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)

СЛУЧАЙНЫЕ ПОСТЫ

Terminator Salvation
Отдельный блог для переводов
Эээ... сезон слухов Delphi 2009?
Создаём систему плагинов, часть 9

19 АПРЕЛЯ 2011 Г.

Архитектура памяти в Windows

В прошлый раз мы рассмотрели вопросы использования глобальных переменных. Прежде, чем двинуться дальше, нужно дать небольшой вводный курс по памяти в Windows.

Любая вещь в вашей программе занимает "память компьютера". Это может быть строка, число, открытый файл, запись, объект, форма и даже сам код. Даже хотя вы явно никого не просили выделять память, она всё равно выделена автоматически - либо компилятором, либо операционной системой.

Адресное пространство и все, все, все...

Кратко говоря, память программы может рассматриваться как один очень-очень длинный ряд байтов. Байт - это единица измерения количества информации, в мире Delphi и Windows он равен восьми битам и может хранить одно из `256` различных значений (от `0` до `255`). На память можно смотреть как на массив байт. Что именно содержат эти байты - зависит от того, как интерпретировать их содержимое, т.е. от того, как их используют. Значение 97 может означать число `97`, или же ANSI букву `'a'`. Если вы рассматриваете вместе несколько байт, то вы можете хранить и большие значения. Например, в 2-х байтах вы можете хранить одно из `256*256 = 65536` различных значений, две ANSI буквы `'ab'` или Unicode букву `'a'` - и т.д.

Чтобы обратиться к конкретному байту в памяти (адресовать его), можно присвоить каждому байту номер, пронумеровав их целыми положительными числами, включив ноль за начало отсчёта. Индекс байта в этом огромном массиве и называется его **адресом**, а весь массив целиком - **памятью программы**. Диапазон адресов от `0` до максимума называется **адресным пространством** программы. А максимум (длина) массива называется **размером адресного пространства**.

(примечание: ну, на самом деле, есть тысяча и один способ

адресовать память, но в рамках современного мира и этой статьи мы ограничимся только этим способом).

Адресное пространство (вернее, его размер) определяет способность программы работать с данными. Чем оно больше - тем с большим количеством данных программа сможет работать (в один момент времени). Если у программы заканчивается свободное место в адресном пространстве (т.е. все адреса в нём выделены под какие-то объекты в программе) - то у программы заканчивается память (out of memory).

Как адресное пространство соотносится с вашим исходным кодом

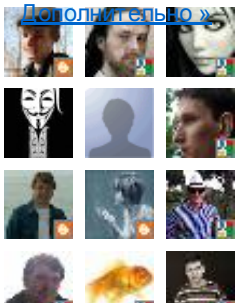
С точки зрения языка высокого уровня (Паскаль) все вещи в вашей программе характеризуются **именем** (идентификатором), **типом** ("сколько памяти выделять") и **семантикой** ("что с этим можно делать"). Например, целое число занимает `4` байта и их можно читать, писать, складывать и т.п. И число A - это не то же самое, что число B. Строки же занимают переменный объём памяти, их, к примеру, можно соединять и редактировать. И так далее.

Но на уровне машинного языка, железа и операционной системы все они характеризуются только **местоположением**, **размером** (в байтах) и **атрибутами доступа**. Местоположение - это адрес объекта. К примеру, число A может иметь адрес `1234`, а число B - `1238`. И поэтому это два разных числа - потому что у них разный адрес, т.е. они лежат в разных местах. Атрибут доступа является упрощённой "семантикой", которая определяет то, что можно делать с памятью. А таких вещей всего три: читать, писать и выполнять. Последнее означает исполнение машинного кода. Тут нужно пояснить, что ваши данные (числа, строки, формы и т.п.) находятся в одном "контейнере" (том самом "массиве памяти из байт") вместе с кодом программы - .exe файлом. Иными словами, код рассматривается наравне с данными, а чтобы их отличать и служат атрибуты доступа.

Можно увидеть, как понятия языка высокого уровня ("имя", "тип" и "семантика") проецируются в понятия низкого уровня ("адрес", "размер" и "атрибуты доступа").

Древний мир

В давние времена память программы была тождественно равна



оперативной памяти машины (т.н. ОЗУ или RAM - Random Access Memory). Иными словами, размер адресного пространства программы был равен размеру установленной оперативной памяти. Вот, установлено на вашей машине две планки памяти по 64 Кб - значит, у вашей программы есть 128 Кб памяти. Ну, за вычетом той памяти, что уже занята, конечно же. Адрес объекта программы был равен адресу физической ячейке оперативной памяти (физическому адресу). И если у вас заканчивалось место в ОЗУ, то у вас заканчивалась память в программе.

