Проект пособия по курсу «Языки системного программирования»

Игорь Жирков
Кафедра вычислительной техники ИТМО

3 июля 2016 г.

Оглавление

Введение **9**

I Основы ассемблера 13

1 Базовая архитектура компьютера **15**
1.1 Архитектура фон Неймана 15
1.2 Развитие архитектуры фон Неймана 17
1.3 Регистры и локальность 19
1.3.1 Регистры общего назначения в Intel 64 21
1.3.2 Другие регистры прикладного назначения 23
1.3.3 Служебные регистры 25
1.4 Кольца защиты 25
1.5 Аппаратный стек 26

2 Основы ассемблера **29**
2.1 Hello, world 30
2.1.1 Взаимодействие с файлами в *nix 30
2.1.2 Организация кода 31
2.1.3 Основные команды 33
2.2 Вывод содержимого регистра 36
2.3 Вызов процедур 40
2.4 Endianness 43
2.5 Строки 45
2.6 Указатели 46
2.7 Предподсчёт констант 46
2.8 Адресация 46
2.9 Задание: Ввод-вывод 48

3 Наследие* 51
3.1 Реальный режим . . . 51
3.2 Защищённый режим . . . 53
3.3 Минимальная сегментация для Intel 64 . . . 57
3.4 RISC или CISC? . . . 59

4 Память 61
4.1 Принцип кэширования в широком смысле . . . 61
4.2 Виртуальная память . . . 62
4.2.1 Зачем нужна виртуальная память? . . . 62
4.2.2 Адресные пространства . . . 63
4.2.3 Пример: доступ к запрещенным адресам . . . 67
4.2.4 Эффективность . . . 68
4.3 Реализация виртуальной памяти . . . 69
4.3.1 Memory mapping . . . 73

5 Цикл компиляции 79
5.1 Препроцессинг . . . 80
5.2 Трансляция . . . 82
5.3 Компоновка . . . 83
5.3.1 Объектные файлы . . . 83
5.3.2 Релоцируемые объектные файлы . . . 86
5.3.3 Исполняемые объектные файлы . . . 91
5.4 Загрузка файла . . . 92
5.5 Динамические библиотеки . . . 93

6 Прерывания и системные вызовы 99
6.1 Прерывания . . . 99
6.2 Системные вызовы . . . 102

7 Модели вычислений 107
7.1 Конечные автоматы . . . 107
7.1.1 Определение . . . 107
7.1.2 Реализация на ассемблере . . . 110
7.1.3 Практическая ценность . . . 113
7.2 Forth-машина . . . 115
7.2.1 Архитектура . . . 115
7.2.2 Пример программы . . . 117
7.2.3 Словарь . . . 118
7.2.4 Компиляция в Forth . . . 123
7.3 Задание: компилятор и интерпретатор Forth . . . 124
7.3.1 Статический словарь, интерпретация . . . 124

7.3.2 Расширяемый словарь, компиляция 129
7.3.3 Bootstrap* 131

II Язык C **133**

8 Основные концепции языка **135**
8.1 Чем особен C? 135
8.2 Структура программ 136
8.2.1 Числовые типы данных 137
8.2.2 Пример программы 141
8.2.3 Предложения и выражения. lvalue и rvalue 143
8.2.4 Препроцессор 145
8.3 Типы данных в C 146
8.3.1 Обзор 146
8.3.2 Приведение типов 147
8.3.3 Массивы 148
8.3.4 Типы функций* 149
8.3.5 Типы-указатели 149
8.3.6 const-типы 150
8.3.7 Определения типа 151
8.3.8 Задание: суммирование массива 153
8.3.9 Задание: проверка числа на простоту 154
8.3.10 Структуры, перечисления, объединения 155
8.4 Типы данных в программировании* 158
8.4.1 Виды типизации 160
8.4.2 Полиморфизм 162
8.4.3 Полиморфизм в C 165
8.5 Функции, процедуры 165

9 Организация кода **167**
9.1 Объявление и определение 167
9.1.1 Функции и процедуры 167
9.1.2 Структуры. Рекурсивные типы данных 168
9.1.3 Неполные типы 169
9.2 Обращение к коду из других файлов 169
9.2.1 Обращение к данным из других файлов 172
9.2.2 Стандартная библиотека 172
9.2.3 Как связаны заголовочные файлы и библиотеки? 174
9.3 Использование препроцессора 174
9.3.1 Основные возможности 174

9.3.2 Include Guard . . . 176
9.3.3 Include Guard . . . 177
9.4 Побочные эффекты выражений . . . 179

10 Работа с памятью **181**
10.1 Указатели . . . 181
10.1.1 Зачем нужны указатели? . . . 181
10.1.2 Адресная арифметика . . . 182
10.1.3 Тип void* . . . 184
10.1.4 NULL . . . 184
10.1.5 Указатели на функции . . . 184
10.1.6 Массивы и указатели . . . 186
10.1.7 Детали синтаксиса . . . 187
10.2 Пример: программа, печатающая свои аргументы . . . 188
10.3 Пример: программа с указателем на функцию . . . 190
10.4 Модель памяти языка C . . . 191
10.4.1 Виды выделения памяти . . . 191
10.4.2 Задание: суммирование динамически созданного массива . . . 194
10.4.3 Строковые литералы . . . 194
10.4.4 Задание: связный список . . . 196
10.4.5 Ключевое слово static . . . 198
10.4.6 Задание: функции высшего порядка на списках . . . 200
10.5 Потоки данных . . . 202

11 Синтаксис, семантика и прагматика языков **207**
11.1 Синтаксис* . . . 208
11.2 Семантика . . . 211
11.2.1 Undefined behavior . . . 211
11.2.2 Unspecified behavior . . . 213
11.2.3 Точки следования . . . 213
11.3 Прагматика C . . . 214
11.3.1 Выравнивание . . . 214
11.3.2 Платформонезависимые типы данных . . . 216

12 Как писать на C **219**
12.1 Типы . . . 220
12.2 Переменные . . . 221
12.3 Функции . . . 222
12.4 Модули . . . 223
12.4.1 Сокрытие содержимого структур . . . 226
12.5 Организация файлов . . . 226

12.6 Иммутабельность или производительность 227
12.7 Задание: поворот картинки на 90 градусов 228
12.7.1 Формат BMP-файла 228
12.7.2 Архитектура 229
12.7.3 Бонусные задания 232
12.8 Задание: аллокатор памяти 232

III С и ассемблер 235

13 Детали трансляции 237
13.1 Соглашения вызова 237
13.1.1 Стековые фреймы 237
13.1.2 Поведение стека при вызове 239
13.1.3 Red zone 243
13.1.4 Переменное количество аргументов 244
13.1.5 vprintf 245
13.2 `volatile` 246
13.3 `setjmp` 249
13.3.1 `volatile` и `setjmp` 250
13.4 Классические уязвимости 254
13.4.1 Переполнение стекового буфера 255
13.4.2 return-to-libc 256
13.4.3 shellcode 256
13.5 Меры противодействия 256
13.5.1 Использование security cookie 257
13.5.2 Address space layout randomization 258
13.5.3 DEP 258

14 Производительность 259
14.1 SSE и AVX 259
14.1.1 Задание: сепия-фильтр для изображения 263
14.2 Оптимизации 266
14.2.1 Миф о скорости C 266
14.2.2 Как писать быстрый код? 268
14.2.3 Пропуск инициализации стековых фреймов 268
14.2.4 Концевая рекурсия 270
14.2.5 Common Subexpressions Elimination 274
14.2.6 Constant propagation 275
14.2.7 Return value optimization 276
14.3 Кэширование 282

14.3.1 Эффективное использование кэшей 282
14.3.2 Пример: инициализация матрицы 287

IV Приложение 291

15 Отладка в `gdb` 293
15.1 Отладка на уровне ассемблера 293

16 Сборка `make` 299

17 Некоторые системные вызовы 301

18 Информация о тестах производительности 303

Введение

Цели курса

Эта книга написана для того, чтобы:

- помочь понять принципы работы компьютера на уровне машинных команд;
- дать навыки программирования на языке C и языке Ассемблера, актуальные в настоящий момент;
- выработать системный взгляд на процесс написания, компиляции и запуска программ, а также на сущность языков программирования вообще;
- дать обзор архитектуры Intel 64, максимально очищенной от наследия старых режимов работы (реальный, виртуальный, защищённый).

Мы надеемся, что эта книга поможет сформировать базу знаний студента, с помощью которой он сможет легко ориентировать в области и самостоятельно развиваться. При составлении курса мы старались уйти от «игрушечности», чрезмерно тривиальных упражнений и заданий, слишком отдалённых по уровню сложности от реальных задач.

Структура книги

Книга состоит из трёх частей. Первая часть посвящена ассемблеру и той среде, в которой исполняется машинный код. Вторая часть посвящена в основном языку C. Разумеется, мы не претендуем на полноту описания языка (для этого есть надёжный и достоверный источник:

стандарт языка). Однако мы надеемся, что предложенный системный взгляд на язык C поможет максимально эффективно и полно освоить его. Систематизация особенно важна для таких старых языков, которые не были изначально придуманы цельно, и потому не всегда легко поддаются элегантному и лаконичному описанию. В третьей главе мы сконцентрируемся на аспектах, требующих понимания материала из обеих первых глав, таких, как отладка и низкоуровневые уязвимости программ.

Наличествуют также специальным образом помеченные вопросы. Они оставляются вам на самостоятельную проработку — вы обязательно столкнётесь с ними на защите лабораторных работ, контрольных работах и экзамене. Не пропускайте их – вы не сможете защитить лабораторные работы.

Также приведен список некоторых источников, полезных для самообразования в данной области. Знакомство с фундаментальными трудами почти всегда предпочтительно для получения глубоких знаний в любой области.

Жирным шрифтом выделены новые термины, курсивом – важные фрагменты текста.

Постольку поскольку темы зависят друг от друга, часто мы будем использовать ссылки между частями. В электронной версии книги на них можно кликнуть, чтобы перейти к тем материалам, на которые они указывают. Также в виде ссылок оформлены термины, ссылки ведут на их определения.

Мы будем стараться часто давать ссылки на источники по теме частей книги, которые рассказывают материал в расширенной форме и/или под другим углом, что тоже может быть полезно для лучшего запоминания.

Предварительные требования

Мы рассчитываем, что читатель знаком с двоичной и шестнадцатиричной системами счисления, основными битовыми операциями (И, ИЛИ, НЕ), а также понимает запись алгоритмов в псевдокоде, используя конструкции перехода (goto), условия (if-then-else) и циклы.

Контакты

Любые исправления, мнения и пожелания крайне приветствуются:

- e-mail: igorjirkov@gmail.com

- vk: id425000; группа, посвященная курсу: http://vk.com/spifmo

Часть I

Основы ассемблера

Глава 1

Базовая архитектура компьютера

1.1 Архитектура фон Неймана

В чём суть программирования? Интуитивно это обычно составление алгоритмов и их кодирование в вид, понятный компьютеру. Однако составить алгоритм из ничего нельзя — попробуйте составить алгоритм похода в университетскую столовую с третьего этажа! Каждый человек предложит свою версию, и все они будут одинаково бесполезны, пока никто чётко не зафиксировал набор базовых операций, из которых, собственно, можно построить алгоритм.

Модель вычислений это набор базовых операций, из которых строится алгоритм, и их стоимости

Стоимости позволяют оценивать сложность выполнения алгоритмов.

Большинство моделей вычислений являются **абстрактными вычислителями**, то есть описывают модель гипотетического компьютера в терминах входных данных, выходных данных и базовых операций. В этом курсе мы ограничиваемся только ими.

Теперь представим, что мы перенеслись на много лет назад, когда компьютеров еще не существовало. Существовала лишь необходимость каким-то образом организовать автоматический вычислительный процесс. Грубо говоря – сделать автоматизированный калькулятор. Естественно, разные люди на бумаге описали очень разные модели вычислений, например, машину Тьюринга и лямбда-исчисление.

Нужно было реализовывать эти модели в реальном мире, строя соответствующие машины (которые называются **конкретные вычислители**, в противовес абстрактным).

Поднявшись на немного более высокий уровень над моделями вычислений, мы для начала поговорим об архитектуре компьютера.

Архитектура компьютера — концептуальная структура вычислительной машины, определяющая процесс вычислений, а также то, как взаимодействует аппаратура и программное обеспечение.

Описание архитектуры более поверхностно по сравнению с описанием модели вычислений.

В связи с тем, что ранее электронные компоненты были не очень надёжны, а также со своей относительной простотой в создании и написании для неё программ прижилась **архитектура фон Неймана**.

Взглянем на схему компьютера с этой архитектурой:

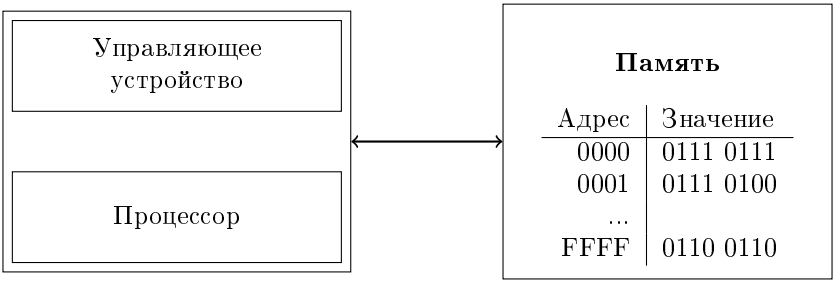


Рис. 1.1: Архитектура фон Неймана

Процессор умеет выполнять команды. **Память** хранит данные и команды, а **управляющее устройство** указывает процессору, какие команды выбирать из памяти для выполнения.

При выполнении команд процессор может читать и записывать в память.

Основные принципы архитектуры фон Неймана таковы:

Двоичное кодирование В памяти хранятся только нули и единицы.

Однородность памяти Никаким способом нельзя узнать, команда ли лежит в данной ячейке, или данные. Процессор может попытаться выполнить данные, что, скорее всего, вызовет ошибку, так как произвольные данные вряд ли соответствуют корректно

закодированной команде. Если же данные по стечению обстоятельств кодируют конкретную команду процессора, он выполнит её и ничего не заметит.

Адресуемость памяти Каждая ячейка имеет свой номер. Ячейки нумеруются последовательно: за нулевой ячейкой идёт первая и т.д. Единица адресации в реальных компьютерах — байт, равный восьми битам, то есть каждые 8 бит имеют свой адрес. Один бит это ноль или единица.

Последовательное управление Программа состоит из набора команд. Они выполняются последовательно.

Исключение – команды перехода, которые явно заставляют компьютер изменять порядок выполнения команд.

Языком ассемблера для того или иного процессора называется язык программирования, в котором каждой машинной команде этого процессора дано символьное имя. Это позволяет писать программы с помощью команд процессора, не переводя их самому в двоичное представление из нулей и единиц.

Жесткость архитектуры Все связи и блоки на схеме остаются на протяжении работы компьютера. Никаких новых связей не возникает.

Для полного описания абстрактного вычислителя здесь не хватает перечисления конкретных команд с их значением и способом кодирования.

Сейчас наша задача – схематично проследить, как из фон Неймановской основы вырос типичный современный персональный компьютер.

1.2 Развитие архитектуры фон Неймана

Во времена фон Неймана, компьютеры были проще, чем сейчас. К тому же они создавались сразу целиком: и память, и сетевые интерфейсы, и процессор. При таком подходе создатели намеренно не делали один компонент существенно быстрее, чем другой, так как это не имело смысла, лишь удорожая компьютер. Однако как только появились устойчивые архитектуры, разработчики аппаратного обеспечения стали концентрироваться на оптимизации частей компьютера по отдельности. Тут и возникли *узкие места в системе* – компоненты, которые

работали значительно медленнее и тормозили работу более быстрых. Иногда технологии бы и позволили пропорционально увеличить производительность тормозящих компонентов, но это было бы слишком дорого, и потому экономически нецелесообразно.

Так сложилась ситуация, когда архитектура фон Неймана в чистом виде оказалась малоэффективной. Более того, она обладает некоторыми фундаментальными недостатками, такими, как неинтерактивность. По этим причинам сегодняшние распространённые архитектуры обладают новыми чертами по сравнению с канонической моделью, описанной выше, хотя и являются её потомками.

Мы сконцентрируемся на описании архитектуры Intel 64, которой обладают большинство современных персональных компьютеров.

Архитектура Intel 64 является результатом многолетнего (с 1970-х) развития одной линейки процессоров. Каждая следующая модель старалась сохранить совместимость с предыдущей, поэтому в современных процессорах многие возможности остались еще от тех давних времён, когда размер оперативной памяти исчислялся килобайтами. Сохранились также разные устаревшие режимы работы: реальный, защищённый, виртуальный и т.д. *Мы будем описывать то, как работает процессор в самом новом, 64-разрядном режиме работы (**long mode**), если не оговорено иное.*

В современном процессоре существуют следующие важные расширения архитектуры фон Неймана, которые видны программисту:

Регистры это ячейки памяти, расположенные на процессоре. Обращение к ним не использует канал обмена с памятью и происходит быстрее. Подробнее см. 1.3.1

Стек вообще – это структура данных, контейнер, работающий по принципу Last in – first out. Часто её сравнивают со стопкой тарелок. Стек поддерживает две операции: `push` кладёт элемент на вершину стопки, `pop` вытаскивает элемент с вершины стопки. Аппаратная поддержка стека помогает реализовывать вызов функций и возврат из них в языках высокого уровня, а также выделение памяти для локальных переменных. Подробнее см. 1.5.

Прерывания позволяют прерывать последовательное исполнение программы по внешнему сигналу или в результате какой-то исклю-

чительной ситуации (деление на ноль, некорректно закодированная инструкция, попытка выполнить привилегированную инструкцию в непривилегированном режиме...). Они также помогают организовывать обмен данными с внешними устройствами. Подробнее см. 6.1

Кольца защиты Процессор постоянно находится в одном из состояний, называющихся кольца защиты. В простейшем случае этих состояний два: привилегированное и непривилегированное. В привилегированном состоянии процессор может исполнять любые команды, в непривилегированном попытка исполнить некоторые команды ведет к ошибке. Так аппаратно реализуется ограничение, по которому команды, необходимые для организации ввода и вывода и вообще взаимодействия с внешними устройствами, может исполнять только операционная система. Подробнее см. 1.4

Виртуальная память является уровнем абстракции над физической памятью, который помогает более эффективно распределять её между программами. Помимо этого она помогает изолировать выполняющиеся одновременно программы друг от друга и контролировать доступ к регионам памяти. Подробнее см. 4.2

Некоторые расширения архитектуры напрямую недоступны программисту. Это, например, различные кэши (кэш инструкций, кэши данных L1-L3, TLB и др.). Они прозрачно влияют на скорость выполнения программ, но не поддаются прямому контролю со стороны программиста, поэтому мы ограничимся их обзорным упоминанием.

Замечание. *Вы должны выработать в себе привычку при вопросах по работе процессора обращаться прежде всего к его документации [7]. См. список литературы для прямой ссылки. Мы часто будем явно отсылать читателя к ней.*

Важнейшая часть документации – второй том – содержит список всех поддерживаемых команд и их полное описание.

1.3 Регистры и локальность

Узкое место архитектуры фон Неймана – канал обмена между памятью и процессором. Для каждой команды процессору необходимо

не только обратиться в память, чтобы считать её, но и обращаться туда за операндами, возможно, также записывать результат. Каким бы быстрым ни был процессор, медленное взаимодействие с памятью будет снижать производительность компьютера.

Процессор снабдили малым количеством собственных ячеек памяти – регистров, которые расположены непосредственно на кристалле процессора и не задействуют канал обмена с памятью. Большую часть операций компьютер осуществляет именно с содержимым регистров. На данный момент разница в скорости доступа к регистрам и к оперативной памяти составляет десятки раз.

Со схемотехнической точки зрения регистры реализованы иначе, чем оперативная память: они существенно сложнее конструктивно. Отсюда их дороговизна, а потому заменить всё оперативную память на аналогичную регистровой экономически нецелесообразно.

Разумеется, использование регистров в худшем случае замедляет работу компьютера. Быстрее было бы совершить действие с ячейкой памяти, нежели еще копировать её содержимое в регистр и из регистра. Однако существует объективное свойство компьютерных программ — **локальность** — которое делает использование регистров оправданным. Оно эмпирическое, то есть программы просто пишут так, что они им обладают.

Локальность бывает двух видов: **пространственная** и **временная**.

Временная локальность – если мы обратились в память по некоторому адресу, то следующее обращение к этому адресу будет скорее всего в ближайшее время (а не с большим перерывом).

Пространственная локальность – если мы обратились в память по некоторому адресу, то в ближайшее время мы будем обращаться к близким ему адресам (отличающимся на небольшие числа).

В типичных программах основную часть времени мы работаем одновременно с очень маленьким набором переменных, большинство из которых смогут храниться в регистрах. Мы считаем данные в регистры однажды, поработаем с ними, а потом вернём результат в память. Данные, не лежащие в регистрах, будут требоваться нам редко, и в этих редких случаях мы проигрываем в производительности (придется освободить используемый регистр и записать в него потребовавшиеся данные). Однако в других случаях мы наоборот выиграем, причём так, что в среднем мы получим прирост производительности.

Замечание. *Это очень распространённая ситуация в инженерной практике. За счёт дополнительных механизмов мы ухудшаем производительность системы в худшем случае, но поднимаем её в среднем.*

1.3.1 Регистры общего назначения в Intel 64

Прежде всего, программист работает с **регистрами общего назначения** (РОН). Они универсальны и могут использоваться в очень многих командах. Часто если команда допускает указание одного из РОН, то любой другой также годится на его место.

Регистры общего назначения `r0`, `r1`, ..., `r15` имеют размер 64 бита.

Некоторые команды неявным образом используют тот или иной регистр. Альтернативные названия регистров отражают это их специализированное применение: например, регистр `rcx` (то же, что и `r1`) может использоваться не только в арифметических операциях, но и как счётчик для циклов в таких специальных командах, как `loop`. Это отражает буква `c` (cycle) в его названии. Разумеется, если команда неявно использует регистры, это будет описано в соответствующем ей разделе документации на процессор.

Обычно вместо `r0-r7` используют именно альтернативные имена, т.к. они исторически появились раньше .

Имя	Другое имя	Описание
`r0`	`rax`	Своего рода «аккумулятор». Например, команда целочисленного деления `div` принимает только один операнд: делитель, а в качестве делимого неявно берётся `rax`. Он не единственный может хранить результаты арифметических операций.
`r3`	`rbx`	Base register. В более ранних моделях процессоров базовую адресацию можно было осуществлять только от этого регистра, но теперь от любых.
`r1`	`rcx`	Используется для организации циклов, например, в команде `loop`.
`r2`	`rdx`	Хранит данные при операциях ввода и вывода.
`r4`	`rsp`	Адрес вершины аппаратного стека. Подробнее см. 1.5.

r5	rbp	Адрес начала текущего стекового фрейма. Подробнее см. 13.1.1.
r6	rsi	Адрес начала строки-источника в командах работы со строками
r7	rdi	Адрес начала строки-получателя в командах работы со строками
r8-r15	нет	Используются в основном для хранения временных переменных.

Таблица 1.1: 64-битные регистры общего назначения в Intel 64

Мы привели объяснения названий регистров для справочных целей. Пока запоминать их не нужно, но далее мы осознаем их значение.

В основном вы будете использовать регистры rax, rbx, rcx, rdx, rsp, rbp, rsi, rdi и r8-r15. Запомните, что rsp и rbp лучше не использовать, так как они имеют служебное назначение.

Вот отдельно соответствие между именами регистров:

r0	r1	r2	r3	r4	r5	r6	r7
rax	rcx	rdx	rbx	rsp	rbp	rsi	rdi

Существуют способы обращаться к меньшим частям РОН.

Для соглашения именования r0-r15 нужно добавить к имени регистра:

- d для double word – младших 32 битов;
- w для word – младших 16 битов;
- b для byte – младших 8 битов.

Например: r7b, r3w, r0d.

Альтернативные имена также позволяют обращаться к меньшим частям регистров. Вот соответствие:

- Младшие 32 бит:

r0d	r1d	r2d	r3d	r4d	r5d	r6d	r7d
eax	ecx	edx	ebx	esp	ebp	esi	edi

- Младшие 16 бит:

r0w	r1w	r2w	r3w	r4w	r5w	r6w	r7w
ax	cx	dx	bx	sp	bp	si	di

- Младшие 8 бит:

r0b	r1b	r2b	r3b	r4b	r5b	r6b	r7b
al	cl	dl	bl	spl	bpl	sil	dil

Наконец, можно обращаться к старшему байту регистров `ax`, `bx`, `cx`, `dx` как `ah`, `bh`, `ch`, `dh`.

Итак, схема:

```
0x 11 22 33 44 55 66 77 88 : rax = r0
               ========== : eax = r0d
                     ===== : ax  = r0w
                     ==    : ah
                        == : al  = r0b
```

Вопрос 1. *Какие другие имена есть у регистров: r4d, r8w, r1w, r3b, r4b, r2w?*

Существует тонкость, связанная с записью в части 64-битных РОН, которые перезаписывают 64-разрядные регистры полностью, что неочевидно. См. 3.4.

1.3.2 Другие регистры прикладного назначения

Программисту доступен 64-х разрядный регистр `rip` (аналогично его части – `eip` (32 бита) и `ip` (16 бит)). Он является счётчиком команд, т.е. хранит адрес следующей команды, которая будет выполнена. Во время выполнения любой команды он указывает на адрес, следующий за данной командой.

Замечание. *Команды в архитектуре Intel 64 бывают разного размера!*

Другим важным регистром является 64-х разрядный `rflags` (32-разрядная часть называется `eflags`, 16-ти разрядная часть – `flags`). В нём хранятся флаги, характеризующие текущее состояние выполнения программы.

Вопрос 2. *Обратитесь к документации процессора и узнайте, зачем нужны следующие флаги из `rflags`: CF, PF, AF, ZF, SF, OF.*

Регистры, доступные прикладному программисту, не ограничиваются упомянутыми выше. Помимо этого существуют регистры, использующиеся в операциях с числами с плавающей точкой, а также в специализированных командах, удобных в алгоритмах декодирования и трансформации изображений и звука. Это 16 128-битных регистров `xmm0-xmm15`. Для работы с ним нужны специальные команды, такие, как `movq`, а не обычный `mov`. В нашем курсе мы ограничимся работой с расширением SSE (Streaming SIMD Extension), позволяющим осуществлять, например, 4 сложения чисел с плавающей точкой одновременно.

Вопрос 3. *Узнайте про деление команд на SIMD, MIMD, SISD и MISD.*

Также есть набор регистров, зависящий от конкретной модели процессора (model specific register). См. 6.2.

Рис. 1.2: Intel 64

1.3.3 Служебные регистры

Некоторые регистры важны для работы операционной системы: они связаны с теми ресурсами, которые она должна контролировать (например, распределение физической памяти между программами или взаимодействие с внешними устройствами). Важно, чтобы программы не могли контролировать эти ресурсы в обход операционной системы, поэтому доступ на запись к таким регистрам из прикладных программ ограничен, см.1.4 .

cr0, cr4 хранят разнообразные флаги настройки системы, например, включение механизма виртуальной памяти;

cr2, cr3 используются в механизме виртуальной памяти, см. 4.2, 4.3;

cr8 или tpr указывает классы прерываний, которые не будут обрабатываться. Процессор вообще позволяет определить до 15 классов прерываний: от 1 (наименьшего) до 15 (наибольшего). Например, при `cr8` = 7 все возникающие прерывания класса 7 и ниже будут игнорироваться. См. 6.1.

cr1 зарезервирован.

efer – это еще один флаговый регистр, использующийся для управления Long-режимом и включения организации системных вызовов с помощью инструкции `syscall`.

idtr хранит адрес таблицы дескрипторов прерываний, см. 6.1.

gdtr и ldtr хранят адреса таблиц глобальных и локальных дескрипторов.

cs, ds, ss, es, gs, fs – так называемые **сегментные** регистры. На данный момент почти не используются. Подробнее см. 3.2

1.4 Кольца защиты

Кольца защиты (protection rings) это один из механизмов, ограничивающих выполнение программ в целях безопасности и надежности. Как концепция впервые они были придуманы при создании ОС Multics, предшественника Unix. Каждое кольцо защиты соответствует уровню привилегий; каждая команда может быть выполнена только на определенном уровне привилегий или еще более высоком уровне. В каждый момент времени мы выполняем код с определенным уровнем

привилегий. В Intel 64 четыре кольца защиты, ОС обычно умеет использовать только два: нулевое (самое привилегированное – для ядра и системного ПО) и третье (самое непривилегированное – для прикладного ПО). Первое и второе по задумке должны были предназначаться для драйверов, системного ПО и служб ОС, а нулевое – только для ядра ОС, однако подобные архитектуры ОС не прижились (см. экзоядро).

С точки зрения реализации, в 64-битном режиме номер текущего кольца защиты хранится в младших двух битах регистров `cs` и `ss`. Он может меняться в сторону больших привилегий при выполнении обработчика прерывания, если это описано в соответствующей записи **IDT**, или инструкции `syscall`. Младшие два бита `ds` задают уровень привилегий для доступа к данным (например, команды `mov`). В более старом защищённом 32-разрядном режиме работы была возможность использовать аппаратный механизм сегментации, чтобы делить адресное пространство на кусочки с разными привилегиями – сегменты. Сейчас можно считать, что каждая программа целиком лежит в одном сегменте, начинающемся с адреса 0. Использование сегментных регистров для задания уровня привилегий, однако, осталось и в 64-разрядном режиме, см. 3

1.5 Аппаратный стек

Вообще, стек это структура данных типа Last In — First Out, которая ведёт себя подобно стопке тарелок. Новый элемент с помощью команды push помещается на вершину стека, а с помощью команды pop достаётся последний помещённый в стек элемент (с вершины).

Нечто похожее на эту структуру данных реализовано аппаратно.

Память линейно адресуема, и она одна, значит, *никакой особенной стековой памяти нет.* Стек эмулируется с помощью машинных команд `push`, `pop` и регистра `rsp`. Регистр `rsp` хранит адрес вершины стека.

- `push` <параметр>
  1. `rsp` уменьшается на размер параметра (2, 4 или 8 байт);
  2. По адресу из регистра `rsp` записывается значение (содержимое регистра или число, в зависимости от параметра команды).
- `pop` <параметр>

1. В регистр, указанный в качестве параметра, записывается значение, прочитанное по адресу из `rsp`;
2. `rsp` увеличивается на размер параметра (2, 4 или 8 байт).

Рис. 1.3: Intel 64, регистры и стек

Несколько важных выводов:

1. Не бывает ситуации, когда «в стеке ничего нет», даже если «мы туда ничего не положили». Команда `pop` всегда будет следовать вышеуказанному алгоритму и выдавать ячейку, на которую указывал `rsp`);
2. Стек растёт к младшим адресам;
3. Существуют разные версии команд `push`, которые кладут в стек 2, 4 или 8 байт. *Почти всегда считается, что операнд знаковый.*

Например, при попытке положить в стек однобайтовое число $B9_{16}$ мы обнаружим, что старший знаковый бит равен 1, значит число отрицательное. Потому в стек мы положим число `0xff b9`, `0xffffffb9` или `0xff ff ff ff ff ff ff b9`.

По умолчанию для 64-разрядного кода считается, что операнды имеют размер 8 байт. Поэтому `push -1` положит в стек число `0xff ff ff ff ff ff ff ff`.

4. В разных архитектурах используется похожий принцип с регистром, который задает стек. Однако на одних архитектурах в этом регистре хранится адрес, куда следующая команда push положит новый элемент, а на других – адрес последнего элемента, который **уже** лежит в стеке.

Вопрос 4. *А как в нашем случае? В `rsp` адрес следующего элемента в стеке или последнего положенного в него элемента?*

Вопрос 5. *Когда маленький операнд при выполнении `push` расширяется нулями, а не знаковым битом? Обратитесь к документации на команду `push`.*

Глава 2

Основы ассемблера

– What time is it?
– Adventure time!

Finn & Jake

Среда

Из соображений единообразности мы будем использовать следующий инструментарий:

Операционная система Debian Linux x64. Мы предоставляем минимальный сконфигурированный образ виртуальной машины, в котором уже установлены все необходимые для работы утилиты. Логин `stud`, пароль `qwerty`, пароль суперпользователя `qwerty`.

```
> uname -sr
Linux 3.16.0-4-amd64
```

Почему *nix система? Практика показывает, что для неё наиболее просто писать приложения на ассемблере.

Компилятор ассемблера Nasm 2.11.05. Существует два основных синтаксиса языка ассемблера для изучаемой архитектуры: AT&T

и Intel версии. Набор машинных команд, конечно, один, но способы их именования – разные; порядок аргументов иногда тоже отличается. Nasm поддерживает синтаксис Intel, и именно его мы будем придерживаться.

Компилятор C gcc 4.9.2. Когда мы рассматриваем ассемблерный код, полученный из программ на C, мы используем именно эту версию для компиляции.

Система сборки GNU Make 4.0

Отладчик gdb 7.7.1

Текстовый редактор на ваше усмотрение, например, `vim`.

Информация о системе для тестов производительности находится в приложении 18

2.1 Hello, world

По традиции начнём изучение ассемблера с программы, выводящей на экран надпись “hello, world”.

Листинг 2.1: ./listings/hello.asm

```
1  global _start
2
3  section .data
4  message: db 'hello, world!', 10
5
6  section .text
7  _start:
8     mov   rax, 1      ; номер системного вызова write
9     mov   rdi, 1      ; дескриптор stdout
10    mov   rsi, message ; адрес строки
11    mov   rdx, 14     ; количество байт для записи
12    syscall
```

2.1.1 Взаимодействие с файлами в *nix

В идеологии Unix принят принцип “Everything is a file”. Под **файлом** имеется в виду любая последовательность байтов. Файлы, таким

образом, являются универсальной абстракцией для обращения к данным на диске, обменом данными между программами и с внешними устройствами. [17]

Операции ввода-вывода должны контролироваться операционной системой, т.к. они зависят от оборудования, которое, к тому же, может использоваться многими параллельно выполняющимися на процессоре программами. **Системные вызовы** это функции, которые являются частью кода операционной системы. Нам нужен системный вызов `write`, который умеет *записывать заданное количество байт* из памяти *начиная с указанного адреса* в *файл с заданным* **дескриптором** – уникальным числом, которое идентифицирует открытый файл или иной ресурс.

При запуске программы она сразу получает доступ к трём открытым файлам: `stdin`, `stdout`, `stderr`. Их дескрипторы всегда равны 0, 1 и 2 соответственно; `stdin` используется для ввода, `stdout` для вывода результатов работы программы, `stderr` для вывода сообщений об ошибках и диагностических сообщений – *как именно* работает программа.

Итак, программа должна вывести строчку "hello, world!", то есть записать её в файл с дескриптором 1.

Замечание. *Ассемблер нечувствителен к регистру букв (кроме имен глобальных меток, таких, как `_start`).*

2.1.2 Организация кода

Как мы помним, память общая для инструкций и данных, они неразличимы (и те и другие – нули и единицы). Однако программисту удобнее складывать в памяти данные отдельно от кода, чтобы не запутаться. Поэтому программы на ассемблере делят на **секции**. Каждая имеет своё предназначение: так в секции `.text` находятся инструкции, в `.data` – **глобальные данные**, доступные в любом месте программы. Описывать части любой секции можно в любом месте программы, написав `section name`, где name – имя секции.

Чтобы не работать напрямую с числовыми адресами, программисты используют **метки** – названия для адресов. В `nasm` они объявляются с помощью двоеточия, как, например, метки `_start` и `message`. Метки могут предшествовать любой команде.

Говоря про переменные в ассемблере мы имеем в виду адреса памяти, по которым лежат соответствующие им данные. Иначе говоря, переменные это метки.

Метка `_start` обязательно должна быть в одном из файлов с исходным кодом программы. Это **точка входа** – адрес инструкции, с которой начинается исполнение программы.

Кроме того, необходима строчка `global _start`, её значение мы изучим позднее.

Всё, что идёт после точки с запятой до конца строки – комментарий, который не несёт логической нагрузки.

