**Боевые действия в условиях ограниченной видимости**

Даже при наличии 512 мегабайт, расходовать память, не думая о последствиях глупо. Дело в том, что Windows XP в домашней редакции уже съедает от этого объема 128 метров, а профессиональная редакция отнимает все 256. Всякие примочки и побрякушки в районе часов, антивирусы и сетевые экраны могут отнять еще 64 метра. Получается, что для других приложений остается не так уж и много места. Если одновременно будет запущен Delphi 2006, 3DS Max и Photoshop, то работа станет невыносимой, ведь эти монстры сжирают оперативку хуже вирусов.

В такие моменты вспоминаются времена MS-DOS с его ограничениями и советские машины (за которыми мне удалось поработать в начале 90-х), когда приходилось экономить каждый байт памяти. Сейчас байты и тем более биты уже никто не считает, а вот кило и мега считать необходимо в любой программе, поэтому данная статья может пригодиться даже тем, кто не собирается писать программы в тяжелых условиях с ограниченным объемом памяти. Необходимо всегда расходовать предоставленные ресурсы разумно.

**Массивы**

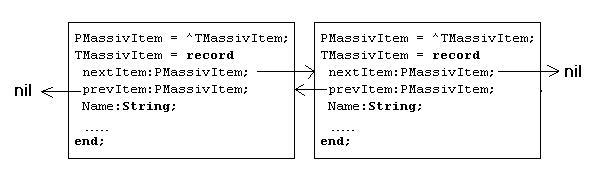
Самый банальный и бессмысленный расход памяти происходит благодаря использованию массивов фиксированной длины. Допустим, что тебе нужно хранить в памяти заранее неизвестное количество чисел. Как поступить в этом случае? Можно зарезервировать максимально возможный буфер и никаких проблем. Да, проблем никаких, а вот утечка памяти произойдет достаточно серьезная.

Допустим, что ты выделил массив из 1024 чисел. Если каждое число типа Integer, то из-под ног утекает 4096 байт памяти. А если тип данных Int64? А если это не число, а структура, размером в 100 байт? Это уже сто килобайт памяти. Десять таких массивов и драгоценный мегабайт уходит в небытие. Простейший вариант решения проблемы – в массиве хранить не сами структуры, а указатели. В этом случае размер массива значительно сокращается, но он все равно не оптимален. Мы резервируем 1024 элемента, а реально в программе может использоваться только один.

Идеальное решение – использование динамических массивов. Не стоит бояться динамики, она безобидна по сравнению с Фредди Крюгером. Можно выделить два основных типа динамических массива:

1. Просто выделяем память для хранения элементов массива, а по мере необходимости память расширяется. Данный вариант самый простой и самый компактный, но очень неудобный, когда необходимо изменять последовательность элементов, например, для сортировки, или удалять элементы из середины массива.

2. Можно написать класс, который будет управлять динамическим массивом. Простейший вариант такого класса я набросал в листинге 1. Данный код упрощен до минимума. На компакт диске ты найдешь более полный вариант с примером использования. Да, такое хранение данных требует некоторой избыточности для каждого элемента, но зато наш массив полностью динамичен и занимает в памяти ровно столько, сколько необходимо. Ничего лишнего резервировать ненужно. А главное, размерность массива ограничена размером типа данных, который используется для хранения количества элементов.

  
*Представление связанного списка в памяти. Стрелками показано, как наведена связь, между элементами*

Как вариант, для хранения динамического массива указателей можно использовать класс TList (в данном случае совет относиться к программистам Delphi). Лично я регулярно использую этот класс, потому что он прост и удобен, но иногда пишу свою реализацию динамики, все зависит от задачи.

На самом деле алгоритмов создания списков очень много, но я выделяю эти два, как наиболее эффективные и достаточно простые в реализации. В зависимости от задачи можно придумать и другой алгоритм, все зависит от того, как хранятся данные, нужно ли их сортировать, есть ли необходимость удалять из середины, куда добавляются новые элементы – в конец или могут вставляться в середину списка и т.д.