Конечно, такой способ хотя и весьма прост, имеет две проблемы:

1. Память программы ограничена оперативной памятью. А раньше эта память была дорогой и её было очень мало.
2. Если нужно запустить две программы, то они будут работать "в одной песочнице": и первая и вторая программа будут размещать свои данные в одном месте - оперативной памяти. И если первая программа по ошибке запишет что-то в данные второй, то... ой.

Виртуальная память и виртуальное адресное пространство

Поэтому в современном мире используется совершенно другая схема: во-первых, память программы теперь больше не тождественна оперативной памяти. Теперь программа работает исключительно с так называемой "виртуальной памятью". Виртуальная память - это имитация реальной памяти. Она позволяет каждой программе:

1. считать, что установлено максимальное теоретически возможное количество оперативной памяти;
2. считать, что она является единственной программой, запущенной на машине.

Иными словами, адресное пространство программы более не ограничено размером **физической памяти** (так называют оперативную память компьютера, чтобы специально указать на её отличие от виртуальной памяти) - адресное пространство имеет теперь **максимально возможный размер**. К примеру, если для адресации используются 32-битные указатели (4 байта), то размер адресного пространства равен 2^32 = 4'294'967'296 байт. Т.е. 4 миллиарда (если угодно: биллионов) или 4 Гб. А **размерность** адресного пространства - равна 32.

В связи с новомодным "переходом на 64 бита" нужно упомянуть, что этот переход заключается в замене 4-байтных (32-битных)

ТЭГИ

7 (4)

указателей на 8-байтные (64-битные) - что увеличивает размер адресного пространства программы аж до `2^64 = 18'446'744'073'709'551'616` байт. Т.е. `18` с лишним квинтиллионов байт или `16` Эб (эксабайт) для краткости. Соответственно, `32`-битный указатель может быть любым числом от `0` до `4'294'967'296` (от `$00000000` до `$FFFFFFFF`). `64`-разрядный указатель может варьироваться от `$00000000'00000000` до `$FFFFFFFF'FFFFFFFF`.

А из второго пункта следует, что `4` Гб или `16` Эб есть **у каждой программы**. Т.е. каждой программе отводится своё личное закрытое адресное пространство. Такая изолированность означает, что программа А в своем адресном пространстве может хранить какую-то запись данных по адресу `$12345678`, и одновременно у программы В по тому же адресу `$12345678` (но уже в его адресном пространстве) может находиться совершенно иная запись данных. Если программа А попробует прочитать данные по адресу `$12345678`, то она получит доступ к своей записи (записи программы А), а не данным программы В. Но если к адресу `$12345678` обратится программа В, то она получит свою запись, а не запись программы А. Иными словами, программа А не может обратиться в памяти (адресному пространству) программы В и наоборот.

Таким образом, при использовании виртуальной памяти упрощается программирование, так как программисту больше не нужно учитывать ограниченность памяти, или согласовывать использование памяти с другими приложениями. Для программы выглядит доступным и непрерывным всё допустимое адресное пространство, вне зависимости от наличия в компьютере соответствующего объема ОЗУ. Если программы выделяют в их адресных пространствах больше памяти, чем есть в системе физической памяти, то часть памяти из ОЗУ переносится на диск ("винчестер") - в т.н. файл подкачки (его ещё называют страничным файлом, page file, SWAP-файлом или "свопом"). Когда программа обращается к своим данным, которые были выгружены на диск, то операционная система автоматически загрузит данные из файла подкачки в ОЗУ. И всё это происходит под капотом - т.е. совершенно незаметно для программы. С точки зрения программы, ей кажется, что она работает с `4` Гб или `16` Эб RAM.

Применение механизма виртуальной памяти позволяет:

- упростить адресацию памяти программами;
- рационально управлять оперативной памятью компьютера (хранить в ней только активно используемые области памяти);

- изолировать программы друг от друга (программа полагает, что монопольно владеет всей памятью).

