УДК 62-50

© 2004 И.В. Русских

Оптимизация системы распределения памяти в приложениях на примере библиотеки Colorer-take5.

В статье приведены базовые подходы к оценке производительности приложений, на примере библиотеки Colorer-take5 выявлены проблемы в производительности системы распределения памяти, разработаны и опробованы общие пути оптимизации методов работы с динамической памятью.

Введение.

В последнее время при разработке программного обеспечения все меньше внимания уделяется эффективности работы алгоритмов, оптимизации различных показателей функционирования приложений. В основном это связано с бурным развитием аппаратных средств, которое зачастую опережает программные системы и позволяет авторам не заду-мываться о многих параметрах функционирования своих приложений. В то же время, применяя простые средства профилирования и отладки программ, можно существенно сократить время работы, сэкономить аппаратные ресурсы и в конечном итоге предоставить пользователям более комфортную среду для работы.

Одним из значимых компонентов современных объектно-ориентированных систем является подсистема работы с динамической памятью. На примере библиотеки Colorer-take5 можно рассмотреть основные способы выявления узких мест в производительности приложений и возможные пути оптимизации работы с динамической памятью.

1. Библиотека Colorer-take5, обзор.

Библиотека Colorer является библиотекой синтаксического анализа узко специализированной на анализе исходных текстов с целью подсветки и выделения синтаксиса [5]. Ядро библиотеки написано на языке C++ и предоставляет набор программных интерфейсов для доступа к нему из внешних приложений (систем редактирования, интегрированных сред разработки и т.д.).

Особенностью библиотеки является то, что для своей работы ей необходимо загружать в память и использовать объемную базу описаний синтаксисов

поддерживаемых языков программирования. Эти описания хранятся в собственном формате HRC, базирующемся на стандарте XML и обеспечивающем возможности гибкой и динамической конфигурации функциональности.

Загрузка и обработка этих описаний в некоторых случаях может занимать значительное время, что может сказаться на работе пользователя. Для оценки эффективности работы библиотеки при загрузке был проведен набор тестов, на основе которых оценивалось использование времени и ресурсов различными компонентами библиотеки.

В качестве теста проводилась загрузка всех HRC описаний в библиотеке, их компиляция и обработка. Общий размер загружаемых файлов составил 5384 Kb, всего в базе хранится 144 раздельных типов (описаний синтаксисов), общее число схем (синтаксических контекстов, входящих в состав каждого типа) составило более 12000. Для тестирования использовался компилятор из среды **Microsoft Visual Studio 7**. При сборке приложения использовался режим максимальной оптимизации с ключами командной строки `/Ogtiyb2`.

Выполнение теста на исходной версии библиотеки дало результат 20.9 сек. Тест проводился на аппаратной конфигурации
AMD450/384Mb/40Gb/Windows2000. При тестировании делалось несколько предварительных проходов для того, чтобы убедиться в загрузке большинства описаний в кэш файловой системы – это позволило более точно оценить реальное время работы библиотеки.

Затем на этой же конфигурации проводилось профилирование работы библиотеки. В качестве средства анализа использовалась система Rational Quantify [9]. В результате была получена статистика по количеству и времени вызовов всех функций и методов библиотеки. При детальном анализе выяснилось, что довольно большой процент времени занимает работа функций выделения/освобождения памяти. В системе win32 эти функции отображаются на системные вызовы HeapAlloc/HeapFree. Было получено распределение числа вызовов функций и времени, проведенного в каждой функции (Таблица 1).

Из анализа таблицы профилирования вызовов видно, что 59.4% (12.4 сек.) всего времени работы тратится на операции выделения и освобождения памяти. Библиотека использует системные вызовы HeapAlloc и HeapFree для выделения динамической памяти под нужны приложения. При более детальном анализе теста обнаруживается, что профилировщик в этом случае несколько завышает время, проводимое в функциях new и delete. Это связано с тем, что системные вызовы HeapAlloc и HeapFree искажают распределение времени. Искажение можно объяснить тем, что профилировщик не проводит инструментирования функций операционной системы, а код, работающий под профилировщиком, использует намного больше памяти, чем при обычном запуске. Как следствие, на указанные системные функции ложится большая нагрузка.

