Математические структуры и моделирование 2018. № 3(47). С. 121–125

УДК 004.258
DOI: 10.25513/2222-8772.2018.3.121-125

ПОЛУАВТОМАТИЧЕСКОЕ УПРАВЛЕНИЕ ПАМЯТЬЮ ПРИЛОЖЕНИЯ

А.Д. Городецкий
аспирант, e-mail: overln2@gmail.com
Д.Н. Лавров
к.т.н., доцент, e-mail: lavrov@omsu.ru

Омский государственный университет им. Ф.М. Достоевского, Омск, Россия

Аннотация. В статье представлен подход к управлению памятью приложения на основе полуавтоматического управления, который сочетает в себе удобства автоматического «сборщика мусора» и полностью ручного управления, как это принято в языке C. Это достаточно гибкий подход, который может менять виды менеджеров памяти во время выполнения программы и/или даже в зависимости от платформы.

Ключевые слова: сборщик мусора, память приложения, алгоритмы управления памятью.

Введение

В работах [1, 2] рассматривается более 100 алгоритмов менеджмента динамической памяти программ. Каждый из этих алгоритмов имеет свои достоинства и недостатки. Есть высокая вероятность, что менеджер памяти, работающий максимально эффективно на любых сценариях, так и не будет создан.

По этой причине языки системного уровня (C / C++/ Rust), позволяющие манипулировать сырой памятью программ, до сих пор широко применимы. Языки, предназначенные для создания прикладных программ, предлагают автоматическое управление памятью по какому-то одному алгоритму. Некоторые среды, например Oracle HotSpot VM, позволяют выбрать один из нескольких алгоритмов на этапе старта программы.

В итоге мы имеем 2 полярных метода — полностью ручное управление памятью, что приводит к необходимости контроля корректности каждой строчки кода, или же абсолютно автоматическое управление (пусть хоть и одним из нескольких способов).

На самом деле прикладные программы работают иначе. Любая программа — это совокупность подпрограмм (подсистем), алгоритм работы каждой из которых подразумевает использование какого-то конкретного менеджера памяти, максимально эффективного в данном случае. Более того, часто эти подсистемы фактически имеют изолированные сегменты памяти.

Ручное управление памятью максимально эффективно с точки зрения машинного времени, но максимально неэффективно в качестве использования

временных затрат на разработку. Автоматическое — наоборот. Стоит ли нам рассмотреть гибридные схемы? Схемы ручного управления памятью в *подсистемах* программ, а не в строчках кода. Внутри же самой подсистемы память будет управляться автоматически. Назовём такую схему управления *полуавтоматической*.

1. Полуавтоматическое управление памятью

Реализация подразумевает как поддержку на этапе компиляции, так и на этапе исполнения программы. На этапе компиляции в код программы должны быть добавлены вызовы указателей на функции сборщика мусора. А на этапе исполнения должна быть возможность эти функции подменить для переключения блоков кода (подсистем) между разными сборщиками мусора.

Также на этапе компиляции должна быть организована проверка корректности управления памятью на уровне блоков в соответствии с двумя правилами.

1. Контроль времени жизни — переменные, выделенные в одном блоке (менеджере памяти) не могут его пережить.
2. Блоки кода могут разделять переменные только на чтение.

Вопросы корректности требуют отдельного рассмотрения и выходят за рамки данной статьи.

2. Обобщённый интерфейс сборщика мусора

Алгоритмы менеджмента памяти можно разделить на 3 категории: трассирующие, подсчёт ссылок, арены. Кратко их рассмотрим.

Трассирующие сборщики — в классическом варианте освобождают память на этапе запроса новой памяти. Реально собирают не мусор, а достижимые объекты. В большинстве случаев осуществляют дефрагментацию памяти путём копирования достижимых объектов в новый регион.

Подсчёт ссылок — дополнительно для каждого объекта хранят счётчик ссылок на этот объект. При каждой перезаписи переменных или полей объектов счётчики ссылок обновляются.

Арена — изначально выделяет регион памяти достаточно большого размера. В момент запроса на создание объекта отрезает от региона фрагмент, равный размеру объекта. Реально сборку мусора не осуществляет. Все объекты удаляются во время освобождения области памяти, выделенной ареной.

Для возможности использования этих трёх типов менеджеров памяти компилятор должен делегировать менеджеру памяти следующие вызовы:

1. Создание объекта.
2. Выход объекта из области видимости.
3. Запись одного объекта в поле другого; или одной переменной в другую.

Так как трассирующие сборщики мусора используют корни (gc roots) [1] как стартовые узлы для обхода графов объектов, то нам следует явно разграничить вызовы для создания (выхода из области видимости) локальных переменных (alloc root) и неименованных объектов (alloc). Функции, необходимые

для реализации менеджера памяти, в зависимости от типа менеджера указаны в таблице 1.

Менеджерам памяти нужна информация о типах объектов. В простейшем случае (C malloc) в функцию выделения памяти передаётся размер объекта. Однако в общем случае также необходима информация о выравнивании, типе объекта, ссылках на другие объекты, деструкторах.

