Тема 9. Среды времени выполнения (из главы 7 в Ахо, 2-е изд.)

Компилятор должен точно реализовывать абстракции, воплощенные в определении исходного языка программирования. Обычно эти абстракции включают такие концепции, как имена, области видимости, связывание, типы данных, операторы, процедуры, параметры и конструкции потока управления. Компилятор должен сотрудничать с операционной системой и другим программным обеспечением для поддержки этих абстракций на целевой машине.

Для этого компилятор создает среду времени выполнения (run-time environment), в которой, как предполагается, будет выполняться целевая программа, и управляет ею. Эта среда решает множество вопросов, таких как схема размещения и память для именованных объектов исходной программы, механизмы, используемые целевой программой для доступа к переменным, связи между процедурами, механизмы передачи параметров, взаимодействие с операционной системой, устройствами ввода-вывода и другими программами.

В этой теме будут рассмотрены выделение памяти и доступ к переменным и данным.

9.1. Организация памяти

С точки зрения разработчика компилятора, целевая программа работает в собственном адресном пространстве, в котором у каждого программного значения есть свое местоположение в памяти. Управление этим логическим адресным пространством и его организация разделяются между компилятором, операционной системой и целевой машиной. Операционная система отображает логические адреса на физические, которые обычно разбросаны в памяти.

Представление времени выполнения объектной программы в пространстве логических адресов состоит из областей данных и программ (рис. 9.1).

Рис. 9.1. Типичное разделение памяти времени выполнения на области кода и данных

Предполагается, что память времени выполнения имеет вид блоков смежных байтов, где байт – наименьшая адресуемая единица памяти. Четыре байта образуют машинное слово. Многобайтные объекты хранятся в последовательных байтах, а их адреса определяются адресами их первых блоков.

Количество памяти, необходимое для конкретного имени, определяется его типом. Элементарные типы данных, такие как символы, целые и действительные числа, могут храниться в целом количестве байтов. Память, выделенная для составных типов, таких как массивы и структуры (записи), должна быть достаточно велика для хранения всех компонентов.

На схему размещения объектов данных в памяти сильное влияние оказывают ограничения адресации на целевой машине. На многих машинах команды сложения целых чисел могут требовать *выравнивания* целых чисел, т.е. их размещения в адресах, кратных 4. Хотя массив из 10 символов требует только 10 байтов для хранения своего содержимого, компилятор может выделить ему для корректного выравнивания 12 байт, оставляя 2 байта неиспользуемыми. Память, остающаяся неиспользуемой из-за выравнивания, известна как *заполнение* (padding). В случаях, когда вопросы экономии памяти выходят на первое место, компилятор может *паковать* (pack) данные так, чтобы заполнения не было; однако при этом могут потребоваться дополнительные команды времени выполнения для позиционирования упакованных данных таким образом, чтобы они могли быть обработаны так, как если бы они были корректно выровнены.

Размер сгенерированного целевого кода фиксируется во время компиляции, так что компилятор может разместить выполняемый целевой код в статически определенной области (*Код* на рис. 9.1), обычно в нижних адресах памяти. Аналогично в процессе компиляции может быть известен размер некоторых объектов данных программы, таких как глобальные константы, и данных, сгенерированных компилятором, таких как информация для поддержки сборки мусора; такие данные также могут быть размещены в другой статически определяемой области (*Статические данные* на рис. 9.1). Одна из причин статического выделения памяти для максимально возможного количества объектов данных заключается в том, что адреса этих объектов могут быть встроены в целевой код.

Для максимального использования памяти во время выполнения программы две другие области (*Стек* и *Куча*) находятся по разные стороны оставшегося адресного пространства. Это динамические области – их размеры могут изменяться в процессе работы программы, причем области при необходимости растут одна навстречу другой. Стек используется для хранения структур данных, называющихся *записями активации* и генерируемых при вызовах процедур.