А теперь, пока вы не перевозбудились от колоссального объема адресного пространства, предоставляемого вашей программе: вспомните, что оно — виртуальное, а не физическое. Другими словами, (виртуальное) адресное пространство — всего лишь диапазон адресов памяти. Конечно, нехватка памяти теперь не происходит, когда заканчивается свободное место в оперативной памяти. И на машине с 256 Мб ОЗУ, любая программа может выделить, скажем, один кусок в 512 Мб памяти. Конечно же, это не означает, что вы можете выделить аж 16 эксабайт - ведь реальный размер ограничен размером диска. И не факт, что в системе будет диск на 16 эксабайт. Тем не менее, это значительно лучше, чем просто 256 Мб оперативной памяти, установленные на вашем "старичке".

(примечание: по непонятной мне причине, некоторые люди *не верят* в тот простой факт, что программа может спокойно выделить больше памяти, чем установлено физической памяти в системе; звучит как сюжет для разрушителей легенд (MythBusters)).

Чем чаще системе приходится копировать данные из оперативной памяти в файл подкачки и наоборот, тем больше нагрузка на жесткий диск и тем медленнее работает операционная система (при этом может получиться так, что операционная система будет тратить всё свое время на подкачку памяти, вместо выполнения программ). Поэтому, добавив компьютеру оперативной памяти, вы снизите частоту обращения к жёсткому диску и, тем самым, увеличите общую производительность системы. Кстати, во многих случаях увеличение оперативной памяти дает больший выигрыш в производительности, чем замена старого процессора на новый. А с падением цен на память уже не проблема собрать систему с 16 или 32 Гб оперативной памяти по доступной цене.

Факты о виртуальном адресном пространстве

Хотя в самом начале мы рассматривали память программы (адресное пространство) как один непрерывный однородный блок, сейчас настало время сделать уточнение, что я вам наврал: таковым он не является. Адресное пространство, хотя действительно однородно и непрерывно более чем на 99%, но в нём есть несколько специальных областей. Я не буду подробно разбирать их все, скажу только о самых важных.

Во-первых, это область для отлова нулевых указателей. Это, определённо, самая важная специальная часть адресного пространства. Начинается она в нуле и заканчивается на адресе `65'535`. Т.е. имеет размер в `64` Кб и расположена в диапазоне `$00000000-$0000FFFF` - самом начале адресного пространства. Специальна эта область тем, что она всегда заблокирована: в ней нельзя выделить память, а любое обращение по этим адресам всегда возбуждает исключение access violation (примечание: это не единственная причина возбуждения access violation). Эта область сделана исключительно для нашего удобства. Как вы узнаете потом (или уже знаете), нулевой указатель `nil` по числовому значению равен `0`. И если вы случайно (по ошибке) обратитесь к нулевому указателю - то эта область поможет вам возбудить исключение и поймать вашу ошибку.

А что такого особенного в числе `65'535`? Ну, `64` Кб - это **гранулярность** выделения памяти. Гранулярность выделения памяти определяет, блоками каких размеров вы можете оперировать при выделении и освобождении памяти. Т.е. гранулярность выделения памяти в `64` Кб означает, что вы можете выделять только блоки памяти, размер которых кратен `64` Кб. Зачем так делается? Ну, если вы попробуете вести учёт "выделенности" каждого байта в программе, то размер управляющих структур у вас превысит размер самих данных. Поэтому память выделяют "кластерами". Иными словами, если вы хотите расположить область в начале адресного пространства, то вы не можете выделить меньше, чем `64` Кб. А больше? Больше - можно. Например, `64 + 64 = 128` Кб. Но большого смысла в этом нет.

Почему гранулярность выделения памяти равна именно `64` Кб, а не, скажем, `8` Кб? Ну, на это есть исторические причины.

(примечание: полностью аналогичный блок расположен на границе `2` Гб - но уже по совершенно другим причинам).

Далее, что вам ещё нужно знать про виртуальное адресное пространство - оно доступно вам не полностью. Грубо говоря, в виртуальном адресном пространстве каждой программы сосуществуют сама программа и операционная система. Та часть, где работает ваша программа (и о котором мы говорили всё это время выше), называется разделом для кода и данных пользовательского режима (user mode). Та часть, где работает операционная система, называется разделом для кода и данных режима ядра (kernel mode). Обе эти части находятся в едином адресном пространстве программы.