Таблица 1: Распределение времени по функциям

Function	**Calls**		**Function**	**FD-Time%**	**F Time**
DString::[]	32207557		loadFileType	87,7	12
SString::[]	23758340		getBaseScheme	87,7	4
toWChar	11092606		parseHRC	87,5	59
SString::length	9461251		addType	50,8	1712
free	6762081		addScheme	49,5	3705
delete	**6762079**		addSchemeNodes	44,6	23643
DString::length	5549304		**new**	**33,7**	**38276**
EntityString::[]	3887736		nh_malloc	33,5	49213
isWhitespace	3576562		heap_alloc	33,2	87536
HeapAlloc	2460681		HeapAlloc	32,8	6348031
nh_malloc	2460674		CXmlEl::parse	26,0	266908
heap_alloc	2460674		**delete**	**25,7**	**15027**
new	**2460618**		free	25,6	143458
HeapFree	2460242		HeapFree	24,9	4811121
String::S̃tring	2150918		setRELow	21,3	7736

Исходя из предположения, что при замене функций распределения памяти в библиотеке на выполнение основного кода будет тратиться столько же времени и после сравнения времени работы исходного теста с другими, было установлено, что реальное время, которое система проводит в функциях new и delete, составляет около 40% от общего времени выполнения теста. Хотя это и на 19% меньше изначально предполагаемого, все равно это неприемлемо большой процент. В первую очередь это связано с особенностью теста – загрузкой и компиляцией всех HRC описаний библиотеки Colorer во внутреннее представление, которое максимальным образом подготавливается к дальнейшей обработке синтаксическим анализатором. Большое число разветвленных конструкций языка порождает множество объектов со сложными взаимосвязями. На высшем уровне это множество поддерживаемых языков в библиотеке, множество схем в каждом языке (типе). Далее представление разбирается на отдельные составляющие (компоненты схем, взаимосвязи между схемами); на низшем уровне библиотека активно оперирует такими примитивами, как хэш-таблицы, вектора (динамические массивы), массивы строк и т.д. Помимо этого память временно используется и освобождается анализатором XML-синтаксиса файлов HRC. Ввиду большого объема описаний (более 5 мегабайт) эта часть библиотеки выделяет для своей работы довольно много памяти (около 40% всех вызовов оператора new), которая к тому же практически сразу освобождается.

Можно сделать выводы, что производительность системы значительно улучшится, если оптимизировать скорость операций управления памятью, которые в конечном итоге используются для создания всех описанных ком-

понентов.

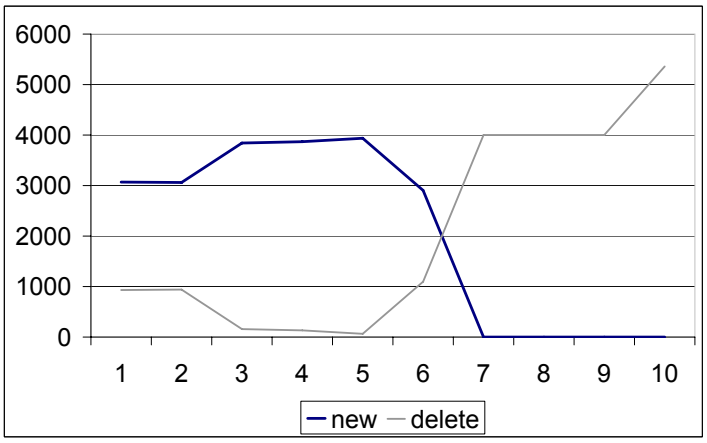
2. Анализ путей оптимизации.

Существует много различных методик и подходов к реализации и оптимизации систем динамического выделения памяти. Как правило, основной проблемой выбора становится баланс между скоростью работы и размером потребляемой памяти.

Из анализа особенностей выполнения приведенного выше теста можно установить, что создание и компиляция динамических структур в библиотеке Colorer происходит единовременно и используется на протяжении всей работы приложения. Освобождение основного объема памяти происходит лишь при прекращении работы всей системы.

Рисунок 1 показывает примерное распределение количества вызовов new/ delete в равные промежутки времени в процессе загрузки синтаксиса одного из языков в библиотеке Colorer.

Рис. 1: Распределение вызовов new/delete



Для оптимизации процесса выделения памяти в подобных случаях можно воспользоваться созданием статических блоков памяти большого объема, в пределах которых будет реализована упрощенная схема распределения памяти. Для максимального ускорения всех операций с памятью запрашиваемые блоки выделяются последовательно в пределах указанных кусков большого размера (ячеек, chunks), при этом не ведется никакого внутреннего слежения за распределением занятых/свободных блоков. Это намного уменьшает время выполнения операций new/ delete, фактически сводя их к константе.

Система управления памятью содержит единственный указатель на свободную область, который только растет. Помимо него ведется подсчет баланса всех вызовов new/ delete. Так как в такой реализации невозможно заново выделять уже освобожденные блоки памяти, этого вообще не происходит при вызове операции delete: она лишь уменьшает общий счетчик выделенных блоков. В момент, когда этот счетчик зануляется, системе возвращаются выделенные ячейки большого размера.

В случае с библиотекой Colorer, из анализа функционирования ее кода можно установить, что под такую модель работы с памятью подходит часть модулей, ответственных за представление синтаксической структуры языков программирования.

