Модели утилизации динамической памяти

Д. А. Сурков, К. А. Сурков, Ю. М. Четырько

Редкая программа может обойтись без использования динамических структур данных, таких как списки, деревья, массивы переменной длины, графы. Заранее предусмотреть и разместить эти данные в памяти невозможно, поэтому программы запрашивают память для данных динамически, по мере необходимости.

Когда потребность в данных отпадает, их память должна быть утилизирована, т.е. возвращена системе для повторного использования. Существуют различные модели утилизации динамической памяти, из которых широкое практическое применение получили следующие:

- модель с ручным освобождением памяти;
- модель со счетчиками ссылок;
- модель с иерархией владения;
- модель с автоматической сборкой мусора;
- модель с автоматической сборкой мусора и принудительным освобождением памяти.

Выбор модели критически влияет на надежность, безопасность, производительность и ресурсоемкость как отдельно взятой программы, так и всей системы в целом. В этой статье рассматриваются достоинства и недостатки вышеперечисленных моделей утилизации динамической памяти, чтобы помочь разработчикам систем сделать правильный выбор.

Модель с ручным освобождением памяти

Это наиболее распространенная модель. В распоряжении программиста есть две процедуры или оператора, с помощью которых он может соответственно запрашивать и освобождать участки (блоки) памяти. В языке программирования C для этой цели служат соответственно процедуры malloc и free, а в языке C++ — операторы new и delete. Операторы new и delete мощнее упомянутых процедур, они позволяют создавать и уничтожать объекты в динамической памяти, поэтому в статье речь пойдет о них.

В модели с ручным освобождением памяти система не следит за наличием или отсутствием ссылок на объекты. Программист должен сам заботиться об уничтожении ненужных объектов и о возвращении их памяти системе.

Когда программа создает объект оператором new, менеджер памяти просматривает список имеющихся свободных блоков памяти в поисках блока, подходящего по размеру. Когда такой блок найден, он изымается из списка свободных блоков и его адрес возвращается программе. Когда программа уничтожает объект, то менеджер памяти добавляет занимаемую объектом память в список свободных блоков.

Обычно список свободных блоков является двусвязным и хранится внутри свободной памяти. Перед добавлением в этот список освобождаемого блока памяти система выполняет дефрагментацию, сливая смежные свободные блоки в один.

Достоинство такой модели в ее детерминизме — временные задержки на выделение и освобождение памяти заранее предсказуемы. Кроме того, если при создании и уничтожении объектов выполняются подпрограммы инициализации и очистки, то порядок работы этих подпрограмм и связанные с этим накладные расходы тоже предсказуемы.

Недостаток модели — ненадежность и подверженность ошибкам. В больших прикладных системах, где данные передаются между несколькими модулями, очень трудно поддерживать соответствие операторов delete операторам new, поэтому выделенная память может вообще никогда не освобождаться. Происходит так называемая «утечка памяти» — объекты уже не используются, ссылок на них уже нет, но система считает память занятой. Утечки памяти могут критически влиять на работоспособность программ, работающих продолжительное время (это относится к СУБД, прикладным серверам, системам управления физическими объектами и пр.).

Еще более опасно так называемое «зависание ссылок», суть которого в том, что в программе остаются ссылки на уничтоженные объекты. Если программа обращается по такой ссылке и изменяет данные, то она не только выполняет пустую работу, но, скорее всего, меняет служебные данные менеджера памяти, используемые для организации списка свободных блоков памяти. Такие ошибки очень трудно найти и исправить, поскольку возникающие из-за них сбои происходят не сразу, а спустя некоторое время, когда уже непонятно, какая подпрограмма нарушила целостность данных.

Еще один недостаток модели состоит в том, что при интенсивном выделении и освобождении памяти, как правило, возникает сильная фрагментация — выделенные блоки памяти перемежаются занятыми блоками. В результате может наступить момент, когда суммарный объем свободной памяти очень велик, но сплошного участка нужного размера нет. При этом выполнить дефрагментацию невозможно, поскольку созданные объекты нельзя перемещать в адресном пространстве программы (ведь неизвестно, где в программе имеются ссылки на эти объекты, а значит, ссылки невозможно правильно корректировать).

Модель со счетчиками ссылок

Анализируя проблемы модели с ручным освобождением памяти, легко прийти к заключению, что для надежной работы с памятью нужно уничтожать объект лишь тогда, когда пропадают все ссылки на него. Стремление сделать уничтожение объектов автоматическим, причем в рамках существующих языков программирования, породило модель утилизации памяти на основе счетчиков ссылок.