**Выделение памяти**

Допустим, что ты пишешь программу воспроизведения/записи звука или видео. На первый взгляд, для воспроизведение mp3 файла необходимо столько же памяти, сколько данные занимают в файле. Достаточно только загрузить все память и отправить данные звуковой карте. Ошибочка вышла. Не факт, что звуковая карта сможет понять сжатые данные. Придется производить декомпрессию самостоятельно и направлять звуковой карте не сжатые данные. Если разложить по полочкам mp3 файл, то для хранения всех данных может понадобиться 60, а то и все 100 мегабайт памяти. Я думаю, за такое расточительство тебя ни один пользователь не простит.

Еще один пример – простой ZIP архиватор. Что если необходимо сжать файл размером в 500 метров? Загрузить его в память, а потом заархивировать его там будет достаточно проблематичным.

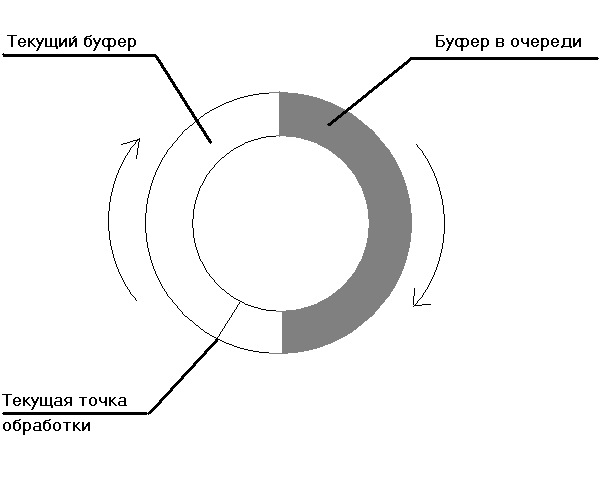
Во всех этих случаях вполне логичным будет выделить два небольших участка памяти. В первый необходимо загружать данные из файла для последующей манипуляции (компрессии/декомпрессии), а во второй будет помещаться результат работы. Тут нужно быть аккуратным, потому что размеры сжатых и не сжатых данных разные и легко выйти за границу буфера.

Не стоит пытаться запихнуть в оперативку все сразу, потому что это бессмысленно. Загружайте всегда данные небольшими порциями.

**Циклическое использование памяти**

При работе со звуком Microsoft уже предоставляет нам очень удобные механизмы работы с памятью, которые мы сейчас рассмотрим. Возможно, данный алгоритм ты будешь использовать в своих программах при решении других задач.

Итак, MMSystem получает при воспроизведении и возвращает при записи звуковые данные через специализированные буферы. Так как данные загружаются порциями и для предотвращения задержек в воспроизведении/записи музыки, используется очередь из буферов. Для нормальной работы достаточно двух блоков памяти, но можно сделать и больше.

  
*Циклическое использование памяти*

Для примера, выделяем память для хранения двух буферов данных. Размер памяти можно выбрать любой, но он должен быть не слишком маленьким, дабы избежать слишком частой смены буферов и не слишком большой, чтобы сэкономить память для других целей. Точный размер подбирается в зависимости от качества звука, потому что от этого зависит и размер необходимых данных. Из личного опыта я всегда выделяю столько, чтобы хватило на воспроизведение/запись не менее 5 и не более 10 секунд звука.

Теперь, говорим звуковой карте, что нужно использовать буфер 1, а буфер 2 помещаем в очередь. Так как MMSystem работает в отдельном потоке, ждем, когда буфер 1 освободиться. Когда это произойдет, система переключиться на следующий буфер в очереди (буфер 2), и пока она будет с ним работать, мы освобождаем буфер 1 или заполняем его новыми данными, и снова помещаем в очередь. Таким образом, с помощью цикла можно по частям обработать данные любого размера с минимальными расходами памяти.