После применения указанной модели работы с памятью время выполнения исходного теста уменьшилось до 17.6 секунд. В процентном соотношении прирост скорости загрузки составил 16%.

Неоспоримыми преимуществами данного способа оптимизации являются легковесность, простота модификации и гибкость подстройки под параметры конкретного приложения. Идентичные алгоритмы могут быть применены в любых подобных ситуациях, когда в системе заранее известны типы объектов, подлежащих долговременному использованию и единовременному освобождению.

К недостаткам метода можно отнести освобождение каждого блока только после освобождения всех его составляющих. Как следствие метод можно применять только к глобальным данным, все время присутствующим в работе. При существовании утечек памяти выделенные блоки не будут возвращены в систему до завершения программы.

С точки зрения операционной системы алгоритм требует немного большего объема памяти в сравнении со стандартными функциями распределения. Это связано с возможным незаполнением выделенных блоков до конца. На приведенном тесте общий объем используемой памяти увеличился в среднем на 5-10%.

Как следствие, этот способ подходит к использованию на завершающих этапах оптимизации и в случаях, когда известно, что выделение памяти происходит малыми частями в больших количествах. При этом оптимизация будет направлена на конкретные модули приложения. В случае с языком C++ это легко реализуется с использованием переопределения операторов new и delete для нужных классов. Весь остальной код продолжает при этом использовать стандартную систему управления памятью.

Существуют методы еще более тонкой подстройки распределения памяти под особенности функционирования отдельных компонентов в приложении, которые можно использовать для узконаправленной оптимизации. Методы, подобные описанному, хорошо применимы к различным представлениям объектов-контейнеров (объектов, управляющих множествами однородных

объектов) и описаны в работе [1].

3. Использование альтернативных систем управлению памятью.

Хотя подобные узконаправленные оптимизации и улучшают параметры системы в частных случая, обычно операции выделения/освобождения памяти равномерно распределены во времени жизни приложения и в случае объектно-ориентированных систем могут активироваться большое количество раз.

Для оптимизации функционирования библиотеки Colorer в целом, в нее была внедрена система альтернативного распределения памяти, предоставляющая ряд преимуществ по сравнению с встроенными системными процедурами. Модуль dlmalloc (автор Doug Lea, [2]) реализует компактную переносимую систему управления памятью, которая может быть легко внедрена в целевое приложение и использована в нем.

Модуль использует ряд принципов оптимизации, в число которых входит кэширование мало-размерных запросов выделения, стратегии оптимального размещения для крупных запросов, смешанные стратегии для неоднородных по размеру запросов (работа [3] содержит подробный анализ существующих проблем и их решений). В качестве языка в модуле используется ANSI C, что позволяет легко внедрить его практически в любое приложение.

В случае с библиотекой Colorer глобальные операторы new/ delete/ new[]/ delete[] переопределялись с использованием специализированных методов dlmalloc/ dlfree, тем самым обеспечивая полную замену системы распределения памяти. Интеграция модуля в библиотеку Colorer дала более чем 30% ускорение работы в случае выполнения теста загрузки всех определений библиотеки. Из анализа результатов профилирования (Таблица 2) видно, что процент времени, затрачиваемый на работу с памятью сократился с 40 до 13%.

Таблица 2: Улучшенные показатели

Function	**FD Time**
SString::[]	10,56
CharacterClass::createCharClass	10,32
new	10,13
String::==	10,08
dlmalloc	**10,07**
HRCParserImpl::loadRegions	9,25
HRCParserImpl::updateLinks	8,54
...	
delete	2,79
dlfree	**2,63**
CharacterClass::addCategory	2,62

При комбинировании этого метода оптимизации с узконаправленной оптимизацией по ограниченному числу классов был получен дополнительный прирост производительности 3-4%.

Таким образом, исходя из замеров времени, на выделении памяти экономится 6,9 секунды, что составляет 33% от всего исходного времени выполнения теста. Выполнение самих операций работы с памятью ускорилось в шесть раз. Конечно, такой прирост общей производительности в основном объясняется особенностями самого теста. Но даже в обычных приложениях оптимизация может оказаться весьма весомой.

При работы основного алгоритма библиотеки Colorer – синтаксического анализа, время, затрачиваемое на операции с памятью несколько меньше. Это связано с тем, что на этой фазе работы библиотеки создается намного меньше динамических структур, работа происходит в основном на основе уже созданных структур анализа текста.