Чем они отличаются? Про пользовательский раздел мы уже много чего сказали: он свой у каждой программы и это полностью ваш раздел - делайте что хотите. Раздел ядра является здесь особенным в двух моментах: во-первых, у вашей программы нет к нему никакого доступа. Вообще и в принципе это невозможно. Там орудует только операционная система, но не вы. Если вы попробуете обратиться к памяти в этом разделе, то получите просто access violation. Во-вторых, особенность раздела в том, что он разделяется между всеми программами. Да, вот так: пользовательская часть у каждого адресного пространства своя, но часть ядра - одна и та же, общая. По сути, раздел ядра является "адресным пространством режима ядра".

Какой размер имеют эти две части адресного пространства? Ну, если мы говорим про 32-разрядную программу, то пользовательский раздел занимает от 2 до 4 Гб (по умолчанию - 2 Гб). Соответственно, режим ядра занимает от 0 до 2 Гб (ибо суммарно должно быть 4 Гб). Конечно же, это за вычетом уже упоминаемых специальных областей. Итого: по умолчанию адресное пространство 32-разрядной программы делится пополам. Половина - вам, и половина - операционной системе.

(примечание: 0 Гб под режим ядра - это специальный особый случай, достижимый только при запуске 32-битной программы на 64-битной машине. В обычных условиях граница между разделами может двигаться от 2 до 3 Гб).

Если говорить совсем точно, то раздел для ваших данных в случае 32-х бит имеет диапазон $0000FFFF-$7FFEFFFF (или $BFFFFFFF в максимуме на 3 Гб, с дыркой на 64 Кб в районе 2 Гб), а раздел режима ядра - $80000000-$FFFFFFFF (или $C0000000-$FFFFFFFF в максимуме для user mode). В случае 64-разрядной программы ситуация будет несколько иная. На сегодняшний день в Windows соотношение выглядит так: user mode - $00000000'00010000-$000003FF'FFFEFFFF (8 Тб); kernel mode - $00000400'00000000-$FFFFFFFF'FFFFFFFF. Ну, это всё ещё недостаточно точно, ведь, на самом деле, режим ядра в случае 64-х бит использует только максимум несколько сотен Гб, оставляя большую часть адресного пространства попросту неиспользуемой. Т.е. у нас в дополнение к двум областям (user mode и kernel mode) появляется ещё и третья: зарезервированная область. Которую, впрочем, со стороны user mode удобно считать частью kernel mode. Сделано это по той простой причине, что 64-битное адресное пространство настолько огромно, что user mode и kernel mode выглядели бы в нём тонюсенькими полосочками, вздумай бы вы изобразить их графически и в масштабе. А если место просто зарезервировано, то и не нужно делать для него управляющих данных. Даже 8 Тб

памяти для user mode - это **очень** много. Если бы вы выделяли мегабайт памяти в секунду, у вас бы ушло три месяца, чтобы исчерпать такое адресное пространство.

Это что касается изолированности одной программы от других и от операционной системы. Внутри программы её модули (exe, DLL, bpl) друг от друга, вообще говоря, никак не изолированы. Однако на практике граница всё же появляется, но связана она с языковыми различиями и особенностью управления памятью в разных языках программирования. Но это разговор для другого раза.

Если вы забудете всё то, что я тут говорил, то вот факт, который вы должны вынести из этого обсуждения: размер памяти программы ограничен 2 Гб (32-битная программа) или 8 Тб (64-битная программа), либо суммарным размером оперативной памяти и файлом подкачки - смотря что меньше, а что больше. Т.е. на практике вы получаете "out of memory" только когда превышаете размер в 2 Гб.

Операции, производимые с виртуальной памятью

Ну, вполне очевидно, что прежде чем использовать память, вы должны её **выделить** (commit), а после окончания работы - **освободить** (release). Конечно, вам не обязательно делать это в явном виде - как я уже сказал, часто за вас это делает кто-то другой автоматически. Но об этом в следующий раз.

Помимо двух операций (выделения и освобождения памяти) существует и третья операция - **резервирование** (reserve) памяти. Смысл её заключается в том, что под зарезервированную виртуальную память не выделяется никакой реальной памяти (будь то оперативная память или файл подкачки), но при этом память считается занятой, как если бы она была выделена. Позднее, вы можете выделить реальную память этому зарезервированному блоку (полностью или частями).

Зачем нужна такая операция? Ну, предположим, вам нужен непрерывный блок памяти для работы (к примеру, чтобы обработать его за один проход одним циклом), но вы выделяете память не сразу а частями - по мере надобности. Вы не можете просто выделить первый блок, а потом - второй: ведь тогда нет гарантии, что они будут идти друг за другом. Вот поэтому и придумали операцию резервирования: вы резервируете достаточно большой регион. Это - "бесплатно". Потом вы выделяете в нём реальную память. Обе цели достигнуты: вы и выделяете память по мере необходимости (а не сразу целиком), и

вы получаете свою непрерывную область памяти.