В языке ассемблера каждая команда соответствует какой-то машинной инструкции, но не все ключевые слова являются командами. Эти слова называются **директивами**, т.к. они контролируют процесс создания исполняемого кода, но сами не являются им.

В примере директивами являются ключевые слова `global`, `section` и `db`.

Директива `db` создает байтовые данные (data byte). Можно задавать данные [15, секция 3.2.1]:

- `db` – байтами;
- `dw` – словами, или двойными байтами;
- `dd` – двойными словами, или по 4 байта;
- `dq` – учетверёнными словами, или по 8 байт;

Несколько примеров:

Листинг 2.2: ./listings/data_decl.asm

```
1  section .data
2    example1: db 5, 16, 8, 4, 2, 1
3    example2: times 999 db 42
4    example3: dw 999
```

`times` n `cmd` это директива, которая повторяет `cmd` n раз в коде программы. Её можно использовать не только для директив объявления данных, но и для инструкций процессора.

Вышеперечисленные директивы допускают объявление нескольких единиц данных сразу, они будут идти в памяти последовательно. Пример:

Листинг 2.3: Hello world

```
4  message: db 'hello, world!', 10
```

Буквы, цифры и другие символы кодируются с помощью таблицы ASCII, которая каждому символу сопоставляет его уникальный номер – **код символа**. Начиная с адреса, соответствующего метке `message`, в памяти расположены сначала коды символов строчки `"hello, world!"` по таблице символов ASCII, а затем один байт, равный 10. Почему 10? По таблице ASCII 10-й символ соответствует переводу строки, и именно он используется в *nix-системах чтобы показать, что последующий текст начинается с новой строки.

Замечание. *В терминологии Intel есть некоторая путаница, которая связана с тем, что вообще машинным словом называется «родной» для машины формат данных, завязанный на разрядность её регистров. Размер машинного слова на Intel 64 равен 64 бит, однако в документации словами называются двухбайтовые числа чтобы сохранить преемственность с терминологией 20-летней давности. Так 32-битные данные называют удвоенными словами, а 64-битные – учетверёнными словами.*

2.1.3 Основные команды

Теперь обратимся собственно к командам, которые составят суть программы.

Команда `mov` может записывать в регистр или память содержимое другого регистра, памяти или непосредственно заданное числовое значение. `mov` не может копировать из памяти в память. Она также требует, чтобы и операнд-источник, и операнд-получатель были одинакового размера.

С помощью команды `syscall` организован механизм **системных вызовов** в *nix системах. Системные вызовы это функции, которые являются частью кода операционной системы. Операции ввода-вывода зависят от оборудования, которое, к тому же, может использоваться многими параллельно выполняющимися на процессоре программами, поэтому нельзя давать программистам контролировать оборудование напрямую – их взаимодействие с ним должно проходить через операционную систему.

Каждому системному вызову соответствует уникальный номер. Чтобы осуществить системный вызов необходимо:

1. Поместить в `rax` его номер;
2. Поместить в регистры параметры. Каждый системный вызов требует свой набор параметров. Следующие регистры используются по порядку для параметров: `rdi`, `rsi`, `rdx`, `r10`, `r8` и `r9`.

Системные вызовы не могут принимать больше шести аргументов;

3. Выполнить инструкцию `syscall`.

Заметим, что после выполнения инструкции `syscall` значения регистров `rcx` и `r11` непредсказуемо изменятся. Пока что механизм, по которому организованы системные вызовы, будет для нас чёрным ящиком: более детально инструкцию `syscall` мы изучим позднее.

Какие существуют системные вызовы и какие они принимают параметры? Существует несколько способов это узнать; пока что мы будем пользоваться сторонней документацией, например, [5]

Что за системный вызов был использован в примере? он называется `write`, и принимает следующие параметры:

1. Файловый **дескриптор**;
2. Адрес, начиная с которого лежит массив данных, который необходимо в соответствующий файл записать;
3. Количество байтов, которое нужно записать в файл.

Чтобы скомпилировать пример, сохраните его в файле `hello.asm` и введите:

```
> nasm -felf64 hello.asm -o hello.o
> ld -o hello hello.o
> chmod u+x hello
```

Смысл этих команд будет ясен после прочтения раздела 5 (Цикл компиляции).

Запускаем:

```
> ./hello
hello, world!
Segmentation fault
```

Хорошо, что мы вывели нужную строчку, но программа аварийно завершила работу, о чем говорит сообщение Segmentation fault. В чем причина?

После завершения выполнения системного вызова, порожденного `syscall`, программа будет продолжать свою работу. После команды `syscall` других инструкций мы не писали, но в памяти по следующему адресу лежат какие-то байты данных. Какие – неизвестно, их значение случайно. Процессор никаким способом не может понять, что это мусор, и так как `rip` указывает на них, он просто интерпретирует содержимое этих ячеек как команды и попытается их выполнить. Очень быстро он наткнётся на неправильно закодированную команду среди этого мусора (скорее всего – сразу), сгенерируется прерывание 6 (неправильный код операции). Другая возможность – в том, что он выйдет за границы выделенных адресов (см. 4.2). В конце концов его обработчик (являющийся частью ОС) аварийно завершит работу программы.

Что делать? существует специальный системный вызов `exit`, который нам и нужно вызвать, чтобы корректно завершить программу.

Вопрос 6. *В чем смысл строчки* `xor rdi, rdi`*?*

Вопрос 7. *Что такое код возврата?*

Вопрос 8. *Что принимает своим первым аргументом системный вызов exit?*

Листинг 2.4: ./listings/hello_proper_exit.asm

```
section .data
message: db 'hello, world!', 10

section .text
global _start

_start:
   mov   rax, 1      ; номер системного вызова write
   mov   rdi, 1      ; дескриптор stdout
   mov   rsi, message ; адрес строки
   mov   rdx, 14     ; длина строки
   syscall
   .lab:
   jmp .lab

  mov   rax, 60     ; номер системного вызова close
   xor   rdi, rdi
   syscall
```

Теперь при запуске программа корректно завершает свою работу.

2.2 Вывод содержимого регистра

Попробуем более сложный пример: выведем на экран содержимое регистра `rax`.

Листинг 2.5: ./listings/print_rax.asm

```
section .data
codes:
   db    '0123456789ABCDEF'

section .text
global _start
_start:
   ; число 1122... в 16-ричной системе счисления
   mov rax, 0x1122334455667788

```

```
11      mov rdi, 1
12      mov rdx, 1
13      mov rcx, 64
14      ; Каждые 4 бита нужно вывести на экран как одну 16-ричную цифру.
15      ; Для этого с помощью сдвига и побитового И с маской 0xf выделим
16      ; каждую тетраду, и используем её как смещение относительно
17      ; метки codes
18    .loop:
19      push rax
20      sub rcx, 4
21      ; cl это однобайтовый регистр, часть rcx
22      ; rax -- eax -- ax -- ah + al
23      ; rcx -- ecx -- cx -- ch + cl
24      sar rax, cl
25      and rax, 0xf
26      lea rsi, [codes + rax]
27      mov rax, 1
28
29      ; при выполнении syscall регистр rcx не сохраняется
30      push rcx
31      syscall
32      pop rcx
33
34      pop rax
35      ; проверка на 0 быстрее с помощью команды test, нежели cmp
36      test rcx, rcx
37      jnz .loop
38
39      mov   rax, 60       ; для вызова close
40      xor   rdi, rdi
41      syscall
```

Мы выделяем из большого числа 16-ричные цифры по отдельности с помощью сдвига `rax` и логического И с маской `0xf`. Каждая 16-ричная цифра – это число в диапазоне [0, 15], его можно прибавить к адресу `codes` и получить адрес, по которому в памяти находится код символа, изображающего соответствующую цифру.

Стек мы можем использовать для сохранения и восстановления регистров, как рядом с инструкцией `syscall`.

Вопрос 9. *Чем отличаются команды `sar` и `shr`?*

Вопрос 10. *Как задавать числа в разных системах счисления (десятичной, шестнадцатиричной и двоичной)? Обратитесь к документации на nasm.*

Замечание. *При начале выполнения программы значение большинства регистров неопределено, т.е. может быть вообще любым.*

Локальные метки

Обратите внимание на метку `.loop` – она начинается с точки. Такие метки локальны, то есть они могут повторяться в разных частях программы, при этом не будет возникать конфликта имен. Реализовано это так: последняя введенная метка без точки служит «базовой» для меток с точкой. Имя метки `.loop` в примере выше, на самом деле – `_start.loop`. Именно по этому имени к ней можно обратиться из любого места программы, даже из тех, которые связаны с другими нелокальными метками, не `_start`).

Относительная адресация

Строчка ниже демонстрирует возможность более сложной адресации, нежели простого указания адреса.

Листинг 2.6: Относительная адресация

```
26      lea rsi, [codes + rax]
```

Квадратные скобки означают **косвенную адресацию**, то, что внутри них, считается адресом. `mov rsi, rax` копирует содержимое `rax` в `rsi`, а `mov rsi, [rax]` копирует в `rsi` содержимое памяти (8 последовательно идущих байт) начиная с адреса, хранящегося в `rax`.

Команда `lea` похожа на `mov`. Её мнемоника означает load effective address. Она позволяет подсчитать адрес ячейки памяти и сохранить его в регистре. Это не совсем тривиальная операция, ведь адрес ячейки может быть закодирован в команде не только числом, но и суммой двух компонент (при относительной адресации) или еще более сложным образом.

Чтобы лучше прочувствовать различие между `lea` и `mov`:

Листинг 2.7: ./listings/lea_vs_mov.asm

```
; rsi <- адрес метки codes, число
mov rsi, codes

; rsi <- содержимое памяти по адресу codes
mov rsi, [codes]

; rsi <- адрес метки codes
; почти то же, что и mov rsi, codes, но в более
; общем случае адрес может состоять из многих
; компонент, а не только быть константным числом
lea rsi, [codes]

; rsi <- содержимое памяти по адресу codes+rax
mov rsi, [codes + rax]

; rsi <- сумма codes и rax
lea rsi, [codes + rax]
```

Команды перехода

Стандартный способ организовывать переходы между частями программы в зависимости от условий – использование команд `test` и `cmp`. Команды перехода срабатывают или не срабатывают в зависимости от установленных флагов. Например, команда `jz address` осуществляет переход по адресу, если установлен флаг Zero Flag. Команда `cmp` вычитает один операнд из другого, но никуда не записывает результат. Однако она устанавливает флаги в зависимости от результата, которые уже будут использовать команды перехода. Команда `test` отличается лишь тем, что производит побитовое И двух операндов и устанавливает флаги, никуда не записывая результат. Команда `test reg, reg` установит флаг ZF только если `reg` = 0, что является быстрым и общепринятым способом проверить, хранится ли в регистре ноль.

Для каждого арифметического флага существуют две команды: jF и jnF, где F – первая буква названия флага. Например, для sign flag: `js` и `jns`. Кроме того полезными являются команды `ja` (jump if above) / `jb` (jump if below) для прыжка после сравнения *беззнаковых чисел* с помощью `cmp`, и `jg` (jump if greater) / `jl` (jump if less) аналогично для знаковых.

Также для проверки на нестрогое неравенство используются такие команды, как `jae` (jump if above or equal), `jle` (jump if less or equal) и им подобные.

Листинг 2.8: ./listings/jumps.asm

```
1  mov rax, -1
2  mov rdx, 2
3
4  cmp rax, rdx
5  jg location
6  ja location   ; разная логика!
7
8  cmp rax, rdx
9  je location   ; если rax равен rdx
10 jne location  ; если rax не равен rdx
```

Вопрос 11. *В чем разница между командами `je` и `jz`?*

Листинг 2.9: Условный переход

```
36      test rcx, rcx
37      jnz .loop
```

В следующем примере мы покажем, как работает механизм вызовов процедур.

2.3 Вызов процедур

Процедуры позволяют изолировать кусочек логики программы и использовать его как черный ящик. Необходимые для его работы данные передают через аргументы. Процедуру можно вызвать с определёнными аргументами, после её работы выполнение программы должно продолжиться с инструкции, которая лежит в памяти следующая после инструкции её вызова.

Для вызова процедуры используют инструкцию `call <address>`, которая комбинирует действия:

```
1  push rip
2  jmp <address>
```

Положенный в стек адрес называется **адрес возврата**.

Концом процедуры можно считать инструкцию `ret`, совершающую действие `pop rip`.

Как можно догадаться, работать этот механизм будет только в том случае, если стек после вызова `call` был в том же состоянии, что и при вызове соответствующего ему `ret`, иначе мы достанем не адрес возврата, а что-то другое. Об этом должен заботиться сам программист.

Аргументы процедура получает через регистры `rdi`, `rsi`, `rdx`, `rcx`, `r8`, `r9`, именно в таком порядке.

Естественно, выполнение функции меняет содержимое регистров. Существуют два типа регистров:

- **Callee-saved регистры** должны сохраняться вызванной процедурой, если ей нужно их поменять. Иначе говоря, процедура не имеет права по завершению её выполнения оставлять эти регистры в измененном состоянии.

  Это `rbx`, `rbp`, `rsp`, `r12-r15`

- **Caller-saved регистры** должны быть сохранены частью программы, которой вызывается процедура.

  Другими словами, эти регистры любая процедура может изменить. Если вам важны их значения, сохраните их перед тем, как исполнять `call`!

  Все не перечисленные в предыдущем пункте регистры могут быть разрушены вызванной процедурой.

Некоторые процедуры могут **возвращать значение** – такие процедуры мы будем называть **функциями**. Это значение обычно является смыслом её вычислительного процесса. Например, можно написать функцию, которая принимает число и возвращает его квадрат.

С точки зрения реализации, *значения мы будем возвращать, записывая их в регистр* `rax`. Поместили аргументы в регистр, запустили функцию с помощью `call` – по возвращении из неё в `rax` записано возвращаемое значение.

Некоторые системные вызовы тоже возвращают значение – внимательно читайте документацию на них.

Использовать регистры `rbp` и `rsp` для своих целей можно только очень осторожно – они имеют служебное назначение. Регистр `rsp` указывает, как вы уже знаете, на вершину стека.

Замечание. *Обратите внимание: для системных вызовов аргументы передаются по-другому! Четвёртый аргумент системному вызову передаётся через `r10`, а в обычную процедуру – через `rcx`.*

Замечание. *Процедуре мы можем передать больше, чем 6 аргументов (излишки положим в стек в обратном порядке), а для системного вызова не поддерживается более 6 параметров.*

Обычно важно сохранять нужные вам caller-saved регистры перед вызовом функции, даже если то, что вы вызываете, написали вы сами на ассемблере.

Следование соглашениям вызова позволяет в любой момент подменить одну реализацию процедуры другой.

Мы напишем две процедуры: `print_newline`, которая будет выводить на экран символ перевода строки, и `print_hex`, которая будет принимать один аргумент (число) и выводить его на экран.

Листинг 2.10: ./listings/print_call.asm

```
section .data

newline_char: db 10
codes: db '0123456789abcdef'

section .text
global _start

print_newline:
   mov rax, 1       ; номер write
   mov rdi, 1       ; дескриптор stdout
   mov rsi, newline_char ; откуда брать данные
   mov rdx, 1       ; сколько байт записать
   syscall
  ret

print_hex:
   mov rax, rdi

   mov rdi, 1
   mov rdx, 1
   mov rcx, 64      ; на сколько сдвинуть rax вправо?
iterate:
   push rax         ; Сохраним исходное значение rax
   sub rcx, 4
   sar rax, cl      ; Вычленим очередные 4 бита, сдвинув их вправо
                    ; Регистр cl это младший байт rcx
```

```
28      and rax, 0xf     ; Вырежем все биты слева от младших четырёх
29      lea rsi, [codes + rax]; По этому адресу находится код символа
30                       ; для представления четырёх бит.
31      mov rax, 1       ;
32
33      push rcx         ; syscall разрушит rcx
34      syscall          ; rax = 1 (31) -- код write, rdi = 1, stdout,
35                      ; rsi = адрес кода символа
36      pop rcx
37
38      pop rax          ; см. строчку 24
39      test rcx, rcx    ; rcx = 0 когда выведены все четверки бит
40      jnz iterate
41
42      ret
43
44  _start:
45      mov rdi, 0x1122334455667788
46      call print_hex
47      call print_newline
48
49      mov rax, 60
50      xor rdi, rdi
51      syscall
```

2.4 Endianness

Попробуем вывести значение, сохранённое в памяти, с помощью написанной функции. Для этого необходимо сначала положить его в регистр.

Листинг 2.11: endianness.asm

```
43  demo1: dq 0x1122334455667788
44  demo2: db 0x11, 0x22, 0x33, 0x44, 0x55, 0x66, 0x77, 0x88
45
46  section .text
47
48  _start:
49      mov rdi, [demo1]
```

```
50      call print_hex
51      call print_newline
52
53      mov rdi, [demo2]
54      call print_hex
55      call print_newline
56
57      mov rax, 60
58      xor rdi, rdi
59      syscall
```

При запуске мы получим два разных результата для demo1 и demo2.

```
> ./main
1122334455667788
8877665544332211
```

Получается, что положив в память число размером 8 байт, мы на деле записываем его байты в обратном порядке!

Разные процессоры по-разному относятся к тому, какой должен быть порядок байт в данных, лежащих в памяти.

Big endian многобайтовое число хранится в памяти в прямом порядке, т.е. начиная со старших байтов;

Little endian по меньшим адресам хранятся менее значимые байты.

Как видно из примера, архитектура Intel 64 подразумевает порядок Little Endian. В целом выбор между этими двумя соглашениями – вопрос договорённости.

Little endian не влияет на то, как в памяти хранятся строки текста – в строке текста единица данных это байт. Однако если каждый символ будет кодироваться двумя байтами, то внутри кода одного символа байты будут переставлены.

Плюс Little endian в том, что фрагмент числа большого формата, содержащий только младшие разряды, расположен по тому же адресу.

Например, пусть `demo3: dq 0x1234` . Тогда чтобы перевести это число в формат `dw` нужно просто считать двухбайтовое число по тому же адресу `demo3`.

Адрес	Значение – LE	Значение – BE
demo3	0x34	0x00
demo3 + 1	0x12	0x00
demo3 + 2	0x00	0x00
demo3 + 3	0x00	0x00
demo3 + 4	0x00	0x00
demo3 + 5	0x00	0x00
demo3 + 6	0x00	0x12
demo3 + 7	0x00	0x34

Таблица 2.1: Little Endian и Big Endian для числа 0x1234

Big endian часто используется при передаче данных по сети, например, в пакетах TCP/IP. Также это внутренний формат представления чисел для виртуальной машины Java.

Существует также понятие **Middle endian**. К примеру, мы хотим организовать программно арифметику 128-битных чисел, которые будут представляться как два 64-битных числа каждое. В таком случае байты числа будут расположены так: сначала младшие 8 байт в обратном порядке (первая компонента), потом старшие 8 байт в обратном порядке (вторая компонента).

2.5 Строки

Символы, как мы уже знаем, кодируются с помощью таблицы символов, например, ASCII. Каждому символу ставится в соответствие его код, и строчка хранится в памяти как последовательность кодов её символов. Однако как определить её границы?

1. Строка может задаваться своей длиной и символами, последовательно лежащими в памяти.

```
1   db 27, 'Selling England by the Pound'
```

2. Выделен специальный символ, который означает окончание строки и который не может в ней встречаться. Строка задаётся адресом начала. Традиционно в качестве признака конца строки берут символ с кодом 0x00. Строки в этом случае называются **нуль-терминированными** (null-terminated).

2.6 Указатели

Указатели это адреса памяти. Они могут храниться в регистрах или других ячейках памяти. Размер указателя на Intel 64, очевидно, 8 байт. Не всегда понятно, однако, каков размер данных, на которые мы указываем? К примеру, записывая непосредственное значение в память, мы получим ошибку компиляции если не уточним размер операнда так, как в этом примере:

```
1  section .data
2  test: dq -1
3
4  section .text
5
6  mov byte[test], 1 ;1
7  mov word[test], 1 ;2
8  mov dword[test], 1 ;4
9  mov qword[test], 1 ;8
```

Вопрос 12. *Чему будет равняться* `test` *после выполнения каждой из команд из этого примера? Запишите число в 16-ричной системе счисления.*

2.7 Предподсчёт констант

Вы можете нередко встретить код вида:

```
1  lab: db 0
2  ...
3    mov rax, lab + 1 + 2*3
```

Nasm поддерживает арифметические выражения со скобками и разными битовыми операторами, в которых могут участвовать только константы, известные на этапе компиляции. Они подсчитаются во время компиляции, и в исполняемый код попадут только результаты. Выражение выше НЕ будет считаться во время выполнения программы.

2.8 Адресация

Как задаются операнды в командах?

1. Непосредственно;

   Команда сама по себе лежит в памяти. Естественно, операнды являются её частями, закодированными каким-то способом. У этих частей, как и у команды, есть адрес, по которому они лежат в памяти. Когда операнд берется изнутри самой команды «как есть», говорят о **непосредственной адресации**.

   Так мы переместим в регистр `rax` число 10.

   ```
   1  mov rax, 10
   ```

2. Через регистр;

   Так мы переместим в регистр `rax` содержимое регистра `rbx`

   ```
   1  mov rax, rbx
   ```

3. Косвенно (direct memory addressing);

   Так мы переместим содержимое 8 байтов начиная с адреса 10 в регистр `rax`:

   ```
   1  mov rax, [10]
   ```

   Мы также можем использовать регистр:

   ```
   1  mov r9, 10
   2  mov rax, [r9]
   ```

   Памятуя о возможностях `nasm` по подсчёту константных выражений во время компиляции, мы можем быть уверены, что в следующем коде в машинную команду будет включен именно адрес как число:

   ```
   1  buffer: dq 8841, 99, 00
   2  ...
   3  mov rax, [buffer+8]
   ```

4. Базово-индексная со смещением и масштабированием.

   Остальные интересные виды адресации являются частными случаями базово-индексной со смещением и масштабированием. В ней адрес состоит из следующих компонент (некоторые из которых могут отсутствовать:

   Адрес = база + индекс * масштаб + смещение

- База задана непосредственно адресом или в регистре;
- Масштаб это число 1, 2, 4 или 8;
- Индекс задан непосредственно или через регистр;
- Смещение задано как непосредственное число.

Некоторые примеры:

Листинг 2.12: ./listings/addressing.asm

```
1  mov rax, [rbx + 4* rcx + 9]
2  mov rax, [4*r9]
3  mov rdx, [rax + rbx]
4  lea rax, [rbx + rbx * 4] ; rax = rbx * 5
5  add r8, [9 + rbx*8 + 7]
```

2.9 Задание: Ввод-вывод

В ходе первых двух лабораторных работ мы сделаем интерпретатор диалекта языка Forth.

Прежде всего, однако, нам нужно написать некоторое количество функций для ввода-вывода. Пока что мы не умеем даже считывать числа с клавиатуры. Это будет разминкой перед немного более серьёзным заданием.

1. Изучите документацию на команды:

   - `xor`
   - `jmp`, `ja` и другие команды переходов
   - `cmp`
   - `mov`
   - `inc`, `dec`
   - `add`, `imul`, `mul`, `sub`, `idiv`, `div`
   - `neg`
   - `call`, `ret`
   - `push`, `pop`

   и на системный вызов `read` (его код 0, а в остальном он похож на `write`).

2. Изучите содержимое файлов, создаваемых скриптом `test.py`. Что означает строчка `%include "lib.inc"` ?

3. Дайте определения следующим функциям в файле `lib.inc`:

`string_length`	Принимает указатель на строку, возвращает её длину.
`print_string`	Вывод нуль-терминированной строки, первый аргумент – указатель a на неё;
`print_char`	Аргумент – символ, который нужно вывести в `stdout`.
`print_newline`	Перевод на новую строку (вывод символа с кодом 0xA);
`print_uint`	Вывод беззнакового 8-байтового числа в десятичной системе счисления. Можно выделить буфер в стеке на фиксированное количество байт, делить выводимое значение на 10 и записывать в буфер коды цифр остатков. Делать это удобно справа налево (от больших адресов к меньшим, соответственно). Затем просто вывести заполненную часть буфера.
`print_int`	Вывод знакового 8-байтового числа в десятичной системе счисления.
`read_char`	Прочитать с потока ввода один символ
`read_word`	Прочитать с потока ввода следующее слово, пропустив перед ним произвольное количество пробельных символов [1]. Слово это последовательность непробельных символов. Слово следует сохранить в буфере в **секции** `.data` как нуль-терминированную строку. Гарантируется, что его размер не превышает 255 символов. В `rax` вернуть указатель на буфер, в `rdx` длину прочитанного слова.
`parse_uint`	Прочитать из буфера в памяти беззнаковое число. `rax` = число, `rdx` = длина прочитанного числа. Например, `rdi` указывает на строку 12343sdsw12, результат: `rax` = 12343, rdx = 5
`parse_int`	Прочитать из буфера в памяти знаковое число. `rax` = число, `rdx` = длина прочитанного числа.

[1] Пробельные символы это переносы строк, пробелы и табуляция (код символа табуляции 0x09)

`string_equals`	Принимает два указателя на строки, сравнивает их посимвольно. Если они равны, то возвращает 1, иначе 0.
`string_copy`	Принимает указатель на строку и указатель на место в памяти. После выполнения по второму адресу лежит копия первой строки.

Порядок, выбранный в таблице, не случаен: многие функции можно выражать через уже написанные.

Обязательно пользуйтесь локальными метками.

Скрипт `test.py` создаёт и компилирует многочисленные файлы с тестами вашего кода и запускает их. Для каждого теста вы увидите результат: `ok` или `fail`, а также входные данные для него.

Не забудьте соблюдать правила передачи аргументов, а также сохраняйте нужные вам caller-saved регистры перед вызовом функции – это один из основных источников ошибок.

Не используйте буферы в `.data`, кроме как для функции `read_word`.

Выводите числа не посимвольно, а с помощью `print_string` (минус можно вывести с помощью `print_char`).

Примерный объём кода в `lib.inc` – 220 строк.

Самопроверка

Прежде всего проверьте следующие вещи про каждую функцию в вашей лабораторной:

- Все необходимые в дальнейшем caller-saved регистры сохраняются перед `call`, восстанавливаются после него.
- Все callee-saved регистры сохраняются в начале функции и восстанавливаются в конце.
- `rdx` правильно устанавливается в `parse_int` и `parse_uint`

Глава 3

Наследие*

Из-за долгого эволюционного развития, некоторые возможности процессора уже практически не используются или используются всё меньше.

Процессор может работать в реальном, защищённом, специальном, виртуальном и long-режимах.

Специальный режим — режим сна, который вы, скорее всего, часто используете. Для студентов характерно работать в специальном режиме.

Виртуальный режим (режим «эмуляции реального») нас не очень интересует.

Long-режим мы используем в этой книге как основной. Вкратце опишем некоторую специфику реального и защищённого режима работы.

3.1 Реальный режим

Исторически первым режимом работы был реальный режим работы, в котором не было виртуальной памяти, физическая память адресовалась напрямую, а все немногочисленные регистры общего назначения были 16-разрядными.

Регистров в 16 бит достаточно, чтобы задавать числа от 0 до 65535, значит, всего с помощью регистра мы можем адресовать 65536 байт. Такая область памяти называлась тоже **сегмент**. Не путайте её с сегментами, которые описываются в ELF файлах, или сегментами защищённого режима!

Главным образом использовались следующие регистры:

- `ip`, `flags`;
- `ax`, `bx`, `cx`, `dx`, `sp`, `bp`, `si`, `di`;
- `cs`, `ds`, `ss`, `fs`, `gs`, `es`.

Оперировать более, чем с $2^{16} = 65536$ байтами памяти, было затруднительно. **Сегментные регистры** использовались для того, чтобы адресовать больше, чем 2^{16} байт памяти следующим образом:

- Все физические адреса были 20-разрядные;
- Каждый логический адрес в программе состоял из двух 16-разрядных компонент: адреса начала сегмента и смещения внутри него. Этот вид адреса преобразовывался аппаратурой в физический 20-разрядный вот так:

  ```
  физический адрес = начало сегмента * 16 + смещение
  ```

  Наиболее распространенной была запись адреса как `сегмент:смещение`, например, `4a40:0002`, `ds:0001`, `7bd3:ah`.

  Как мы уже говорили, программисты стремятся разнести код и данные в разные места памяти, не перемешивая их. Сегментные регистры имеют соответствующую специализацию: `cs` хранит адрес начала сегмента кода, `ds` соответствует сегменту данных, `ss` – стеку, а остальные сегментные регистры используются для дополнительных сегментов данных.

  Обратите внимание: строго говоря компонента сегмента это *не адрес начала сегмента*, а его кусочек (старшие четыре 16-разрядные цифры). Дописав еще одну цифру 0 мы, таким образом, умножим его на 16 и получим 20-разрядный линейный адрес начала сегмента, к которому уже затем прибавим смещение.

  Каждая команда, использующая 16-разрядный адрес, неявно подразумевала, что есть еще одна его компонента, *которая хранится в сегментном регистре*. Каком именно? Это варьировалось в зависимости от команды (и могло быть явно переопределено). Документация, конечно, явно оговаривает этот момент для каждой команды.

  Например, команда `mov` по умолчанию подразумевала работу с данными, и потому адрес `0004` отсчитывался от начала сегмента данных, получаемого с помощью `ds`:

```
1   mov al, [0004] ; === mov al, ds:0004
```

Но можно было бы переопределить начало сегмента только для конкретного адреса:

```
1   mov al, cs:[0004]
```

При запуске программы загрузчик устанавливал регистры `ip`, `cs`, `ss`, `sp` так, чтобы адрес начала сегмента кода совпал с началом области кода в памяти, `cs:ip` указывал на точку входа, а `ss:sp` – на вершину стека.

Реальный режим не устраивал программистов по нескольким причинам:

- Никакой поддержки многозадачности. Программы (если и не одна) должны были загружаться по разным адресам, так как адресное пространство было одно на все программы (включая ОС);
- Программы могут случайно переписать друг друга или ОС, так как память одна на всех;
- Любая программа может выполнить любую инструкцию, то есть, потенциально, любая программа может порушить всю систему.

Эти причины привели к появлению защищённого режима.

3.2 Защищённый режим

В Intel 80286 впервые появился защищённый 32-разрядный режим работы, с регистрами `eax`, `ebx`, ..., `esi`, `edi`.

В нём появились некоторые механизмы *защиты* программ от ошибок: кольца защиты, виртуальная память, а также более совершенный механизм сегментации. Благодаря им существенно усложнилось написание вредоносного кода. Кроме того, изолированные друг от друга и от ОС программы не рушили работу системы даже в случае аварийной остановки или при попытке изменить область памяти, которая им не принадлежит.

По сравнению с реальным режимом, изменился способ получения начала адреса сегмента: из специальной таблицы, а не напрямую через сегментный регистр.

```
физический адрес = база сегмента из таблицы + смещение
```

Теперь сегментные регистры `cs`, `ds`, `ss`, `es`, `gs`, `fs` хранили не кусочки абсолютных адресов начал сегментов, а **селекторы сегментов**, содержащие служебную информацию, а также смещение в таблицах дескрипторов сегментов с более подробной информацией о них. Может существовать много таблиц LDT (Local Descriptor Table) и одна GDT (Global Descriptor Table).

LDT были предназначены для механизма аппаратного переключения задач, который не прижился. В настоящее время разделение адресных пространств происходит с помощью механизма виртуальной памяти, а LDT не используются.

Регистр `gdtr` хранит адрес GDT и её размер.

Рис. 3.1: Строение селектора сегмента в одном из сегментных регистров

Индекс обозначает номер дескриптора в таблице GDT или LDT; в какой именно зависисит от бита `T`.

Поле RPL (Request Privilege Level) реализует кольца защиты. Напомним, что колец защиты четыре, они пронумерованы от нуля до трёх. Каждое кольцо защиты – это некоторый уровень привилегий. Внутри таблиц GDT/LDT хранятся дескрипторы сегментов – структуры данных, описывающие их. Одно из полей дескриптора сегмента – номер кольца защиты, к которому этот сегмент привязан.

`RPL` характеризует уровень привилегий, с позиции которого мы просим доступ к выбранному этим селектором сегменту. Если этот сегмент привязан к более привилегированному уровню, нежели тот, с которого мы делаем запрос, то попытка доступа к этому сегменту не удаётся.

- Значения `RPL` в `cs` и `ss` должны быть равны и характеризуют *номер текущего кольца защиты.*

- Значение `RPL` в `ds` может не совпадать. Так можно переопределить уровень привилегий в сторону более слабого для доступа к сегментам данных.

  Например, находясь в кольце 0 при `ds = ???? ???? ???? ??10`$_2$ мы не сможем обратиться к сегменту данных, в дескрипторе которого будет указан уровень привилегий 0 (несмотря на биты

cs/ss). Только обращения к сегментам данных с уровнем привилегий 2 или 3 будут разрешены. Код, однако, можно выполнять и из самых привилегированных сегментов.

Замечание. *Изменить напрямую значение* `cs` *нельзя. Для этого используются такие инструкции, как* `jmp 0x02:new_start`.

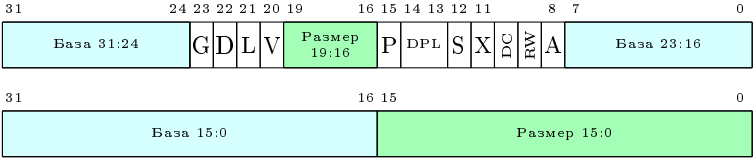


Рис. 3.2: Формат дескриптора сегмента

G – Гранулярность, в чем задаётся размер (0 = в байтах, 1 = в страницах размера 4KB)
D – Размер операнда для операций по-умолчанию (0 = 16 бит, 1 = 32 бита)
L – Описывает 64-разрядный сегмент
V – Доступен системному программисту
P – Присутствует в памяти
S – Тип (0 системный, 1 код или данные)
X – Тип (0 данные, 1 код)
RW – Для данных разрешение на запись (чтение всегда разрешено), для кода – на чтение (запись всегда запрещена)
DC – Для данных направление роста сегмента (вверх или вниз), для кода – возможность вызвать с более высоких уровней привилегий.
A – Был ли доступ к сегменту.
DPL – Уровень привилегий сегмента (к какому кольцу защиты привязан?)

Компьютер стартует в реальном режиме. Чтобы войти в защищённый режим нужно создать GDT и настроить на неё `gdtr`, установить специальный бит в регистре `cr0` и произвести т.н. **far jump**. Приставка far означает, что используется не просто 32-разрядный адрес, но также задаётся и селектор сегмента.

Если мы хотим после загрузки GDT перейти на адрес `addr`, мы напишем:

```
1   jmp 0x08:addr
```

Вопрос 13. *Расшифруйте, что означает селектор сегмента 0x08.*

Вот пример кода для включения защищёного режима работы:

Листинг 3.1: ./listings/loader_start32.asm

```
    lgdt [cs:_gdtr]

    mov eax, cr0        ; !! Privileged instruction
    or al, 1            ; this is the bit responsible for protected mode
    mov cr0, eax        ; !! Privileged instruction

    jmp (0x1 << 3):start32 ; assign first seg selector to cs

align 16
_gdtr:                  ; stores GDT's last entry index + GDT address
dw 47
dq _gdt

align 16

_gdt:
; Null descriptor:
dd 0x00, 0x00
; x32 code descriptor:
db 0xFF, 0xFF, 0x00, 0x00, 0x00, 0x9A, 0xCF, 0x00 ; differ by exec bit
; x32 data descriptor:
db 0xFF, 0xFF, 0x00, 0x00, 0x00, 0x92, 0xCF, 0x00 ; execution off (0x92)
; size size base base base util util|size base
```

Можно подумать, что при сегментации на каждое обращение к памяти нужно еще и прочитать что-то из GDT. Это не так: для всех сегментных регистров (хранящих селекторы) существуют специальные **теневые регистры**, хранящие вхождение в GDT (дескриптор), соответствующий этому селектору. Из него берется вся необходимая информация о сегменте.

Формат дескриптора в таблице GDT вы можете посмотреть на рисунке 3.2.