На практике стек растет по направлению к меньшим адресам, а куча – по направлению к бóльшим.

Для хранения при вызове процедуры информации о состоянии машины, такой как счетчик команд и регистры машины, используется запись активации. Когда управление возвращается из вызова, активация вызывающей процедуры может быть возобновлена после восстановления значений регистров и установки счетчика команд в точку, следующую непосредственно за вызовом. Объекты данных, время жизни которых находится в пределах времени жизни активации, могут располагаться в стеке вместе с другой информацией, связанной с активацией.

Многие языки программирования позволяют программисту выделять память для данных и освобождать ее под управлением программы. Например, в САОД мы использовали процедуры *New* и *Dispose* для получения и освобождения блоков памяти требуемого размера. Для управления данными такого вида используется куча.

Выделение памяти для данных и схема их размещения в памяти в среде времени выполнения являются ключевыми вопросами управления памятью. Это достаточно сложные вопросы, поскольку одно и то же имя в исходном тексте программы может обозначать несколько разных мест в памяти во время работы программы.

Два прилагательных – *статическое* и *динамическое* – предназначены для того, чтобы различать действия во время компиляции и во время выполнения программы. Говорят, что решение о выделении памяти статическое, если оно принимается во время компиляции, когда известен только исходный текст программы, но не то, что именно программа делала в процессе работы. И наоборот, решение является динамическим, если оно может быть принято исключительно при выполнении программы.

Многие компиляторы используют для динамического выделения памяти некоторую комбинацию двух следующих стратегий.

1. *Хранение в стеке*. Память для локальных имен процедуры выделяется в стеке. Стек поддерживает обычную политику вызова процедур и возврата из них.

2. *Хранение в куче*. Память для данных, которые должны «пережить» вызов создающей их процедуры, обычно выделяется в куче, которая представляет собой многократно используемую память. Куча представляет собой область виртуальной памяти, которая позволяет объектам или другим элементам данных получать место для хранения при создании и освобождать его, когда эти объекты становятся недействительными.

Одно из средств поддержки управления кучей – «сборка мусора» (garbage collection) — позволяет системе времени выполнения находить ненужные элементы данных и повторно использовать занимаемую ими память, даже если программист явно не освободил ее. Автоматическая сборка мусора представляет собой существенный элемент многих современных языков программирования, несмотря на трудность эффективного выполнения данной операции; в ряде языков применение такой технологии попросту невозможно.

9.2. Выделение памяти в стеке

Почти все компиляторы языков программирования, в которых применяются пользовательские функции, процедуры или методы, как минимум часть своей памяти времени выполнения используют в качестве стека. Всякий раз при вызове процедуры (это обобщенный термин для процедур, функций, методов или подпрограмм) в стеке выделяется пространство для ее локальных переменных, а по завершении процедуры это пространство снимается со стека. Такой метод не только позволяет совместно использовать память не перекрывающимся по времени процедурам, но и обеспечивает возможность компиляции кода процедуры таким образом, что относительные адреса ее нелокальных переменных всегда остаются одними и теми же, независимо от последовательности вызовов процедур.

Выделение памяти в стеке не имело бы столько преимуществ, если бы *вызовы*, или *активации* (activation), не могли бы быть вложенными во времени.

Если активация процедуры *p* вызывает процедуру *q*, то активация *q* должна завершиться раньше, чем завершится активация *p*. Обычно складывается одна из трех ситуаций.

1. Активация *q* завершается нормально. Тогда, по сути, в любом языке программирования управление возвращается в точку *p*, следующую непосредственно за точкой, в которой был сделан вызов *q*.

2. Активация *q* или некоторой процедуры, вызванной *q* прямо или косвенно, завершается аварийно, т.е. продолжение выполнения программы невозможно. В этом случае *p* завершается одновременно с *q*.