Кстати, все три операции выполняются функциями `VirtualAlloc` и `VirtualFree`. Не забудьте только, что мы говорили про гранулярность выделения памяти в `64` Кб.

И снова: какое это имеет отношение к Delphi?

Ну, почти самое прямое. Ведь программа на Delphi должна выделять и освобождать память. Это значит, что ей нужно вызывать функции `VirtualAlloc` и `VirtualFree`. А выделять память она будет в своём (виртуальном) адресном пространстве - причём, только в пользовательской его части.

Операции с памятью в Delphi проводятся через функции `GetMem` и `FreeMem`. Конечно же, кроме этих функций в Delphi существует и много других - но они являются лишь обёртками или переходниками к `GetMem` и `FreeMem`. Эти обёртки (например: `AllocMem`, `New`, `Dispose`, `SetLength` и т.п.) предоставляют дополнительную функциональность и удобство (кстати, в системе тоже есть обёртки к вызовам `VirtualAlloc` и `VirtualFree`). В некоторых случаях, эти вызовы и вовсе скрыты и происходят автоматически под капотом языка. Например, при сложении двух строк:

```
1  var
2    S1, S2, S3: String;
3  begin
4    S1 := S2 + S3;
```

вы не видите вызов `GetMem`, но он здесь есть.

Зачем нужны "свои" подпрограммы управления памятью? Почему нельзя просто использовать `VirtualAlloc` и `VirtualFree`? Ну, Delphi тут не уникальна - большинство языков используют т.н. менеджеры памяти - это код, который в Delphi стоит за вызовами `GetMem` и `FreeMem`, который служит переходником к `VirtualAlloc` и `VirtualFree`. А делается это по причине всё той же гранулярности выделения в `64` Кб. Т.е. если вы создаёте `100` объектов по, скажем, `12` байт, то вместо двух килобайт (`12` б `* 100 = 1.2` Кб + служебные данные менеджера памяти) вы занимаете уже почти `6.5` Мб (`64 * 100 = 6'400` Кб) - на несколько порядков больше! Использовали бы вы VirtualAlloc - вы бы очень быстро исчерпали свободную память. Менеджер памяти же "упаковывает" несколько запросов на выделение памяти в один блок.

(примечание: "упаковка" ни в коем случае не означает "сжатие"

или "кодирование" - это просто размещение нескольких маленьких кусочков памяти в одном 64 Кб блоке).

Заметьте, что операции резервирования памяти у Delphi нет, т.к. подобная операции не имеет большого смысла при "упаковке" запросов менеджером памяти. Для работы с резервированием используются функции операционной системы.

Продолжение следует...

Вот и все базовые сведения про устройство памяти в Windows, которые вам нужно знать для начала. В следующий раз мы более близко посмотрим на то, как архитектура памяти соотносится с переменными в ваших программах.

См. также: Архитектура памяти в Windows: мифы и легенды (spin-off).
Читать далее: Адресное пространство под микроскопом.

-= АЛЕКСАНДР АЛЕКСЕЕВ =- 7:54 ТЭГИ DELPHI, НАЧИНАЮЩИМ, СТАТЬЯ

5 КОММЕНТАРИЙ(ЕВ):

Анонимный комментирует...

Очепятка: "ибо суммарно **болжно** быть 4 Гб"

19 АПРЕЛЯ 2011 Г., 12:34

Torbins комментирует...

Ух, мега-пост! С нетерпением жду продолжения :)

19 АПРЕЛЯ 2011 Г., 15:39

Анонимный комментирует...

Гранулярность в 64Кб относится только к резервированию, передовать физическую память можно постранично (4Кб).

26 АПРЕЛЯ 2011 Г., 0:23

Анонимный комментирует...

Автор нагло понадергал абзацев у Рихтера и даже не указал нигде об этом. Книга называется "Windows для профессионалов. Создание эффективных WIN32-приложений с учетом специфики 64-разрядной версии Windows."

28 ИЮНЯ 2011 Г., 21:33

Анонимный комментирует...

Ваша статья о распределении памяти - лучшая в сети, спасибо!

8 НОЯБРЯ 2013 Г., 15:11

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