Значение флага `D` нуждается в некоторых пояснениях, т.к. оно зависит от типа сегмента:

- Сегмент кода: длина эффективного адреса и операндов команд по умолчанию. Если он установлен, то используются 32-битные адреса и 32-битные или 8-битные операнды, иначе 16-битные адреса и 16-битные или 8-битные операнды. На уровне машинных инструкций можно добавить перед кодом операции префикс

0x66, чтобы выбрать другой размер операнда, или префикс 0x67, чтобы выбрать другой размер адреса.

- Сегмент стека (сегмент данных + ему соответствует регистр `ss`)[1]. Опять же речь идёт о размере операнда по умолчанию для `call`, `ret`, `push`/`pop` и др. Установленному флагу соответствуют 32-битные операнды и регистр `esp`, сброшенному - 16-разрядные.

- Для сегментов данных, растущих к младшим адресам, это также граница (64 КБ если флаг сброшен, 4 ГБ если флаг установлен).

В long-режиме этот бит должен быть установлен.

Сегментация не получила поддержки со стороны распространённых ОС. Почему?

- Это проще (не нужно постоянно менять сегментные регистры);

- Практически все распространённые языки используют плоскую модель памяти, где все байты памяти линейно нумеруются. Сегменты сложно использовать в таких условиях.

- Нет острой проблемы фрагментации памяти;

- Сегментация даёт с помощью GDT возможность использовать не более 8192 сегментов. Может быть непросто эффективно использовать их.

При дальнейшем развитии процессора и появлении long-режима от сегментации решили отказаться. К сожалению, полностью её выкинуть из процессора невозможно, и в урезанном виде она присутствует и в long-режиме.

3.3 Минимальная сегментация для Intel 64

Даже в Long-режиме каждый раз при выборке инструкции из памяти процессор использует сегментацию. Благодаря механизму сегментации мы получаем линейный виртуальный адрес, который затем преобразуется в физический с помощью механизма виртуальной памяти (см. 4.2).

LDT являются частью механизма аппаратного переключения контекстов, считающегося медленным и устаревшим, поэтому LDT не используются в Long-режиме.

[1] В этом случае в документации этот флаг называют `B`.

Все обращения к памяти с использованием регистров `cs`, `ds`, `es`, `ss` подразумевают, что начало сегмента будет нулевым, а размер – неограниченно большим (поэтому проверки на невыход за размер сегмента не происходят). В GDT по прежнему есть вхождения для как минимум трёх дескрипторов (специальный нулевой сегмент, сегмент кода и сегмента данных), но их поля базы и размера сегментов не имеют смысла. Отдельные дескрипторы для кода и данных необходимы потому, что никакая комбинация флагов в дескрипторе не позволяет регулировать одновременно разрешение на чтение/запись и на выполнение

Кроме того, в реальности нужны еще как минимум два дескриптора для кода приложений, чтобы указать им иной уровень привилегий по сравнению с кодом ОС. Итого можно ограничиться пятью дескрипторами: два для кода (ОС и приложений) и два для данных (ОС и приложений), а также null-дескриптор, который должен быть в любой GDT.

Чтобы не быть голословными, мы приведем фрагмент реального кода загрузчика Pure64 (это название самого загрузчика), описывающего GDT для 64-разрядного режима работы. Загрузчик системы, конечно, исполняется на нулевом уровне привилегий, поэтому дескрипторов для кода приложений тут пока еще нет.

Листинг 3.2: ./listings/gdt64.asm

```
1  align 16
2  GDTR64:           ; Global Descriptors Table Register
3     dw gdt64_end - gdt64 - 1 ; limit of GDT (size minus one)
4     dq 0x0000000000001000 ; linear address of GDT
5
6
7  ; This structure is copied to 0x0000000000001000
8  gdt64:
9  SYS64_NULL_SEL equ $-gdt64 ; Null Segment
10    dq 0x0000000000000000
11 ; Code segment, read/exec, nonconforming
12 SYS64_CODE_SEL equ $-gdt64
13    dq 0x0020980000000000 ; 0x00209A0000000000
14 ; Data segment, read/write, expand down
15 SYS64_DATA_SEL equ $-gdt64
16    dq 0x0000900000000000 ; 0x0020920000000000
17 gdt64_end:
18
```

```
19  IDTR64:           ; Interrupt Descriptor Table Register
20      dw 256*16-1    ; limit of IDT (4096 bytes - 1)
21      dq 0x0000000000000000 ; linear address of IDT
```

3.4 RISC или CISC?

В рамках одной из классификаций, система команд процессора определяет принадлежность его к одной из двух категорий: CISC (Complete Instruction Set Computer) или RISC (Restricted/Reduced Instruction Set Computer).

CISC обладает большой системой команд «на все случаи жизни», которые могут выполняться долго, но и делать многое, а RISC – минималистичной системой команд, в которой есть очень мало базовых, но очень быстрых команд.

В пользу CISC говорит, например, скорость исполнения определенных сложных операций (например, можно совершать много операций с данными одновременно, как реализовано в расширении SSE), но производители компиляторов с языков высокого уровня не могут охватить полностью огромное количество команд.

RISC же позволяет писать эффективные компиляторы и эффективно организовывать различные аппаратные техники ускорения выполнения программ, такие, как конвейер и суперскалярная архитектура.

Вопрос 14. *Прочитайте про то, что называют конвейером команд в процессорах.*

Система команд, которую мы изучаем, конечно, относит архитектуру к категории CISC. Однако на более глубоком, микропрограммном уровне, каждая команда транслируется в набор более простых команд, и уже здесь происходят различные оптимизации (!), конвейеризация и пр.

Модель процессора, которая описывается в документации Intel, иногда поражает кажущейся нелогичностью. Например, ситуация с РОН и их частями выглядит очень странно. Деление, например, `rax` на `eax`, `ax`, ... очевидно, пока мы говорим о чтении из регистра. При записи же происходят следующие любопытные вещи:

Листинг 3.3: ./listings/risc_cisc.asm

```
1  mov rax, 0x1111222233334444 ; rax = 0x1111222233334444
```

```
2   mov eax, 0x55556666          ;!rax = 0x0000000055556666
3                                ; почему не rax = 0x1122334455556666?
4
5   mov rax, 0x1111222233334444 ; rax = 0x1111222233334444
6   mov ax, 0x7777               ; rax = 0x1111222233337777
7                                ;(работает??!!)
8   mov rax, 0x1111222233334444 ; rax = 0x1111222233334444
9   xor eax, eax                 ; rax = 0x0000000000000000
10                               ; почему не rax = 0x1111222200000000?
```

Причина в оторванности концепции регисров от того, что происходит внутри процессора. На самом деле регистров `rax`, `eax` не существует.

Подумаем о том, как происходит декодирование инструкций системы команд, видной нам. Одна из задач декодера команд в том, чтобы конвертировать команды из этой устаревшей, унаследованной системы в новую, не видную программисту, похожую на систему команд RISC-процессора. Можно будет выполнять много (до шести) таких маленьких инструкций одновременно). Чтобы это осуществить *нужно виртуализировать понятие регистров*. При выполнении декодер выбирает свободный физический регистр из большого банка регистров. Как только большая CISC-инструкция исполнилась, нужно записать значение этого рабочего регистра в тот, который на данный момент хранит в себе значение нужного виртуального регистра, например, `rax`.

Что может помешать этой идиллии? Зависимости между инструкциями по данным (наихудший сценарий – когда несколько подряд идущих инструкций работают с одним и тем же регистром и изменяют его). К примеру, большие проблемы должен вызывать регистр `rflags`.

Если бы запись в `eax` не обнуляла бы весь `rax`, декодеру бы пришлось соединять значения двух регистров в момент завершения CISC-инструкции. Эта зависимость между регистрами сильно бы тормозила работу процессора и не позволяла бы осуществлять столько оптимизаций работы на уровне микрокоманд. Поэтому здесь логичность и красота принесены в жертву сермяжной необходимости быстроты.

Глава 4

Память

В этой главе мы постараемся дать представление о том, чем отличается та память, которую использует программист, от памяти, физически существующей в компьютере.

4.1 Принцип кэширования в широком смысле

Интернет это огромное хранилище данных, однако на то, чтобы обращаться к ним, необходимо достаточно большое время. Поэтому браузер кэширует веб страницы и их элементы (картинки, таблицы стилей и т.д.), чтобы не скачивать их каждый раз. Иначе говоря, он сохраняет их на жёстком диске или в оперативной памяти, в обоих случаях доступ к ним осуществляется значительно быстрее. Однако скачать весь интернет, конечно, нельзя, так как объём хранилища на компьютере гораздо меньше.

Жёсткий диск многократно превышает своим объёмом количество оперативной памяти, установленной в компьютере. Однако оперативная память на порядки быстрее. Поэтому вся работа с данными совершается в оперативной памяти. Оперативная память выступает как кэш для данных, загруженных с жёсткого диска.

Впрочем, на жёстком диске тоже есть своя кэш-память.

На кристалле процессора существуют многоуровневые кэши данных (обычно трёх уровней – L1, L2, L3). Их объём гораздо меньше, чем объём оперативной памяти, но они быстрее. Процессор старается оперировать с данными в кэше, и только если в кэше их нет, загру-

жает данные из оперативной памяти. Помимо этого в большинстве процессоров есть как минимум **кэш инструкций** (очередь команд) и **Translation Lookaside Buffer** для убыстрения механизма виртуальной памяти.

Время доступа к регистрам еще меньше, чем к L1-L3 кэшам, поэтому регистры тоже можно рассматривать как кэш данных.

Почему всё это хорошо работает в информационных системах, обеспечивая *уменьшение среднего времени доступа*? Дело всё в той же **локальности**: в каждый момент времени мы работаем с небольшим набором данных. Итак, кэширование очень универсальный и всепроникающий принцип, когда есть выбор между малым количеством более быстрой памяти и большим количеством более медленной.

Механизм виртуальной памяти помогает, помимо прочего, использовать оперативную память как кэш, в который загружаются необходимые для работы в данный момент куски программного кода и данных.

4.2 Виртуальная память

Виртуальная память это абстракция над физической памятью. Если её нет, то мы работаем в программах напрямую с физической памятью. Иначе же мы абстрагируемся от физической памяти и ведем себя так, будто вся память принадлежит одной программе. На современных системах обычно это достигается с помощью страничной организации памяти, о которой мы подробно поговорим.

4.2.1 Зачем нужна виртуальная память?

Конечно, если на компьютере всегда выполняется только одна программа, то разумно без изысков поместить её в физическую память начиная с фиксированного адреса. Если есть необходимость добавить другие компоненты (код из библиотек, драйвера внешних устройств и т.д.), можно поместить их все в память в определённом порядке.

Зачастую однако аппаратура вкупе с операционной системой должны создать среду, которая будет поддерживать выполнение многих программ параллельно или псевдопараллельно (переключаясь между ними). В этом случае ОС требуется какая-то форма управления памятью, которая бы решала следующие задачи:

- Уметь выполнять программы любого размера (даже большего, нежели размер физической памяти). Возможно, подгружать в

память только те части программы, которые действительно нужны для её выполнения при текущем запуске;

- Размещение в памяти нескольких программ одновременно.

  Программы могут взаимодействовать с внешними устройствами, чьё время отклика на запросы обычно велико. Тогда при запросе, например, к жесткому диску, мы хотели бы на время, пока аппаратура обрабатывает его, загрузить процессор другой полезной работой: отдать его ресурс программе, которая что-то считает и которой жесткий диск не нужен для работы. Конечно, быстро отдать процессор другой программе можно, если она хотя бы частично уже находится в физической памяти.

- Уметь загружать программы в любое место физической памяти.

  Так мы сможем подгружать необходимые куски программ в любое свободное место памяти, даже если внутри программ используется абсолютная адресация, а значит, точные значения адресов уже прописаны в машинном коде.

- По возможности освободить программиста от задачи управления памятью.

  Меньше мыслей о том, как заставить программу надежно работать на конкретном аппаратном обеспечении, и больше о самой логике программы.

- Более эффективное использование общих данных и кода.

  Если две программы сообща используют некоторые данные (загружают в оперативную память файл с жесткого диска) или код (библиотеку с какими-то общеупотребимыми функциями), нерационально держать в памяти несколько копий одинаковых сущностей. К тому же можно использовать такие общие участки памяти для обмена данными между процессами.

Один из механизмов, помогающих достичь этих целей, это виртуальная память.

4.2.2 Адресные пространства

Концепция виртуальной памяти требует для начала раскрытия понятия адресного пространства.

Адресное пространство – некоторая последовательность адресов, от минимального до максимального. Мы сталкиваемся с тремя

типами адресных пространств: **логическим**, **физическим** и **адресным пространством процесса**.

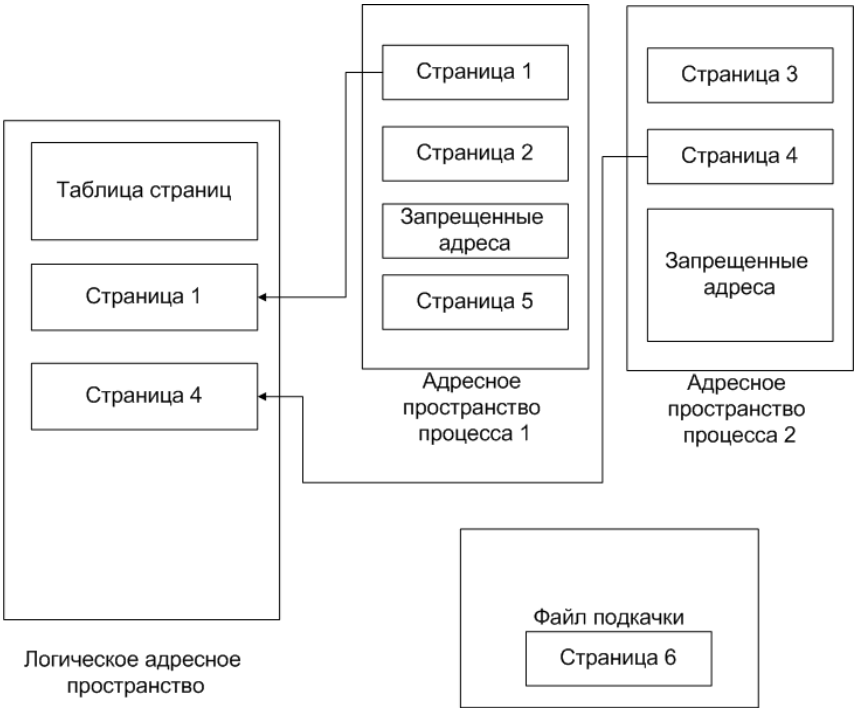
То, сколько реальной памяти имеет компьютер, определяет его **физическое адресное пространство**. **Логическое адресное пространство** – сколько всего ячеек памяти он может адресовать. Бывает, что логическое адресное пространство больше физического (например, в 32-разрядной машине стоит 512 Мбайт оперативной памяти, а потенциально она способна работать почти с четырьмя гигабайтами). Возможна и обратная ситуация: если мы поставим в неё 8 гигабайт памяти, то она сможет работать только с четырьмя из них.

Если логическое адресное пространство больше физического, то некоторые его адреса становятся запрещёнными. При попытке обращения к ним процессор генерирует ошибку, ведь им не соответствует реальных ячеек физической памяти.

Адресное пространство процесса – то, как видна память прикладному программисту. Когда же он пишет прикладное программное обеспечение в современном окружении, у него часто есть иллюзия того, что вся память принадлежит только его программе. В какую бы ячейку памяти он ни обратился по адресу, он не увидит другие прикладные программы, которые, на деле, параллельно или псевдопараллельно выполняются на процессоре вместе с его собственной и делят с ней физическую память.

Очевидно, адреса в смысле адресного пространства процесса совершенно другие, нежели адреса в смысле физической памяти. Процесс перевода одних адресов в другие осуществляет **блок управления памятью** (Memory Management Unit).

Адресное пространство процесса делится на **страницы** фиксированного размера. В каждый момент времени только некоторые, нужные для работы страницы находятся в физической памяти компьютера.



Разные области адресного пространства имеют одинаковый интерфейс доступа (чтение и запись по адресу). Однако реальный смысл действий с ними различен.

- Некоторые адреса соответствуют частям конкретных файлов, находящихся в файловой системе – приложений, библиотек, файлов данных и т.д.

  Сюда также можно отнести memory mapped IO (области памяти для обмена данными с внешними устройствами).

  Страницы в физической памяти могут быть общими для нескольких процессов (**shared**) или эксклюзивными (**private**) и быть доступны только для одного.

- Некоторые страницы появляются для того, чтобы удовлетворить потребности программы в дополнительной памяти (**анонимные страницы**). В частности, к ним относятся страницы, на которых размещены стеки и область **динамически расширяемой памяти (кучи)**. Анонимными они называются, потому что им не соответствуют никакие имена в файловой системе. В отличие от

образа исполняемого файла, содержащего код и данные, файлов с данными и устройств, которые в рамках философии Unix тоже имеют соответствующие им вхождения в файловой системе.

- Большая часть адресов являются **запрещёнными** – их значение неопределено, и при доступе к ним процессор генерирует прерывание `#PF`(Page Fault), о котором в конечном итоге уведомит само приложение. Обычно это ведёт к аварийному завершению работы приложения.

  В *nix принято использовать механизм сигналов для сообщения приложениям об исключительных ситуациях. Возможно задать пользовательский код для обработки почти всех сигналов.

  В такой системе обращение к запрещенной странице приводит к вызову прерывания #PF, его обработчик в операционной системе вызовет передачу сигнала `SIGSEGV` приложению. Распространённое сообщение об ошибке `Segmentation fault` часто возникает именно в этой ситуации.

Если свободной памяти не осталось, то в соответствии с одним из правил (называемых *стратегии вытеснения*) некоторые страницы вытесняются из основной памяти в файл подкачки, который хранится во внешней памяти (на жестком диске или твердотельном накопителе). В *nix системах для этого часто выделяют отдельный раздел на накопителе, в Windows файл, хранящий вытесенные из основной памяти страницы, называется `PageFile.sys`. Процесс вытеснения приложений из основной памяти на более объемную и медленную (целиком или только некоторые страницы) называется **своппинг (swapping)**.

Вопрос 15. *Узнайте, какие существуют стратегии вытеснения.*

Для каждого процесса в контексте виртуальной памяти со страничной организацией существует важное понятие **working set**. Это набор тех страниц, которые принадлежат ему и которые на данный момент загружены в физическую память.

Замечание. *Процесс может обратиться к операционной системе с просьбой выделить ему память. На самом деле он, таким образом, получает не физическую память, а дополнительные адреса в своё распоряжение. Динамическое выделение памяти, в конечном счёте, происходит именно так.*

4.2.3 Пример: доступ к запрещенным адресам

Продемонстрируем, как доступ к запрещенным адресам ведёт к ошибке.

Существует интерфейс доступа к информации о виртуальной памяти через файловую систему, т.н. `procfs`. Просматривая специальные файлы (которые, конечно, не являются блоками данных на носителе), мы можем посмотреть содержимое памяти процесса, переменные окружения и др. В частности, файл `/proc/идентификатор_процесса/maps` показывает карту памяти адресного пространства процесса.

Напишем простую программу, которая входит в бесконечный цикл, и в которой есть секция данных и секция кода.

Листинг 4.1: ./listings/examples/pages/mappings_loop.asm

```
1  section .data
2  correct: dq -1
3  section .text
4
5  global _start
6  _start:
7  jmp _start
```

Используем файл `/proc/?/maps`, где ? – ID процесса.

```
> ./main &
[1] 2186
> cat /proc/2186/maps
00400000-00401000 r-xp 00000000 08:01 144225 /home/stud/main
00600000-00601000 rwxp 00000000 08:01 144225 /home/stud/main
7fff11ac0000-7fff11ae1000 rwxp 00000000 00:00 0 [stack]
7fff11bfc000-7fff11bfe000 r-xp 00000000 00:00 0 [vdso]
7fff11bfe000-7fff11c00000 r--p 00000000 00:00 0 [vvar]
ffffffffff600000-ffffffffff601000 r-xp 00000000 00:00 0 [vsyscall]
```

Регион памяти это набор последовательно идущих страниц. Секции оказались отображены в разные регионы памяти. Первый из регионов в этом примере соответствует секции кода, второй – секции данных.

Из всего большого адресного пространства лишь немногие его адреса не запрещены.

Вопрос 16. *Прочитайте о том, что означает четвертый (`08:01`), пятый (`144225`) столбцы в `man procfs`.*

Как видно из второго столбца, на регионах памяти (а, скорее, на отдельных страницах) стоят разрешения на чтение (r, read), запись (w, write), выполнение (x, execute). Буква `p` соответствует **private**, то есть региону памяти, который не делят несколько программ одновременно, в отличие от `s` (**shared**).

Теперь модифицируем программу так, чтобы происходила запись туда, куда не следует.

Для начала это будут запрещённые адреса.

Листинг 4.2: ./listings/examples/pages/segfault_badaddr.asm

```
section .data
correct: dq -1
section .text
global _start
_start:
mov rax, [0x400000-1]

; exit
mov rax, 60
xor rdi, rdi
syscall
```

Здесь мы обращаемся в память по адресу 0x003FFFFF, предшествующему началу региона памяти, в котором находится сегмент кода. Первый байт по этому адресу – из диапазона запрещённых адресов, поэтому при запуске программы видим ошибку:

```
> ./main
Segmentation fault
```

4.2.4 Эффективность

На то, чтобы подгрузить отсутствующую в физической памяти страницу, тратится очень много времени – это сложная и дорогая опе-

рация. Так почему такой громоздкий механизм оказался не только эффективным по затратам памяти, но и достаточно быстрым?

Эффективности этого механизма способствуют следующие факторы:

1. Работа по трансляции адресов осуществляется, преимущественно, аппаратурой;

2. Благодаря уже упомянутому свойству локальности, механизм подгрузки недостающих страниц запускается достаточно редко, в сравнении с доступом к уже загруженным. Как и в случае с регистрами, в худшем случае у нас падение производительности, но в среднем это работает быстро.

3. Благодарaря кэшу оттранслированных *адресов начал страниц* TLB (translation lookaside buffer) и локальности, чаще мы осуществляем не полную трансляцию адреса с подключением таблиц трансляции, а облегченную версию, в которой из TLB очень быстро достаётся адрес начала страницы. Смещения же внутри страницы совпадают вне зависимости от того, идет ли речь о странице виртуальной памяти или о смещении относительно адреса, по которому она загружена в физическую память.

Вопрос 17. *Что такое ассоциативная кэш-память?*

4.3 Реализация виртуальной памяти

Для начала следует оговорить, что размер страницы может быть различным (4 килобайта, 1 гигабайт, ...). Мы опишем распространённый случай системы, настроенной на страницы размером 4 килобайта.

Каждый виртуальный 64-битный адрес (то есть, адрес, который мы используем в программах на ассемблере) на самом деле состоит из нескольких частей:

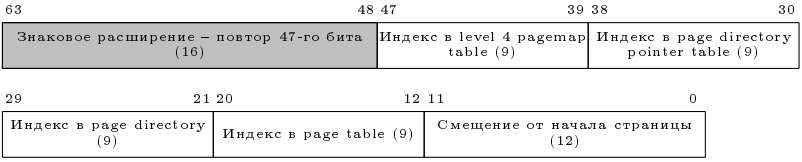


Рис. 4.1: Виртуальный адрес

Собственно, сам виртуальный адрес занимает 48 бит, но он расширяется до т.н. **канонического (canonical)** 64-битного виртуального адреса, в котором старшие 16 бит заполнены знаковым битом исходного числа (47-ым). Поэтому 17 старших битов виртуального адреса всегда одинаковы; только такие канонические адреса считаются корректными.

В процессе трансляции 48 значащих битов виртуального адреса преобразуются в 52 бита физического адреса с помощью иерархии специальных таблиц.[1]. Крайняя правая часть – младшие 12 бит – соответствуют смещению внутри страницы.

Физическое адресное пространство также поделено на слоты для страниц (**page frame**), они находятся «вплотную» друг к другу. Значит, каждый слот начинается с такого адреса, который кратен размеру страницы, иначе говоря, в котором его младшие 12 бит – нули. Поэтому в физическом и виртуальном адресе младшие 12 бит совпадают.

Остальные 4 части виртуального адреса задают индексы в иерархической системе таблиц, использующихся в процессе трансляции. Размер каждой из таблиц этой иерархии равен 4 кбайт (чтобы уместиться на одну страницу физической памяти). Размер одного вхождения в таблицах – 64 бит, но в нём хранятся не только индексы в таблицах следующего уровня, но и служебная информация (например, для защиты некоторых регионов памяти от записи).

Регистр `cr3` используется для того, чтобы найти самую первую таблицу, называемую **Page Map Level 4 (PML4)**. Выбор элемента из PML4 (получение 52-разрядного физического адреса элемента этой таблицы) происходит так:

- Биты 51:12 берутся из `cr3`;
- Биты 11:3 берутся из битов 47:39 виртуального адреса;
- Оставшиеся 3 бита – нули.

Назовём найденную в таблице запись PML4E. Затем ищем запись в таблице следующего уровня (Page Directory Pointer Table): аналогично биты 51:12 берутся из PML4E, следующие 9 бит виртуального адреса составят биты 11:3, последние биты – нули. Дойдя до Page table мы достанем оттуда, наконец, биты 51:12 физического адреса. Это будет тот самый физический адрес, начиная с которого в память загружена страница. Добавив к нему младшие 12 бит виртуального адреса мы

[1] Теоретически можно поддержать 64 бита физического пространства, но на данный момент процессоры ограничиваются 52-мя, ибо их более, чем достаточно

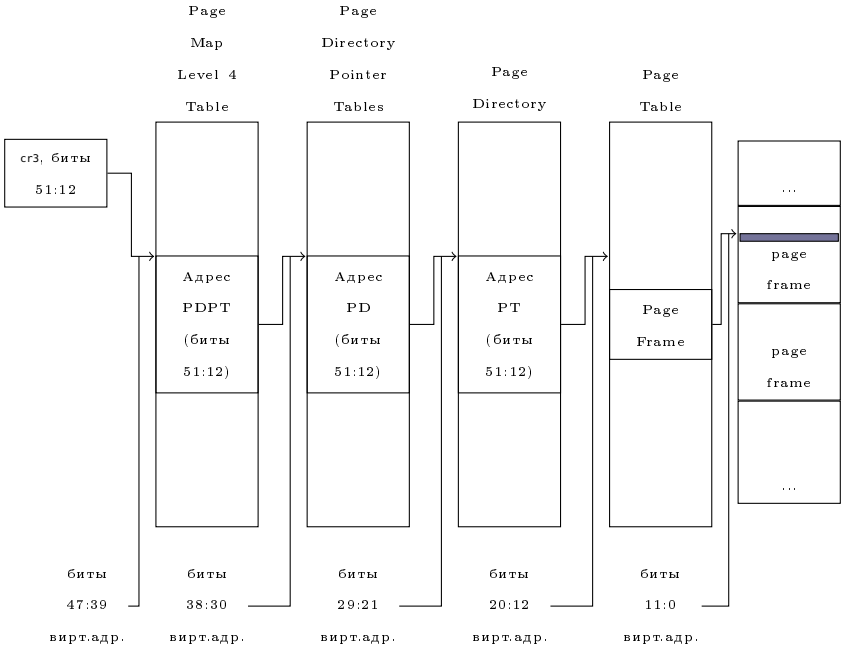


Рис. 4.2: Трансляция виртуального адреса

получим, наконец, физический адрес, которому соответствует виртуальный адрес.

Четыре чтения из физической памяти вместо одного? Да, выглядит громоздко. Однако благодаря кэшу адресов *страниц* **TLB**, процесс трансляции большую часть времени использует уже оттранслированные до этого адреса страниц, к которым нужно всего лишь прибавить смещение внутри страницы (младшие 12 байт виртуального адреса). Свойство локальности обеспечивает нам обращение в основном к небольшому набору страниц. Сам же TLB является ассоциативной кэш-памятью, и потому обеспечивает очень быстрый поиск вхождения по ключу (виртуальному адресу начала страницы). Кроме того, таблицы могут попадать в кэш-память процессора.

Рассмотрим формат вхождений в Page Table.

Записи в таблице страниц

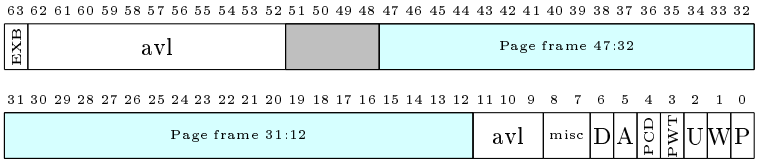


Рис. 4.3: Вхождение в таблицу страниц

P	Present (находится в физической памяти)
W	Writable (разрешена запись)
U	User (разрешено обращение в кольце защиты 3)
A	Accessed (к странице было обращение)
D	Dirty (на страницу было что-то записано)
EXB	Execution-Disabled Bit (запрещает выполнять код со страницы)
avl	Available (доступно разработчику ОС)
PCD	Page Cache Disable (не кэшировать страницу)
PWT	Page Write-Through (записывать на страницу данные минуя кэш)

Если бит `P` сброшен, то при попытке обратиться к странице сгенерируется исключение #PF. Операционная система может в ответ подгрузить страницу с диска в физическую память.

Бит `W` можно использовать для защиты от записи. Это необходимо, например, если мы захотим разделить один фрейм кода в физической памяти между несколькими адресными пространствами ради экономии физической памяти. Это могут быть и общие данные (см. 5.5). Если не помечать эти страницы как защищённые от записи, их можно использовать для обмена данными между процессами.

Страницы с кодом ОС или драйверов имеют сброшенный бит `U`. При попытке обратиться к ним из кольца защиты 3 произойдёт исключение.

`EXB` (или, `NX`-бит) используется для запрещения выполнения кода. На этом основана технология **DEP**. В процессе работы программы злоумышленник может создать ситуацию, когда полученные от него программой данные будут кодировать машинные команды. Эти данные-команды будут храниться в буфере в памяти в одной секции вместе с другими данными. Затем, эксплуатируя найденную им уязвимость, злоумышленник создаст ситуацию, при которой процессор попытается исполнить этот вредоносный код. Однако если область данных программы помечена битом **EXB**, попытка исполнить зловредный код, который находится вместе с другими данными, приве-

дет к исключению. Секция `.text`, код в которой может быть выполнен, располагается на страницах, защищенных от записи битом `W`, что исключает возможность её злонамеренной модификации.

4.3.1 Memory mapping

Термин **mapping** означает «отображение». Речь идет об установлении соответствия между чем-то (файлами, устройствами, физической памятью) и регионами виртуальной памяти. Когда загрузчик заполняет адресное пространство процесса, когда процесс просит у ОС для своих нужд дополнительные страницы, когда ОС отображает файлы с носителя в адресное пространство процесса – всё это можно назвать memory mapping.

Отсюда системный вызов *nix `mmap`, который можно использовать для этих целей: динамически выделить память, отобразить в память файл/устройство.

Отображать файлы в адресное пространство выгоднее, чем просто целиком и сразу копировать их в физическую память. После этого можно обращаться к ним как будто они уже целиком загружены в память.

Как вызвать `mmap`? Конечно, с помощью `syscall`, как и любой другой системный вызов.

`rax`	9	Код вызова
`rdi`	addr	Рекомендация для ОС, с какого адреса начать область, в которую производится отображение. Нужно, конечно, чтобы она начиналась с адреса, кратного размеру страницы. Можно оставить 0, чтобы ОС сама выбрала подходящую область памяти.
`rsi`	len	Размер
`rdx`	prot	Флаги защиты, которые будут установлены на новых страницах (чтение, запись, исполнение...)
`r10`	flags	Флаги типа (shared/private, анонимные страницы и др.)
`r8`	fd	Дескриптор файла, который отображается в память. Не всегда используется.
`r9`	offset	Смещение в файле или ином объекте, начиная с которого он отображается в память

Этот системный вызов вернёт в `rax` указатель на свежесозданную область памяти.

Продемонстрируем отображение обычного файла в память. Для этого нам понадобится еще один системный вызов – `sys_open`, с помощью которого мы откроем файл по имени и получим его **дескриптор**.

`rax`	2	Код вызова
`rdi`	filename	Имя файла – указатель на начало нуль-терминированной строки.
`rsi`	flags	Флаги открытия (только для чтения, для записи, для записи и чтения...)
`rdx`	mode	Если `sys_open` используется для создания файла, здесь будут флаги, задающие его права в файловой системе (чтение, запись, выполнение для владельца, его группы и всех остальных).

Итак, мы хотим отобразить текстовый файл в память и отобразить его содержимое с помощью функции `print_string` из первой лабораторной работы, выводящей в поток вывода нуль-терминированную строку. Для этого мы должны сделать следующее:

1. Открыть файл с помощью `sys_open`. В `rax` он вернёт дескриптор файла;
2. Вызвать `sys_mmap` с нужными параметрами. Одним из них будет дескриптор, полученный на первом шаге.
3. Передать указатель на полученную область памяти в `print_string`.

Проверкой ошибок и закрытием файла в мини-примере пренебрежем, хотя в реальном программировании так делать нельзя.

Создадим файл `test.txt`, затем скомпилируем следующий ассемблерный файл:

Листинг 4.3: ./listings/examples/mmap/main.asm

```
1  ; Эти макроопределения скопированы из кода Linux.
2  ; Там используются аналогичные наименования
3  ; для макроопределений языка C, на котором написан Linux.
4  ; Мы могли бы просто подставить числа в нужные места,
5  ; но макроопределения более наглядно показывают смысл
6  ; этих констант в коде программы.
```

```
%define O_RDONLY 0
%define PROT_READ 0x1
%define MAP_PRIVATE 0x2

section .data
; Имя того файла, который мы отобразим в память
fname: db 'test.txt', 0

section .text
global _start

; Эти функции нужны для отображения нуль-терминированной строки.
print_string:
   push rdi
   call string_length
   pop rsi
   mov rdx, rax
   mov rax, 1
   mov rdi, 1
   syscall
   ret
string_length:
   xor rax, rax
.loop:
   cmp byte [rdi+rax], 0
   je .end
   inc rax
   jmp .loop
.end:
   ret

_start:
; sys_open открывает файл
mov rax, 2
mov rdi, fname
mov rsi, O_RDONLY ; только для чтения
mov rdx, 0      ; мы не будем создавать файл,
                ; так что этот параметр нам неважен
syscall

; sys_mmap
```

```
48  mov r8, rax      ; в rax лежит дескриптор открытого файла,
49                   ; он будет 4-ым параметром mmap
50  mov rax, 9       ; номер системного вызова
51  mov rdi, 0       ; пусть ОС выбирает, куда именно отобразить файл
52  mov rsi, 4096    ; размер страницы
53  mov rdx, PROT_READ ; Новый регион будет помечен только для чтения
54  mov r10, MAP_PRIVATE ; страницы принадлежат только одному
55                   ; адресному пространству
56  mov r9, 0        ; смещение внутри файла test.txt
57  syscall          ; теперь rax указывает на созданный регион памяти
58
59  mov rdi, rax
60  call print_string
61
62  mov rax, 60      ; корректный выход с sys_exit
63  xor rdi, rdi
64  syscall
```

Стоит уточнить, что мы сейчас работаем на ассемблере. ОС написана на языке C, и страницы справки `man` рассчитаны на C-программистов. К примеру, вот фрагмент из справки на `mmap`, описывающий её третий аргумент `prot`:

```
The prot argument describes the desired memory protection of the mapping (an
       PROT_NONE or the bitwise OR of one or more of the following flags:

       PROT_EXEC  Pages may be executed.

       PROT_READ  Pages may be read.

       PROT_WRITE Pages may be written.

       PROT_NONE  Pages may not be accessed.
```

`PROT_NONE`, `PROT_EXEC` это определения препроцессора, аналогичные конструкциям `%define` в `nasm`. Они определяют какие-то фиксированные числа, которые используются для контроля поведения `mmap`: в зависимости от значения `prot`, поведение `mmap` изменится. Чтобы узнать,

какие числа стоят за этими определениями, можно воспользоваться следующими способами:

1. Искать определение в заголовочных файлах Linux API в директории `/usr/include`;
2. Воспользоваться одной из систем Linux Cross Reference (lxr), как, например, http://lxr.free-electrons.com.

Мы советуем использовать второй способ. Можно даже набрать в Google `lxr PROT_READ`, и это сразу выдаст нужные ссылки.