Два буфера в данном случае минимально необходимы, но желательно использовать 3 или 4. Дело в том, что в момент освобождения буфера, система может быть занята и не успеет его очистить, подготовить и вернуть обратно в очередь. В этом случае может произойти заикание в воспроизведении или даже сбой, если идет запись.

**Сортировка**

Когда данных очень много, то просматривать их все очень сложно и проблематично даже для очень мощного процессора, если памяти не достаточно. Дело в том, что процессор – не самое главное в современном компьютере. Слабым местом по-прежнему остается жесткий диск. Я уже много раз говорил о том, что это механика и, не смотря на то, что производители пытаются из нее выжать максимум, она остается очень слабой.

Можно попытаться оптимизировать процесс поиска данных через дефрагментацию, дабы головка жесткого диска не прыгала по блину как угорелая, а читала их последовательно, но затраты в соотношении с результатом слишком большие. Можно читать данные большими кусками, а обработку перенести в отдельный поток, дабы максимально распараллелить задачи, но это тоже не совсем то.

Если данных действительно очень много, а памяти очень мало, в бой вступают мозги и алгоритмы. Мы должны реализовать алгоритм так, чтобы не было необходимости просматривать все данные. И в этом нам помогут – сортировка и индексация. Давай посмотрим, как они могут нам помочь.

Для начала возьмем сортировку. Когда данные упорядочены, с ними проще работать и искать необходимую информацию. Допустим, что у нас есть список всех жителей города и тебе нужно найти номер телефона Иванова. С такой фамилией в целом городе может быть не один человек, а сотня. Просматривать весь список достаточно проблематично, а если он будет упорядочен по фамилии, то достаточно будет просмотреть все записи на букву "И" и готово. Экономия времени и памяти на лицо.

**Индексация**

Поддерживать списки в упорядоченном виде достаточно проблематично, особенно, если записи очень часто изменяются, добавляются, удаляются и при этом каждая из них может иметь произвольный или очень большой размер. В этом случае записи хранят на диске в произвольном виде, а для быстрого поиска используют индексы. Так поступает большинство баз данных, дабы сэкономить память и процессорное время.

Индексы – могут состоять из двух колонок: поле, по которому происходит упорядочивание данных и указатель на место, где находятся данные этой записи. Сами данные хранятся в произвольном порядке, а вот индексная табличка сортируется. Так как она очень маленькая и состоит только из двух полей, такую табличку достаточно легко поддерживать в упорядоченном виде, а для хранения ее в памяти нужно намного меньше места, чем для хранения всего списка.

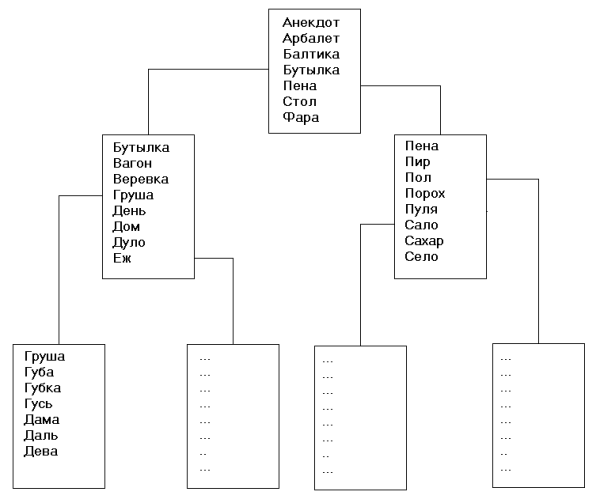
Теперь, если нам снова нужно найти номер телефона по фамилии, мы бежим по индексной таблице и ищем необходимую фамилию. Когда она найдена, переходим по указателю из индексной таблицы и получаем данные человека, в том числе его номер телефона и адрес.

Индексные таблички хороши, но только в меру. Для телефонного справочника, может понадобиться три индекса: для фамилии, телефона и адреса. Благодаря трем индексам мы можем легко упорядочить данные по любому из этих параметров, но содержать в упорядоченном виде придется уже три таблички, а это уже сложнее.