3. Активация *q* завершается из-за сгенерированного исключения, которое *q* обработать не в состоянии. Процедура *p* может обработать исключение; в этом случае активация *q* завершается, в то время как активация *p* продолжается, хотя и не обязательно с точки, в которой была вызвана *q*. Если *p* не может обработать исключение, то данная активация *p* завершается одновременно с активацией *q* и, скорее всего, исключение будет обработано некоторой другой открытой активацией процедуры.

Активации процедур в процессе выполнения всей программы можно представить в виде дерева, называемого *деревом активаций* (activation tree). Каждый узел соответствует одной активации, а корень представляет собой активацию главной процедуры, которая инициирует выполнение программы. В узле активации процедуры *p* дочерние узлы соответствуют активациям процедур, вызываемых данной активацией *p*. Эти активации указываются слева направо в порядке их вызовов. Заметим, что активация, соответствующая дочернему узлу, должна быть завершена до того, как начнется активация его соседа справа. Пример такого дерева активации можно посмотреть в Ахо, 2-е изд., рис. 7.4., стр. 532.

Вызовы процедур и возвраты из них обычно управляются стеком времени выполнения, именуемым *стеком управления* (control stack). Каждая активная активация имеет *запись активации* (activation record), иногда именуемую *кадром* (frame), в стеке управления. На дне стека находится запись активации корня дерева активации, а последовательность записей активации в стеке соответствует пути в дереве активации от корня до текущей активации, в которой в настоящий момент находится управление. Запись последней по времени активации находится на вершине стека.

Как и в Ахо, 2-е изд. будем изображать стеки управления так, чтобы низ стека находился выше его вершины, так что элементы записи активации, находящиеся ниже на странице книги, на самом деле были ближе к вершине стека. Это несколько непривычно (обычно наверху находится вершина стека), но сделаем это для облегчения совместного чтения презентации и книги Ахо, 2-е изд.

Содержимое записи активации варьируется в зависимости от реализуемого языка программирования. Вот список данных, которые могут находиться в записи активации (элементы данных и их возможный порядок в стеке показаны на рис. 9.2).

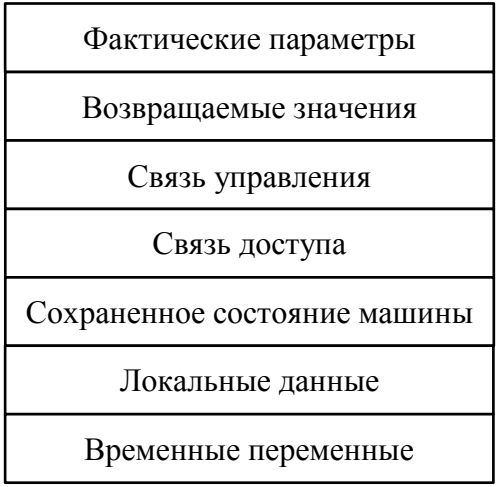

Рис. 9.2. Общий вид записи активации

1. Временные значения, появляющиеся, например, в процессе вычисления выражений, когда эти значения не могут храниться в регистрах.

2. Локальные данные процедуры, к которой относится данная запись активации.

3. Информация о состоянии машины непосредственно перед вызовом процедуры. Эта информация обычно включает адрес возврата (значение счетчика программы, которое должно быть восстановлено по выходу из процедуры) и содержимое регистров, которые использовались вызывающей процедурой и должны быть восстановлены при возврате из вызываемой процедуры.

4. Связь доступа может потребоваться для обращения вызванной процедуры к данным, хранящимся в другом месте, например в другой записи активации (см. разделе 7.3.5).

5. Связь управления указывает на запись активации вызывающей процедуры.

6. Память для возвращаемого значения вызываемой функции, если таковое имеется. Не все вызываемые процедуры возвращают значения, а кроме того, для большей эффективности возвращаемое значение может находиться не в стеке, а в регистре.

7. Фактические параметры, используемые вызываемой процедурой. Зачастую эти значения также располагаются не в стеке, а по возможности для большей эффективности передаются в регистрах. Однако в общем случае место для них отводится в стеке.