К примеру, для `PROT_READ` система выдаст следующую информацию:

```
PROT_READ

Defined as a preprocessor macro in:
arch/mips/include/uapi/asm/mman.h, line 18
arch/xtensa/include/uapi/asm/mman.h, line 25
arch/alpha/include/uapi/asm/mman.h, line 4
arch/parisc/include/uapi/asm/mman.h, line 4
include/uapi/asm-generic/mman-common.h, line 9
```

Кликнув на любую из перечисленных ссылок мы сразу получим вхождение:

```
 18 #define PROT_READ       0x01            /* page can be read */
```

Значит, можно написать в ассемблерном файле `%define PROT_READ 0x01` и использовать его имя в тексте программы, не запоминая точного значения.

Глава 5

Цикл компиляции

Компиляция программ происходит в несколько этапов. Мы рассмотрим типичный процесс компиляции программы на языках, транслирующихся в машинный код, таких, как ассемблер, C/C++ и др.

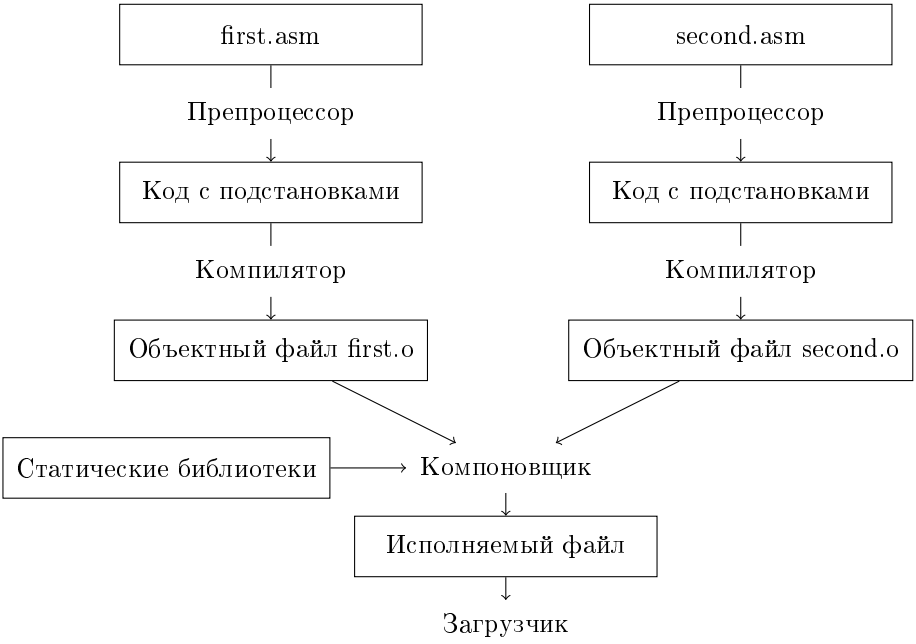


Рис. 5.1: Цикл компиляции

Препроцессор выполняет команды преобразования текста в текст, выдавая программу на исходном языке программирования, в которой произведены текстовые замены.

Компилятор из каждого файла с исходным кодом создаёт файл с машинным кодом, который, однако, обычно еще не готов к исполнению, ибо не связан с другими файлами, к которым обращается.

Компоновщик устанавливает связи между разными файлами и собирает из них один, который уже можно исполнить.

Загрузчик принимает исполняемый файл и загружает его в основную память специальным образом, чтобы машинный код можно было начать выполнять.

Теперь мы поговорим про некоторые из этих стадий более подробно.

5.1 Препроцессинг

Изначально мы пишем текст программы, набор символов. Первая стадия компиляции — препроцессинг исходного кода программы. Он сводится к исполнению специальных команд для программы-препроцессора для обработки исходного текста. В результате получается какой-то другой исходный текст программы на том же языке программирования.

Типичный пример такой инструкции для препроцессора ассемблера — `%define`, осуществляющая **макроподстановку**.

```
1  %define cat_count 3
2  ...
3  mov ax, cat_count
```

Встретив такую конструкцию, препроцессор поймёт, что во всей программе далее по тексту нужно заменить последовательность символов `cat_count` на 3. Подстановка тела макроса вместо его имени называется **инстанцирование**. После того, как исходный текст пройдёт через препроцессор, он будет выглядеть так:

```
1  mov ax, 3
```

Важно отметить, что такой препроцессор ничего не знает про синтаксис языка. Синтаксис языка описывает разрешенные в нём конструкции. Например, конструкция if (*cond*) *statement* else *statement* является цельной сущностью с четко определенной структурой. Макросы позволяют свободно манипулировать частями этой сущности, пока в результате получается корректная строчка на исходном языке

программирования. Один макрос может сгенерировать часть строчки if (*cond*, другой –) *statement*, а третий – всё остальное. Противоположностью обычным, текстовым макросам являются **синтаксические макросы**, используемые в таких языках, как LISP, Ocaml и Scala. Они понимают структуру выражений языка и производят трансформации программы с учетом её иерархической структуры.

С помощью макроподстановок мы можем *задавать имена для констант*. Зачем?

Обратимся к последнему примеру. Пусть нам нужно изменить количество котиков в нашей программе. Если мы повсюду писали число 3 вместо того, чтобы использовать именованную константу `cat_count`, то нам придётся заменять вручную тройки во всей программе на новые значения. Более того, нам каждый раз будет требоваться решать, соответствует ли по логике программы данное число количеству котиков, или чему-то другому.

Другой подход – завести глобальную переменную в памяти с соответствующим именем. У этого подхода есть ряд недостатков:

- Потребление памяти чуть-чуть увеличивается;
- Во многих инструкциях можно задавать операнды непосредственно. Это экономит обращение к памяти за операндом и, соответственно, быстрее. Если речь идёт о языке высокого уровня, то их компиляторы стремятся произвести оптимизацию **constant folding** – подстановку константных значений вместо обращений за ними в память, где это только возможно. Использование макроопределений освобождает их от части работы и гарантирует результат.

Замечание. *Хорошим тоном считается именовать все важные константы в вашей программе.*

В низкоуровневых языках (C, ассемблер) принят способ задания констант через макроопределения.

Можно задавать макроподстановки с параметрами. Число после имени макроподстановки означает количество аргументов, к которым можно будет обращаться по их индексам внутри описания тела макроса; индексы аргументов начинаются с единицы:

```
1  %macro test 3
2  dq %1
3  dq %2
4  dq %3
5  %endmacro
```

Тогда эта строчка:

```
test 666 555 444
```

будет преобразована в:

```
dq 666
dq 555
dq 444
```

Вопрос 18. *Обратитесь к документации на* `nasm` *и прочитайте про* `%define` *и* `%macro` / `%endmacro`.

Препроцессор `nasm` *достаточно мощен. Он может задавать макроподстановки с параметрами, осуществлять операции над строками, например, считать конкатенацию строк и их длину и т.д.*

Вы можете посмотреть, какой код будет сгенерирован после макроподстановок. Для этого запустите `nasm` с ключом `-E`.

5.2 Трансляция

Обычно компилятор принимает исходный файл с кодом и **транслирует** его на другой язык. В этом процессе много внутренних стадий, на каждой из которых компилятор может трансформировать код из одного промежуточного представления в другое. Эти представления называются IR (intermediate representation). Непосредственно перед генерацией кода для целевой архитектуры программа будет представлена в виде, близком к ассемблерному коду, поэтому можно легко получить и листинг с ассемблерным кодом.

Транслирование на ассемблер с языка высокого уровня это сложное преобразование в одну сторону с потерей информации, точное обратное преобразование невозможно.

Модулем называется единица трансляции исходного кода. Обычно это один файл с исходным кодом, из которого генерируется **объектный файл**. Он содержит машинные команды и некоторую служебную информацию. Объектный файл обычно нельзя исполнять. Причины, например, в том, что из одного **модуля** может вызываться код, находящийся в другом. Для этого необходим адрес начала вызываемой процедуры, который мы на данный момент не знаем в силу того, что все файлы компилируются раздельно. Аналогичная проблема возникает с обращением к глобальным данным, объявлённым в других **модулях**.

Замечание. *Конечно, для нас процедура будет задаваться адресом её первой команды. Концом процедуры будем условно считать момент, когда встречается инструкция возврата из процедуры* `ret`*. Однако компьютер не мыслит в терминах процедур вообще; некоторые инструкции, такие, как* `call` *и* `ret` *могут быть использованы для организации механизма вызова процедур человеком или компилятором с языка высокого уровня, а могут и для совершенно иных целей.*

5.3 Компоновка

5.3.1 Объектные файлы

Когда мы компилировали ассемблерный файл в первой лабораторной работе в объектный файл, мы использовали ключ `-f` для указания формата `elf64`. Затем мы пропускали получившийся файл с расширением `.o` через компоновщик `ld` и получали исполняемый файл.

ELF (executable and linkable format) это формат объектных файлов, типичный для *nix систем. Мы ограничимся рассмотрением его вариации для 64-разрядных архитектур.

Объектные файлы состоят из заголовка (см. рис. 5.2) и секций, хранящих самую разнообразную информацию, включая данные и код.

Стандарт ELF подразумевает три типа объектных файлов:

Релоцируемые объектные файлы (relocatable object file). В комбинации с другими такими файлами компоновщик может сделать из него исполняемый объектный файл (см. следующий пункт). Термин **релокация** означает процесс привязывания адресов к различным частям программы и изменение программы таким образом, чтобы ссылки на эти части были правильными. Это необходимо, например, когда компоновщик составляет один большой исполняемый файл из многих объектных, которые ссылаются на адреса внутри друг на друга.

Компиляторы производят объектные файлы этого типа.

Исполняемые объектные файлы (executable object file), содержащие код и данные в таком виде, в котором он может быть воспринят загрузчиком операционной системы, помещен в память и выполнен. Этот файл можно получить после компоновки.

Разделяемые объектные файлы (shared object file), специальный тип объектного файла, который можно загружать в память во время выполнения или загрузки основной программы и связывать с ней *динамически*. В ОС Windows это всем известные `.dll` файлы (dynamically loaded library), в *nix системах их принято снабжать расширением `.so` (shared object).

Компоновщик должен сделать из набора релоцируемых объектных файлов исполняемый. Для этого он выполняет следующие основные задачи:

1. **Релокация**
2. Разрешение символов (symbol resolution) – в каждом месте объектного файла, в котором идёт ссылка на какой-то символ, проставить правильный адрес символа. Да, компоновщик меняет машинный код, сгенерированный компилятором!

Некоторые секции вы уже встречали – секции кода и данных. Однако это лишь частный случай. Некоторые возможные секции ELF-файла (полный список см. в [13]):

.text скомпилированный машинный код программы;

.rodata данные только для чтения;

.data глобальные переменные, инициализированные некоторыми значениями.

.bss (block storage start) – изменяемые глобальные переменные, инициализированные нулями. Так как их значения известны и одинаковы, нет нужды хранить так много нулей подряд, увеличивая размер объектного файла, достаточно лишь их количества. К тому же операционная система может предоставлять более эффективные способы для быстрой инициализации памяти нулями, чем её обнуление вручную;

.symtab таблица символов. Под **символами** подразумеваются адреса, которые соответствуют любым меткам ассемблера. *Символы это тот посредник, через который взаимодействуют разные части программы.* Также тут содержится дополнительная служебная информация, которую мы изучим позднее.

```
ELF Header:
  Magic:   7f 45 4c 46 02 01 01 00 00 00 00 00 00 00 00 00
  Class:                             ELF64
  Data:                              2's complement, little endian
  Version:                           1 (current)
  OS/ABI:                            UNIX - System V
  ABI Version:                       0
  Type:                              EXEC (Executable file)
  Machine:                           Advanced Micro Devices X86-64
  Version:                           0x1
  Entry point address:               0x4000ce
  Start of program headers:          64 (bytes into file)
  Start of section headers:          704 (bytes into file)
  Flags:                             0x0
  Size of this header:               64 (bytes)
  Size of program headers:           56 (bytes)
  Number of program headers:         2
  Size of section headers:           64 (bytes)
  Number of section headers:         7
  Section header string table index: 4
```

Рис. 5.2: Заголовок типичного ELF-файла, полученный с помощью `readelf -h`

.rel.text таблица релокаций для секции `.text` – тех мест в `.text`, которые должны быть модифицированы, когда компоновщик комбинирует этот объектный файл с другими;

.rel.data таблица релокаций для глобальных переменных, которые не определены в этом **модуле**, но на которые он ссылается;

.debug таблица символов, которые нужны именно для отладки программы; если файл был написан на C или C++, то сюда включается информация и о локальных переменных функций, а не только глобальных. Эту секцию `gcc` создаст только в случае, если запущен с ключом `-g`.

.line необходима, например, когда мы написали программу не на ассемблере, а на языке более высокого уровня. Тогда каждая строчка превратится в несколько ассемблерных команд. Чтобы было удобнее отлаживать программу и было возможно выполнять программу по строчкам на исходном языке, в этой секции сохраняют информацию о том, какая строчка соответствует каким командам.

.strtab хранит информацию о строках символов. Адреса этих строчек используются в секциях `.symtab` и `.debug`. Это общий механизм, который также используется в секциях **.dynstr**;

5.3.2 Релоцируемые объектные файлы

В демонстрационных целях напишем простую программу и создадим из неё релоцируемый объектный файл:

Листинг 5.1: ./listings/examples/nm/main.asm

```
1  section .data
2  datavar1: dq 1488
3  datavar2: dq 42
4
5  section .bss
6  bssvar1: resq 4*1024*1024
7  bssvar2: resq 1
8
9  section .text
10
11 extern somewhere
12 global _start
13    mov rax, datavar1
14    mov rax, bssvar1
15    mov rax, bssvar2
16    mov rdx, datavar2
17 _start:
18 jmp _start
19    ret
20 textlabel: dq 0
```

Мы использовали директивы `extern` и `global`. `extern` помечает символ как «определённый в другом месте» (внешний), `global` – как «определённый в этом модуле и доступный другим модулям» (глобальный). Другие символы считаются локальными. Мы увидим, что эти директивы *управляют созданием таблицы символов*. Таким образом, именно таблица символов – важнейшая часть объектного файла, которая обеспечивает взаимодействие кода между объектными файлами.

Чтобы посмотреть заголовок объектного файла и дизассемблировать его, можно воспользоваться утилитой `objdump`. Чтобы посмотреть специфическую для ELF информацию, существует специальная утилита `readelf`. Просто имена символов можно посмотреть с помощью `nm`.

Начнём исследование с `objdump` (см. 5.3)

```
> nasm -f elf64 main.asm && objdump -tf -m intel main.o
main.o:     file format elf64-x86-64
architecture: i386:x86-64, flags 0x00000011:
HAS_RELOC, HAS_SYMS
start address 0x0000000000000000
SYMBOL TABLE:
0000000000000000 l    df *ABS*  0000000000000000 main.asm
0000000000000000 l    d  .data  0000000000000000 .data
0000000000000000 l    d  .bss   0000000000000000 .bss
0000000000000000 l    d  .text  0000000000000000 .text
0000000000000000 l       .data  0000000000000000 datavar1
0000000000000008 l       .data  0000000000000000 datavar2
0000000000000000 l       .bss   0000000000000000 bssvar1
0000000002000000 l       .bss   0000000000000000 bssvar2
0000000000000029 l       .text  0000000000000000 textlabel
0000000000000000         *UND*  0000000000000000 somewhere
0000000000000028 g       .text  0000000000000000 _start
```

Рис. 5.3: Таблица символов в релоцируемом объектном файле (до компоновки)

Смысл столбцов в таблице следующий:

- Виртуальный адрес символа. Пока что мы не знаем, с каких ад-

ресов будут начинаться секции, поэтому все виртуальные адреса задаются от начал секций. Например, `datavar1` является первым числом, что лежит в секции `.data`, поэтому его адрес 0, а следующее – `datavar2 -` находится в той же секции со смещением 8 (размер `datavar1`). Для `somewhere`, определённого где-то в другом модуле, очевидно, виртуальный адрес пока смысла не имеет, поэтому он оставлен нулевым.

- Набор из семи букв или пробелов; каждая буква характеризует символ. Некоторые из них представляют для нас интерес:
  1. `l`, `g`, _ – локальный, глобальный, ни тот ни другой;
  2. –
  3. –
  4. –
  5. `I`, _ – ссылка на другой символ или просто символ;
  6. `d`, `D`, _ – символ для отладки, динамический символ или нормальный символ;
  7. `F`, `f`, `O`, _ – это имя функции, файла, объекта или обычный символ
- Секция, которой принадлежит эта метка (`*UND*` соответствует неизвестной пока секции, `*ABS*` означает никакой секции);
- Загрузочный адрес символа (имеет смысл только в экзотических случаях, например, при создании прошивок для встраиваемых систем);
- Имя символа.

Например, первый символ в таблице это имя файла (`f`) `main.asm`, и необходимо оно только для отладки (`d`), локально для объектного файла (`l`).

Обратим внимание, что глобальная метка `_start`, которая является **точкой входа** для программы, помечена буквой `g` во втором столбце.

Замечание. *Компоновщик чувствителен к регистру! _start и _STaRT это разные символы.*

Теперь посмотрим, как выглядят адреса в машинном коде. В таблице символов они же указаны относительно начала секций, а не абсолютным образом. Воспользуемся `objdump` с ключами `-D` (для дизассемблирования) и `-M intel-mnemonic` для отображения дизассемблированного кода в синтаксисе Intel, а не AT&T.

```
> objdump -D -M intel-mnemonic main.o
main.o:     file format elf64-x86-64
Disassembly of section .data:
0000000000000000 <datavar1>:        ...
0000000000000008 <datavar2>:        ...
Disassembly of section .bss:
0000000000000000 <bssvar1>:        ...
0000000002000000 <bssvar2>:        ...
Disassembly of section .text:
0000000000000000 <_start-0x28>:
   0:   48 b8 00 00 00 00 00    movabs rax,0x0
   7:   00 00 00
   a:   48 b8 00 00 00 00 00    movabs rax,0x0
  11:   00 00 00
  14:   48 b8 00 00 00 00 00    movabs rax,0x0
  1b:   00 00 00
  1e:   48 ba 00 00 00 00 00    movabs rdx,0x0
  25:   00 00 00
0000000000000028 <_start>:
  28:   c3                      ret
0000000000000029 <textlabel>:
```

Операнд команд `mov` в секции `.text` со смещениями 0 и 14 относительно её начала должен быть адресом метки `datavar1`, но он равен нулю! Аналогичная ситуация с меткой `bssvar`. Это значит, что компоновщик должен будет *изменять машинный код, записывая вместо нулевых абсолютных адресов правильные*. Каждому символу должны быть сопоставлены места в объектном файле, где на него происходит ссылка (**таблица релокаций**). Как только компоновщик понимает, каким будет адрес символа, он заполняет ссылки на него его адресом. Таблица релокаций указывает, в каких местах требуется произвести изменение машинного кода, проставив адрес.

```
> readelf --relocs main.o
Relocation section '.rela.text' at offset 0x440 contains 4 entries:
  Offset          Info       Type            Sym. Value    Name+Addend
```

```
000000000002  000200000001 R_X86_64_64       0000000000000000 .data + 0
00000000000c  000300000001 R_X86_64_64       0000000000000000 .bss + 0
000000000016  000300000001 R_X86_64_64       0000000000000000 .bss + 2000000
000000000020  000200000001 R_X86_64_64       0000000000000000 .data + 8
```

Если есть необходимость просто посмотреть имена символов в объектном файле, воспользуйтесь `nm`, который покажет только виртуальные адреса, букву, характеризующую тип символа и его имя:

```
> nm main.o
0000000000000000 b bssvar
0000000000000000 d datavar
                 U somewhere
000000000000000a T _start
000000000000000b t textlabel
```

5.3.3 Исполняемые объектные файлы

Перейдём ко второму типу файлов – **исполняемому**. Создадим второй файл `other.asm`:

Листинг 5.2: ./listings/examples/nm/other.asm

```
1  global somewhere
2  section .data
3  somewhere: dq 999
4  private: dq 666
5  section .code
6
7  global func:function
8
9  func:
10 mov rax, somewhere
11 ret
```

Скомпилируем его так же, как и предыдущий, и скомпонуем получившиеся объектные файлы в исполняемый. Что находится внутри скомпонованного объектного файла?

```
> nasm -f elf64 main.asm
> nasm -f elf64 other.asm
> ld main.o other.o -o main
> objdump -tf main

main:     file format elf64-x86-64
architecture: i386:x86-64, flags 0x00000112:
EXEC_P, HAS_SYMS, D_PAGED
start address 0x00000000004000ce

SYMBOL TABLE:
00000000004000b0 l    d  .text  0000000000000000 .text
00000000006000d8 l    d  .data  0000000000000000 .data
00000000006000e8 l    d  .bss   0000000000000000 .bss
0000000000000000 l    df *ABS*  0000000000000000 main.asm
00000000006000d8 l       .data  0000000000000000 datavar
00000000006000e8 l       .bss   0000000000000000 bssvar
```

```
00000000004000cf l       .text  0000000000000000 textlabel
0000000000000000 l    df *ABS*  0000000000000000 other.asm
00000000006000e0 g       .data  0000000000000000 somewhere
00000000004000ce g       .text  0000000000000000 _start
00000000006000e8 g       .bss   0000000000000000 __bss_start
00000000006000e8 g       .data  0000000000000000 _edata
00000000006000f0 g       .bss   0000000000000000 _end
```

Изменились флаги: теперь файл можно исполнять (`EXEC_P`), исчезла таблица релокаций (больше нет флага `HAS_RELOC`). Виртуальные адреса символов теперь заполнены, как и адреса в коде, и файл готов для загрузки.

Как видим, таблица символов есть и в исполняемом файле. Её можно удалить с помощью утилиты `strip`.

Вопрос 19. *Почему если не пометить `_start` как глобальный символ, `ld` выдаст предупреждение? Продемонстрируйте, где хранится адрес точки входа, с помощью `readelf`.*

5.4 Загрузка файла

Загрузчик это часть операционной системы, которая подготавливает исполняемый файл с кодом к тому, чтобы его можно было выполнить. Подготовка включает в себя размещение в памяти секций исполняемого файла, необходимых для выполнения программы (`.data`, `.text`, ...), инициализацию `.bss`, а также иногда отображение файлов с диска в память. Каждая секция загружается в **регион памяти**, состоящий из последовательно идущих страниц.

Благодаря механизму виртуальной памяти, все программы можно загружать по одному адресу. Обычно это 0x400000.

Почти любая программа использует код из библиотек. Библиотеки бывают двух видов: статические и динамические.

Статические библиотеки состоят из набора релоцируемых объектных файлов, которые связываются компоновщиком с основной программой и встраиваются в итоговый исполняемый файл.

Динамические библиотеки это то же, что и **разделяемые объектные файлы**, которые связываются с программой во время её загрузки или даже во время её выполнения.

5.5 Динамические библиотеки

Программа работает с заранее неизвестным количеством динамических библиотек. Мы должны поместить их все в одно адресное пространство. Значит, нужно уметь загружать каждую библиотеку в память *начиная с любого адреса*. Чтобы этого достичь есть два пути:

1. Загружать библиотеку в свободное место в памяти и производить релокацию так же, как это делает компоновщик. Это зачастую лишает нас возможности загрузить библиотеку один раз в физическую память и разделять её между разными процессами: каждый процесс загрузит её по своему адресу, а значит в каждой копии должна произойти релокация на свой адрес; релокация изменяет машинный код, значит, ни о каком «разделить одну копию между всеми» не может быть речи.

2. **PIC (position independent code)** Можно писать код, который будет выполняться, будучи загруженным по любому адресу. Для этого необходимо избавиться от абсолютных адресов полностью. В последние годы процессоры стали массово поддерживать адресацию относительно `rip`, что позволяет компиляторам достаточно легко генерировать PIC.

Сделаем из файла `other.asm` динамическую библиотеку и посмотрим, как будет выглядеть её заголовок.

```
> nasm -f elf64 other.asm
> ld -shared -o other.so other.o
> readelf -h other.so
ELF Header:
  Magic:   7f 45 4c 46 02 01 01 00 00 00 00 00 00 00 00 00
  Class:                             ELF64
  Data:                              2's complement, little endian
  Version:                           1 (current)
  OS/ABI:                            UNIX - System V
  ABI Version:                       0
  Type:                              DYN (Shared object file)
  Machine:                           Advanced Micro Devices X86-64
  Version:                           0x1
  Entry point address:               0x0
```

```
  Start of program headers:          64 (bytes into file)
  Start of section headers:          1168 (bytes into file)
  Flags:                             0x0
  Size of this header:               64 (bytes)
  Size of program headers:           56 (bytes)
  Number of program headers:         3
  Size of section headers:           64 (bytes)
  Number of section headers:         10
  Section header string table index: 7
```

Напомним исходный код:

Листинг 5.3: ./listings/examples/nm/other.asm

```
1  global somewhere
2  section .data
3  somewhere: dq 999
4  private: dq 666
5  section .code
6
7  global func:function
8
9  func:
10 mov rax, somewhere
11 ret
```

В динамической библиотеке больше секций:

```
readelf -S other.so
There are 12 section headers, starting at offset 0x598:

Section Headers:
  [Nr] Name              Type             Address           Offset
       Size              EntSize          Flags  Link  Info  Align
  [ 0]                   NULL             0000000000000000  00000000
       0000000000000000  0000000000000000           0     0     0
  [ 1] .hash             HASH             00000000000000e8  000000e8
       0000000000000030  0000000000000004   A       2     0     8
```

```
  [ 2] .dynsym           DYNSYM           0000000000000118  00000118
       00000000000000a8  0000000000000018   A       3     2     8
  [ 3] .dynstr           STRTAB           00000000000001c0  000001c0
       0000000000000028  0000000000000000   A       0     0     1
  [ 4] .rela.dyn         RELA             00000000000001e8  000001e8
       0000000000000018  0000000000000018   A       2     0     8
  [ 5] .text             PROGBITS         0000000000000200  00000200
       000000000000000a  0000000000000000   A       0     0     1
  [ 6] .eh_frame         PROGBITS         0000000000000210  00000210
       0000000000000000  0000000000000000   A       0     0     8
  [ 7] .dynamic          DYNAMIC          0000000000200210  00000210
       00000000000000f0  0000000000000010  WA       3     0     8
  [ 8] .data             PROGBITS         0000000000200300  00000300
       0000000000000010  0000000000000000  WA       0     0     4
  [ 9] .shstrtab         STRTAB           0000000000000000  00000310
       000000000000005a  0000000000000000           0     0     1
  [10] .symtab           SYMTAB           0000000000000000  00000370
       00000000000001c8  0000000000000018          11    14     8
  [11] .strtab           STRTAB           0000000000000000  00000538
       0000000000000059  0000000000000000           0     0     1
```

Секции `.hash`, `.dynsym` и `.dynstr` – минимальный необходимый набор секций для релокации загрузчиком.

.hash является хэш-таблицей, необходимой для ускорения поиска символа в секции `.dynsym`;

.dynsym хранит символы «на экспорт» из библиотеки. Её содержание ожидаемо:

```
> readelf --dyn-syms other.so

Symbol table '.dynsym' contains 7 entries:
   Num:    Value          Size Type    Bind   Vis      Ndx Name
     0: 0000000000000000     0 NOTYPE  LOCAL  DEFAULT  UND
     1: 0000000000000200     0 SECTION LOCAL  DEFAULT    5
     2: 0000000000200300     0 NOTYPE  GLOBAL DEFAULT    8 somewhere
     3: 0000000000200310     0 NOTYPE  GLOBAL DEFAULT    8 __bss_start
```

```
> objdump -ft other.so

other.so:     file format elf64-x86-64
architecture: i386:x86-64, flags 0x00000150:
HAS_SYMS, DYNAMIC, D_PAGED
start address 0x0000000000000000

SYMBOL TABLE:
00000000000000e8 l    d  .hash  0000000000000000 .hash
0000000000000118 l    d  .dynsym        0000000000000000 .dynsym
00000000000001a8 l    d  .dynstr        0000000000000000 .dynstr
00000000000001d0 l    d  .eh_frame      0000000000000000 .eh_frame
00000000002001d0 l    d  .dynamic       0000000000000000 .dynamic
0000000000200280 l    d  .data  0000000000000000 .data
0000000000000000 l    df *ABS*  0000000000000000 other.asm
0000000000000000 l    df *ABS*  0000000000000000
00000000002001d0 l     O .dynamic       0000000000000000 _DYNAMIC
0000000000200280 l     O .data  0000000000000000 _GLOBAL_OFFSET_TABLE_
0000000000200280 g       .data  0000000000000000 somewhere
0000000000200288 g       .data  0000000000000000 __bss_start
0000000000200288 g       .data  0000000000000000 _edata
0000000000200288 g       .data  0000000000000000 _end
```

Рис. 5.4: Таблица символов динамической библиотеки

```
    4: 0000000000000200     0 NOTYPE  GLOBAL DEFAULT    5 func
    5: 0000000000200310     0 NOTYPE  GLOBAL DEFAULT    8 _edata
    6: 0000000000200310     0 NOTYPE  GLOBAL DEFAULT    8 _end
```

.dynstr хранит строчки, на которые по индексам ссылается `.dynsym`

Вопрос 20. *Что хранит переменная среды `LD_LIBRARY_PATH`?*

Вопрос 21. *Разделите первую лабораторную работу на два модуля (там был один модуль, файл lib.inc включался полностью в файл,*

тестирующий правильность его работы). Вынесите все описанные функции в статическую библиотеку.

Вопрос 22. *Перенесите описанные в lib.inc функции в динамическую библиотеку. Протестируйте их примерно тем же кодом, который тестировал её раньше.*

Вопрос 23. *Возьмите любую из стандартных утилит Linux (coreutils). Изучите её с помощью* `readelf` *и* `objdump`.

Глава 6

Прерывания и системные вызовы

6.1 Прерывания

Архитектура фон Неймана не интерактивна. В то же время хочется, чтобы компьютер мог работать с внешними устройствами. Устройства, в свою очередь, в процессе взаимодействия с компьютером нуждаются в его внимании, даже если он в этот момент выполняет какую-то программу. Ситуацию исправляет механизм прерываний.

Прерывания это (при-)остановки программы по специальному сигналу процессору. Сигнал может приходить *как от внешних устройств, так и изнутри* системы (деление на ноль, специальная команда вызова прерывания, неправильно закодированная инструкция...). Сигналу соответствует **обработчик** – программа для задания поведения компьютера в случае прихода именно этого прерывания.

Каждое прерывание имеет жёстко фиксированный номер. Для нас неважно, как именно процессор получает номер прерывания от контроллера прерываний. Имея номер N, процессор обращается в **таблицу дескрипторов прерываний (IDT, Interrupt Descriptor Table)**, хранящуюся в памяти. Каждое вхождение в ней занимает 16 байт. N-ое вхождение в этой таблице описывает релевантную информацию, в том числе адрес программы, которая запустится при поступлении прерывания. Вхождения в неё это **дескрипторы прерываний**.

Ситуации возникновения первых 30 прерываний уже жестко описаны в документации и используются во время функционирования

процессора для предопределенных целей, например, сигнализирование о неправильно закодированной команде. Остальные могут использоваться программистом.

Регистр `idtr` хранит адрес IDT и её размер в байтах.

Рис. 6.1: Регистр `idtr`

Если флаг `IF` установлен, то процессор реагирует на прерывания, если сброшен, то игнорирует их.

Вопрос 24. *Что такое Non maskable interrupts? Как они связаны с прерыванием под кодом 2? Как на них влияет флаг IF?*

Рис. 6.2: Дескриптор прерывания

DPL – Descriptor Privilege Level
Текущий уровень привилегий должен быть $\leq$ DPL.

Type – 1110 (interrupt gate, `IF` автоматически сбрасывается в обработчике) или 1111 (trap gate, `IF` не сбрасывается).

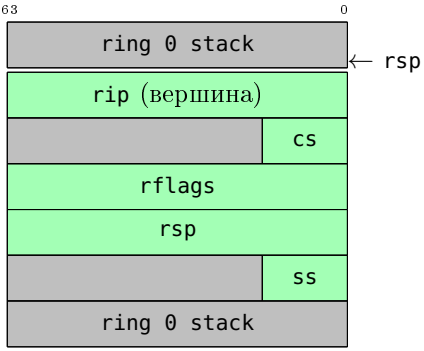
Команда `int` n позволяет вручную вызвать обработчик прерывания под номером n. Она работает даже при сброшенном флаге `IF`.

Прерывания позволяют переключаться с программ на их обработчики и сигнализировать об исключительных ситуациях. Так, в процессоре существует большое количество механизмов защиты от некорректного поведения программ. Например, при обращении к 94-ому прерыванию, если в IDT лежат всего 60 дескрипторов, возникает прерывание с кодом 13.

Многим прерываниям, которые используются процессором в своих целях (как в примере выше) соответствуют специальные мнемоники, отражающие контекст их появления. Так, 13-ое прерывание обозначается #**GP** (от general protection). [1]

Очень часто код, который исполняется на процессоре, имеет низкий уровень привилегий (т.е. выполняется в 3-ем кольце защиты). На нём прямой контроль над внешними устройствами невозможен. Поэтому обработчики должны исполняться в нулевом кольце, и при прерывании нужно изменять текущий уровень привилегий на нулевой.

Перед тем, как выполнить обработчик прерываний, в стек заносятся регистры `ss`, `rsp`, `rflags`, `cs`, `rip`:



Итак, прерывание с номером n обрабатывается так:

1. Из `idtr` берется адрес **IDT**;
2. Из `idt` со смещением $128 * n$ берем дескриптор прерываний, вычленяем обработчик;
3. Изменяем уровень привилегий на тот, что указан;
4. Устанавливаем новые `cs`, `rip`; одновременно в стек сохраняются старые значения `ss`, `rsp`, `rflags`, `cs`, `rip`;
5. Если дескриптор описывает Interrupt gate, то `if = 0`;

Для возврата используется команда `iretq`, которая восстанавливает сохранённые регистры с текущей вершины стека и продолжает исполнение программы с сохранённого адреса `cs:rip`.

[1] См. секцию 6.3.1 третьего тома документации.

6.2 Системные вызовы

Системные вызовы, как вы уже знаете, являются частью кода ОС. Соответственно, выполняются они часто *с другим уровнем привилегий*, нежели код, их вызывающий. Возникает вопрос: как передать выполнение коду из менее привилегированного кольца в более привилегированное?

Решить это можно так: необходимо вызвать прерывание процессора, а дальше пусть разбирается обработчик. Обработчики прерываний обычно выполняются всегда в нулевом кольце. Раньше в *nix для этих целей использовалось прерывание с кодом 0x80, т.е. команда `int 0x80`. Обработчик этого прерывания занимался только обработкой системных вызовов. Однако можно было бы решить эту проблему и иным образом, например, использовать некорректно заданную инструкцию – любой способ вызвать прерывание в принципе годился бы.

Сейчас ситуация немного изменилась: разработчики хотели убыстрить механизм системных вызовов, чтобы не нужен был доступ к IDT (и лишнее обращение к памяти).

Для этого используется пара новых инструкций `syscall` и `sysret`.

У этого подхода есть несколько ограничений, которые нельзя назвать существенными:

- Переход происходит только между третьим и нулевым кольцом (и обратно);
- Все обращения к ядру ОС проходят через один и тот же код;
- Некоторые РОН теперь обладают дополнительным значением; так `rcx` будет хранить адрес возврата для системного вызова, а `r11` – сохранённый регистр флагов.

Абстрагируясь от конкретной архитектуры, для механизма системных вызовов нужно уметь вызывать строго определенный код в более привилегированном режиме, а затем возвращаться к исполняемой программе.

Model Specific Registers

Существует набор регистров, разнящийся от модели к модели процессора. Они называются **Model specific registers**, доступ к ним осуществляется через их номер и специальные инструкции: `rdmsr` для чтения и `wrmsr` для записи.

`rdmsr` принимает в `ecx` номер MSR, а в паре `edx:eax` возвращает значение из запрошенного регистра.

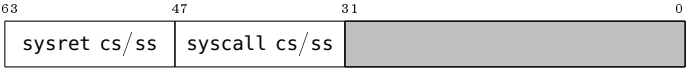
`wrmsr` принимает в `ecx` номер MSR и записывает в него значение из `edx:eax`.