Модель со счетчиком ссылок широко применяется в технологии COM, во многих системных библиотеках и языках программирования. Она часто реализуется как надстройка над уже рассмотренной моделью с ручным освобождением памяти.

В модели со счетчиком ссылок с каждым объектом ассоциируется целочисленный счетчик ссылок. Обычно он хранится в одном из полей объекта (хотя может быть «навешен» и снаружи). При создании объекта этот счетчик устанавливается в нулевое значение, а потом увеличивается на единицу при создании каждой новой ссылки на объект. При пропадании каждой ссылки значение счетчика уменьшается на единицу и, когда оно становится равным нулю, объект уничтожается (оператором delete). Таким образом, программисту не нужно думать о том, когда следует уничтожить объект — это автоматически происходит тогда, когда пропадает последняя ссылка на него.

Увеличение и уменьшение счетчика ссылок выполняется с помощью двух специальных методов объекта, в технологии COM называемых AddRef и Release. Метод AddRef вызывается при любом копировании ссылки, а также при ее передаче в качестве параметра подпрограммы. Метод Release вызывается при пропадании или обнулении ссылки, например, в результате выхода программы за

область видимости ссылки или при завершении подпрограммы, в которую ссылка была передана в качестве параметра.

В зависимости от языка программирования за вставку в код вызовов методов AddRef и Release отвечает либо компилятор, либо макроподстановка (шаблон) системной библиотеки, либо сам программист.

Очевидный недостаток этой модели — наличие дополнительных накладных расходов на элементарное копирование ссылок. Еще более серьезный недостаток состоит в том, что счетчики ссылок не учитывают возможных циклических связей между объектами. В этом случае счетчики ссылок никогда не уменьшаются до нуля, что ведет к утечкам памяти.

Для решения проблемы циклических связей используется следующий прием. Ссылки делят на два вида: «сильные» ссылки и «слабые» ссылки. Сильные ссылки влияют на счетчик ссылок, а слабые ссылки — нет. При уничтожении объекта слабые ссылки автоматически обнуляются. Для доступа к объекту слабую ссылку нужно предварительно превратить в сильную ссылку (это предотвращает уничтожение объекта во время операций с ним).

Реализация сильных и слабых ссылок вносит дополнительные расходы: увеличивается потребление памяти и значительно замедляется доступ к объектам. Так, в библиотеке boost для языка C++ каждая ссылка в действительности представляет собой запись, в которой помимо указателя на реальный объект хранятся служебные данные, причем, они размещаются в динамической памяти и требуют в ней лишнего места. За каждой операцией доступа к объекту скрывается дополнительный доступ к служебным данным для проверки того, что объект еще «жив». Библиотека шаблонов и компилятор скрывают от программиста эти детали, он думает, что работает с обычными ссылками и не осознает потери памяти и снижения производительности.

Деление ссылок на виды больше запутывает программиста, чем помогает ему. При написании программы ответить на вопрос, какого вида должна быть ссылка, порой затруднительно. Кроме того, в программе все равно возникает опасность циклических связей, образованных сильными ссылками. Попытка минимизации количества сильных ссылок (вплоть до одной на каждый объект) и повсеместное использование слабых ссылок приводят нас, по сути, к модели с ручным освобождением памяти, с той лишь разницей, что уничтожение объекта выполняется не вызовом оператора delete, а обнулением главной ссылки на объект. Единственная проблема, которая при этом решается — это проблема зависших ссылок.

Модель с иерархией владения

Анализ структуры многих программ показывает, что динамические объекты часто объединяются в иерархию. Например, в программах с графическим пользовательским интерфейсом главный объект управления программой содержит в себе объекты окон, которые в свою очередь содержат объекты панелей и кнопок. Отношением подчиненности могут быть связаны не только объекты пользовательского интерфейса, но и любые данные в программе. Используя эту особенность можно реализовать модель утилизации памяти, которая будет существенно более надежной, чем предыдущие.

Эта модель — модель с иерархией владения — основана на том, что при создании любого объекта ему назначается объект-владелец. Владелец отвечает за уничтожение подчиненных объектов. Создав объект и назначив ему владельца, можно больше не заботиться о том, что ссылки на него пропадут, и произойдет утечка памяти. Этот объект будет обязательно уничтожен при удалении владельца.

Объект можно уничтожить принудительно, даже если у него есть владелец. При этом объект либо изымается из списка подчиненных объектов своего владельца, либо помечается как уничтоженный для предотвращения повторного уничтожения.

Объект может быть создан без владельца, в этом случае он требует явного уничтожения. Модель управления временем жизни объектов без владельца ничем не отличается от уже рассмотренной модели с ручным освобождением памяти.

