное спасибо за их систематизацию и доступное разъяснение!

27 АПРЕЛЯ 2011 Г., 11:46

GunSmoker комментирует...

>>> Что с к ключом /3GB (и {$SetPEFlags IMAGE_FILE_LARGE_ADDRESS_AWARE}) что без него результат одинаков: Out of memory. на 1,9Gb (2`008`453

Kb). Использую D2007, Win32 XP SP3

Можно использовать VMMap, чтобы произвести анализ адресного пространства.

(И на будущее: это лучше спрашивать на форумах).

27 АПРЕЛЯ 2011 Г., 12:44

Юрий комментирует...

>Можно использовать VMMap, чтобы произвести анализ адресного пространства.

Александр, большое спасибо, и за отличную статью и за ответ

27 АПРЕЛЯ 2011 Г., 17:04

Анонимный комментирует...

Спасибо, Вас также интересно читать даже после того как я перестал программировать на Delphi

28 АПРЕЛЯ 2011 Г., 16:11

quwy комментирует...

Тема уместности модуля Forms в консольном приложении не раскрыта :)

29 АПРЕЛЯ 2011 Г., 17:32

GunSmoker комментирует...

>>> *Тема уместности модуля Forms в консольном приложении не раскрыта*

Не понял: а это к чему?

29 АПРЕЛЯ 2011 Г., 17:46

quwy комментирует...

Вот об этом: http://www.gunsmoker.ru/2010/12/blog-post.html

Я не то, чтобы не согласен, но модуль Forms в консольном приложениее НЕ НУЖЕН, а та статья прямо говорит: суйте все, юзер на пару со своим компом стерпит.

А вспомнилось это от мифов о размере занимаемой памяти. Просто так.

30 АПРЕЛЯ 2011 Г., 2:18

GunSmoker комментирует...

Гм, а я разве говорю, что его надо туда всенепременно пихать? Зачем он там?

30 АПРЕЛЯ 2011 Г., 7:34

Анонимный комментирует...

Очепятка: "Конечно же, вы не сможете спроецировать *сразу* всю эту память *сразу*, но вы можете делать это по частям."
Слово "сразу" написано дважды.

4 МАЯ 2011 Г., 9:57

FrosT комментирует...

Отличная статья, дала много пищи для размышлений!

>Но наша работа на этом ещё не закончена. Когда же размер файла имеет значение?
Ответ: при упаковке или шифровании.

+ при наличии антивиря. Запуск портабельного офиса 2007 (exe размером в 200 мегов) дает достаточно времени для вдумчивой медитации.

31 АВГУСТА 2011 Г., 23:16

GunSmoker комментирует...

>>> *+ при наличии антивиря. Запуск портабельного офиса 2007 (exe размером в 200 мегов) дает достаточно времени для вдумчивой медитации.*

Тогда ещё дополню, что ещё и при наличии цифровой подписи: перед запуском система обязана проверить неизменность файла, что приводит к его полному считыванию.

И ещё один вариант: в заголовок PE можно установить флаг "кэширования" - что приведёт к копированию файла в файл подкачки при его запуске с сети или сменного накопителя. Предназначен флаг как раз для портабельных программ, чтобы они могли продолжать работать, когда их носитель отключается при их работе.

Я не упоминал все эти случаи, потому что тут нагрузка разовая: один раз прочитали файл целиком - и забыли. А дальше файл выполняется как обычно. И, кроме того, тут речь про чтение файла, а не потребление памяти - про что идёт речь в посте. Хотя теперь думаю, что стоило бы

упомянуть эти случаи для полноты картины - ну, вот теперь будут в комментариях.

А сжатие существенно хуже: мы не только дали разовый лаг на запуск, так ещё и заблокировали разделение памяти. Т.е. 10 запущенных программ - в 10 раз больше потребление памяти. Как говорится: за что боролись - на то и напоролись.

31 АВГУСТА 2011 Г., 23:33

Анонимный комментирует...

На многих картинках ничего не видно.

12 СЕНТЯБРЯ 2011 Г., 16:14

GunSmoker комментирует...

Картинки кликабельны.

12 СЕНТЯБРЯ 2011 Г., 21:14

Анонимный комментирует...

Привет я не могу включить 3-х разрядные проги в винде 7 64-х разрядном, может что то посоветуешь?

25 НОЯБРЯ 2011 Г., 14:15

Анонимный комментирует...

Здорово было бы в шапку добавить оглавление по всем мифам

31 ЯНВАРЯ 2012 Г., 15:10

Александр Алексеев комментирует...

Добавил.

1 ФЕВРАЛЯ 2012 Г., 0:05

Анонимный комментирует...

Отличная статья! Спасибо!

14 МАРТА 2012 Г., 13:35

Анонимный комментирует...

Хочу добавить по поводу пункта 10. Если взять программу размером 100Мб и подписать ее цифровой подписью - запускаться будет долго, т.к. сначала винда весь файл прочитает и проверит что цифровая подпись действительна, потом антивирус сделает то же самое. А если файл запускается с DVD - то сразу смерть. Я

тестировал, exe файл размером 1Гб, на машине с касперским запускается 15 минут. Поэтому пихать 100 меговые файлы в ресурсы не лучшая затея.

14 АПРЕЛЯ 2012 Г., 10:07

Анонимный комментирует...

Вы можете ПО РУССКИ(читай-литературный,а не технический язык)обьяснить,откуда выделяется 2-4 гб памяти 1-му процессу?

22 ФЕВРАЛЯ 2013 Г., 14:58

Александр Алексеев комментирует...

Гб? Да ещё и 4? Кроме вашего кода никто столько выделять не будет.

22 ФЕВРАЛЯ 2013 Г., 22:17

ОТПРАВИТЬ КОММЕНТАРИЙ

Можно использовать некоторые HTML-теги, например:

Жирный
Курсив
Ссылка

Вам необязательно регистрироваться для комментирования - для этого просто выберите из списка "Анонимный" (для анонимного комментария) или "Имя/URL" (для указания вашего имени и (опционально) ссылки на сайт). Все прочие варианты потребуют от вас входа в вашу учётку (поддерживается OpenID).

Пожалуйста, по возможности **используйте "Имя/URL" вместо "Анонимный"**. URL можно просто не указывать.

Ваше сообщение может быть помечено как спам спам-фильтром - не волнуйтесь, оно появится после проверки администратором.

Введите комментарий...

Подпись комментария: Andrey Alisov ▼

Публикация Просмотр

ССЫЛКИ

Создать ссылку

Следующее Главная страница Предыдущее

NEVER SEND A HUMAN TO DO A MACHINE'S JOB