Инструкции `syscall` и `sysret`

Эта инструкция реализована в расчёте не то, что ОС использует flat-модель памяти, то есть без сегментов, только одно большое однородное адресное пространство на процесс.

Чтобы её использовать в long-режиме, ОС должна установить следующие регистры:

- STAR (MSR номер 0xC0000081), из которого загружается `cs`/`ss` для обработчика системных вызовов и `cs`/`ss` для инструкции `sysret` (см. далее).



- LSTAR (MSR номер 0xC0000082), из которого загружается `rip` обработчика системных вызовов. Иными словами это адрес процедуры в ОС, которая занимается обработкой системных вызовов;
- SFMASK (MSR номер 0xC0000084), обозначающий, какие биты в `rflags` нужно сбросить при системном вызове.

Алгоритм работы `syscall` таков:

- Загрузить `cs` из `STAR`;
- Изменить `rflags` в соответствии с `SFMASK`;
- Сохранить `rip` в `rcx`;
- Загрузить `rip` из `LSTAR` и `ss` из `STAR`.

Так мы окажемся в обработчике системных вызовов.

Обратите внимание на объяснение того, что мы уже и так знаем как факт: в процедурах четвертый аргумент передается в `rcx`, в системных вызовах – через `r10`, т.к. `rcx` занят адресом возврата.

Коду обработчика нужно самому установить правильное значение `rsp`, если он хочет иметь свой собственный стек.

Завершается обработка системного вызова инструкцией `sysret`, которая загружает `cs` и `ss` из `STAR`, а `rip` из `rcx`.

Как мы уже поняли, `syscall` загружает `cs` и `ss` из разных MSR. Однако при этом дескрипторы в **теневые регистры**, соответствующие `cs` и `ss`, загружаются не из GDT. Процессор загружает в них фиксированные значения. В GDT, однако, должны найтись дескрипторы, которые соответствуют значениям `cs` и `ss` (по смещениям, находящимся в новых селекторах сегментов), и это обязанность программиста.

Таким образом, для работы `syscall` в GDT должны рядом находиться два дескриптора: дескрипторы для 64-битного кода и данных, оба с DPL=0.

Возникает вопрос: что за фиксированные значения загружаются в `cs` и `ss`? Ответ можно найти в документации на команду `syscall`. Вот отрывок из документации, описывающий её алгоритм работы (сверьтесь с 3.2):

```
RCX <- RIP;     Вот, что происходит с rcx и r11!
R11 <- RFLAGS;

RIP <- IA32_LSTAR;

RFLAGS <- RFLAGS AND NOT(IA32_FMASK); (* Сбрасываем некоторые флаги *)

(* RPL обязательно установлен в 0, CS предоставлен ОС*)
CS.Selector <- IA32_STAR[47:32] AND FFFCH
(* Здесь начинаются поля теневого регистра*)
CS.Base <- 0;
CS.Limit <- FFFFFH;
CS.Type <- 11; (* Сегмент кода (можно выполнять), accessed = 1 *)
CS.S <- 1;
CS.DPL <- 0;
CS.P <- 1;
CS.L <- 1; (* сегмент 64-разрядный *)
CS.D <- 0;
CS.G <- 1; (* Необходимо для 64-битных сегментов *)
CPL <- 0;

(* Аналогично для SS и его теневого  регистра *)
```

```
SS.Selector <- IA32_STAR[47:32] + 8;

SS.Base <- 0;
SS.Limit <- FFFFFH;

(* Сегмент данных, разрешения на чтение, запись, accessed = 1 *)
SS.Type <- 3;
SS.S <- 1;
SS.DPL <- 0;
SS.P <- 1;
SS.D <- 1;  (* 32-разрядный сегмент стека *)
SS.G <- 1; (* Гранулярность 4Кб *)
```

Глава 7

Модели вычислений

Модель вычислений это язык, на котором вы решаете задачу. Часто сложная задача, сформулированная на языке X, тривиально решается после достаточно простой процедуры переформулировки.

Распространённая ошибка при освоении этой темы – думать о новой, неизвестной модели вычислений в терминах старой. Это ведет к тому, что свойства старой модели вычислений переносятся на новую, что бессмысленно.

Пусть мы нашли решение задачи на каком-то языке, который сами же и придумали. Вообще говоря, мы таким образом придумали модель вычислений, которая удобна для формулировки решения этой задачи, так как её базовые операции красиво ложатся в логику решения. Тогда надёжный путь к реализации на некотором языке программирования – идти сверху вниз и выражать *понятия удобного языка* через *язык реализации*.

Мы знакомы с архитектурой Intel 64 и, соответственно, её моделью вычислений, наследуемой от фон Неймановской. Познакомимся еще с некоторыми.

7.1 Конечные автоматы

7.1.1 Определение

Детерминированный конечный автомат — некая абстрактная машина, которая выполняет операции над входным потоком символов в соответствии с заданными правилами.

Конечный автомат полностью задаётся следующими параметрами:

1. Множество состояний, в которых может находиться автомат;
2. Набором символов, которые могут встречаться на входе (алфавит);
3. Начальным состоянием (перед началом работы он находится в нём);
4. Конечными состояниями;
5. Правилами перехода между состояниями. Каждое правило имеет вид «если автомат находится в состоянии X и на входе символ Y, то перейти в состояние Z».

Если на вход автомата попал такой символ, для которого из текущего состояния нет подходящего правила перехода, поведение автомата считается неопределённым. О том, какую роль занимает **неопределённое поведение** при описании моделей вычислений/языков программирования, мы поговорим еще неоднократно.

Некоторые задачи очень просто решить в терминах конечных автоматов. В частности это поиск подстрок по заданному шаблону или проверка строки на соответствие каким-то условиям.

Пример 1. *Пусть нам необходимо проверить, является ли строка целым числом. Перед числом может стоять знак + или -.*

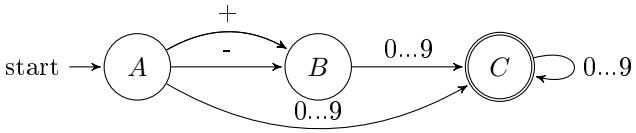


Рис. 7.1: Проверка на число

Пусть алфавит этого автомата – латинские строчные и прописные буквы, пробельные и пунктуационные символы, а также цифры. Множество состояний: $\{A, B, C\}$. Переходы изображены на диаграмме, начальное состояние отмечено стрелкой от надписи start. Конечные состояния помечены двойными кругами.

Мы начнём работу из состояния A и будем двигаться по правилам-стрелочкам, руководствуясь тем, какой символ следующий на входе.

Заметим, что стрелочки с надписью 0...9, на самом деле, изображены таковыми для краткости. Говоря строго, это 10 правил-стрелочек,

Откуда	По какому правилу	Куда переходим
A	+	B
B	3	C
C	4	C

Таблица 7.1: Трассировка работы конечного автомата 7.4 на входе +34

каждое из которых соответствует переходу по своей цифре на входе автомата.

В ходе выполнения работы автомата на входе `+34` мы дошли до конечного состояния C. Однако в случае строки, например, `idkfa`, мы бы натолкнулись на ситуацию, когда непонятно, что делать (нет подходящих правил), и поведение автомата неопределено. Чтобы всегда давать ответ ДА или НЕТ, нужно добавить еще одно состояние и из каждого состояния исходного автомата направить туда стрелки для всех символов алфавита, для которых в этих состояниях не было правил. Этот трюк пригодится нам при реализации автоматов.

Пример 2. *Дана строчка из нулей и единиц. Количество единиц четное?*

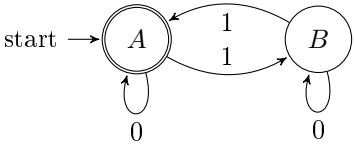


Рис. 7.2: Четно ли количество единиц в битовой строке?

В пустой строчке количество единиц равно нулю – четному числу. Состояние A и начальное и конечное, что отражает этот факт. В любом состоянии чтение нуля не должно приводить к смене состояния. Как только мы прочитали единицу, нужно сменить состояние A на состояние B или наоборот. Можно думать о них так: если мы находимся в состоянии A, то количество единиц в уже обработанной части ввода четно, если в B – нечётно.

Замечание. *Обратите внимание: ни памяти, ни присваиваний, ни if-then-else, ничего подобного в конечных автоматах нет и не может быть, а есть только состояния и переходы между ними. То есть, это описание совершенно другого компьютера, нежели того,*

который мы программируем на ассемблере. Вся возможная информация кодируется текущим состоянием. Однако работу этого, другого компьютера, можно имитировать на привычных языках программирования. Как ни удивительно, это ведет к простым, надёжным решениями, особенно в области встраиваемых систем.

7.1.2 Реализация на ассемблере

Конечные автоматы легко реализуются на императивных языках, в том числе на ассемблере. Гораздо удобнее решить задачу, нарисовав диаграмму с переходами (на которой сразу видны ошибки), а потом тривиально закодировав её.

Чтобы закодировать автомат, необходимо:

1. Избавиться от неопределённого поведения. Для этого в каждом состоянии должны быть правила перехода для *всех* символов алфавита. Поэтому добавим в автомат конечное состояние-ошибку или «нет», в зависимости от задачи, решаемой автоматом. Во все состояния добавим правила, ведущие в это новое состояние, если считанный символ не попадал под все изначальные правила. Условно назовём это **else-переходом**, но с точки зрения чистой математики это просто много правил для всех остальных символов. Реализация может быть более эффективной.

2. Реализовать «считывание символа со входа». Вход конечного автомата – совершенно не обязательно `stdin`; это может быть, например, буфер в памяти вкупе с указателем на текущий элемент;

3. Каждому состоянию:

   - Сопоставить ассемблерную метку;
   - Считать символ;
   - Для каждого правила закодировать проверку с переходом на соответствующие метки;
   - Добавить else-переход.

Реализуем первый автомат на ассемблере. Для этого сначала избавляемся от неопределённого поведения:

Также нужно понять, когда строчка заканчивается. Обычно для этого используют какой-то специальный символ окончания строки. Не будем ходить далеко – возьмём в качестве него нуль-терминатор.

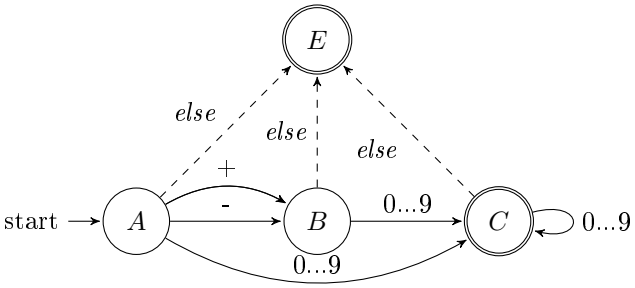


Рис. 7.3: Проверка на число – полный автомат

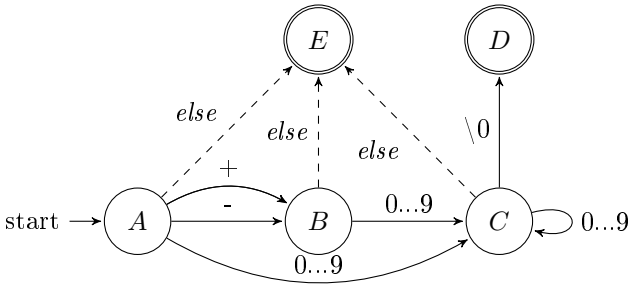


Рис. 7.4: Проверка на число – полный автомат для нуль-терминированной строки

Листинг 7.1: ./listings/automaton_example_bits.asm

```
1  section .text
2  ; будем считать, что getsymbol возвращает
3  ; в al код следующего символа
4  ; начальное состояние:
5  _A:
6      call getsymbol
7      cmp al, '+'
8      je _B
9      cmp al, '-'
10     je _B
11 ; цифры в ascii таблице идут последовательно.
12 ; переход в b если коды символов от '0' до '9',
13 ; иначе переход на метку _E
14 ; удобнее закодировать "наоборот":
```

```
15  ; переход на _E если код меньше 0 или больше 9
16     cmp al, '0'
17     jb _E
18     cmp al, '9'
19     ja _E
20     jmp _B
21
22  _B:
23     call getsymbol
24     cmp al, '0'
25     jb _E
26     cmp al, '9'
27     ja _E
28     jmp _C
29
30  _C:
31     call getsymbol
32     cmp al, '0'
33     jb _E
34     cmp al, '9'
35     ja _E
36     test al, al
37     jz _D
38     jmp _C
39
40  _D:
41  ; успех и процветание
42  _E:
43  ; неблагополучие и сплин
```

Этот автомат приходит в состояние D или E, и, соответственно, на адреса $\_D$ или $\_E$. Можно оформить его изолированно в виде функции, которая будет возвращать, например, 1 (истину) или 0 (ложь). По меткам $\_D$ и $\_E$ тогда будет лежать код устанавливающий `rax` и возвращающийся из функции:

```
1  _D: mov rax, 1
2     ret
3  _E: xor rax, rax
4     ret
```

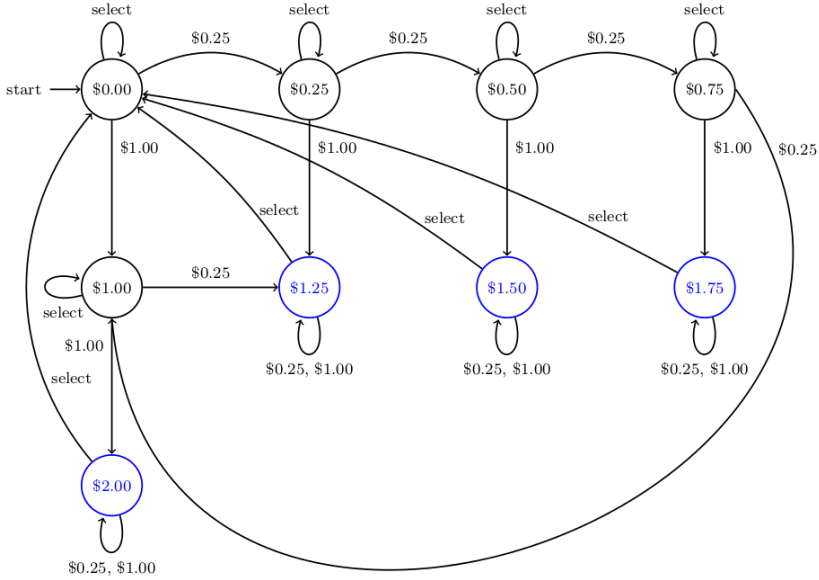
7.1.3 Практическая ценность

Конечные автоматы не тьюринг-полны. Это означает, что как язык программирования они более бедны, чем ассемблер, C и другие языки общего назначения: есть алгоритмы, выразимые на этих языках, которые нельзя выразить через модель конечных автоматов.

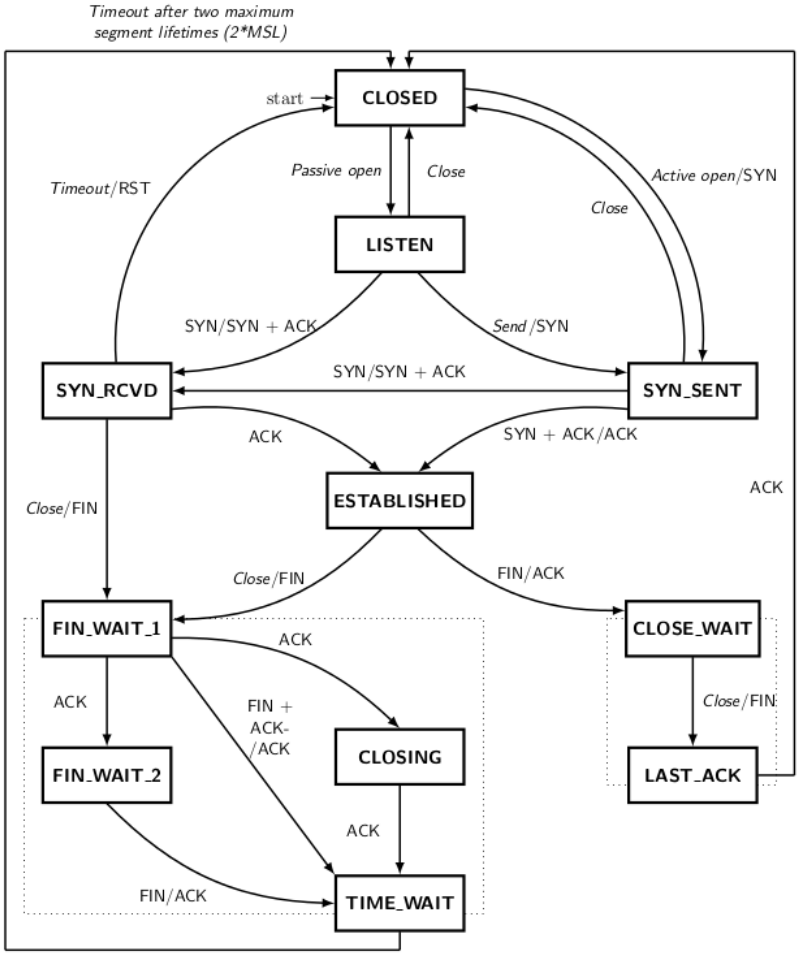
Например, мы не можем с помощью конечных автоматов посчитать количество слов в строке (если только длина строки не ограничена заранее каким-то числом символов).

Вопрос 25. *Нарисуйте конечный автомат для подсчёта количества слов в строке, чья длина ограничена восемью символами.*

Однако автоматы часто используют для описания логики работы встраиваемых систем. Например, это алгоритм работы автомата по продаже напитков, где события (нажатия на кнопки, кидание монетки внутрь) считаются алфавитом, а вход – та последовательность действий, которую совершает пользователь.



А это представление алгоритма работы сетевого протокола TCP в виде конечного автомата. Оно компактно и даёт представление о том, что происходит, значительно более наглядно, нежели текстовое описание или громоздкий псевдокод. Алфавитом в данном случае выступают события: приход пакетов разных типов, таймаут. Состояния это различные режимы, в которых работает система.



Кроме того, существуют техники верификации, например, Model Checking, которые могут проверять истинность логической формулы, задающей утверждение о конечном автомате. Такие проверки увеличивают степень доверия к программе, которая получится в процессе автоматической генерации кода для такого автомата.

Вопрос 26. *Нарисуйте и закодируйте автомат, который проверит, четное ли в строке количество слов.*

Вопрос 27. *Нарисуйте и закодируйте автомат, который определит, необходимо ли обрезать строку слева, справа или с двух сторон, или обрезка не нужна. Если строка начинается или заканчивается*

последовательностью пробельных символов, то её нужно обрезать.

7.2 Forth-машина

Forth это язык, придуманный Чарльзом Муром для управления телескопом. Его особенности с практической точки зрения в том, что можно очень быстро написать его интерпретатор *и компилятор* для любой архитектуры, и удобнее всего делать это на ассемблере. При этом он позволяет писать высокоуровневый, надежный и многократно переиспользуемый код.

Интерпретаторы различных диалектов Forth вы можете встретить:

- В загрузчике FreeBSD;
- В программах управления роботами (ведь очень удобно написать интерпретатор форта на голом железе, а затем описывать логику работы на высокоуровневом языке!);
- Во встраиваемом ПО принтеров и другой бытовой техники;
- В ПО космических аппаратов;
- и во многих других местах.

Это даёт полное право назвать Forth языком системного программирования.

Существует очень много диалектов Forth. Мы придумаем свой, усредненный и простой.

7.2.1 Архитектура

Сначала мы посмотрим на абстрактный вычислитель для языка Forth. Он состоит из процессора, стека данных, стека адресов возвратов и линейно адресуемой памяти.

Forth-машина имеет такую характеристику как «размер ячейки» – обычно его берут соответствующим размеру машинного слова физического процессора, на котором реализуют Forth. Стеки состоят из ячеек такого размера.

Программы состоят из слов и чисел, разделённых пробелами. Слова выполняются последовательно. *Слова-числа кладутся в стек данных*, для чего не нужно писать каких-то дополнительных команд. Следующая программа положит в стек последовательно числа 42, 13, 9:

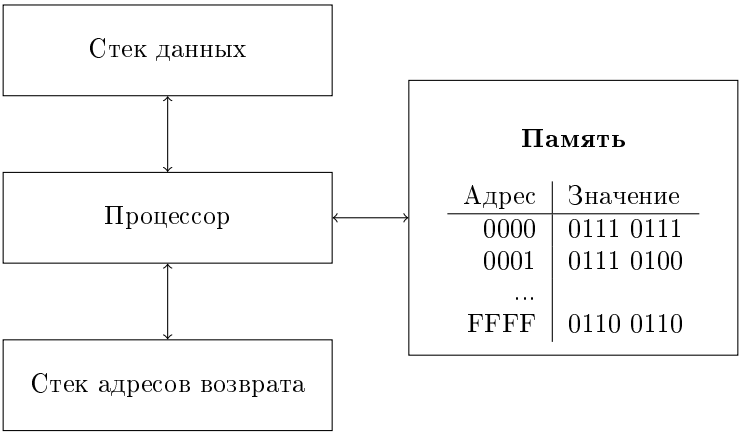


Рис. 7.5: Архитектура Forth-машины

```
42 13 9
```

Логика работы слова может быть описана на ассемблере или как последовательность других Forth-слов (назовём эти невстроенные слова **colon-слова**). Стек адресов возвратов необходим, чтобы после выполнения невстраиваемого слова вернуться в место программы, с которого оно было вызвано (как в случае с `call` и `ret`, но отдельный стек).

Большинство слов манипулируют со стеком данных. В дальнейшем написав «стек» мы почти всегда будем подразумевать именно стек данных. Например, слова +, -, *, / вынимают из стека два операнда, совершают соответствующее арифметическое действие и записывают результат в стек.

После выполнения этой программы на вершине стека будет лежать число $(8 + 8) * 4 + 1 = 65$:

```
1 4 8 8 + * +
```

Ради единообразия мы примем соглашение, что *сначала мы снимаем со стека второй операнд*. Следующее выражение поместит на вершину стека -1, а не 1:

```
1 2 -
```

Новое слово можно определить с помощью *двоеточия*. За ним идет *название нового слова*, список команд и *точка с запятой*, завершающая определение. Двоеточие и точка с запятой тоже слова, поэтому должны быть отделены пробелами.

```
: sq dup * ;
```

Теперь каждый раз когда мы напишем в программе слово `sq`, будет выполняться последовательность действий `dup` (положить в стек еще раз число, которое было на его вершине) и * (взять из стека два числа, перемножить их, результат положить в стек).

Для описания слов в Forth приняты **стековые диаграммы**:

```
swap (a b -- b a)
```

В скобках слева даны имена ячейкам у вершины стека *до выполнения слова*, а справа указывается состояние стека *после выполнения слова*. Так, `swap` меняет местами две последние ячейки стека.

Слово `rot` переставит на вершину третье число из стека:

```
rot        (a b c -- b c a)
```

7.2.2 Пример программы

Листинг 7.2: Подсчёт дискриминанта уравнения $1x^2 + 2x + 3 = 0$

```
: sq dup * ;
: discr rot 4 * * swap sq swap - ;
1 2 3 discr
```

Выполним программу `discr a b c` по шагам для каких-то чисел a b и c. После каждого слова напишем состояние стека после его выполнения.

```
a    ( a )
b    ( a b )
c    ( a b c )
```

Далее выполняется слово `discr`, развернём его:

```
rot  ( b c a )
4    ( b c a 4 )
*    ( b c (a*4) )
*    ( b (c*a*4) )
swap ( (c*a*4) b )
sq   ( (c*a*4) (b*b) )
swap ( (b*b)  (c*a*4) )
-    ( (b*b - c*a*4) )
```

Теперь для конкретных значений 1 2 3:

```
1    ( 1 )
2    ( 1 2 )
3    ( 1 2 3 )
rot  ( 2 3 1 )
4    ( 2 3 1 4 )
*    ( 2 3 4 )
*    ( 2 12 )
swap ( 12 2 )
sq   ( 12 4 )
swap ( 4 12 )
-    ( -8 )
```

7.2.3 Словарь

Ключевая часть Forth-машины – словарь, хранящийся в памяти. Он определяет слова, некоторую служебную информацию и алгоритм их

выполнения. Каждое слово состоит из **заголовка** и последовательности команд. В заголовке хранится ссылка на предыдущее слово, само название слова и служебные флаги.

Ссылка на предыдущее слово нужна, чтобы организовать связный список из вхождений словаря. Forth запоминает адрес последнего из определенных слов. Как только есть необходимость найти слово по его имени, он начинает просматривать словарь с последнего определённого слова. Если его имя не совпадает с искомым, он обращается к предыдущему слову в словаре через ссылку в его словарном заголовке и повторяет процедуру. Останавливается процесс когда ссылка на предыдущее слово равна нулю – это специальное, зарезервированное значение, означающее, что мы дошли до конца словаря.

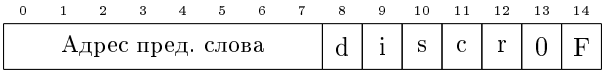


Рис. 7.6: Пример заголовка записи словаря для слова `discr`

Также в машине есть некоторое количество служебных виртуальных регистров, чье количество меняется в зависимости от реализации.

Внутренняя структура слов

Слова можно условно поделить на **встроенные** (написанные на ассемблере) и определённые через другие слова (мы назвали их **colon-слова**). В первом приближении слова это просто последовательности адресов других слов.

Существуют три распространённых подхода к организации связей между словами:

- Indirect Threaded Code
- Direct Threaded Code
- Subroutine Threaded Code

Мы воспользуемся Indirect Threaded Code.

Indirect Threaded Code Для этого классического подхода необходимы две специальные ячейки памяти или регистра:

PC указывает на следующую выполняемую forth-**команду**;

W в начале выполнения *невстроенных* слов указывает на первый адрес сразу после заголовка.

В реализации на конкретной машине это могут быть выделенные регистры или ячейки памяти.

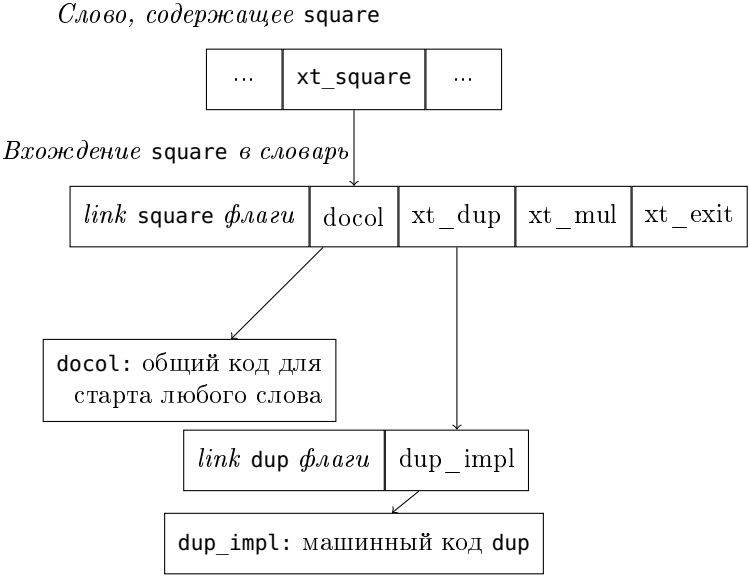


Рис. 7.7: Indirect threaded code

Каждое встроенное слово хранит после заголовка один адрес – адрес исполняемой процедуры. Эта процедура вызывается с помощью `jmp`.

Каждое невстроенное слово начинается с адреса специальной процедуры. Для встроенных слов это просто адрес их реализации на ассемблере. Для colon-слов это адрес специальной процедуры `docol`. Адрес этой ячейки памяти называется **execution token (XT)**. Фактически команды это адреса таких ячеек, то есть execution token'ы.

Colon-слова состоят из адреса `docol` и последовательно идущих XT слов, которые необходимо выполнить.

Иными словами, команда это адрес, по которому лежит адрес кода. Поэтому переходы по командам происходят с *двумя уровнями косвенности*.

Листинг 7.3: ./listings/forth_dict_sample.asm

```
section .data
w_plus:
    dq 0     ; В первом слове указатель на предыдущее отсутствует
    db '+',0
    db 0     ; Никаких флагов
xt_plus:     ; Execution token для plus, иначе говоря
             ; адрес ячейки, указывающей на реализацию
    dq plus_impl
w_dup:
    dq w_plus
    db 'dup', 0
    db 0
xt_dup:
    dq dup_impl
w_double:
    dq w_dup
    db 'double', 0
    db 0
    dq docol  ; Адрес docol - 1 уровень косвенности
    dq xt_dup ; Адрес адреса реализации dup
    dq xt_plus
    dq xt_exit

last_word: dq w_double
section .text
    plus_impl:
        pop rax
        add rax, [rsp]
        mov [rsp], rax
        jmp next
    dup_impl:
        push qword [rsp]
        jmp next
```

Обратите внимание на отличия между встроенными словами (`dup`) и другими (`square`).

PC указывает на ячейку памяти, в которой лежит следующий XT. Значит, после выполнения одной команды необходимо считать следу-

ющую по адресу [pc] и увеличить его на размер команды. Каждая встроенная процедура завершается переходом на специальную последовательность команд next, которая это и делает:

Листинг 7.4: ./listings/forth_next.asm

```
1  next:
2      mov w, pc
3      add pc, 8  ; предполагая размер ячейки 8 байт
4      mov w, [w]
5      jmp [w]
```

Из кода видно, что в момент перехода в пятой строчке **W** будет указывать:

- Для встроенных слов – на ячейку, в которой лежит адрес реализации;
- Для **colon-слов** – на ячейку, в которой лежит адрес docol.

Регистр **W** не играет роли при вызове встроенных слов, но играет её при вызове colon-слов, таких, как square; последний состоит из других слов, а значит, нам нужно будет выполнять их *и возвращаться к исполнению* square. Для этого нужно будет задействовать стек адресов возвратов, что сделает процедура docol. Кроме того она перенаправит **PC** на первый XT слова, которое должно сейчас выполниться. Нетрудно понять, что адрес этого XT будет начало текущего слова + 8. **W** во время начала выполнения слова указывает на начало текущего слова (см. реализацию next), что даёт возможность процедуре docol положить в **PC** значение (**W** + 8)

Листинг 7.5: ./listings/forth_docol.asm

```
docol:
  sub rstack, 8
  mov [rstack], pc
  add w, 8   ; предполагая размер ячейки 8 байт
  mov pc, w
  jmp next

exit:
  mov pc, [rstack]
  add rstack, 8
  jmp next
```

Любое невстроенное слово заканчивается XT слова `exit`. Если `docol`, аналогично инструкции ассемблера `call`, кладёт в стек адресов возврата значение **PC**, то `exit` восстанавливает его, аналогично `ret`. Только стеки для адресов возврата и данных различны, в отличие от Intel 64.

Вопрос 28. *Прочитайте статью Moving Forth (Brad Rodriguez) и узнайте про Direct Threaded Code и Subroutine Threaded Code. Какие преимущества и недостатки вы можете назвать?*

7.2.4 Компиляция в Forth

Forth работает в двух режимах: режиме интерпретации и режиме компиляции. В режиме интерпретации он выполняет команды последовательно.

В режим компиляции он входит когда программист определяет новое слово. В словаре создается новая запись с XT, соответствующим `docol`, и каждая следующая команда дописывается к ней.

Соответственно, Forth должен иметь некоторую внутреннюю переменную `here`, хранящую адрес, в который в режиме компиляции мы будем записывать слова. После каждой записи нужно будет продвигать `here` на размер ячейки (адреса).

Из режима компиляции нужно как-то выходить, иначе мы, начав определять слово, так и будем в него записывать другие слова до бесконечности. Это одна из причин для появления флага **Immediate**. Если слово помечено этим флагом в словаре, то оно всегда будет ин-

терпретироваться, даже в режиме компиляции. Слово `;` является таковым, благодаря чему оно позволяет выйти из режима компиляции.

В режиме интерпретации числа помещаются на стек. В режиме компиляции просто записать число к другим XT нельзя: оно не является адресом. Решается эта проблема так: вводится дополнительное слово `lit`. Когда Forth компилирует слово и встречает там число, необходимо дописать в `here` сначала XT для слова `lit`, а затем число.

Инструкции перехода в Forth

В нашем диалекте будет два специальных слова: **branch** n и **branch0** n. Первое означает безусловный переход (прибавить n к `PC`), второе – условный переход, если на вершине стека 0. Эти слова могут быть использованы *только в режиме компиляции*, т.е. только в определениях слов.

Они похожи на `lit` n тем, что смещение для перехода хранится в коде сразу после них.

7.3 Задание: компилятор и интерпретатор Forth

Пришло время, используя созданную в первой лабораторной работе библиотеку, написать интерпретатор языка Forth, использующий Indirect Threaded Code.

Прежде чем приступать, удостоверьтесь в том, что вы достаточно хорошо понимаете, как должен работать Forth. При необходимости поэкспериментируйте с существующей реализацией `gforth`, чтобы привыкнуть к нему.

Вопрос 29. *Прочитайте документацию на команды `sete`, `setl` и им подобные, а также `cqo`.*

7.3.1 Статический словарь, интерпретация

Для начала напишем цикл интерпретатора и какой-то небольшой словарь встроенных слов.

Пока у нас не будет режима компиляции, и мы не сможем определять новые слова.

За **PC** и **W** удобно взять какие-то РОН. Не забудьте, что они могут разрушаться при вызове функций, если они **caller-saved**!

Словарь удобно определять с помощью макросов, иначе придётся поддерживать в каждом элементе ссылку на предыдущий и, в общем, писать много лишних команд.

Из макросредств `nasm` нам понадобятся: `%macro` / `%endmacro`, `%define`, `%%`, `%+`.

Вопрос 30. *Обратитесь к разделу документации `nasm`, посвященому макросам, и прочитайте про назначение вышеуказанных конструкций.*

Благодаря тому, что `%define` можно использовать многократно для одного и того же препроцессорного символа, мы можем с его помощью поддерживать такое макроопределение, которое содержит последнее определённое статически слово.

```
%define link 0
%macro example 0
%%link: dq link
%define link %%link
%endmacro

example
example
example
example
```

При **инстанцировании** макроса `example` каждый раз будет создаваться новая метка с именем, основанным на `link`, а затем вставляться строчка `%define link %%link`, которая переопределит символ `link` на адрес новой метки. В следующий раз при его использовании будет подставлен адрес метки, которая была создана в последнем example по ходу описания программы.

Последовательность `%%link` позволяет генерировать уникальные метки, добавляя к `link` случаные символы. Эти случайные символы, однако, будут всякий раз одинаковы внутри одного `%macro` / `%endmacro`.

Вопрос 31. *Создайте ассемблерный файл с содержимым примера выше. Скомпилируйте его с флагом `-l out.lst` и изучите содержимое полученного файла. Объясните, почему у ячеек данных, которые были созданы, именно такое содержимое.*

Создайте макрос `native`, который принимает 3 параметра: *имя слова*, *часть идентификатора*, из которой мы составим имя метки для

реализации, а также *флаги*. Он создаёт в секции данных кусок словаря, а в секции кода – метку, по которой находится реализация слова на ассемблере. Так как большинство слов не снабжены никакими флагами, то можно создать перегрузку макроса `native` с двумя параметрами, которая содержит в себе вызов полной версии с нулём в качестве флагов:

```
%macro native 2
native %1, %2, 0
%endmacro
```

Сравните удобство описания слов на ассемблере без макросов и с их помощью.

Без макросов:

```
section .data
w_plus:
  dq w_mul  ; previous
  db '+',0
  db 0
xt_plus:
  dq plus_impl
section .text
  plus_impl:
    pop rax
    add [rsp], rax
    jmp next
```

С макросами:

```
native '+', plus
    pop rax
    add [rsp], rax
    jmp next
```

Также создайте макрос colon, полностью аналогичный предыдущему, но рассчитанный на то, чтобы создавать colon-слова таким образом:

```
colon '>', greater
  dq xt_swap
  dq xt_less
  dq exit
```

Не забудьте про адрес `docol` в первой ячейке после заголовка записи в словаре!