Модель с иерархией владения не избавляет программиста полностью от необходимости явно освобождать память, однако значительно сокращает риск утечек памяти. Эта модель, также как и предыдущие, не решает проблему фрагментации памяти, но позволяет более успешно бороться с зависшими указателями, например, путем рассылки сообщений об уничтожении объектов по иерархии. Обрабатывая эти сообщения, объекты-получатели могут обнулять сохраненные ссылки на уничтожаемые объекты.

Модель с иерархией владения применяется во многих графических библиотеках и средах визуального программирования (для управления компонентами), она успешно использовалась авторами этой статьи для управления объектами в САПР СБИС.

Модель с иерархией владения иногда совмещается с моделью на основе счетчиков ссылок. Такая гибридная модель используется, например, в новейшей технологии драйверов для ОС Windows. [MS06]

Модель с автоматической сборкой мусора

Главными требованиями, которые предъявляются к современным программным системам, являются надежность и безопасность. Чтобы их обеспечить, нужно в принципе устранить возможность утечек памяти и избавиться от зависания ссылок. Это достижимо лишь в модели с автоматической утилизацией памяти на основе так называемой сборки мусора.

Модель с автоматической сборкой мусора предусматривает лишь возможность создавать объекты, но не уничтожать их. Система сама следит за тем, на какие объекты еще имеются ссылки, а на какие — уже нет. Когда объекты становятся недостижимы через имеющиеся в программе ссылки (превращаются в «мусор»), их память автоматически возвращается системе.

Эта работа периодически выполняется сборщиком мусора, и происходит в две фазы. Сначала сборщик мусора находит все достижимые по ссылкам объекты и помечает их. Затем он перемещает их в адресном пространстве программы (с соответствующей корректировкой значений ссылок) для устранения фрагментации памяти.

Обход графа достижимых объектов начинается от «корней», к которым относятся все глобальные ссылки и ссылки в стеках имеющихся программных потоков. Анализируя метаданные (информацию о типах данных, которая размещается внутри выполняемых модулей), сборщик мусора выясняет, где внутри объектов имеются ссылки на другие объекты. Следуя по этим ссылкам, сборщик мусора обходит все цепочки объектов и выясняет, какие блоки памяти стали свободными. После этого достижимые по ссылкам объекты перемещаются для устранения фрагментации, а ссылки на перемещенные объекты корректируются.

Эта модель вроде бы решает все проблемы: нет утечек памяти, нет фрагментации памяти, нет зависших указателей.

По скорости выделения памяти данная модель сравнима со стеком, ведь выделение объекта — это, по сути, увеличение указателя свободной области памяти на размер размещаемого объекта. Однако по достижении этим указателем определенного предела запускается сборка мусора, которая может потребовать много времени и привести к ощутимой задержке в работе программы.

Моменты наступления таких задержек и их длительности обычно непредсказуемы. Поэтому одна из проблем сборщика мусора — это недетерминизм связанных с его работой задержек.

Для амортизации задержек в сборщиках мусора применяются различные подходы. Например, в среде .NET используется принцип поколений, основанный на том наблюдении, что объекты, создаваемые раньше, как правило, живут дольше. Вся память делится на поколения (их количество обычно соответствует числу уровней кэширования с учетом ОЗУ; в современных архитектурах обычно три поколения). Нулевое (младшее) поколение самое маленькое по объему, первое поколение в несколько раз больше, чем нулевое, а второе в несколько раз больше, чем первое. Объекты создаются в младшем поколении и перемещаются в старшие поколения, пережив сборку мусора. Сборка мусора выполняется не во всей памяти, а лишь в тех поколениях, в которых исчерпалось свободное место — чаще в нулевом, реже в первом, и еще реже во втором поколении. Таким образом, задержек при сборке мусора много, но их средняя длительность небольшая. [JR00]

Другой подход к амортизации задержек используется в сборщиках мусора реального времени среды Java (JRTS — Java Real-Time Specification). В них дефрагментация выполняется эпизодически и лишь в самом крайнем случае, когда не может быть найден свободный блок памяти нужного размера. Кроме того, сборка мусора выполняется в течение фиксированных интервалов времени (квантов), которые обязательно чередуются с квантами работы программы. [BS07]

В модели с автоматической сборкой мусора программный код завершения жизни объекта (метод Finalize или, говоря иначе, деструктор) выполняется асинхронно в контексте сборщика мусора. Момент и порядок вызова этого метода у того или иного объекта никак не детерминирован, что порождает проблему, если объект управляет некоторым ресурсом, например, сетевым соединением. Открытие соединения происходит при создании и инициализации объекта, т.е. предсказуемо, а закрытие соединения — во время сборки мусора, т.е. непредсказуемо и далеко не сразу после потери последней ссылки на объект. В результате лимит сетевых соединений, или других ресурсов, может временно исчерпаться.