**Деревья**

Индексы – очень мощное средство для экономии памяти и упрощения сортировки. Но для поиска данных все равно приходиться сканировать всю таблицу, пока мы не найдем необходимую запись. Если искомый текст начинается на букву "А", то мы найдем его быстро, ведь он будет находиться в самом начале. Но если искомая строка начинается на "Я", то во время поиска мы придется сканировать до самого конца и экономия скорости будет небольшой. Да, мы все еще экономим память, потому что загружаем в оперативку только индексную таблицу, а не все данные, но скорость все еще не высокая. А если построить индекс в виде дерева, то мы сэкономим и место и время.

Индексы в виде деревьев очень часто используются в базах данных, в том числе и в MS SQL Server. Так что, понимание этого индекса пригодиться не только программистам (для реализации в собственных проектах), но и администраторам, для лучшего понимания внутренностей баз данных.

  
*Пример древовидного индекса*

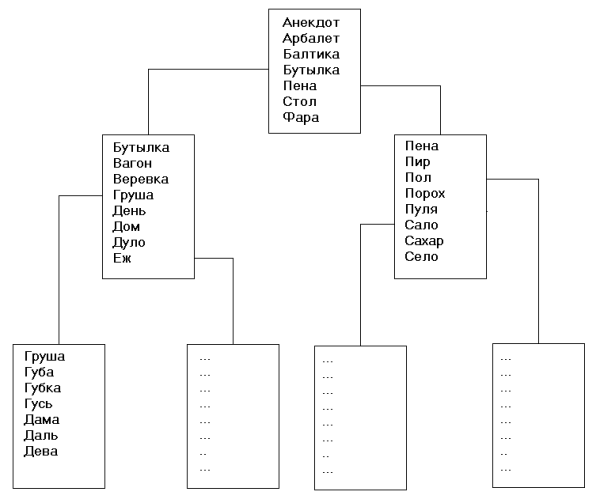
Индекс не зря называют деревом, потому что все данные в нем располагаются именно в таком виде. Давай рассмотрим, как выглядит дерево в памяти, и ты сразу увидишь все преимущества и недостатки. Я попытался его изобразить графически, но художник из меня не очень хороший (это если мягко выразиться). Во главе всего находиться главный блок, с которого всегда начинается поиск. На рисунке я набросал дерево, состоящее из первых попадавших в мою голову слов. Да, сегодня в моей голове явно бред, пора бы отдохнуть.

**Прогулка по дереву**

Итак, дерево состоит из блоков равного размера. Конечно же, размер блока должен быть достаточен для хранения хотя бы нескольких записей, но он и не должен быть слишком большим. Некоторые специалисты рекомендуют делать его равным 8 кило (мой любимый размерчик!). Не знаю, откуда взялась эта рекомендация, но ей можно найти логическое объяснение. Блок получается не большой и не маленький, а в самый раз.

Внутри блока все строки желательно упорядочить, дабы упросить в нем поиск. Так как блоки у нас небольшие, то поддерживать все записи в нем в упорядоченном состоянии достаточно просто.

Попробуем найти слово "Гусь". Начинаем поиск с первого блока и последовательно просматриваем все строки. Находим слово "Бутылка", которое меньше, чем искомый гусь, но следующее слово (Пена) будет уже больше. Переходим в следующий блок, на который указывает найденная бутылочка. Ищем здесь и видим, что ближайшее к искомому слово – Груша. Переходим по ссылке на следующий блок, где и как раз и есть искомый Гусь.