Теперь создайте и протестируйте процедуры:

- `find_word`, возвращающую адрес начала слова с переданным в первом аргументе именем. Если слово не нашлось, вернуть 0. Для удобства следует завести **глобальную переменную**, хранящую в себе адрес последнего слова из словаря;
- `cfa` (code from address), которая по адресу начала слова получит его XT, то есть прибавит к аргументу правильный размер заголовка.

Вкупе с библиотекой, которую вы написали, выполняя первую лабораторную работу, этого достаточно, чтобы легко написать основной цикл интерпретатора.

Интерпретатор будет формировать небольшую программу-заглушку, состоящую из XT слова, которое нужно интерпретировать, и адреса самого интерпретатора с двумя уровнями косвенности:

```
program_stub: dq 0
xt_interpreter: dq .interpreter
.interpreter: dq interpreter_loop
```

Начать исполнение программы мы можем так, и это завершающий фрагмент кода инициализации интерпретатора:

```
mov pc, xt_interpreter
jmp next
```

Так мы попадем в `next` с **PC**=`xt_interpreter`. Внутри `next` мы запустим как раз `interpreter_loop`.

Листинг 7.6: ./listings/forth_next.asm

```
next:
    mov w, pc
    add pc, 8  ; предполагая размер ячейки 8 байт
    mov w, [w]
    jmp [w]
```

Тот, в свою очередь, должен будет считать слово с потока ввода, и если оно есть в словаре, то записать его XT по адресу `program_stub`, настроить на него **PC** и перейти к `next`.

Алгоритм работы интерпретатора представлен ниже:

1: **interpreter_loop:**

```
2: word ← слово из stdin
3: if word пустое then
4:    выход
5: if word есть в словаре и его адрес addr then
6:    xt ← cfa(addr)
7:    [program_stub] ← xt
8:    PC ← program_stub
9:    goto next
10: else
11:    if word это число n then
12:       push n
13:    else
14:       Ошибка: неизвестное слово
```

Наконец, вспомним картинку с архитектурой Forth-машины: там есть не только стеки, но и память! Выделим для этой *пользовательской памяти* 65536 forth-ячеек.

Вопрос 32. *В какой секции лучше выделить память под пользовательские данные?*

Чтобы forth знал про то, где находится память, создадим слово `mem`, которое будет просто класть на вершину стека адрес начала пользовательской памяти.

Задание реализуйте интерпретатор, который умеет класть числа в стек и выполнять следующие команды:

- `.S` – не разрушая стек печатает всё его содержимое. Для реализации нужно будет перед запуском интерпретатора запомнить `rsp`
- Арифметика: `+ - * /`, `= <` (операторы сравнения кладут в стек 1 если условие выполняется, 0 если условие не выполняется);
- Логика: `and`, `not`; они записывают в стек 1 в случае успеха и 0 в случае неуспеха. Истинными значениями считаются любые ненулевые, нулевые – ложными.
- ```
  rot  (a b c -- b c a)
  swap (a b -- b a)
  dup  (a -- a a)
  drop (a -- )
  ```
  ```

- `. ( a -- )` точка вынимает знаковое число с вершины стека и печатает его;
- Ввод-вывод:

  `key ( -- c)` – читает один символ с `stdin`; в ячейке стека только один символ.

  `emit ( c -- )` – пишет один символ в `stdout`.

  `number ( -- n )` – читает знаковое число с `stdin` (гарантированно помещается в ячейку).
- `mem` загрузит в стек константу – адрес начала пользовательской памяти.
- Команды работы с памятью:

  `! (data address -- )` – записывает данные по адресу;

  `@ (address -- value)` – читает содержимое памяти по адресу.

Протестируйте полученный интерпретатор.

Затем создайте область памяти под стек адресов возврата и реализуйте слова `docol` и `exit`. Выделите регистр под указатель вершины этого стека.

Задайте colon-слова `or` и `greater` с помощью макроса `colon` и протестируйте их работоспособность.

Вопрос 33. *Выберите одно встроенное слово и одно невстроенное, заданные с помощью макросов. Разверните макросы вручную. Проверьте результат, сопоставив его с листингом, который генерирует `nasm` с ключом `-l`.*

7.3.2 Расширяемый словарь, компиляция

Теперь добавим возможность расширить словарь. Для этого выделим какое-то количество байт заранее, например, 65536 forth-ячеек.

Вопрос 34. *Какую секцию следует выбрать для выделения пространства под расширяемую часть словаря?*

1. Добавим переменную `state`, которая равна 1 в режиме компиляции и 0 в режиме интерпретации;
2. Добавим переменную `here`, хранящую адрес начала свободной области для словаря;

3. Добавим переменную `last_word`, хранящую адрес последнего определённого слова;

4. Добавим определения для `:` и `;`.

   Двоеточие:

   1: *word* ← `stdin`
   2: Заполняем заголовок словаря, начиная с `here`;
   3: Дописываем в начало слова адрес реализации `docol`;
   4: Обновляем `last_word` и `here`;
   5: *state* ← 1;
   6: Переход на `next`.

   Точка с запятой (помечена **Immediate**!)

   1: `here` ← XT слова `exit` ;
   2: *state* ← 0;
   3: Переход на `next`.

5. В режиме компиляции Forth работает так:

```
1:  compiler_loop:
2:  word ← слово из stdin
3:  if word пустое then
4:      выход
5:  if word есть в словаре и его адрес addr then
6:      xt ← cfa(addr)
7:      if word помечено как Immediate then
8:          интерпретировать word
9:      else
10:         [here] ← xt
11:         here ← here + 8
12: else
13:     if word это число n then
14:         if предыдущее слово было branch или 0branch then
15:             [here] ← n
16:             here ← here + 8
17:         else
18:             [here] ← xt_lit
19:             here ← here + 8
20:             [here] ← n
21:             here ← here + 8
22:     else
23:         Ошибка: неизвестное слово
```

Возможно, легче объединить compiler_loop и interpreter_loop в один цикл.

Добавьте реализацию команд `0branch` и `branch` и протестируйте все команды.

Вопрос 35. *Почему необходимо рассматривать особый случай для инструкций `branch` и `0branch`?*

7.3.3 Bootstrap*

Вообще можно выделить две части интерпретатора Forth. Первая и главнейшая – `next`, это **внутренний интерпретатор** (inner interpreter). Другая – `interpreter_loop` / `compiler_loop`, **внешний интерпретатор** (outer interpreter). Но мы на самом деле можем закодировать её на самом forth! Ведь чтобы составить colon-слово из XT других слов, совершенно необязательно уже иметь внешний интерпретатор.

Чтобы это сделать нужно немного расширить набор слов, которыми мы располагаем. Некоторые слова, которые мы написали на ассемблере, можно будет после этого перевести на forth, например, `docol`, `exit`, двоеточие и точку с запятой.

Вот пример набора дополнительных слов:

- `,` сохраняет вершину стека в `here` и увеличивает `here`;
- `word (addr -- length)` читает с `stdin` слово до пробельных символов и записывает её в нуль-терминированном виде по указанному адресу; в стек кладётся длина прочитанной строчки.
- `create (string -- )` создаёт новое слово в словаре с именем, соответствующим нуль-терминированной строке по адресу `string`.
- `find (string -- addr)` возвращает адрес заголовка слова `string` в словаре или 0, если его не нашёл.
- `'` читает слово с `stdin` и возвращает его XT. Например, `' square` положит на стек XT слова `square`.
- `[` для входа в режим компиляции и `]` для выхода из него.

Задание реализуйте внешний интерпретатор forth на нём же и заставьте исполняемый файл работать без интерпретатора, написанного на ассемблере.

Задание оказывается, есть смысл кэшировать последнее значение стека данных (или даже несколько) в регистре. Перепишите реализации команд так, чтобы последним значением в стеке после `[rsp]` считался какой-то регистр.

Часть II

Язык C

Глава 8

Основные концепции языка

8.1 Чем особен C?

Итак, пришло время познакомиться с самым распространённым языком, используемым для системного программирования. Язык C достаточно высокоуровневый, чтобы на его примере проиллюстрировать также некоторые концепции, присущие и многим другим языкам, такие, как типизация или полиморфизм.

Главная особенность C в том, что он очень близок к машинному представлению программы – своеобразный «высокоуровневый ассемблер». Например, вместо того, чтобы давать полную абстракцию от реальной памяти, сборщик мусора, как в Java или C#, программист сам может (и должен) управлять выделением и освобождением зарезервированной памяти. Программы на C, однако, в отличие от ассемблерных программ, переносимы (если написаны правильно). Это изначально достигалось благодаря тому, что все распространённые архитектуры компьютеров очень похожи, а модель вычислений C, в свою очередь, похожа на них. О смысле этого мы поговорим особо.

Замечание. *With great power comes great responsibility. (Ben Parker)*

В C есть невероятное количестов способов «выстрелить себе в ногу». Утечки памяти, обращение по испорченному указателю — лишь самые очевидные и распространённые.

Замечание. *Язык C чувствителен к регистру и нечувствителен*

к расстановке пробелов, пока транслятор может определить, где заканчивается один «элемент языка» и начинается второй. То есть, следующий код эквивалентен:

```
int main  (int argc , char * * argv)
{
  return 0;
}
```

```
int main(int argc, char** argv)
{
  return 0;
}
```

Существует некоторое количество вариаций языка, относительно стандарта ANSI. Мы будем описывать версию C89 или ANSI C, оставляя за рамками новый стандарт C99 и дополнительные возможности, добавленные разработчиками компиляторов (например, `gcc`).

Чтобы придерживаться стандарта мы будем компилировать файлы с исходным кодом используя ключи `-ansi`, `-pedantic`, `-Wall` и `-Werror`:

```
> gcc -o main -ansi -pedantic -Wall -Werror file1.c file2.c ...
```

8.2 Структура программ

Из чего же состоит программа на C? Вообще говоря выбор невелик:

- Описание типов данных (структуры, новые типы и т.п.) на основе уже описанных или встроенных. Например:

  ```
  typedef int new_name_for_int_type;
  ```

- Описание глобальных переменных:

  ```
  int variable_name = 42;
  ```

- Описания **функций**:

```
1   int square( int x ) { return x * x; }
```

- Комментарии между `/*` и `*/`.

Внутри функций можно описывать переменные, локальные для этой функции, типы данных, а также вызывать другие функции (но не описывать вложенные функции).

Функции выполняют какую-то работу. Они состоят из предложений языка (statement), разделённых точкой с запятой.

8.2.1 Числовые типы данных

Выполнение программы на C это оперирование с ячейками данных.

Абстрактный вычислитель языка C имеет архитектуру фон Неймана. Это осознанный выбор, обеспечивающий близость к аппаратуре, благодаря которой C и является основным языком для системного программирования. Переменные – адреса памяти, по которым лежат данные, так же, как и метки в ассемблере.

Язык C обладает статической нестрогой типизацией. Статическая типизация означает, что все типы данных фиксируются во время компиляции. Нестрогая типизация проистекает от низкоуровневости C, его близости к аппаратуре. Так как для компьютера всё нетипизировано, то мы можем с лёгкостью, например, взять ячейку, которая хранит адрес другой ячейки памяти (указатель), и интерпретировать её как число, которое мы затем можем вывести на экран, явно не попросив компилятор перевести её из одного типа (адрес) в другой (целое число).

Литерал это последовательность символов в программе, которая кодирует непосредственно заданное значение. Число 42, код символа `'a'`, указатель на строку `"abcde"` – литералы.

В C есть следующие встроенные типы данных:

1. `char`
   - Бывает `signed` (знаковый) и `unsigned` (беззнаковый). По-умолчанию обычно `signed`, но стандартом языка это не регламентировано;
   - Занимает всегда 1 байт;

- Несмотря на слегка запутывающее имя (`char` – character, символ) это тип, который хранит **число**. Часто это число интерпретируется как код символа по таблице ASCII, но отнюдь не обязательно.
- Литерал `'x'` обозначает код символа 'x' по таблице ASCII.

```
char number = 5;
char symbol_code = 'x';
char null_terminator = '\0';
```

---

2. `int`

- Целое число;
- Бывает `signed` и `unsigned`. По-умолчанию `signed`;
- Есть синонимы: `signed`, `signed int`;
- Может быть `short` (2 байта), `long` (4 байта на 32-разрядных архитектурах, 8 на 64-разрядных) и `long long` (8 байт);
- Также можно писать `short`, `short int`, `signed short`, `signed short int`;
- Если написать просто `int`, то будет иметься в виду размер машинного слова для конкретного вычислителя — компьютера, для которого мы компилируем программу. Для 32-разрядной архитектуры это `long`, для 16-разрядной — `short`. 64-разрядная архитектура, к сожалению, по причине совместимости, не подчиняется этому правилу. Для неё просто `int` занимает те же 4 байта.
- По умолчанию все числовые литералы считаются типа `int`. Дописывая к ним суффикс `L` или `LL` мы получим числа типа `long` или `long long`. Вот пример с операцией побитового сдвига, когда это важно!

```
1 << 48
```

Значение этого выражения не 2^{48}, а 0. Почему? Единица имеет тип `int`, он занимает 4 байта, т.е. 32 бита, поэтому если сдвинуть единицу влево на 48 бит она просто выйдет за границы формата. Другое дело:

```
1L << 48
```

Или

```
1LL << 48
```

---

3. `float`
   - Число с плавающей запятой;
   - Занимает 4 байта;
   - Диапазон представления: $1,175494351 \times 10^{-38}...3,4028235 \times 10^{38}$
4. `double`
   - Число с плавающей запятой;
   - Занимает 8 байт;
   - Диапазон представления: $2,225074 \times 10^{-308}...1,79769 \times 10^{308}$

Создать переменную одного из вышеперечисленных типов можно так:

```
typename variable_name;
```

Сначала идёт имя типа, потом имя переменной. Запомните этот паттерн. В именах можно использовать латиницу, цифры (но не начинать имя с цифры), подчеркивания.

Можно сразу присвоить переменной константное значение, если это необходимо.

```
int hello = 1488;
```

Замечание. *Не забывайте, что нельзя объявлять переменные в функции где попало! В ANSI C постулируется, что переменные можно объявлять только в начале блока. Блоком считается любая последовательность statement'ов, заключенная в фигурные скобки. Переменные считаются существующими до тех пор, пока не закроют тот блок, в начале которого они были объявлены.*

Можно:

```
int square( int x ) {
  int y;
  y = x;
```

```
4     y = y * y;
5     return y;
6   }
```

Можно:

```
1   int square( int x ) {
2     int y;
3     y = x;
4     if (x == y) {
5       int z;
6       z = x * x;
7       return z;
8     }
9     return 0;
10  }
```

Нельзя:

```
1   int square( int x ) {
2     int y;
3     y = x;
4     int one_more_red_nightmare;
5     y = y * y;
6     return y;
7   }
```

Помимо упомянутых типов, в C существуют также структуры, перечисления, объединения. Это отдельные классы типов данных, о них вы прочитаете позднее.

Распространённые ошибки

В C нет типов данных `bool`, `string`. Когда есть необходимость в проверке условий (внутри if/ while/ тернарного оператора : ?), то нулевое значение считается ложью, а ненулевое – истиной.

```
1   while(-1) {
2   /* will loop forever */
3   }
4   while (20) {
5   /* will loop forever too */
6   }
```

```
7
8   if (0) {
9   /*is never executed*/
10  }
11
12  int x = 1? -3 : 999; /* x = -3 */
```

Строки **нуль-терминированы** и задаются адресом их начала. Соответственно, их тип – «указатель на символ». О типах-указателях мы поговорим особо.

8.2.2 Пример программы

Посмотрим на какую-нибудь совсем простую программу на C.

Листинг 8.1: listings/hello_world.c

```
1   #include <stdio.h>
2
3   /* Исполнение программы на C начинается с функции main */
4   /* main это функция, которая возвращает данные типа int
5      Это код возврата программы. */
6   int main() {
7     /* Локальная переменная функции main
8        Как только выполнение функции завершится
9        она станет недоступна */
10    int x = 43;
11
12    /* Вызов функции printf
13       с тремя аргументами через запятую*/
14    printf( "Hello,␣world!␣%d␣%d\n", x-1, x);
15
16    /* Напечатает строчку Hello, world! 42 43
17       На место первого спецификатора вывода %d
18        будет подставлено число 42
19       На место второго -- 43 и т.д.*/
20    return 0;
21  }
```

Забудем пока о том, что происходит в самой первой строчке – будем считать, что она позволяет нам пользоваться «встроенными» функциями языка C.

Выполнение программы начинается с функции `main`. Её запускает функция `_start`, содержащаяся в стандартной библиотеке языка.

Нетрудно догадаться, что функции задаются так:

```
return_type function_name( type0 arg0, type1 arg1, ... ) {
...
}
```

Если функция должна просто совершить некоторые действия и ничего не возвращает, то её типом возврата `return_type` указывается `void`. Определённую функцию можно вызывать:

```
function_name( 1, "characters", 9.4 );
```

Если тип её возвращаемого значения не `void`, то можно использовать её результат выполнения как выражение. В таком случае выполнение функции должно завершаться исполнением инструкции `return`, показывающей, что именно является результатом выполнения функции.

```
int square( int x ) { return x * x; }

int sx = square( 5 ); /* sx = 25 */
```

Определённая в стандарте языка функция `printf` умеет печатать строку, в которую вставлены текстовые представления каких-то данных. Первым аргументом ей передаётся строка, а затем всё то, что нужно в эту строку вставить на место **спецификаторов вывода**. Каждый спецификатор начинается на знак процента.

Внимание! `printf` – всегда источник ошибок. Вы можете передать ей больше/меньше аргументов, чем есть спецификаторов в строке, вы можете неправильно указать спецификаторы и компилятор вас ни о чём не предупредит.

Запись строкового литерала (содержимого строки в кавычках) означает, что где-то в памяти будет лежать нуль-терминированная строка с таким содержимым, и в месте, где мы вписали эту строку, будет использоваться указатель на её первый символ, т.е. адрес её начала.

Обратите внимание, что все переменные (`int x`) объявлены в начале блока, составляющего тело функции.

Вопрос 36. *Какие бывают спецификаторы вывода? Посмотрите в любом источнике основные спецификаторы для вывода строк, чисел разных форматов.*

8.2.3 Предложения и выражения. lvalue и rvalue

В программах на C используются т.н. **выражения**. Выражения соответствуют единицам данных.

Литералы и имена переменных являются выражениями. Выражения можно конструировать из других выражений с помощью операторов (+, - и прочих арифметических, логических и побитовых операций) и вызовов функций (кроме возвращающих `void`).

Например, выражениями являются:

```
1
14 + 88
14 + 22 * square( 2 )
x
```

Вопрос 37. *Узнайте, какие арифметические, побитовые и логические операторы есть в C.*

Выражения представляют собой единицы данных, поэтому можно использовать их справа от оператора присваивания.

Слева от оператора присваивания можно использовать только те выражения, которые обозначают единицы данных, *у которых есть адрес*. Они называются **lvalue**, все другие выражения называются **rvalue**.

Конечно, нет смысла в подобных присваиваниях:

```
4 = 2;
square(3) = 9;
```

Предложения (statement) языка заставляют программу что-то делать. В этом и есть основная идея императивного стиля программирования: описание последовательности действий. Они бывают трёх типов:

1. Выражения, завершенные точкой с запятой;

   ```
   1 + 3;
   42;
   square(3);
   ```

   Смысл этих предложений в вычислении значения соответствующих выражений.

2. Блок, ограниченный { и }. Внутри него находятся другие предложения языка;

```
int y = 1 + 3;
{
  int x;
  x = square(2) + y;
  printf("%d\n", x);

}
```

После блока точка с запятой не нужна.

3. Управляющие предложения (`if`, `while`, `for`). Они тоже не завершаются точкой с запятой.

Вопрос 38. *Узнайте про то, как использовать конструкции `if`, `while`, `for`.*

В прошлом разделе мы говорили также о присваиваниях. Присваивания считаются выражениями, их результатом является присвоенное значение. Поэтому присваивания попадают в категорию «выражения, завершённые точкой с запятой». Например:

```
int x;
int y;

x = y = 4; /* x = 4 , y = 4 */
```

Присваивания являются *правоассоциативной операцией*. Это означает, что в длинной цепочке присваиваний скобки ставятся справа налево. Противоположность присваиванию в этом плане, например, операция деления.

Левоассоциативные операции эквивалентны при такой расстановке скобок:

```
40 / 2 / 4
((40 / 2) / 4)
```

Правоассоциативные операции:

```
x = y = z
(x = (y = z))
```

8.2.4 Препроцессор

Помимо вышеперечисленных сущностей, в программах на C встречаются директивы препроцессора. Препроцессор в C похож на препроцессор `nasm`. Основные директивы, которые вы увидите:

- `#define`
- `#include`
- `#ifndef`
- `#endif`

Директива `#define` аналогична директиве `nasm` `%define` для объявления констант. Также она позволяет задавать макроподстановки с параметрами, как `%macro` / `%endmacro`.

```
1  #define MY_CONST_VALUE 42
2  #define MACRO_FUN( x ) ((x) * (x))
```

Важно внутри тела макроподстановки (`MACRO_FUN`) все вхождения аргументов брать в скобки. Причина в том, что макросы ничего не знают про синтаксис программы. Вот пример, когда это приводит к неочевидным результатам:

```
1  #define SQUARE( x ) (x * x)
2
3  int x = SQUARE( 4+1 )
```

После макроподстановки:

```
1  int x = 4+1 * 4+1
```

Очевидно, что значение этого выражения, в силу приоритета операции * над +, 9, а не 25.

Директива `#include` включает содержимое указанного файла в текущий. Имя файла подаётся ей в угловых скобках (`#include <stdio.h>`) или в одинарных кавычках (`#include "file.h"`).

- Файлы в угловых скобках ищутся компилятором в наборе стандарных директорий. Для `gcc` это обычно:
  - `/usr/local/include`
  - `libdir/gcc/target/version/include`

  - `/usr/target/include`
  - `/usr/include`

  Ключ `-I` для `gcc` позволяет добавить к этому списку дополнительные директории.

- Файлы в двойных кавычках ищутся там же, а также в текущей директории.

8.3 Типы данных в C

8.3.1 Обзор

Все типы данных в C можно разделить на следующие категории:

- Встроенные числовые типы (`char`, `int`, `float` и т.д.);
- Массивы – фиксированное количество последовательно лежащих в памяти элементов одного типа;
- Указатели – ячейки, хранящие адреса других переменных, причем в программе мы явно указываем, на данные какого типа он направлен. Указатели могут хранить в том числе адреса функций;
- Структуры – сборные типы из фиксированного количества элементов разного типа;
- Перечисления – числовые типы, принимающие одно из нескольких фиксированных значений. Эти значения имеют имена;
- Объединения – кусок памяти, который можно интерпретировать как данные разных типов (и даже разного размера). Размер объединения равен размеру его наибольшей интерпретации;
- Типы функций;
- Псевдонимы для других типов, созданные с помощью `typedef`.

Также тип можно сделать константным, то есть создать ячейку памяти такого типа, которую нельзя будет просто так изменить.

8.3.2 Приведение типов

Язык C позволяет очень вольно обращаться с типами. Можно интерпретировать данные одного типа как данные другого типа. Достаточно перед значением указать в скобках новый тип.

```
1  int a = 4;
2
3  double b = 10.5 * (double)a; /* now a is a double */
4
5  int b = 129;
6  char k = (char)b; //???
```

Разумееется, стоит обращаться с такими вещами осторожно. Например, char – это, обычно, знаковое число с диапазоном значений от -128 до 127. Разумеется, число 129 не входит в диапазон его представления, однако в нём хватит битов, чтобы записать его – правда, установится знаковый бит, поэтому значение k будет вообще отрицательным.

Вопрос 39. *Объясните подробнее, чему будет равен k и вычислите его значение.*

Вопрос 40. *Что такое* `sizeof`*?*

Вопрос 41. *Чему равны на архитектуре Intel 64:*

- `sizeof(void)`
- `sizeof(0)`
- `sizeof('x')`
- `sizeof("hello")`

Integer promotion

Когда происходят арифметические операции с различными типами, операнды преобразуются к первому из их общих «предков» в этой иерархии:

`int` → `unsigned int` → `long` → `unsigned long` → `long long` → `unsigned long long` → `float` → `double` → `long double`

Например:

```
int i;
float f;
double d = f + i;
```

Значение выражения в третьей строчке посчитается так:

1. Значение `i` приведётся к типу `float` (сама переменная, конечно, не изменится);
2. Полученное значение суммируется с `f`, результат тоже типа `float`;
3. Эта сумма приведётся к типу `double`, чтобы быть записанной в переменную `d`.

На каждом шаге мы протаскивали тип значения дальше по иерархии в сторону `long double`. При этом компилятор неявно для нас вставлял код конверсии из одного типа в другой, т.к. машинное представление для них может быть разное.

8.3.3 Массивы

Массив в C это структура данных, которая может хранить фиксированное количество данных одинакового типа. Соответственно, для массива нужно знать тип элементов и их количество. Так мы объявляем массив данных типа `int` размером 1024:

```
int myarray[1024];
```

Написать вместо фиксированного числа элементов переменную нельзя. Чтобы выделять память под массивы заранее неизвестной длины используется другой механизм, который даже не всегда имеется в наличии – динамическое выделение памяти.

Доступ к элементам осуществляется по индексу. Индексы начинаются с нуля, т.е. самый первый элемент массива имеет индекс 0.

Вот пример объявления массива, двух чтений из него и модификации элемента по индексу:

```
int myarray[1024];
int y = myarray[64];

int first = myarray[0];

myarray[10] = 42;
```

Можно инициализировать массив значениями сразу; в таком случае компилятор может и догадаться о точном размере массива.

```
1  int sarray[] = {1,2,3,4,5};
```

С точки зрения абстрактного вычислителя C массивы это просто области памяти, в которых последовательно лежат данные (одного типа). Никакой информации о длине массивов не сохраняется! Задача программиста – понимать, сколько на самом деле элементов в массиве и не выходить за его границы.

К статически заданным массивам, как `sarray`, можно применять операцию `sizeof`.

Вопрос 42. *Что вернёт `sizeof` от статически заданного массива?*

Вопрос 43. *Как с помощью `sizeof` от статически заданного массива посчитать количество элементов в нём? Напишите универсальное выражение, в котором используется тип элементов массива.*

8.3.4 Типы функций*

Об этом мало кто знает, но у функций в языке C тоже есть типы. Мы не можем создавать переменные таких типов (но «непосредственно заданные константные значения» – можем: функции – это, фактически, и есть экземпляры), но можем использовать их в списке аргументов.

```
1  double root(int number) {...}
2  int somefunction(int param1, double (fArgument)(int) ) { }
```

Здесь второй аргумент имеет тип «функция из `double` в `int`», и описывается как `double (имя аргумента) (int)`. *Экземпляр этого типа ассоциируется с адресом начала кода функции.*

8.3.5 Типы-указатели

Для любого типа (включая типы функций) мы можем использовать производный от него тип – тип-указатель. Обычно для этого нужно дописать звёздочку справа от имени типа. Если у ячейки памяти, хранящей целое число, тип `int`, то у ячейки, хранящей адрес ячейки, в которой находится целое число, тип – `int*`. Об указателях мы поговорим особо.

```
1  int a = 40;
2  int* b = &a;
```

В переменную b мы записали адрес переменной a с помощью оператора взятия адреса &. Очевидно, что этот оператор применим только к **lvalue**.

Противоположностью оператора & является оператор *: если первый берет адрес ячейки памяти, то второй обращается к памяти, используя переданное ему значение как адрес. Пример:

```
int a = 40;
int* b = &a;
*b = 55;   /* a = 55 */
```

К типам-указателям дописывание еще одной звёздочки тоже применимо: можно делать указатель на указатель на указатель...

Строки

Отдельного типа для строк в C нет. Строки это массивы символов, они **нуль-терминированы** Иначе говоря, они заканчиваются символом с кодом 0 и задаются адресом своего начала. Тип одного символа – char, типом для строк будет char*.

Язык C поддерживает строковые литералы в двойных кавычках. Когда вы используете строковый литерал в выражении, вы используете адрес начала строки такого содержания.

```
char* str = "hey, man!";

char* part = str + 5; // part = "man!"
```

8.3.6 const-типы

Если нам дан тип, помимо того, что мы можем образовать от него производный тип-указатель, мы также можем образовать производный константный тип, дописав ключевое слово const. Это означает, что переменная такого типа в программе напрямую изменяться не сможет. Данные таких типов должны инициализироваться сразу при объявлении.

```
int a;
a = 42 ; /* ok */
```

```
const int a; /* compilation error */
```

```
const int a = 42; /* ok */
a = 99; /* compilation error */
```

```
int const a = 42; /* ok */
const int b = 99; /* ok */
```

Как видите, неважно, в каком месте для типов-неуказателей мы напишем `const`.

Интересно то, как `const` взаимодействует с типами-указателями. Если речь идет о переменной-указателе, то она сама является ячейкой памяти и имеет адрес. Поэтому вопрос: мы защищаем от изменения сам указатель, или то, на что он указывает?

Так мы защищаем то, на что указываем:

```
const int * a;
int const * a;
```

Так мы защищаем сам указатель, а содержимое может меняться:

```
int * const a;
```

Так мы защищаем и то, и другое:

```
int const * const a;
const int* const a;
```

Для запоминания есть простое правило:

Замечание. *Те `const`, которые слева от звёздочки, относятся к типу, на который мы указываем. Справа – к самому указателю.*

8.3.7 Определения типа

В данном случае слово «определение» означает не «ответить на вопрос, какой тип», а «дать определение новому типу данных».

В C можно определять новые типы данных на основе существующих:

```
typedef unsigned short int mytype_t;
```

В этом фрагменте кода создаётся новый тип с именем `mytype_t`. Фактически он ничем, кроме имени, не отличается от типа данных `unsigned short int`.

Суффикс _t, конечно, необязателен. Он лишь помогает визуально легко отличать название типа данных от чего-нибудь еще.

Какие возможности даёт механизм опередения новых типов?

1. Создав типы для расстояний и для веса, мы задумаемся, прежде чем в программе складывать килограммы с метрами. Типы здесь служат не только определению формата представления числа (целое беззнаковое соответствующего размера), но и документации значения данных в логике программы;

2. Если мы внезапно решим, что старое определение «метра», подразумевавшее диапазон значений от 0 до 255, нам больше не годится, нам достаточно изменить определение типа, а не выискивать использование именно значений типа «метр» во всей программе;

3. Благодаря слову `typedef` мы можем создавать имена для типов данных «указатель на функцию», типов структур и объединений. Это радикально повышает удобство работы.

Вопрос 44. *Объясните, чему будет равен `x` после выполнения этого кода:*

```
1  int x = 10;
2  size_t t = sizeof(x=90);
```

Общие требования к заданиям

К заданиям по C есть некоторые общие требования:

1. Разделяйте логику программы и вывод. Старайтесь делать максимально переиспользуемый код.

   Если функция суммирования массива `sum` будет сама что-то выводить на экран, то её уже нельзя будет использовать в других программах как часть более сложной системы, не переписав её. К тому же, исходя из своего названия функция должна *суммировать*, а *не суммировать И выводить результат на экран.*

2. Только C89;

3. Никаких комментариев на русском языке или транслитом;

4. Названия переменных и функций должны происходить от правильных и уместных английских слов.

   Неправильно: `char massive[128];`

   Правильнее: `char array[128];`

   Еще лучше: `char input_buffer[128];`

5. Помните про модификатор `const`. Всё, что вы хотите гарантированно защитить от изменений, помечайте им. В том числе, указатели на то, что вы гарантированно не измените. Например, в функции, которая принимает массив и хочет не изменяя его вычислить какую-то информацию о нём, разумно защитить сам массив от перезаписи внутри функции с помощью `const`.

Пока что не задумывайтесь о том, зачем нужны строчки `#include`. Будем считать, что они позволяют нам использовать функции из стандартной библиотеки C.

8.3.8 Задание: суммирование массива

Есть глобальная переменная-массив из чисел типа `int`. Необходимо:

- Написать функцию sum, которая будет принимать массив чисел типа int и, возможно, что-то еще, и возвращать результат суммирования чисел.

Вам потребуется научиться использовать :

- `sizeof` для правильного определения размера статически заданного массива;
- `size_t` как тип для длины массивов и их индексов;
- модификатор `const` для указателей и не только;
- функция `printf` и модификаторы вывода;

Ограничения:

- Ваша функция `sum` должна работать с любым массивом, в том числе произвольной длины, заранее вам неизвестной.

Листинг 8.2: labs/3/sum_static/stud/sum_static.c

```
1  #include <stdio.h>
2  #include <stdlib.h>
3
4  int array[] = {1,2,3,4,5,6,7,8,9,10};
5
6  int main( int argc, char** argv ) {
7
8  }
```

8.3.9 Задание: проверка числа на простоту

На вход подается число. Необходимо:

- Написать функцию `int is_prime( unsigned long )`, которая проверяет число на простоту;
- Написать функцию `main`, которая вызывает функцию `is_prime` и, в зависимости от результата, печатает `yes` или `no`.

Вам потребуется научиться использовать

- функцию `scanf` с правильными спецификаторами формата;
- функцию `strtoul`;

Ограничения:

- В поток ввода подаётся не более 128 символов
- В поток могут вводиться не числа или отрицательные числа, тогда в ответ нужно написать сообщение об ошибке.

Обратите внимание:

- Функция `is_prime` должна принимать `unsigned long`. Всегда ли это то же самое, что `unsigned int`? Как подсказать компилятору, что написанное число имеет формат `long`, а не `int`?

Листинг 8.3: labs/3/prime/stud/prime.c

```
1  #include <stdio.h>
```

```
#include <stdlib.h>

int is_prime( unsigned long num ) {
}


int main( int argc, char** argv ) {
}
```

8.3.10 Структуры, перечисления, объединения

Один из основных принципов осмысления для человеческого мозга — абстракция. Абстракция означает замену низкоуровневых понятий более удобными, высокоуровневыми. Например, думать о походе в булочную удобнее в понятиях уровня «повернуть за угол» или «перейти дорогу», нежели «переставить правую ногу на 42 сантиметра вперёд». Так же и в программировании, мы создаём взамен базовых сущностей некоторые новые сущности новых типов, чтобы нам было удобнее. Например, вместо того, чтобы оперировать с тройками чисел мы объединяем их под именем «вектор».

Структура — некий «сборный» тип данных.

```
struct { int a; char b; } d;
d.a = 0;
d.b = 'k';
```

В этом фрагменте кода мы объявили структуру с именем d, которая хранит в себе два элемента: `a` типа `int` и `b` типа `char`. Мы можем обращаться к отдельным полям структуры, написав её имя, точку и имя поля.

Вышеуказанный способ «одноразовый» — чтобы объявить новую структуру такого же типа придётся переписать её определение. Мы можем создать новый тип – «структура с именем `struct_name_t`», которым в дальнейшем и пользоваться.

```
struct struct_name_t { int a; char b; };

...

struct struct_name_t d;
d.a = 0;
```

```
7    d.b = 'k';
```

Обратите внимание, что без слова `struct` при объявлении переменной `d` не обойтись. Язык C определяет два разных пространства имён типов: глобальное и `struct`-пространство[1]. Идентификаторы с одинаковыми именами из двух этих пространств не конфликтуют между собой. Например:

```
1   struct S { int a; } ;
2   void S( void ); /* ok */
```

Если мы не хотим каждый раз писать `struct S`, то мы можем сделать в глобальном пространстве имён псевдоним для него:

```
1   typedef struct S S; /* struct S -> S */
```

Теперь можно писать также:

```
1   S example;
```

Структуры можно сразу инициализировать значениями с синтаксисом, похожим на инициализацию массивов:

```
1   struct S {char* name; int value; };
2   ...
3   S new_s = { "myname", 4 };
```

Объединения во всём похожи на структуры, только их поля накладываются друг на друга, а не лежат последовательно одно за другим.

Объединения, как и структуры, имеют своё пространство имён. Рассмотрим такой пример:

```
1   typedef union {
2    int integer;
3    short shorts[2];
4    } dword_t;
5
6    ...
7    dword_t test;
8    test.integer = 0xAABBCCDD;
```

[1] На самом деле, есть и другие.

Мы объявляем объединение, которое хранит в себе целое число размером 4 байта (x32 или x64 архитектура). В то же время оно хранит в себе массив из двух чисел по два байта каждое. Если бы это была структура, они бы лежали в памяти последовательно, но это объединение, так что первое поле содержит в себе оба числа из массива `shorts`. Присвоив полю `integer` число $AABBCCDD_{16}$ мы автоматически изменили и остальные поля, на которые оно накладывается. Теперь если мы попытаемся вывести на экран сначала `integer`, а потом `shorts[0]` и `shorts[1]`, то мы увидим следующее:

```
aabbccdd
ccdd aabb
```

Как видите, поля действительно накладываются друг на друга.

Вопрос 45. *Почему при выводе `short[0]` и `short[1]` (именно в таком порядке) вывелась строка `ccdd aabb`, а не `aabb ccdd`?*

Так как структуры можно вкладывать внутрь других структур (то же с объединениями, их можно и перемешивать), можно достигать интересных вещей.

Например, в этом примере мы исполььзуем объединение структуры и массива из трёх элементов. Структура не имеет имени, и к её полям можно обращать как к полям объединения!