Для решения указанной проблемы в среде .NET используется детерминированное завершение жизни объектов через интерфейс IDisposable. Этот интерфейс имеет единственный метод Dispose, который реализуется в объектах, управляющих ресурсами. Метод Dispose как правило освобождает ресурсы и отменяет работу процедуры-завершителя (метода Finalize), чтобы ускорить освобождение памяти. После вызова метода Dispose объект не уничтожается, а остается в памяти до тех пор, пока не пропадут все ссылки на него.

Применение интерфейса IDisposable на практике выявило новые проблемы. Оказалось, что после вызова метода Dispose в программе могут оставаться ссылки на объект, находящийся уже в некорректном состоянии. Программе никто не запрещает обращаться по этим физически доступным, но логически зависшим ссылкам и вызывать у некорректного объекта различные методы. Метод Dispose может вызываться повторно, в том числе рекурсивно. Кроме того, в программе с несколькими вычислительными потоками может происходить асинхронный вызов метода Dispose для одного и того же объекта.

Для решения проблем метода Dispose программистам было предписано делать следующее: 1) определять в объекте булевский флаг, позволяющий выяснить, работал ли в объекте код завершения; 2) блокировать объект внутри метода Dispose на врямя работы кода завершения; 3) игнорировать повторные вызовы метода Dispose, проверяя упомянутый булевский флаг; 4) в начале public-методов проверять, что объект уже находится в завершенном состоянии, и в этом случае создавать исключение класса ObjectDisposedException.

Заканчивая критику сборщиков мусора, укажем на еще одну серьезную проблему с ними — легальную утечку памяти. Если в модели с ручным освобождением объектов утечка памяти возникает из-за невыполненных операторов delete, то в модели с автоматической сборкой мусора

утечка памяти возникает из-за невыполненного обнуления ссылок. Такой вид утечек памяти характерен для программного кода, в котором одни объекты регистрируют обработчики событий в других объектах. Программисты порой забывают отключать обработчики событий, в результате ассоциированные объекты остаются в памяти, несмотря на кажущееся отсутствие ссылок на них в программе.

Модель с автоматической сборкой мусора и принудительным освобождением памяти

Отвлечемся на время от проблем реализации, и сформулируем некую идеальную с точки зрения программиста модель утилизации динамической памяти. На наш взгляд, в этой модели должны сочетаться: 1) быстрая автоматическая сборка мусора и 2) безопасное принудительное освобождение памяти.

Наличие сборки мусора означает, что программист может полагаться на то, что система следит за потерей ссылок на объекты и устраняет утечку памяти. Наличие безопасного принудительного освобождения памяти означает, что программист вправе уничтожить объект, при этом память объекта возвращается системе, а все имеющиеся на него ссылки становятся недействительными (например, обнуляются).

Эта модель, называемая нами моделью с автоматической сборкой мусора и принудительным освобождением памяти, на самом деле не нова и уже давно применяется в компьютерах Эльбрус (на основе одноименного процессора) и AS/400 (на основе процессора PowerPC), которые обеспечивают очень эффективную реализацию этой модели за счет аппаратной поддержки.

На каждое машинное слово в этих компьютерах отводится два дополнительных бита, называемых битами тегов. Значения этих битов показывают, свободно ли машинное слово, или занято, и если занято, то хранится ли в нем указатель, или скалярное значение. Этими битами управляют аппаратура и операционная система, прикладным программам они недоступны. Программа не может создать ссылку сама, например, превратив в нее число или другие скалярные данные. Созданием объектов занимается система, которая размещает в памяти объекты и создает ссылки на них. При уничтожении объектов соответствующие теги памяти устанавливаются в состояние, запрещающее доступ. Попытка обратиться к свободной памяти по зависшему указателю приводит к аппаратному прерыванию (подобно обращению по нулевому указателю). Поскольку вся память помечена тегами, сборщику мусора нет необходимости анализировать информацию о типах, чтобы разобраться, где внутри объектов располагаются ссылки на другие объекты. Что более важно, сборщику мусора почти не нужно тратить время на поиск недостижимых объектов, поскольку освобожденная память помечена с помощью тех же тегов [FS98, BB03].