  
*B-Tree деревья в Oracle. На кончиках веток находятся значения индекса и ROWID*

Ощутил преимущества древовидного индекса? Если нет, то вот они:

1. Для поиска даже самой последней записи в наборе данных, не нужно просматривать абсолютно все блоки. Вполне возможно, что достаточно будет просмотреть только 3 или четыре индексных блока;

2. На поиск любых данных затрачивается примерно одинаковое время;

3. Для эффективной работы вполне достаточно столько памяти, сколько занимает один индексный блок. Все блоки держать в оперативке нет смысла, но если есть лишняя память, то можно кешировать блоки;

Недостаток, на мой взгляд, только один, но он очень серьезный – поддерживать такой индекс не так уж и просто. О возможных вариантах поддержки можно писать отдельную статью. Я же могу порекомендовать тебе прочитать что-нибудь про архитектуру индексов в базах данных Oracle и MS SQL Server. Архитектуру первой я знаю не очень хорошо, а вот в MS SQL Server индексные деревья сделаны достаточно эффективно.

Вот так вот при минимуме затрат можно быстро искать и обрабатывать большие объемы информации. В данном случае, мы затронули разновидность лиственных под называнием B-Tree (Balanced Tree или сбалансированные деревья). Бывают и другие разновидности, но это уже совершенно другая история.

**Кластер или нет**

Существует две разновидности деревьев – кластерные и не кластерные. В первом случае, на кончиках веток индексного дерева находятся сами данные. Это значит, что прогулявшись по всем блокам мы находим непосредственно искомые данные. Получается, что в данном случае данные упорядочены с помощью такого индекса физически. Конечно же, на один список (таблицу) можно создать только один кластерный индекс, потому что упорядочить физически по двум разным параметрам один и тот же набор данных просто нереально.

В не кластерном дереве на кончиках веток находятся только ссылки на данные, а сами данные могут находиться где угодно на диске и в любом порядке. Таких индексов можно создавать сколько угодно.

**Итого**

Мы поделились с тобой только основными алгоритмами экономии памяти и немного затронули даже оптимизацию. На эту тему можно говорить вечно, потому что оптимизация скорости и памяти – очень интересны сами по себе. Мы же не будем углубляться дальше основ, потому что дальнейшая оптимизация скорости и ресурсов требует хорошего знания математики. Если у тебя нет проблем с этой наукой, то за новыми алгоритмами можешь обратиться к трудам Кнута. Да, тексты его книг завернуты не по детски и без хорошего знания высшей математики и бутылки водки тут не разобраться, но зато пищи для ума хватит на год вперед.

**Листинг 1. Пример динамического массива**

type

PMassivItem = ^TMassivItem;

TMassivItem = record

nextItem:PMassivItem; //следующий элемент массива

prevItem:PMassivItem; //предыдущий элемент массива

Name:String; // пример данных, которые нужно хранить

// здесь могут быть еще поля элемента массива

end;

TMassiv = class

public

FirstItem:PMassivItem; // первый элемент списка

LastItem:PMassivItem; // последний элемент списка

itemsNumber:Integer; // количество элементов

constructor Create(str:String); // конструктор

procedure AddItem(str:String); // добавление элемента

procedure DeleteItem(index:Integer); // удаление

end;

procedure TMassiv.AddItem(str:String);

var

newItem:PMassivItem;

begin

newItem:= new(PMassivItem);

newItem.Name:=str;

newItem.nextItem:=nil;

newItem.prevItem:=LastItem;

LastItem.nextItem:=newItem;

LastItem:=newItem;

Inc(itemsNumber);

end;

constructor TMassiv.Create(str:String);

begin

itemsNumber:=1;

FirstItem:=new(PMassivItem);

FirstItem.Name:=str;

FirstItem.nextItem:=nil;

FirstItem.prevItem:=nil;

LastItem:=FirstItem;

end;

procedure TMassiv.DeleteItem(index: Integer);

var

i:Integer;

currentItem:PMassivItem;

begin

// поиск удаляемого элемента

currentItem:=FirstItem;

for i:=0 to index-1 do

currentItem:=currentItem.nextItem;

// наводим новые связи и освобождаем память

if currentItem.prevItem<>nil then

currentItem.prevItem.nextItem:=currentItem.nextItem;

if currentItem.nextItem<>nil then

currentItem.nextItem.prevItem:=currentItem.prevItem;

Dispose(currentItem);//очистка памяти

Dec(itemsNumber); //уменьшаем счетчик

end;