```
union pixel_t {
   struct {
      char g, b, r;
   }
   char comps[3];
};

...

union pixel_t px;

px.g = 0; /* perfectly fine */
```

Перечисления — простой тип данных на основе целого числа. Мы фиксируем несколько значений и даём им имена.

Например, светофор бывает в нескольких состояниях:

- Красный;
- Красный и жёлтый;
- Жёлтый;
- Зелёный;
- Ничего не горит.

В языке C это будет выглядеть так:

```
1
2  typedef enum {
3     red = 0,
4     red_and_yellow,
5     yellow,
6     green,
7     nothing
8  } light_t;
9
10 ...
11    light_t l = nothing;
12 ...
```

Константа 0 получила имя `red`, константа 1 стала `red_and_yellow` и т.д.

Для перечислений, как и для структур и объединений, есть отдельное пространство имён.

8.4 Типы данных в программировании*

Так как процесс программирования на C сводится к манипуляции данными разных типов, мы поговорим о том, что вообще такое типы данных в языках программирования.

Во многих областях программирования и computer science был пройден путь от состояния «всё на свете одного типа» к типизации. Пройден хотя и по-разному, но по схожим причинам. Например, нетипизированы:

1. Лямбда-выражения в бестиповом лямбда-исчислении;
2. Множества во многих теориях множеств;

3. S-выражения в языке LISP;

4. Битовые строчки.

С точки зрения программирования пока что нас будут интересовать именно битовые строчки. Для компьютера всё на свете — битовые строчки фиксированного размера. Они могут интерпретироваться как числа, строки символов или любым иным способом. Можно считать, что в компьютере данные нетипизированы.

Однако начиная работать в какой-то среде без типов, мы сами начинаем разграничивать объекты на некоторые категории, с каждой из которых мы работаем единообразно. Какие-то конкретные битовые строчки в конкретных ячейках памяти мы договариваемся считать числами, какие-то — числами с плавающей точкой, исполняемым кодом и так далее.

Кажется, что так у нас и появляются типы? Однако на деле это лишь иллюзия. Ведь мы по прежнему можем прибавить к ячейке, содержащей код, число 42. Язык программирования никак нам не мешает ломать программу и творить всяческий беспредел. Чтобы перейти к настоящей типизации, нам необходимы ограничения на операции, которые можно производить с данными. Негоже складывать числа и строки символов! Соответствие этим ограничениям может проверяться во время исполнения программы (**динамическая типизация**) или во время компиляции (**статическая типизация**).

Итак, рассмотрим всё множество данных, с которыми мы работаем в программе. Выделим в нем подмножества таких данных, с которыми мы работаем единообразно. Объявляя такое подмножество типом данных мы также должны определить, какие операции с ним разрешены.

Часто тип также определяет, как именно хранятся данные (если хранятся вообще) и как именно осуществляются вышеупомянутые операции.

В программировании типизация данных защищает от некорректных операций над объектами. Например, мы не можем складывать строки и числа, хотя, так как и те и другие хранятся в памяти как битовые строчки, в принципе, могли бы «наивно» проделать такую операцию с их представлениями, а потом проинтерпретировать результат каким-то образом.

Кроме того, если модель вычислений какого-либо языка допускает такую операцию, как присваивание, типы дают защиту и от её некорректного применения (присвоим, например, целому числу строчку символов). Наконец, передавать аргументы в функцию мы тоже

можем только уместных с точки зрения этой функции типов. В нетипизированном мире мы могли бы взять синус строки «ugly» и получить непредсказуемый результат.

Итак, типизация защищает нас от некорректных операций, в частности, присваивания с разными типами данных и применения функций к аргументам несовместимых типов.

8.4.1 Виды типизации

Работа с типами осуществляется по-разному.

1. **Статическая типизация** означает, что вся работа по проверке корректности выражений просходит во время анализа программы при компиляции. Поэтому после компиляции мы можем быть уверены в том, что некоторых ошибок, от которых нас защищают типы, в программе точно не будет.

   Динамическая типизация, наоборот, работает именно во время выполнения. В случае некорректной с точки зрения типов операции, мы столкнёмся с ошибкой выполнения программы, а не ошибкой её компиляции.

2. **Строгая типизация** подразумевает, что все операции должны быть произведены в точности с теми данными, которые им приходят.

   Нестрогая типизация подразумевает, что существуют неявные преобразования типов, которые делают возможными операции с данными, которые *не совсем* им подходят.

3. Помимо этого иногда говорят о такой классификации:

   Явная типизация когда мы явно указываем, какие данные какого типа.

   Неявная типизация когда мы даем компилятору возможность догадаться о типе, если это возможно.

Строгая статическая типизация

Типы проверяются во время компиляции и отношение к ним строгое.

В OCaml есть два вида сложения: для целых чисел (`+`) и для чисел с плавающей точкой (`+.`). Тип данных целых чисел называется `int`, а с плавающей точкой – `float`. Соответственно, правильно использовать:

```
<float> +. <float>
<int>   +  <int>
```

Следующий код уже приведет к ошибке:

```
4 +. 1.0
```

Мы использовали данные типа `int` (число 4) в том месте, где ожидался `float`, и неважно, что с точки зрения здравого смысла было бы логично неявно преобразовать целое число в `float`. Чтобы разрешить ситуацию придётся использовать специальную операцию преобразования типов из `int` во `float`. Это воплощение принципа строгости к типам.

Нестрогая статическая типизация

В языке C нестрогая статическая типизация. Все типы указываются и проверяются во время компиляции, но ограничения на их использования не такие строгие.

```
1   double x = 4 + 3.0;
```

Здесь мы складываем целое число типа `int` и число с плавающей точкой типа `double`. Это данные разных типов, которые по-разному представлены на уровне машинных кодов. Сложение их – совершенно нетривиальная задача. Компилятор C автоматически вставит код для преобразования целого числа 4 в число с плавающей точкой 4.0, сложения его с 3.0 и записи результата в ячейку, соответствующую переменной x.

Нестрогость в том, как *прозрачно для программиста* компилятор сгенерировал код преобразования типов (из `int` в `double`).

Строгая динамическая типизация

Такая типизация используется в языке Python: все ограничения на типы должны выполняться, но проверка происходит не в момент ком-

пиляции, а в момент выполнения.

```
>>> "e" + 3
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: cannot concatenate 'str' and 'int' objects
>>> 3 if True else 1.0 + "e"
3
```

Сначала мы пытаемся сложить строку и число. При попытке *выполнить* этот код возникнет ошибка. Однако если мы не дойдём до выполнения этой конструкции, ошибки не будет.

Нестрогая динамическая типизация

В последнем примере на Python мы складывали строку и число. Но в строке содержалось тоже число (можно было вычленить его оттуда, опираясь на здравый смысл, думает программист). Поэтому в языке с нестрогой типизацией эта операция **может** быть позволена при условии, что существует такой набор преобразований типов, при котором эта конструкция с этими конкретными значениями данных станет корректной. Например, число мы приведём к типу «строка», затем применим + к двум строкам и получим их конкатенацию.

Для примера рассмотрим язык Javascript.

```
3 + 4  --> 7
'3' + 4 --> '34'
```

Во второй строчке результат её выполнения равен конкатенации двух строк. Несмотря на то, что второй аргумент + это число, оно неявно приводится к типу строки.

8.4.2 Полиморфизм

Теперь, когда мы узнали кое-что про типы, подумаем о том, что вообще говоря конкретная единица данных может одновременно попадать в несколько различных типов. Например, число 1, в принципе, может быть интерпретировано как целое число и как число с плавающей запятой. Полиморфизм как раз и означает, в противовес мономорфизму, возможность для каких-то переменных и значений иметь несколько различных типов, а для функций – работать не только с одним набором типов аргументов.

Виды полиморфизма

Так как данные сами по себе не представляют для нас интереса в отрыве от функций, которые работают с ними, принимая их как параметры, то мы будем говорить о полиморфизме функций (или, что по сути одно и то же, операторов) как о том, как изменяется алгоритм работы одной и той же функции при передаче ей параметров разных типов.

Полиморфизм бывает разный. Часто говорят о двух категориях полиморфизма, каждая из которых делится еще на две:

1. Универсальный полиморфизм (universal):
   - Параметрический (parametric) ;
   - Включающий (inclusion);
2. ad hoc:
   - Перегрузка (overloading);
   - Принудительный (coercion).

Замечание. *Ad hoc — распространённое латинское выражение, означающее примерно «к месту».*

Универсальный полиморфизм — обычно функции, которые принимают аргументы потенциально бесконечного количества типов и работающих на них всех одинаково.

- В случае параметрического полиморфизма, функция явно или неявно принимает дополнительный аргумент-тип, который показывает, данные какого типа передаются ей в другом аргументе.
- В случае включающего полиморфизма, мы говорим о неком супертипе который включает в себя остальные, например, об общем родителе нескольких классов в языках, поддерживающих ООП.

Ad hoc полиморфизм — обычно функции, которые принимают аргументы только фиксированного количества типов и, возможно, работающих по-разному для разных типов.

- Перегруженные функции — фактически несколько разных функций с одним именем, отличающиеся типом аргументов. Их алгоритмы работы могут быть очень разными!
- Принудительный полиморфизм — когда функция объявлена, как принимающая данные некоторого типа X, а если передать ей тип Y,совместимый с X, то данные типа Y сконвертируются в данные типа X. Например, в строчке (язык C)

```
1   1 + 0.5
```

единица имеет тип «целое число» (int), а литерал 0.5 — число двойной точности с плавающей запятой (double). Эти числа представляются в компьютере по-разному и сложить их – нетривиальная задача. Компилятор автоматически вставляет машинный код для того, чтобы сконвертировать число 1 в формат чисел с плавающей точкой, а затем складывает два операнда. Единицу «насильно сконвертировали» в double.

Порой сложно определить, какой именно тип ad hoc полиморфизма используется в данном месте программы. Например, практически в любом языке мы можем написать такие строчки:

```
1   3  + 4
2   3  + 4.0
3   3.0 + 4
4   3.0 + 4.0
```

Оператор «плюс» тут явно полиморфен. Он оперирует с данными как целого типа, так и с числами с плавающей запятой, и даже с теми и другими одновременно. Но как именно? Давайте попробуем выявить возможные объяснения:

- У плюса есть четыре перегрузки для каждого из случаев;
- У плюса есть две перегрузки: для сложения целых чисел и для сложения чисел с плавающей запятой. Если один из аргументов во втором случае — целое число, то мы принудительно переводим его в тип действительных чисел.
- Мы умеем складывать только действительные числа. В первых трёх случаях мы приводим целые к типу действительных чисел, а потом складываем.

Хороший обзор можно найти в статье [4].

8.4.3 Полиморфизм в C

8.5 Функции, процедуры

Мы условно можем разделять функции и процедуры, подразумевая, что процедуры ничего не возвращают, а функции возвращают значение. Соответственно, вызов функции является выражением, а вызов процедуры – нет.

Пример процедуры:

```
1  void myproc ( int a, int b )
2  {
3    printf("%d", a+b);
4  }
```

Процедура называется `myproc`, тип возвращаемых ею данных — `void`, то есть она ничего не возвращает. Процедура принимает два параметра: два целых числа с именами a и b. А это функция:

```
1
2  int myfunc ( int a, int b )
3  {
4    return a + b;
5  }
```

В отличие от процедуры, она возвращает некоторое значение типа `int`. Выполнение функции завершается предложением `return`, которое указывает, что именно будет являться результатом выполнения функции.

Если у функции или процедуры нет аргументов, принято писать в скобках ключевое слово `void`:

```
1  int myfunction( void ) { return 0; }
```

Вызов функции можно использовать как составную часть сложного выражения:

```
1  int a = 42 + myfunc( a, a );
```

В данном примере какой-то переменной a присвоится значение, получаемое по формуле $42 + (a + a)$.

Телом функции является предложение-блок. В нём можно объявить переменные, локальные для этого блока (и для этой функции). Они никак не видны извне функции. По этой причине в разных функциях можно объявлять локальные переменные с одним и тем же именем, из-за разной их области видимости никаких конфликтов имён не происходит.

Замечание. *Область видимости переменных – до конца того блока, в котором они объявлены. Т.е. можно сделать «еще более локальные» переменные, нежели локальные для функции.*

Глобальные переменные, в отличие от локальных, объявляются вне функций, и к ним есть доступ из любого места программы.

Глава 9

Организация кода

9.1 Объявление и определение

9.1.1 Функции и процедуры

Компиляторы C исторически были сделаны так, чтобы компилировать файл за один проход по нему. Поэтому каждая функция должна быть описана перед её вызовом. Если функция определена в другом файле или же в том же файле, но уже после её использования (по тексту программы), то её нужно сначала объявить вместе со всей **сигнатурой** (параметрами и возвращаемым значением) **перед** использованием. Такое объявление называется **прототипом**.

Для функции

```
1  int square(int x) { return x*x; }
```

прототип выглядел бы так:

```
1  int square(int x);
```

Имена аргументов в прототипах можно опускать:

```
1  int square(int);
```

Рассмотрим три сценария:

1. Функция сначала определена, потом вызывается:

   ```
   1  int square( int x ) { return x * x; }
   ```

```
2
3   ...
4   int z = square(5);
```

2. Описание прототипа предшествует вызову, описание тела функции после вызова:

```
1   int square( int x );
2
3   ...
4   int z = square(5);
5
6   ...
7
8   int square( int x ) { return x * x; }
```

3. Ошибка: прототипа нет, описание тела функции после вызова:

```
1   int z = square(5);
2
3   ...
4
5   int square( int x ) { return x * x; }
```

Когда компилятор попытается скомпилировать строчку с вызовом `square`, он сообщит об ошибке, т.к. он не знает ничего о том, что такое `square` и какие аргументы он принимает.

Объявление функции до её использования также называется **forward declaration**.

9.1.2 Структуры. Рекурсивные типы данных

Допустим, мы хотим написать рекурсивную структуру данных, например, **связный список (linked list)**. Каждый элемент связного списка хранит в себе, собственно, значение, и указатель на следующий элемент. Последний элемент в указателе на следующий обязательно хранит 0 как признак окончания списка.

```
1   struct llist_t {
2     int value;
```

```
3     struct llist_t* next;
4   };
```

Можно объединить этот паттерн с `typedef`, создав сразу тип в глобальном пространстве имён:

```
1   typedef struct llist_t {
2     int value;
3     struct llist_t* next;
4   } llist_t;
```

Так мы создали два имени для типа этой структуры:

- `struct llist_t` в пространстве имён `struct`;
- `llist_t` в глобальном пространстве имён.

9.1.3 Неполные типы

Возможно объявить т.н. неполный тип, то есть описать его принадлежность к пространству имён, но не реализацию. Например:

```
1   struct llist_t;
```

При таком подходе можно работать только с указателями на структуры такого типа. При попытке разадресовать его компилятор выдаст ошибку, ведь он не знает внутреннего устройства этой структуры! По этой же причине нельзя реализовывать функции, которые возвращают экземпляр неполного типа по значению, а не по указателю (объявлять – можно).

```
1   struct llist_t;
2
3   struct llist_t* f() { ... } /* ok */
4   struct llist_t g();   /* ok */
5   struct llist_t g() { ... } /* bad */
```

9.2 Обращение к коду из других файлов

Разумеется, можно вызывать код и из других файлов. Чтобы вызвать функцию из другого файла, необходимо добавить её **прототип** в текущий файл.

Листинг 9.1: "square.c"

```
1  int square( int x ) { return x * x; }
```

Листинг 9.2: "main.c"

```
1  int square( int x );
2
3  int main(void) {
4     printf( "%d\n", square( 5 ) );
5     return 0;
6  }
```

Каждый файл с кодом является отдельным модулем, т.е. компилируется раздельно с другими файлами. Из него создаётся объектный файл.

Если мы используем ELF, то при компиляции картина будет такой:

```
> gcc -c main.c
> objdump -t main.o

main.o:     file format elf64-x86-64

SYMBOL TABLE:
0000000000000000 l    df *ABS*  0000000000000000 main.c
0000000000000000 l    d  .text  0000000000000000 .text
0000000000000000 l    d  .data  0000000000000000 .data
0000000000000000 l    d  .bss   0000000000000000 .bss
0000000000000000 l    d  .note.GNU-stack
                                 0000000000000000 .note.GNU-stack
0000000000000000 l    d  .eh_frame
                                 0000000000000000 .eh_frame
0000000000000000 l    d  .comment
                                 0000000000000000 .comment
0000000000000000 g     F .text  000000000000001c main
0000000000000000         *UND*  0000000000000000 square
```

```
> gcc -c square.c
> objdump -t square.o
square.o:     file format elf64-x86-64

SYMBOL TABLE:
0000000000000000 l    df *ABS*  0000000000000000 square.c
0000000000000000 l    d  .text  0000000000000000 .text
0000000000000000 l    d  .data  0000000000000000 .data
0000000000000000 l    d  .bss   0000000000000000 .bss
0000000000000000 l    d  .note.GNU-stack
                              0000000000000000 .note.GNU-stack
0000000000000000 l    d  .eh_frame
                              0000000000000000 .eh_frame
0000000000000000 l    d  .comment
                              0000000000000000 .comment
0000000000000000 g     F .text  0000000000000010 square
```

Как видно, функция `square` и функция `main`, не будучи никак особенно выделены, получили в соответствие глобальные символы. Прототип же функции получил символ из неопределённой секции:

```
0000000000000000         *UND*  0000000000000000 square
```

`gcc` умеет также и связывать файлы с кодом друг с другом; заодно он также подключает стандартную библиотеку языка.

После компоновки таблица символов очень сильно разрастётся благодаря стандартной библиотеке и некоторым вспомогательным символам.

```
> gcc -o main main.o square.o
> objdump -t main | grep square
```

```
0000000000000000 l    df *ABS*  0000000000000000     square.c
00000000004004d2 g     F .text  0000000000000010     square
```

9.2.1 Обращение к данным из других файлов

Если глобальная переменная, которая нам нужна, находится в другом файле, следует упомянуть её с пометкой `extern`:

Листинг 9.3: "square.c"

```
1  extern int z;
2  int square( int x ) { return x * x + z; }
```

Листинг 9.4: "main.c"

```
1  int z = 0;
2  int square( int x );
3
4  int main(void) {
5     printf( "%d\n", square( 5 ) );
6     return 0;
7  }
```

Использование таблицы символов аналогично тому, что происходило с функциями.

9.2.2 Стандартная библиотека

Чтобы использовать стандартные функции библиотеки языка C, нужно включать соответствующие заголовочные файлы. В них, фактически, находится не код самих функций, а их сигнатуры.

Компилятор генерирует ссылки на эти функции, думая, что они находятся в других файлах, а компоновщик связывает те файлы, которые ему подаются на вход. Если компоновщик знает местоположение стандартной библиотеки C, то вы можете явно не указывать ему, откуда её взять.

Именно так происходит подключение библиотек. Не забывайте: сам по себе #include — не подключение библиотеки, а просто подстановка текста файла!

В доказательство:

```
>  cat p.c
#include <stdio.h>

> gcc -E -pedantic -ansi p.c | grep " printf"
extern int printf (const char *__restrict __format, ...);
```

Опуская нестандартное ключевое слово `__restrict`, мы видим, что в файле `stdio.h`, включенном в `p.c`, находится прототип функции `printf`, но не её код. Три точки означают произвольное количество аргументов далее, о чем мы поговорим особо.

Сам же код `printf` в скомпилированном виде находится в стандартной библиотеке C. Покажем это с помощью утилит `ldd` (показывает информацию о связанных с объектным файлом динамических библиотеках) и `readelf`. Скомпилируем файл:

Листинг 9.5: 'main.c'

```
1  #include <stdio.h>
2
3  int main()
4  {
5     printf("Hello␣World!\n");
6     return 0;
7  }
```

```
>  gcc main.c -o main && ldd main
        linux-vdso.so.1 (0x00007fff4e7fc000)
        libc.so.6 => /lib/x86_64-linux-gnu/libc.so.6 (0x00007f2b7f6bf000)
        /lib64/ld-linux-x86-64.so.2 (0x00007f2b7fa76000)

> readelf -s /lib/x86_64-linux-gnu/libc.so.6  | grep " printf"
   596: 0000000000050d50   161 FUNC    GLOBAL DEFAULT   12
printf@@GLIBC_2.2.5
  1482: 0000000000050ca0    31 FUNC    GLOBAL DEFAULT   12
printf_size_info@@GLIBC_2.2.5
```

```
  1890: 0000000000050480  2070 FUNC    GLOBAL DEFAULT   12
printf_size@@GLIBC_2.2.5
```

Действительно, файл связался динамически с библиотекой `libc`,

9.2.3 Как связаны заголовочные файлы и библиотеки?

Вкратце: прямой связи нет. Часто студенты рассуждают так: к примеру, чтобы использовать функцию `printf` необходимо в начале файла написать `#include <stdio.h>`. Это означает подключение библиотеки `stdio.h`, в которой находится код функции.

Это распространённая ошибка, `stdio.h` – не библиотека, а лишь текстовый файл с набором прототипов некоторых функций из стандартной библиотеки. В библиотеке же уже в скомпилированном виде (машинными командами) хранится некоторый объем кода, в том числе – и код, соответствующий функции `printf`.

9.3 Использование препроцессора

9.3.1 Основные возможности

Препроцессор C не обладает большими возможностями. Вот его основные директивы:

`#define`

Эта директива встречается в нескольких вариациях.

```
1   #define FLAG
```

Это означает, что некоторая сущность, называемая «символ препроцессора» (по аналогии с символами в объектных файлах), определена. У неё, однако, нет никакого значения. Необходимо это может быть для условных проверок на то, определен символ или нет (см. `#ifdef`).

```
1   #define C 42
```

Это означает, что при написании в программе токена C вместо него будет подставлена строчка 42.

```
#define MAX(a, b) ((a)>(b))?(a):(b)
```

Это подстановка с параметрами. Если написать в тексте программы

```
int x = MAX(4+3, 9)
```

то препроцессор преобразует её в строчку:

```
int x = ((4+3)>(9))?(4+3):(9)
```

Обратите внимание на скобки, в которых заключены все вхождения параметров. Это необходимо, чтобы сложные выражения (такие, как 4+3) не разваливали выражения внутри макроса, ведя к неочевидной их интерпретации.

Добавить определёные символы можно не только с помощью строчек `#define SYM VALUE`, но и передавая их `gcc` с помощью ключа `-D`, например, `gcc -DMYFLAG=3` или `gcc -DMYFLAG`.

#include

Эта директива просто включает содержимое другого файла как есть.

#ifdef

С помощью этой директивы можно включить кусок кода в файл если определён символ препроцессора.

Включить код если символ определён:

```
#ifdef MYFLAG
/*code*/
#endif
```

Включить код если символ определён, а если нет, то другой код:

```
#ifdef MYFLAG
/*code*/
#else
/*other code*/
#endif
```

Включить код если символ неопределён, а если определён, то другой код:

```
#ifndef MYFLAG
/*code*/
#else
/*other code*/
#endif
\
```

9.3.2 Include Guard

Схема с добавлением прототипов вызываемых функций неудобна. Особенно тяжело становится поддерживать её в случае, когда в каждом модуле много функций, которые можно вызывать.

Чтобы повысить удобство работы для каждого файла с кодом, доступным другим файлам, выполняют следующие действия:

1. Создать так называемый **заголовочный файл** с расширением `.h` и тем же именем;
2. Для каждой экспортируемой функции добавить в заголовочный файл её прототип;
3. Если есть необходимость обращаться к экспортируемым функциям, то в начало файла вписывается строчка:

   ```
   #include "header_file.h"
   ```

   Эта директива включит содержимое файла `header_file` как есть в текущий файл. Файл `header_file` содержит прототипы функций, которые и попадут на место этой строчки, позволяя вызывать их.

Пример:

Листинг 9.6: "square.c"

```
int square( int x ) { return x * x; }
int succ( int x ) { return x+1; }
```

Листинг 9.7: "square.h"

```
int square( int x );
```

Листинг 9.8: "main.c"

```
#include "square.h"

int main(void) {
   printf( "%d\n", square( 5 ) );
   return 0;
}
```

Заголовочные файлы не участвуют в компиляции иначе, как в составе `*.c` файлов.

Почему препроцессор это ужасно Препроцессинг ничего не знает о структуре языка и запутывает (подчас очень сильно!) структуру программ. Казалось бы, простая задача – найти в тексте программы из сотни файлов все вхождения макроса `min` оказывается маловыполнимой. В общем случае, макросы могут быть вложенными,поэтому, чтобы искать вхождения `min` во всех местах, где они на самом деле присутствуют, нужно раскрыть все макросы. Как только препроцессор сделает свою работу, макрос `min` тоже исчезнет из программы, так как будет раскрыт.

9.3.3 Include Guard

В одном файле с кодом нельзя иметь более одного прототипа функции. Мы, конечно, не будем писать сами несколько прототипов для одной и той же функции. Однако если мы включаем много заголовочных файлов, опасность такое получить более, чем реальна.

Пример: пусть есть три файла, из них два заголовочных:

Листинг 9.9: "a.h"

```
void a(void);
```

Листинг 9.10: "b.h"

```
#include "a.h"
void b(void);
```

Листинг 9.11: "main.c"

```
1  #include "a.h"
2  #include "b.h"
```

Как будет выглядеть файл `main.c` после того, как отработает препроцессор?

```
> gcc -E main.c
# 1 "main.c"
# 1 "<built-in>"
# 1 "<command-line>"
# 1 "/usr/include/stdc-predef.h" 1 3 4
# 1 "<command-line>" 2
# 1 "main.c"
# 1 "a.h" 1
void a(void);
# 2 "main.c" 2
# 1 "b.h" 1
# 1 "a.h" 1
void a(void);
# 2 "b.h" 2
void b(void);
# 2 "main.c" 2
```

Теперь `main.c` содержит два прототипа функции `void a(void)`, поэтому при компиляции возникает ошибка.

Чтобы этого не происходило, используются два подхода:

- В начале заголовочного файла написать строчку:

  ```
  1  #pragma once
  ```

  Этот способ работает в большинстве компиляторов, но не является частью стандарта языка.

- Использовать т.н. **Include guards**:

  Листинг 9.12: "newa.h"

  ```
  1  #ifndef _FILE_H_
  ```

```
2  #define _FILE_H_
3
4  void a(void);
5
6  #endif
```

Текст между директивами `#ifndef X` и `#endif` включится препроцессором только в случае, когда препроцессорный символ `X` не было определено с помощью директивы `#define`. Если он будет включен, то в его составе появится и директива `#define X`, которая определит требуемый символ. Если вставить этот же файл повторно далее по тексту, то условие в новой копии проверки `#ifndef X` не выполнится, соответственно, содержимое блока `#ifndef X ... #endif` будет удалено.

Обычно в качестве `X` используется имя файла заглавными буквами, декорированное символами подчёркивания.

9.4 Побочные эффекты выражений

Если бы не существовало побочных эффектов, то процедуры бы не делали абсолютно никакой полезной работы. Любые изменения состояния программы, не связанные с конкретным значением, возвращаемым данной функцией, но инициированные изнутри неё, называются **побочными эффектами**. Например, действия ввода-вывода, изменение глобальной переменной изнутри функции — побочные эффекты. Без побочных эффектов трудно обойтись, но они делают программу менее предсказуемой, некрасивой и более запутанной. По возможности следует избегать их.

Побочные эффекты есть не только у функций. Операторы **преинкремента** и **постинкремента** также обладают побочным эффектом.

- Преинкремент:

```
1  int x = 10;
2
3  int y = 3 + (++x);
```

Здесь сначала содержимое `x` инкрементируется (увеличивается на 1), а затем полученное значение суммируется с числом 3.

- Постинкремент:

```
1  int x = 10;
2
3  int y = 3 + x++;
```

Значением выражения `x++` является исходное значение `x` (10). Однако при выполнении этого выражения значение переменной `x` увеличится на 1.

Аналогично действуют операторы пре- и постдекремента.

Глава 10

Работа с памятью

10.1 Указатели

> Указатели обычно смешивают в одну кучу с операторами GOTO, характеризуя их как чудесный способ написания программ, которые невозможно понять.
>
> Д. Ритчи, создатель языка C

10.1.1 Зачем нужны указатели?

Язык C имеет фон Неймановскую модель вычислений. Это обуславливает то, что процесс выполнения программы на C это манипуляция с данными в памяти. Память адресуема, и это её свойство можно использовать для того, чтобы более эффективно обращаться с данными. У этого, однако, есть своя цена: манипуляции с адресами легко приводят к ошибкам.

Необходимость хранить адреса и манипулировать ими – причина появления **указателей**. Указатели это набор типов данных, хранящих адреса переменных других типов данных. Если существует тип `X`, то существует и тип `X*`, хранящий адрес переменной типа `X`, тип `X**`, хранящий адрес переменной, хранящий адрес переменной типа `X`.

Рассмотрим следующий пример:

```
1  int a = 4;
2  int* p_a = &a;
3  *p_a = 10; /* a = 10*/
```

В первой строчке мы объявляем переменную типа int. Во второй строчке мы объявляем переменную типа «указатель на int», затем с помощью оператора & берём адрес переменной. В третьей — обращаемся «по указателю» (это называется *разыменовывание*) и в ту ячейку, на которую указывает p_a, заносим число 10. То есть, фактически, в ячейку a.

В терминах абстрактного вычислителя:

- `a` – имя для набора ячеек памяти абстрактного вычислителя, которые содержат число 4;
- `p_a` – имя для набора ячеек памяти абстрактного вычислителя, которые содержат адрес переменной типа `int`;
- `p_a` хранит в себе адрес ячейки `a`;
- `*p_a` это то же самое, что `a`;
- `&a` численно равно `p_a` . Однако это не то же самое, так как `p_a` это имя для ячеек памяти, а `&a` это то, что содержится в `p_a`, то есть число-адрес.

Замечание. *Очевидно, что результат применения оператора взятия адреса* `&` *никогда не будет* ***lvalue****.*

10.1.2 Адресная арифметика

С указателями разрешено производить следующие операции:

- Прибавлять целые числа;

  Казалось бы, у нас есть адреса. Указатель хранит в себе адрес ячейки, так почему бы не сделать для всех указателей один тип «указатель на ячейку памяти»? В чём разница между `int*` и `char*`?

  Большое значение при работе с указателями занимает размер элемента, на который мы указываем. Прибавляя к указателю число, мы по факту смещаем его не ровно на столько же байт, а на столько же размеров элементов, на которые он указывает. Посмотрим на то, как меняется значение указателя. Пусть его начальное значение было 1000_{10}:

```
int a = 42;
int* p_a = &a;
p_a += 42;     /* 1000 + 42 * sizeof( int ) */
p_a = p_a + 1; /* 1168 + 1 * sizeof( int ) */
p_a --;        /* 1172 - 1 * sizeof( int ) */
```

- Брать адрес; Так как указатель это переменная, то у него есть адрес, который можно взять с помощью амперсанда.
- Разыменовывать; Разыменовывание — основная операция с указателем. Она означает «взять элемент, лежащий по адресу из указателя». Разыменовывание происходит с помощью оператора *.

```
int catsAreCool = 0;
int* ptr = &catsAreCool;
*ptr = 1; /* catsAreCool = 1 */
```

- Вычесть другой указатель;

  Если мы оперируем с двумя указателями, которые указывают на одну область памяти (например, на разные элементы одного и того же массива), то мы можем вычесть **меньший** из **большего** и получить количество элементов в интервале от меньшего включительно до большего не включая его. Обратите внимание, количество элементов, а не количество байт!

```
int arr[128];
int* ptr1 = &arr[50];
int* ptr2 = &arr[90];
ptr2 - ptr1; // = 40
```

  Во всех других случаях (вычитание большего из меньшего, вычитание указателей на разные области памяти и т.п.) результат вычитания указателей не определен стандартом языка. Сложение, умножение, деление и т.п. для указателей запрещены.

- Сравнивать;

  Мы можем проверить значение указателей на равенство или неравенство, а именно узнать, который из них указывает на больший адрес в памяти. Эта операция имеет смысл и стабильный результат только если оба указателя направлены на разные части одной связной области памяти, например, массива.

10.1.3 Тип void*

Существует специальный тип указателя, называющийся `void*`. В отличие от указателей на обычные типы данных (которые неявно указывают и на их размер), `void*` указывает на данные неопределённого типа и размера. С ним не пройдёт адресная арифметика (потому что размер данных, на которые он указывает, неизвестен). Прежде, чем нормально работать с таким указателем, необходимо явно определить, на данные какого типа он указывает, конвертировав его в соответствующий тип-указатель:

```
1  void* a = (void*)4;
2  short* b = (short*) a;
3  b ++; /* correct, b = 6 */
```

10.1.4 NULL

В C с помощью препроцессора выделяется специальная константа с именем `NULL` для указателя, который «ведёт в никуда». Это означает, что если указатель равен `NULL`, то он не ведёт ни на какой корректный объект и обращаться по нему не следует. `NULL` обычно равен нулю, . Иначе говоря, конечно, это указатель на какую-то ячейку памяти с номером 0, но мы условились, что в ней ничего не лежит.

Замечание. *Всегда ли NULL равен 0 и можно ли использовать 0 вместо него? Стандарт языка [3] постулирует: когда речь идёт об указателях и только тогда можно использовать 0 как NULL. Даже если на данной платформе «указатель в никуда» представлен не числом, в котором все биты сброшены, а как-то иначе, компилятор поймёт, что литерал 0 в контексте указателей это null pointer, и вставит правильное представление вместо нуля. Поэтому следующие выражения корректно проверяют указатель на* `NULL`*:*

```
1  if(x) { ... }
2  if(NULL != x) { ... }
3  if(0 != x) { ... }
```

10.1.5 Указатели на функции

Фон Неймановский абстрактный вычислитель C подчиняется принципу однородности памяти, то есть в его памяти есть и данные, и код.

Значит, функции тоже имеют адреса, по которым начинается их код. Мы можем создавать указатели на функции, передавать их в качестве аргументов другим функциям, а также вызывать их. Зачем это нужно? Например, таким образом можно сделать функцию, которая запускает другую функцию и считает время её выполнения, или же проходит по массиву и применяет её ко всем его элементам.

По аналогии с указателем на данные, указатель на функции подразумевает некоторую информацию о типе функции. Это возвращаемое функцией значение, а также типы аргументов.

Чтобы объявить переменную-указатель на функцию, используется немного странный синтаксис:

```
<тип возвращаемого значения> (*имя) (параметр1, параметр2, ...);
```

Например:

```
1  double doubler (int a) { return a * 2.5; }
2  ...
3  double (*fptr)( int );
4  double a;
5  fptr = &doubler;
6  a = fptr(10); /* a = 25.0 */
```

Мы описали тип указателя, создали переменную `fptr` типа «указатель на функцию, принимающую `int` и возвращающую `double`». Затем этой переменной присвоили адрес функции `doubler`, вызвали её по указателю, а возвращённое функцией значение записали в переменную `a`.

Разумеется, работает и `typedef`:

```
1  double doubler (int a) { return a * 2.5; }
2  typedef double (*megapointer_type)( int );
3
4  ...
5  double a;
6  megapointer_type variable = &doubler;
7  a = variable(10); /* a = 25.0 */
```

`typedef` позволил сделать нам новый тип данных, хранящий указатель на функции соответствующего вида. Теперь мы можем создавать переменные такого типа.

Обратите внимание, что звёздочку ставить после имени такого типа не нужно: она как бы уже скрыта typedef'ом. По аналогии:

```
typedef int* intptr;
...

int a = 4;
intptr b = &a; // no *!
*b = 10; // as usual
```

Можно не скрывать звёздочку внутри `typedef`:

```
double doubler (int a) { return a * 2.5; }
typedef double (megapointer_type_2)( int );


...
double a;
megapointer_type_2* variable = &doubler;
a = variable(10); //a = 25
```

Тип, который мы объявили, соответствует *типу самой функции*, а не указателю, хранящему адрес функции. Получается, что функции – тоже некоторого рода объекты для языка C, только не *объекты первого класса*, потому что мы не можем создавать переменные типа функций.

Объекты первого класса – в языках программирования это сущности, которые могут быть переданы как параметр, возвращены из функции или присвоены переменной.

Иногда функции в C называют объектами второго класса, так как мы имеем право создавать переменные типа «указатель на функцию некоторого вида». Получается как и с обычными типами: существует некоторый базовый тип (например, int, а для функции – тип функции, которая принимает какие-то аргументы X и возвращает Y), а мы можем создать для него производный тип – указатель.

10.1.6 Массивы и указатели

В C массивы особенные. Вообще любой набор значений, которые хранятся последовательно в памяти, являются массивом, и наоборот. В листинге ниже показаны несколько массивов фиксированных размеров. Что же такое имя массива с точки зрения абстрактного вычислителя? На самом деле это имя для адреса первого элемента, то есть, фактически, указатель на его первый элемент!

Обращение к элементу массива происходит одним из следующих, эквивалентных способов:

```
1  a[4] = 2;
2  *(a+4) = 2
```

Конечно, мы можем взять `a` как адрес первого элемента массива, прибавить к нему 4 (в соответствии с правилами адресной арифметики, это, фактически, **4 * sizeof(int)**), и разыменовать получившийся адрес.

Адрес четвертого элемента можно получить так:

```
1  &a[4];
2  a+4;
```

Так как эти записи эквивалентны, *практически любые операции с указателями можно переписать с использованием синтаксиса обращения к массивам и наоборот.*

Если массив объявляется как локальная переменная функции, он хранится в стеке целиком там же, где и другие локальные переменные этой функции. Если массив объявлен вне функций, он глобален.

Массивы любой длины можно кратко инициализировать нулями:

```
1  int a[10] = {0};
```

Вопрос 46. *Объясните запись: 4[a]*

Вопрос 47. *Чему будет равен* `sizeof(pa)` *?*

```
1  int a[10] = {0};
2  int* pa = a;
```

10.1.7 Детали синтаксиса

C позволяет определять несколько переменных одного типа в одну строчку:

```
1  int a,b = 4, c;
```

При объявлении указателей перед каждым именем необходимо поставить *.