В качестве теста использовалось несколько файлов исходных текстов различных языков (C++, XML, Perl) с размером, достаточным для общей оценки производительности. Тест запускался последовательно несколько раз, затем вычислялись усредненные значения. Использование оптимизированных функций выделения памяти дало в этом случае прирост производительности 6-13 процентов. Это низкий результат по сравнению с первым тестом, он связан с тем, что в исходном (неоптимизированном) варианте теста выделение/освобождение памяти занимает около 15% (в сравнении с 40 процентами в первом тесте). Таким образом, аналогичное шестикратное ускорение операций с памятью увеличивает общую производительность в лучшем случае на 12.5%. Это значение колеблется в зависимости от синтаксиса обрабатываемого языка (его сложности, неоднородности) и действий, применяемых на следующих за синтаксическим анализом этапах.

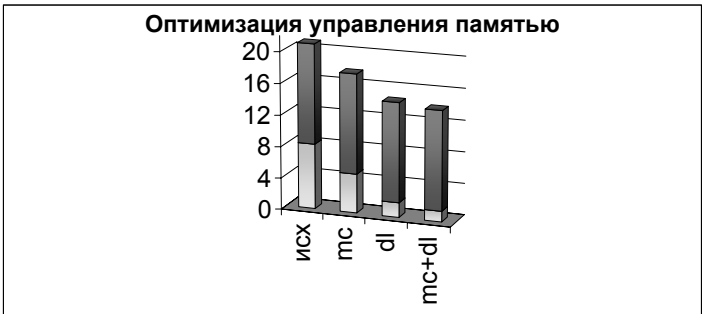
Заключение.

В итоге целенаправленная оптимизация подсистемы работы с памятью в случае с библиотекой Colorer дала довольно значительный прирост производительности, ощутимый даже для конечного пользователя (Диаграмма 2).

Следует отметить, что в реальной работе на скорость влияют различные внешние обстоятельства. Активная работа ОС с жестким диском, параллельное выполнение других задач в системе, различные действия пользователя в реальности еще более замедляют показатели. Описанные методы оптимизации позволяют сгладить этот эффект и сделать работу конечного пользователя более комфортной.

В случае с библиотекой Colorer с целью оптимизации функционирования кода были пересмотрены некоторые из внутренних алгоритмов. Так, например детальный анализ позволил выявить лишние циклы обхода структуры синтаксических описаний и оптимизировать линковку символических

Рис. 2: Результаты оптимизации



	исх	mc	dl	mc+dl
Основной код	12,64	12,64	12,63	12,64
Память	8,26	4,96	1,87	1,36

имен HRC. Это позволило на 20-30% ускорить обработку базы HRC. В процессе анализа и перепроектирования языка HRC были использованы идеи объектно-ориентированного программирования, модульного построения и связывания компонент. Это позволило избежать повторных загрузок файлов синтаксиса, как следствие уменьшилось время, требуемое на разбор XML-структур языка HRC.

Существует большое число других путей оптимизации работы приложений, большей частью они могут быть связаны с той областью, теми алгоритмами, которые реализует приложение. Исследование, выявление критических участков кода позволяет целенаправленно улучшать различные показатели. Существуют и разработаны принципы «низкоуровневой» оптимизации, с помощью которых можно добиться значительного выигрыша в производительности в ущерб стройной архитектуре и объектной модели приложения (работы [7, 8]).

Список литературы

[1] Doug Lea. Some storage allocation techniques for container classes. // C++ Report, 1989.

[2] Doug Lea, dlmalloc. http://gee.cs.oswego.edu/dl/html/malloc.html, ftp://gee.cs.oswego.edu/pub/misc/malloc.c

[3] Paul R. Wilson, Mark S. Johnstone, Michael Neely, and David Boles. Dynamic Storage Allocation: A Survey and Critical Review. // International Workshop on Memory Management, September 1995. ftp://ftp.cs.utexas.edu/pub/garbage/allocsrv.ps.

[4] Igor Russkih. HRC Language Reference, 2004. http://colorer.sf.net/hrc-ref/

[5] Igor Russkih. Colorer-take5 Library http://colorer.sf.net/

[6] Русских И.В. Основные принципы разработки библиотеки Colorer. // Методы и средства обработки сложной графической информации. Тезисы докладов. Н.Новгород, 2003 г. с. 80-82.

[7] G. Varghese, Algorithmic Techniques for Efficient Protocol Implementations, in SIGCOMM '96, (Stanford, CA), pp. 306-317, ACM, August 1996.

[8] A. Gokhale, Douglas C. Schmidt. Principles for Optimizing CORBA IIOP Performance. Washington University.

[9] Rational Quantify. http://www.rational.com

Нижегородский государственный университет им. Н.И.Лобачевского
603950, Нижний Новгород, пр. Гагарина, 23

Memory allocation optimization techniques by the example of Colorer-take5 library.

© 2004 Igor Russkih
Nizhny Novgorod State University
23 Gagarin Ave., 603950 Nizhny Novgorod, Russia

The article introduces some common approaches of application's performance measurement. More attention is focused on dynamic memory allocation performance and its practical optimization techniques in Colorer-take5 library.