Определим структуру `tinfo`

```
typedef void (*walker_routine)
                (void* ctx, void* obj, tinfo* info);
typedef void (*walker)
                (void* ctx, walker_routine callback);
typedef void (*destructor)
                 (void* obj);

struct tinfo {
   const char* tname;
   sizeof size;
   sizeof align;
   destructor destroy;
   walker walk;
}
```

Указатель на функцию `walker` позволяет рекурсивно обойти все дочерние поля объекта и совершить над ними какую-либо конкретную операцию, например, уничтожить, вызвав деструктор. Выразим общий интерфейс менеджера памяти на языке C:

```
   void* alloc(tinfo* info)
   void* alloc_root(tinfo* info)
   void free(tinfo* info, void* obj)
   void free_root(tinfo* info, void* obj)
   void store(tinfo* info, void* dest, void* src)
```

В случае, когда менеджеру памяти не нужна какая-то часть вызовов, используются заглушки.

3. Поддержка на этапе исполнения

На этапе исполнения нам нужна возможность подменять указатели на функции менеджера памяти. Однако нужно не допустить конфликтов на уровне потоков исполнения команд (системных потоков). Для этого глобальные переменные указателей на функции должны быть объявлены как `thread_local[]`. Thread local переменные широко поддерживаются линковщиками, и на многих архитектурах (например x86-64) не вносят дополнительных расходов.

Алгоритм замены менеджера памяти:

Таблица 1. Функции, необходимые для реализации менеджера памяти

	trace gc	**ref count**	**arena**
alloc	+	+	+
alloc root	+	=alloc	=alloc
free	stub	+	stub
free root	stub	=free	stub
store	stub	+	stub

1. Сохранить указатели на функции родительского менеджера памяти.
2. Записать свои указатели на функции.
3. Для получения памяти самому необходимо использовать функции *родительского* менеджера памяти.
4. При деактивации вернуть указатели на функции родительского менеджера памяти.

```
# pool
def withPool = self: MemPool,
               action: () -> None do
  oldAlloc = loadAlloc() # состояние старого аллокатора
  storeAlloc(
poolAlloc,
poolRootAlloc,
poolFree,
poolRootFree,
poolStore) # подменяем на себя
  action() # вызываем пользовательский код под пулом
  storeAlloc(oldAlloc) # возвращаем родительский менеджер

# использование
pool = mkMemPool(conf)
pool withPool lambda
   doSomeCrazyStuff()
```

Делегирование собственных запросов на память родительскому менеджеру позволяет выстраивать иерархии менеджеров памяти с полным контролем лимитов на память.

4. Заключение

Данная схема позволяет получить контроль над управлением памятью в программе, достаточный для требуемого уровня производительности. Чем требуемый уровень производительности выше, тем меньшими блоками кода может

понадобиться управлять. Мы можем очень гибко варьировать производительность итоговой программы, внося дополнительное ручное управление памятью.

Главным преимуществом по сравнению с ручным управлением является то, что для изменения схемы управлением памятью в каком-то блоке кода (рекурсивно на все вызовы) нет необходимости переписывать ***каждую строчку кода***. Достаточно лишь переключить блок кода на новый менеджер и согласовать вход и выход данных. Один и тот же библиотечный код может работать на *разных алгоритмах менеджмента памяти* без каких-либо изменений.

Главным преимуществом по сравнению с автоматическим управлением является то, что вы получаете контроль над памятью, и можете получить дополнительную производительность на этапе *оптимизации*, если вам будет это необходимо.

Данная реализация достаточно проста и близка к libc API для работы с динамической памятью. Накладные расходы на вызов заглушек (stub) достаточно малы, что позволяет иметь скорость, сравнимую с C при использовании арен.

Благодарности

Выражаем признательность Елене Анатольевне Костюшиной и Евгению Александровичу Тюменцеву за обсуждение результатов и конструктивную критику.

Литература

1. Jones R., Hosking A., Moss E. The Garbage Collection Handbook: The Art of Automatic Memory Management // Applied Algorithms and Data Structures series. Chapman & Hall/CRC, 2011. 511 p.
2. Bartlett J. Inside memory management: The choices, tradeoffs, and implementations of dynamic allocation // IBM. Developer Works. 2004. URL: `https://www.ibm.com/developerworks/linux/library/l-memory/` (дата обращения: 03.09.2018).

SEMI-AUTOMATIC APPLICATION MEMORY MANAGEMENT

A.D. Gorodetsky
Postgraduate Student, e-mail: overln2@gmail.com
D.N. Lavrov
Ph.D.(Eng.), Associate Professor, e-mail: lavrov@omsu.ru

Dostoevsky Omsk State University, Omsk, Russia

Abstract. The article presents an approach to memory management of an application based on semi-automatic control that combines the convenience of an automatic "garbage collector" and completely manual control, as is customary in the C language. This is a fairly flexible approach that can change the types of memory managers during program execution and / or even depending on the platform.

Keywords: garbage collector, application memory, memory management algorithms.

Дата поступления в редакцию: 01.09.2018