Ниже сформулированы базовые принципы модели с автоматической сборкой мусора и принудительным освобождением памяти на уровне спецификации для языков программирования:

- Выделение динамической памяти выполняется оператором/процедурой new (это действие считается элементарным в системе). Выделенная память автоматически инициализируется нулями и всегда привязывается к типу созданного в памяти объекта.

- Уничтожение объекта — освобождение занимаемой им динамической памяти — выполняется автоматически при пропадании всех ссылок на объект. Для дефрагментации освободившихся участков памяти периодически выполняется сборка мусора, в результате которой объекты сдвигаются, а ссылки на них корректируются.

- Объекты можно уничтожать принудительно с помощью оператора/процедуры delete. В результате этого действия все ссылки на объект становятся недействительными, а попытка последующего доступа к объекту приводит к исключительной ситуации. Дефрагментация

освобожденной этим способом памяти выполняется во время сборки мусора. При этом оставшиеся ссылки корректируются и получают некоторое зарезервированное недействительное значение, например, -1 (зависшие ссылки можно было бы обнулять, но в этом случае стерлась бы разница между нулевой и зависшей ссылкой, что ухудшило бы диагностику ошибок).

В эти принципы вписываются как аппаратно приближенные языки с ручным освобождением памяти (такие как C и C++), так и высокоуровневые языки с автоматической сборкой мусора (такие как Oberon, Java и C#).

Главный вопрос, который пока остается открытым, — можно ли реализовать эту модель программно, чтобы она эффективно работала для популярных аппаратных архитектур, в которых нет тегирования памяти. Подумаем, каким образом этого можно достичь.

Первое простейшее решение состоит в том, чтобы по каждому вызову оператора delete выполнять просмотр памяти с корректировкой недействительных ссылок. Просмотр занимает значительно меньше времени, чем полная сборка мусора с дефрагментацией памяти. Решение подходит для мобильных и встроенных устройств с небольшим объемом ОЗУ и без поддержки виртуальной памяти.

Второе решение основано на использовании средств аппаратной поддержки виртуальной памяти, которая существует в большинстве современных компьютерных архитектур. Виртуальная память практически всегда имеет страничную организацию. Страницы памяти могут быть выгружены на диск и помечены как отсутствующие. Обращение к данным в выгруженной странице приводит к аппаратному прерыванию. Это прерывание обрабатывает ОС, которая подгружает запрошенную страницу с диска и замещает ею одну из редко используемых страниц. В этом механизме нас интересует возможность аппаратно перехватывать обращения к страницам виртуальной памяти. На самом деле страницы могут оставаться в памяти и на диск не выгружаться. Идея состоит в том, чтобы при вызове оператора delete помечать страницы, в которых располагается удаляемый объект, как отсутствующие. Обращение к данным на этих страницах будет вызывать аппаратное прерывание. Обрабатывая это прерывание, система проверяет, куда именно выполняется обращение: к освобожденному участку памяти, или занятому. Если обращение выполняется к занятому участку страницы, то запрос удовлетворяется и работа продолжается в штатном режиме. Если обращение выполняется к освобожденному участку памяти, то создается программная исключительная ситуация.

Это решение имеет очевидный недостаток — при большом количестве обращений к «живым» объектам, расположенным на одной странице рядом с удаленными объектами, будут возникать холостые аппаратные прерывания, которые снизят производительность системы. Для борьбы с этой проблемой система должна подсчитывать частоту холостых прерываний и в случае превышения этим значением некоторого порога досрочно запускать процесс обнуления недействительных ссылок.

Заключение

По мнению авторов, наиболее перспективной моделью утилизации динамической памяти представляется модель с автоматической сборкой мусора и принудительным освобождением памяти. Эта модель может быть эффективно реализована при наличии аппаратного тегирования оперативной памяти. Она может быть реализована и в отсутствии тегирования, возможные варианты этой реализации предложены в статье.

Литература

[MS06] Architecture of the Kernel-Mode Driver Framework — Microsoft Corporation, сентябрь 2006. Доступно на сайте http://www.microsoft.com/whdc/driver/wdf/KMDF-arch.mspx

[JR00] Jeffrey Richter. Garbage Collection: Automatic Memory Management in the Microsoft .NET Framework, в 2-х частях — MSDN Magazine, ноябрь-декабрь 2000. Доступно на сайте http://msdn.microsoft.com/msdnmag

[BS07] Benjamin Biron, Ryan Sciampacone. Real-time Java, Part 4: Real-time garbage collection. Доступно на сайте http://www.ibm.com/developerworks/java

[FS98] Фрэнк Солтис. Основы AS/400 — Русская редакция, 1998.

[BB03] Б.А. Бабаян. Защищенные информационные системы. 2003. Доступно на сайте http://www.mcst.ru