Вызовы процедур реализованы с помощью *последовательностей вызовов* (calling sequences), состоящих из кода, который выделяет память в стеке для записи активации и вносит информацию в ее поля. *Последовательность возврата* представляет собой аналогичный код, который восстанавливает состояние машины, так что вызывающая процедура может продолжать работу.

Последовательности вызовов и схема записей активации могут существенно отличаться даже в разных реализациях одного и того же языка. Код в последовательности вызова зачастую разделяется между вызывающей и вызываемой процедурами. Не существует четкого разграничения задач времени выполнения между вызывающей и вызываемой процедурами; исходный язык, целевая машина и операционная система налагают свои требования, в силу которых может оказаться предпочтительным то или иное решение. В общем случае, если процедура вызывается *n* раз, то часть последовательности вызова в вызывающих процедурах генерируется *n* раз. Однако часть последовательности вызова в вызываемой процедуре генерируется лишь единожды. Следовательно, желательно разместить как можно большую часть последовательности вызова в коде вызываемой процедуры – с учетом того, что известно вызываемой процедуре при ее вызове. Заметим, что вызываемая процедура не имеет доступа ко всей информации.

При разработке последовательностей вызовов и схемы записей активации помогают следующие принципы.

1. Значения, передаваемые между вызывающей и вызываемой процедурами, в общем случае помещаются в начале записи активации вызываемой процедуры, так что они максимально близки к записи активации вызывающей процедуры. Смысл этого заключается в том, что вызывающая процедура может вычислить значения фактических параметров вызова и разместить их на вершине собственной записи активации, что позволяет ей избежать создания полной записи активации вызываемой процедуры или даже знания ее схемы. Более того, это позволяет реализовать процедуры, которые принимают при каждом вызове различное количество аргументов, как, например, функция *printf* в языке программирования C. Вызываемая функция знает, где в ее записи активации следует разместить возвращаемое значение; ее же аргументы располагаются в стеке последовательно ниже этого места.

2. Элементы фиксированного размера обычно располагаются посредине. Как видно из рис. 9.2, эти элементы обычно включают связь управления, связь доступа и состояние машины. При каждом вызове сохраняются одни и те же компоненты состояния машины, так что сохранение и восстановление состояния для каждого вызова осуществляется одним и тем же кодом. Более того, при стандартизации информации о состоянии машины программы наподобие отладчиков смогут легко расшифровывать состояние стека при возникновении ошибки.

3. Элементы, размер которых может быть не известен заранее, размещаются в конце записи активации. Большинство локальных переменных имеет фиксированную длину, которая может быть определена компилятором исходя из типа переменной. Однако некоторые локальные переменные имеют размер, который не может быть выяснен до выполнения программы; наиболее распространенным примером является массив с динамическим размером, который определяется одним из параметров вызываемой процедуры. Кроме того, количество памяти, необходимой для временных переменных, обычно зависит от того, насколько успешно они распределяются по регистрам на стадии генерации кода. Таким образом, хотя объем памяти, необходимой для временных переменных в конечном счете оказывается известен компилятору, он может не быть известен в момент генерации промежуточного кода.

4. Следует ответственно подходить к вопросу помещения указателя на вершину стека. Распространенный подход состоит в том, чтобы он указывал на конец полей фиксированного размера в записи активации. Тогда обращение к данным фиксированной длины может осуществляться с использованием фиксированных смещений относительно указателя на вершину стека, известных генератору промежуточного кода. Следствием такого метода является размещение полей переменной длины записи активации «выше» вершины стека. Их смещения вычисляются во время выполнения программы, но обращение к ним выполняется также посредством указателя на вершину стека – просто значения смещений оказываются положительными.

Детали см. раздел 7.2 в Ахо, 2-е изд.
Вопросы доступа к нелокальным данным в стеке (раздел 7.3).
Управление кучей (раздел 7.4).