```
int* a, *b, c;
```

В этом примере `a` и `b` являются указателями, а типом переменной `c` будет `int`.

Это правило (звездочка применяется не к типу, а к переменной) можно обойти с помощью `typedef`, создав псевдоним для типа `int*`.

Сложные примеры*

Однако можно создавать очень сложные определения с указателями, массивами, указателями на функции и т.д. Для их расшифровки можно использовать следующий алгоритм:

1. Находим идентификатор, начинаем разбирать с него;
2. Идём вправо до первой закрывающей скобки, находим её пару и интерпретируем то, что в них;
3. Поднимаемся на уровень выше и повторяем, пока не закончим разбор.

Несколько примеров (объяснение в столбце «Интерпретация»):

```
int* (* (*fp) (int) ) [10];
```

Шаг	Интерпретация
Начинаем с имени	`fp`
Справа уже скобка, слева звёздочка	это указатель на
Поднимаемся на уровень выше, идём вправо: `(int)`	функцию, принимающую `int`
Идём налево от уже разобранных скобок	и возвращающую указатель на
Поднимаемся на уровень выше: массив	массив из десяти
Идём опять влево от разобранного	указателей на `int`

10.2 Пример: программа, печатающая свои аргументы

```
#include <stdio.h>

```

```
3  int main( int argc, char** argv ) {
4        int i;
5        if ( argc <= 1 )
6              printf( "No␣arguments␣specified!\n" );
7        else
8        {
9              printf( "%d␣arguments␣specified\n", argc-1 );
10             for( i = 1; i < argc; i++ )
11             {
12                   printf( "argument␣No␣%d␣is␣%s\n",
13                         i, *(argv + i) );
14                   printf( "argument␣No␣%d␣is␣%s\n",
15                         i, argv[i] );
16             }
17             return 0;
18       }
19 }
```

Эта программа печатает параметры, с которыми она запущена. Оболочка разделяет вход на части по пробелам, символам табуляции и переносам строк. При запуске загрузчиком функции `main` ей передаётся два аргумента: количество параметров (`argc`) и массив из них (`argv`).

Каждый параметр это нуль-терминированная строчка текста, строки задаются указателем на первый символ, поэтому каждый параметр это указатель (`char*`). Параметров потенциально много, и эти указатели лежат в массиве. Массив задаётся указателем на свой первый элемент, а длина массива может храниться только отдельно. Поэтому `argv` это указатель на первый из параметров. Второй находится по адресу `argv + 1 * sizeof( char* )`, третий – `argv + 2 * sizeof( char* )` и т.д. Общее количество – в `argc`.

Нулевой аргумент это имя самой программы. Обратите внимание на следующее:

- Спецификатор вывода `"%s"` для строк. Это единственный спецификатор, для которого соответствующий аргумент `printf` должен являться указателем типа `char*`;
- Эквивалентность записей `*(argv + i)` и `argv[i]`;
- `i` был объявлен в начале блока тела функции. Нельзя объявлять его прямо в цикле, как в других языках подобных C:

```
1   for (int i = ...; ...; ...)
```

10.3 Пример: программа с указателем на функцию

Теперь совершим скачок в пропасть указателей и попробуем вызвать функцию по указателю.

```
1   #include <stdio.h>
2
3   int increment( int value ) {
4         return value + 1;
5   }
6
7   typedef int(fptr)(int);
8
9   void call_and_print( fptr function, int value ) {
10        int returned;
11        returned = function( value );
12        printf( "%d\n", returned );
13  }
14
15  int main( int argc, char** argv ) {
16      call_and_print( increment, 41 );
17      return 0;
18  }
```

Обратите внимание на следующее:

- Все функции сначала объявлены, а только потом вызываются;
- Создан новый тип `fptr`, обозначающий адрес функции, принимающей `int` и возвращающей `int`.
- Имя функции обозначает её адрес. Поэтому в вызов `call_and_print` и передаётся просто имя функции безо всяких дополнительных манипуляций.

Указатели на функции часто используются в C – для обобщённого программирования, для построения сложных автоматов и т.д.

10.4 Модель памяти языка C

Пришло время подробнее поговорить о структуре памяти абстрактного вычислителя C.

В памяти есть несколько областей:

- Код;
- Константы – в константной памяти помещаются все константы, такие, как строковые литералы. Обычно операционная система защищает соответствующие ей страницы от записи, поэтому при попытке записать туда что-нибудь программа рушится;
- Данные – глобальные переменные.
- Свободное пространство. В случае, если присутствует механизм динамического выделения памяти, обычно она выделяется тут.
- Стек – здесь живут все локальные переменные функций, адреса возврата, служебная информация стековых фреймов. Если программа выполняется в несколько потоков, то каждый поток имеет свой стек.

10.4.1 Виды выделения памяти

Соответственно количеству домов, где может жить мистер Данные, у него есть выбор из трёх типов выделения памяти:

Автоматическое выделение памяти происходит, когда мы создаём локальную переменную. При заходе в процедуру или функцию ей выделяется кусочек стека для локальных переменных, адреса возврата и (иногда) небольшого количества служебной информации. При выходе из функции вся информация о таких переменных автоматически теряется, так как соответствующий кусок стека выкидывается, а память будет отдана следующей вызванной функции. Соответственно время жизни таких данных — пока мы не завершим выполнение функции.

Замечание. *По этой причине никогда нельзя возвращать из функции адреса локальных для неё данных!*

Статическое выделение памяти происходит еще во время компиляции в области данных, константной или неконстантной. Эти

данные живут до завершения программы. Глобальные данные, инициализированные нулями или не инициализированные ничем, попадают в секцию `.bss`, константные данные попадают в секцию `.rodata`, а изменяемые – в `.data`.

Динамическое выделение памяти необходимо тогда, когда мы в процессе работы программы на основании её хода работы должны решить, сколько именно памяти нам необходимо, а затем зарезервировать её. Для этого используется специальная часть стандартной библиотеки C. Она поддерживает информацию о том, какие блоки использованы, а какие - нет. Обычно ОС выделяет некоторое количество страниц для нашей программы впрок, которые мы можем заполнить динамическими данными, а при необходимости прозрачно для программиста запросить у ОС еще страниц.

Когда память нам больше не нужна, мы должны освободить её, иначе мы не сможем переиспользовать эти ячейки памяти в дальнейшем. *Вышеописанное – не способ выделения памяти, неотъемлемый для языка C, это лишь удобный механизм в стандартной библиотеке. При написании ядра ОС, например, он не будет вам доступен.*

Рассмотрим следующий код:

```
1  #include <malloc.h>
2  ...
3  int* a = (int*) malloc(200);
4
5  a[4] = 2;
```

Прежде всего необходимо подключить заголовочный файл `malloc.h`, в котором объявлены следующие функции:

- `malloc` выделяет `size` байт подряд в куче. Возвращает указатель на первый из них.

  ```
  1  void* malloc(size_t size);
  ```

- `calloc` выделяет `size * count` байт подряд в куче. Удобно для массивов (количество элементов и размер каждого). Кроме того, `calloc` инициализирует память нулями.

  ```
  1  void* calloc(size_t size, size_t count);
  ```

- `free` освобождает кусок кучи, указатель на начало которого передан в аргументе.

```
1   void free(void* p);
```

- `realloc`

```
1   void* realloc(void* ptr, size_t newsize);
```

  Ищет свободный кусок памяти размера `newsize` и перемещает туда данные начиная с адреса `ptr`. Возвращает указатель на новый кусок кучи. Старый автоматически освобождается.

При использовании calloc и malloc указатель, который они возвращают, нужно привести к тому типу-указателю, который нужен. Иначе будет невозможна адресная арифметика. К тому же, C++, в отличие от C, требует явно приводить типы одного указателя к типам других, т.е. следующей конструкцией без явного приведения типа лучше не пользоваться, иначе ваш код будет непереносим на C++:

```
1   int* arr = malloc( sizeof(int) * 42 );
```

Если используемую память не освободить с помощью вызова free, то она останется зарезервированной, и следующие вызовы malloc её не перекроют. Так в программах происходят утечки памяти, когда некоторое количество памяти, которая не используется, остаётся зарезервированным всё время жизни процесса. Из-за ошибок в логике программы количество такой памяти может расти, что еще сложнее простить.

Замечание. *Для любознательных: о полемике на тему того, стоит ли использовать явное приведение типа при вызове `malloc`.*

В чистом C указатель типа `void*` можно приводить к любому другому типу неявно:

```
1   char* arr = malloc( sizeof(int) * 42 ); /* correct! */
```

Более того, использование явного преобразования может сокрыть ошибку. Если вы забыли включить заголовочный файл, в котором определен malloc, компилятор может посчитать, что функция malloc где-то есть, и по умолчанию она возвращает int, и поставить вместо неё заглушку, возвращающую 0.

Если вы не поставили явное приведение типа, то произойдёт попытка присваивания int в char, что запрещено и вызовет ошибку компиляции. Если же есть явное преобразование типа, то int сконвертируется в char*, и ошибка скроется.*

Однако в C++ конструкция без приведения типов просто некорректна. Из соображения совместимости с C++ приведение типов всё-таки обычно ставится.

10.4.2 Задание: суммирование динамически созданного массива

Есть массив из чисел типа `int` *произвольного размера*, который считывается с `stdin`. То есть, в момент написания программы мы не знаем, сколько памяти нужно выделить под массив. Первое число в `stdin` – размер массива, то есть количество чисел, которые за ним последуют и которые нужно просуммировать.

Необходимо:

- Использовать функцию `sum` из задания 8.3.8;
- Сформировать массив из считанных с `stdin` чисел и передать его на вход функции `sum`
- Вывести результат с помощью `printf`

Вам потребуется научиться использовать:

- `malloc` для динамического выделения памяти
- `free` для её освобождения
- `scanf` с правильными спецификаторами для считывания чисел с `stdin`
- `NULL`

Не забудьте про `const` !

10.4.3 Строковые литералы

Напомним, что строки в C — просто массивы символов, заканчивающиеся символом с кодом 0. Ноль выступает признаком конца строки.

Практически все строковые **литералы** в C кладутся в константные глобальные данные. Например:

```
1  char const* str = "when the music is over, turn out the lights";
```

`str` — не более, чем указатель, сама строка (так как является литералом), создаётся в константной части области данных. Поэтому при попытке изменить её, скорее всего, возникнет ошибка времени выполнения, потому что ОС защитила эту область памяти от записи.

```
1  str[15] = '\''; /* runtime error */
```

Поэтому все строковые литералы имеют тип `char const*`.
Такие конструкции тоже допустимы, ведь это обычный указатель:

```
1  char will_be_o = "hello, world!"[4]; /* is 'o' */
2
3  char const* tail = "abcde"+3 ; /* is "de" */
```

Строки вообще могут быть и не константными (`char*`).
Существуют другие способы работать со строками:

1. Создать строку в глобальных переменных. Она будет изменяемой. Единственный случай, когда строковый литерал не ведёт к тому, что в константной области данных создаётся соответствующая строка.

   ```
   1  char str[] = "something_global";
   ```

   Иначе говоря, это просто глобальный массив, который мы заранее заполнили значениями.

2. Создать строку в стеке. Она будет изменяемой.

   ```
   1  void func(void) {
   2    char str[] = "something_local";
   3  }
   ```

   Строка `"something_local"` будет, тем не менее, создана в константной части области данных (если она достаточно велика). При входе в функцию `func` будет выделено достаточное количество байт для строки, затем строка будет скопирована из области данных в локальный буфер.

3. Конечно, можно создать строку и в динамической памяти. Функция `strcpy` из заголовочного файла `string.h` используются для копирования строки.

```c
#include <malloc.h>
#include <string.h>

int main( int argc, char** argv )
{
  char* str = (char*)malloc( 25 );
  strcpy( str, "wow, such a nice string!" );

  free( str );
}
```

Вопрос 48. *Почему мы выделили 25 байт, хотя в строке 24 символа?*

Интернирование строк означает, что константные строчки создаются в константной области данных один раз. Если даже в программе они встречаются много раз, копий создаваться не будет.

```c
char* best_guitar_solo = "Firth of fifth";
char* good_genesis_song = "Firth of fifth";
char* best_1973_live = "Firth of fifth";
```

Эти три указателя, скорее всего, будут указывать на одну и ту же область памяти. Интернирование поддерживается почти всеми современными компиляторами, хотя и не декларировано как часть стандарта.

Интернирование строк — одна из причин, почему константные строчки защищены от записи. Если одна из таких строчек используется в разных местах программы по разным причинам, она никогда не будет изменена в одной из частей программы, нарушив логику работы в другой.

10.4.4 Задание: связный список

На вход подается произвольное количество чисел. Необходимо:

1. Сохранить их в **связном списке** чисел в обратном порядке;
2. Написать функцию для подсчёта суммы элементов связного списка;
3. Вывести их сумму;

4. Вывести элемент списка, соответствующий номеру вашего варианта, если список достаточно длинный, иначе сообщение о недостаточной длине с указанием точной длины;

5. Очистить за связным списком память.

Вам потребуется научиться использовать:

- Структуры, в том числе с полями, ссылающимися на структуры такого же типа;
- Константу `EOF`.

Ограничения:

- В поток ввода попадают только числа, разделенные произвольным количеством пробельных символов;
- Каждое число влезает в формат `int`;

Дополнительные требования:

- Все повторяющиеся операции должны делаться единообразным образом. Сделать это можно с помощью вызовов к функциям.
- Писать маленькие функции - хорошо.
- Конструирование элемента связного списка, обращение к его элементу по индексу, добавление в начало или конец относятся к повторяющимся операциям.

Мудро выражать функции друг через друга, так меньше работы и код надежнее. Рекомендованный набор функций:

- `list_create` – принимает число, возвращает указатель на созданный в куче элемент связного списка;
- `list_add_front` – принимает число и указатель на указатель на связный список. Добавляет число в начало списка.

  Пример: список (1,2,3), число 5 -> список становится (5,1,2,3).
- `list_add_back`, аналогично, добавляет элемент в конец. Это более медленная операция, т.к. приходится пройтись по всему списку;
- `list_get` получает элемент по индексу;
- `list_free` освобождает память для всех элементов списка;

- `list_length` принимает список и считает его длину;
- `list_node_at` принимает список и индекс, возвращает указатель на структуру `list_t`, соответствующую нужному элементу списка. Если индекс слишком большой, вернуть NULL;
- `list_sum` принимает список, возвращает сумму элементов в нём.

10.4.5 Ключевое слово static

У `static` есть несколько значений:

1. Применяя `static` к глобальным переменным или к функциям, мы ограничиваем их область видимости текущим модулем.

Листинг 10.1: listings/examples/static_objects/main.c

```
1  int global_int;
2  static int module_int;
3
4  static int module_function() {
5     static int static_local_var;
6     int local_var;
7     return 0;
8  }
9  int main( int argc, char** argv ) {
10    return 0;
11 }
```

В выводе `nm` глобальные символы помечены заглавными буквами, а локальные – строчными.

```
> gcc main.c --ansi --pedantic -o main
> nm main
00000000006008e0 B __bss_start
00000000006008e0 b completed.6661
00000000006008d0 D __data_start
00000000006008d0 W data_start
00000000004003f0 t deregister_tm_clones
0000000000400470 t __do_global_dtors_aux
00000000006006c0 t __do_global_dtors_aux_fini_array_entry
```

```
00000000006008d8 D __dso_handle
00000000006006d0 d _DYNAMIC
00000000006008e0 D _edata
00000000006008f0 B _end
0000000000400554 T _fini
0000000000400490 t frame_dummy
00000000006006b8 t __frame_dummy_init_array_entry
00000000004006b0 r __FRAME_END__
00000000006008ec B global_int
00000000006008a8 d _GLOBAL_OFFSET_TABLE_
                 w __gmon_start__
0000000000400370 T _init
00000000006006c0 t __init_array_end
00000000006006b8 t __init_array_start
0000000000400560 R _IO_stdin_used
                 w _ITM_deregisterTMCloneTable
                 w _ITM_registerTMCloneTable
00000000006006c8 d __JCR_END__
00000000006006c8 d __JCR_LIST__
                 w _Jv_RegisterClasses
0000000000400550 T __libc_csu_fini
00000000004004e0 T __libc_csu_init
                 U __libc_start_main@@GLIBC_2.2.5
00000000004004c1 T main
00000000004004b6 t module_function
00000000006008e4 b module_int
0000000000400430 t register_tm_clones
00000000004003c0 T _start
00000000006008e8 b static_local_var.1377
00000000006008e0 D __TMC_END__
```

```
1  static int ABSOLUTE_RANDOM_CONSTANT_VALUE = 42;
2  static int square_it( int a ) {
3   return a * a + ABSOLUTE_RANDOM_GLOBAL_VALUE;
4  }
```

---

2. Применяя `static` к локальной переменной, мы заставляем её быть созданной в области данных, а не в стеке, как другие локальные переменные. Инициализируя её каким-то значением, мы

гарантированно сделаем это ровно один раз, а при последующих вызовах функции значение переменной будет сохраняться (ведь она не пересоздаётся каждый раз при входе в функцию).

```
1
2   int demo (void)
3   {
4     static int a = 42;
5     printf("%d\n", a++);
6   }
7
8   ...
9
10  demo(); //outputs 42
11  demo(); //outputs 43
12  demo(); //outputs 44
```

10.4.6 Задание: функции высшего порядка на списках

Что такое функции `foreach`, `map`, `map_mut` и `foldl`?

- `foreach` принимает указатель на начало списка и функцию. Затем он по очереди запускает функцию на всех элементах списка по порядку.
- `map` принимает функцию f и список. Он возвращает новый список (исходный остаётся неизменным), в котором содержимое каждого элемента получается применением функции f к соответствующему элементу исходного списка.
- `map_mut` делает то же самое, только изменяет исходный список.
- `foldl` работает немного сложнее. Эта функция получает:
  - начальное значение аккумулятора;
  - функцию f;
  - список из элементов.

  Возвращает `foldl` сущность такого же типа, что и аккумулятор, полученную следующим образом:

1. Запускаем f от копии аккумулятора и первого элемента списка – запоминаем в аккумуляторе полученное значение a';
2. Запускаем f от значения a' и второго элемента списка – запоминаем в аккумуляторе полученное значение a''; Как только список кончается, мы возвращаем запомненное значение в аккумуляторе.

   Например, выбрав функцию f как произведение двух элементов, мы, запустив `foldl` с аккумулятором 1 и этой функцией получим произведение всех элементов в списке.

- `iterate` принимает один начальный элемент s, длину желаемого списка n и функцию f. Она генерирует список такого вида: $[s, f(s), f(f(s)), f(f(f(s))), ...]$ и длины n.

Вышеперечисленные функции попадают в категорию **функций высшего порядка** – функций, которые в качестве одного из аргументов принимают другую функцию. Распространённый пример – функция сортировки массива, которая одним из аргументов принимает «сравниватель», который сравнивает два элемента и отвечает на вопрос, какой из них должен стать ближе к началу в отсортированном массиве, а какой – ближе к концу.

Вопрос 49. *Изучите функцию* `sort`.

Итак, задание. На вход подается произвольное количество чисел. Необходимо:

- сохранить их в связном списке
- перенести все функции из предыдущего задания в отдельный .c файл, и снабдить его корректным заголовочным. Не забудьте Incude Guard!
- написать функцию `foreach`, с её помощью вывести исходный список на экран дважды: сначала разделяя числа пробелами, потом каждое на отдельной строчке;
- написать функцию `map`; вывести полученные с её помощью квадраты и кубы чисел из исходного списка;
- написать функцию `foldl`; вывести полученные с её помощью сумму, наибольший и наименьший элемент списка.

- написать функцию `map_mut`; вывести полученные с её помощью модули чисел.
- написать функцию `iterate`; вывести созданный с её помощью список из степеней числа 2 (первые 10 значений);
- очистить всю выделенную память

Вам потребуется научиться использовать

- указатели на функции;
- `limits.h` и константы из него;
- Ключевое слово `static` для функций, которые вы хотите использовать только в одном модуле.

Ограничения:

- В поток ввода попадают только числа, разделенные произвольным количеством пробельных символов.
- Каждое число умещается в формат `int`

Разумно вынести чтение списка в отдельную функцию, принимающую указатель на `FILE`

Решение занимает около 90 строчек (не учитывая написанные в прошлом домашнем задании функции).

10.5 Потоки данных

Чтобы организовать ввод-вывод, мы думаем о файлах или о физических устройствах как о чём-то одинаковом. Абстрагируясь от конкретных особенностей тех или других, мы работаем с ними единообразно, как с так называемыми «потоками данных», в которые мы можем что-то писать или что-нибудь оттуда считывать. *Здесь мы будем изучать то, как абстрагируется взаимодействие с файлами в стандартной библиотеке языка C.* Интерфейс доступа к файлам стремится быть похожим на интерфейс доступа у памяти: оба являются последовательностями байтов, оба линейно адресуемы.

Поток данных – упорядоченная последовательность данных.

Потоки данных бывают двух видов: текстовые и бинарные.

- Бинарные потоки состоят из байтов, которые пишутся или читаются «как есть», то есть не претерпевают по пути к коду (или вовне) никаких изменений.

- Текстовые потоки включают символы, сгруппированные в строчки, каждая из которых заканчивается специальным символом завершения строки, зависящим от реализации. Если данные, записанные в двоичный поток, в целостности и сохраности доходят до адресата (устройства или файла), то текстовые потоки гарантируют это только для печатных символов, символов переноса строки и табуляции при следующих условиях:

  – Последний символ – символ перевода строки;
  – Ни один символ перевода строки не следует сразу за пробельными символами.

Для работы с потоками используются специальные библиотечные функции и структуры. Некоторые из них применимы только к одному из двух типов потоков, например,`fscanf` стоит применять только к текстовым потокам.

Потоки могут использовать буфер. Это означает, что поток будет ждать, пока в него запишут достаточно символов прежде, чем передавать их вовне, сохраняя данные в свой буфер, пока тот не заполнится. Это оправдано с точки зрения производительности, так как передача данных вовне использует системные вызовы ОС (например, `write`), обращения к которым скрыты от нас в библиотеке. Операции ввода-вывода дороги (в том числе из-за переключения контекста), поэтому их количество лучше сократить.

При открытии файла с ним ассоциируется соответствующий поток.

Некоторые моменты работы с файлами:

1. Существует тип `FILE`, хранящий всю необходимую информацию для работы с открытым файлом;

2. Существует константа `EOF`. Если она возвращается библиотечной функцией, например, работающей с файлом, то это означает, что достигнут конец файла;

3. Существует константа `BUFSIZ`, в которой хранится такая длина буфера записи, которая на данной системе даёт наилучшие результаты с точки зрения производительности операций чтения-записи. Иногда эту константу удобно использовать для размера пользовательского буфера, куда, например, программист читает строчки, полученные от пользователя.

4. Как мы знаем, уже в начале работы программы программисту доступны три открытых файловых потока: `stdin`, `stdout` и `stderr`. Они соответствуют потокам ввода, вывода и диагностики. Эти три потока буферизованы, но `stderr` может быть небуферизован (или по крайней мере автоматически выдавать своё содержимое после каждой операции записи в него).

Пример работы с файлами в C:

Листинг 10.2: listings/streams_example.c

```
int smth[]={1,2,3,4,5};

/* открыть для чтения и записи как бинарный файл */
FILE* f = fopen( "hello.img", "rwb" );

/* прочитать в память начиная с адреса smth один блок
 * размером sizeof(int) байтов.
 * Указатель на текущую позицию в файле передвинется
 * вперёд */
fread( smth, sizeof(int), 1, f);

/* вернуться к началу файла. SEEK_SET означает не смещение
 * относительно текущей позиции, а абсолютный адрес в файле */
fseek( f, 0, SEEK_SET );

/* записать 1 блок размера (5*sizeof(int)) байтов в файл f */
fwrite(smth, 5 * sizeof( int ), 1, f);

/* закрыть файл */
fclose( f );
```

В данном кусочке происходит открытие файла для чтения и записи в бинарном режиме. Затем из него читается один блок размером `sizeof(int)` в память по адресу «адрес первого элемента буфера». Затем указатель на текущую позицию в файле, который перемещается при чтении из него или записи в него, откатывается на нулевую позицию в файле. Потом `5*sizeof(int)` байт записываются в файл из буфера `smth`, после чего файл закрывается, при этом все буферизованные потоки точно сливают накопленную информацию в него.

Замечание. *Потоки в C это не дескрипторы! Они задаются не числом, а указателем на структуру. Получить дескриптор мож-*

но с помощью функции `int fileno(FILE *stream)`*, объявленной в файле* `stdio.h`*.*

Вопрос 50. *Посмотрите справочную документацию (man) по функциям:* `fread`*,* `fread`*,* `fwrite`*,* `fprintf`*,* `fscanf`*,* `fopen`*,* `fclose`*,* `fflush`*.*

Вопрос 51. *В чем разница между вторым и третьим параметром* `fread`*?* `fwrite`*?*

Вопрос 52. *Как заставить поток «слить» данные из буфера на вывод?*

Вопрос 53. *Что произойдёт, если функцию* `fflush` *применить к потоку, в котором по крайней мере последнее действие было чтением из потока?*

Вопрос 54. *Почему* `stderr` *постоянно сливает данные, не дожидаясь* `fflush`*, а не ждёт заполнения, как* `stdout`*?*

Замечание. *Чтение из бинарного файла, открытого в режиме чтения текста, чревато трудноуловимыми ошибками.*

Глава 11

Синтаксис, семантика и прагматика языков

Что считать описанием языка (например, программирования)? На этот вопрос есть несколько ответов:

- *Сам компилятор и есть описание языка.* Не самый лучший путь, поскольку в компиляторе обязательно найдутся ошибки. И как делать другие компиляторы? К тому же описание языка смешивается с техническими деталями реализации.

- Задать некоторые правила построения правильных конструкций языка. Например, формальные грамматики служат именно этой цели. Если строчка из символов алфавита языка соответствует этим правилам, то она считается корректной.

  Этот подход, однако, вообще никак не рассматривает, каким смыслом наделены те или иные строчки языка.

- Задать соответствие структурных элементов языка действиям абстрактного вычислителя.

  Этот подход не учитывает, что элементы языка должны быть скомпонованы в корректным образом написанные программы.

Хорошо скомбинировать второй и третий подходы. **Синтаксис** языка задаёт множество корректно написаных программ для данного языка. **Семантика** языка задаёт соответствие между написанными программами и смыслом, который в них вкладывает абстрактный вычислитель.

Существует третий аспект — **прагматика**, которая задаёт изменение смысла в зависимости от контекста. В случае языков программирования, контекст – конкретный вычислитель, на котором исполняется программа, а прагматика задаёт детали преобразования программы для абстрактного вычислителя (например, для C) в программы для конкретного вычислителя (машинные коды).

11.1 Синтаксис*

Прежде всего, язык множество строчек из символов некоторого алфавита. Разумеется, при конструировании какого-то осмысленного языка не все строчки из символов алфавита считаются его корректными предложениями, т.е. «принадлежащими языку».

В ходе попыток создать математические модели для естественных языков (русский, английский и т.д.), была предложена концепция грамматик. Она основывается на том, что предложение языка можно в соответствии с т.н. «правилами грамматики» разбить на части (возможно, несколькими способами), каждая из которых носит определенное имя в правилах грамматики (подлежащее, сказуемое..). Эти части, в свою очередь, разбить на более мелкие, и так далее до некоторых базовых частей.

Обычно эти части называют **символами**. Символы бывают терминальными (базовые символы, для которых не определено, из каких частей они сами состоят), и нетерминальные (которые можно проанализировать по частям).

Этот подход применяется для описания синтаксиса языков программирования.

Итак, описание грамматики включает в себя:

- Конечное множество терминальных символов;
- Конечное множество нетерминальных символов (другое название – *синтаксические категории*)
- Конечное множество правил вывода, каждое из которых берет некоторый нетерминал и описывает возможности по его раскрытию в последовательность символов (терминальных и/или нетерминальных);
- Начальный символ – такой нетерминальный символ, который соответствует любому целому предложению языка.

Каждое правило имеет вид:

Нетерминал := <последовательность из терминалов и нетерминалов>

Можно написать несколько различных правил раскрытия для одного и того же нетерминала и для удобства объединять их с помощью символа | («или»)

Иногда используют особенный терминал ϵ – пустую строчку.

Пример 3. *Грамматика языка формул из натуральных чисел со сложением и вычитанием.*

Нетерминалы:

- *Формула;*
- *Число;*
- *Цифра;*
- *Действие.*

Начальным символом будет являться «Формула», что логично, так как начальный символ соответствует любому предложению языка.

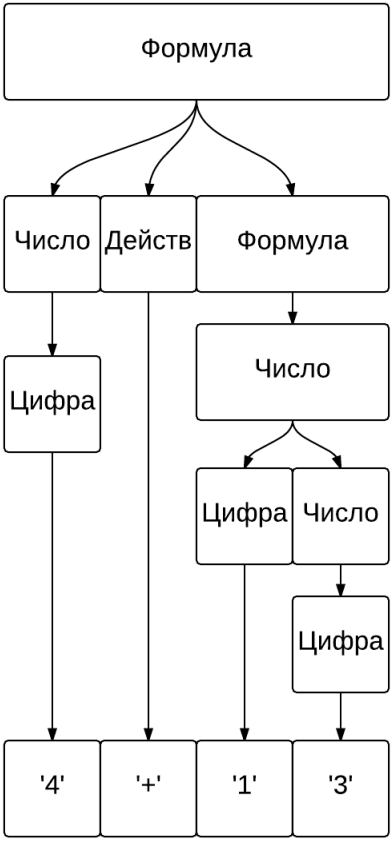
Терминалы:

- *'0', '1', ... '9'*
- *'+', '-'*

Правила:

1. *Формула := Число | Число Действие Формула*
2. *Число := Цифра | Цифра Число*
3. *Цифра := '0' | '1' | ... | '8' | '9'*
4. *Действие := '+' | '-'*

С помощью грамматики над корректной строчкой языка можно построить некоторую структуру – «дерево разбора», которое показывает соответствие между терминалами/нетерминалами и частями строчки. Для строчки «4 + 13» дерево разбора будет выглядеть вот так:



Каждый узел дерева раскрывается в несколько других путём применения правила разбора, и так пока мы не доходим до терминалов.

Вопрос 55. *Как вы думаете, всегда ли можно однозначно построить дерево разбора для любой строчки? Если нет, то придумайте грамматику и строчку, которую можно разобрать двояко.*

Те грамматики, которые мы описали, называются «контекстно свободными», или обыкновенными. Именно они имеют наибольшее распространение при работе с языками программирования. Существуют и другие виды грамматик, которые в совокупности образуют иерархию из четырёх ступеней (иерархия Хомского).

Если нулевой и первый уровень нам неинтересны как слишком сложные и выразительные грамматики, то на втором уровне живут обыкновенные (или «контекстно-свободные») грамматики, о которых

мы рассказали. На третьем же – регулярные языки, которые задаются с помощью **конечных автоматов** (или, что эквивалентно, регулярных выражений). Мы уже приводили пример того, как можно построить автомат, распознающий определенные строчки. С помощью грамматик мы, конечно, можем выразить любой регулярный язык, но не наоборот.

Вопрос 56. *Прочитайте про иерархию Хомского. Придумайте, как можно построить контекстно-свободную грамматику, имея регулярное выражение (конечный автомат), распознающее язык.*

11.2 Семантика

Семантика языка описывается обычно одним из следующих способов:

- **Аксиоматически**. Про текущее состояние программы можно написать некоторый набор правильных утверждений, его описывающих. Каждая конструкция описывается через то, как она влияет на эти утверждения.
- **Денотационно**. Каждому предложению языка ставится в соответствие математический объект в рамках какой-то теории, например, теории доменов. Про поведение этих объектов говорит сама теория.
- **Операционно.** Каждое предложение языка как-то влияет на работу абстрактного вычислителя, его интерпретирующего. Это влияние и подлежит описанию.

Мы уже определили, что алгоритм нельзя описать без базовых, изначально определённых действий (модели вычисления). Так и здесь, описывая значение языка необходимо на чём-то основываться. Фундаментом для описаний семантики обычно служит математическая логика и лямбда исчисление, так как их смысл считается достаточно очевидным.

11.2.1 Undefined behavior

В языках программирования случается, что не каждому предложению языка можно поставить в соответствии какой-то смысл. С точки зрения синтаксиса, однако, предложение будет совершенно корректно сконструированным.

Например, обращение по нулевому указателю в C ведёт к неопределённому поведению (так говорит стандарт языка). Значит на разных платформах/компиляторах обращение к нулевому указателю может привести к чему угодно.

Это сделано намеренно. Язык C стремится к минимальным гарантиям безопасности и корректности работы в сочетании с минимальными накладными расходами. Если бы стандарт языка определил особое поведение при обращении к нулевому указателю (например, вывод сообщения об ошибке или механизм исключений, как в языках высокого уровня), то при каждом обращении по указателю пришлось бы его проверять: нулевой он или нет?

Неопределённое поведение опасно тем, что это не исключительная ситуация, после которой программа аварийно завершит работу (или даст возможность обработать её). Она может продолжить исполнение, но в определённый момент у этого проявтятся непредсказуемые последствия. Например, типичен случай, когда запись за границы массива, выделенного в куче, ведёт к повреждению заголовков структур, размечающих кучу на отдельные части. Так нарушается консистентность кучи, и через несколько вызовов `malloc`/`free` программа внезапно рушится.

Случаи неопределённого поведения, на самом деле, очень многочисленны.

- Знаковое переполнение целых чисел;
- Обращение по некорректным указателям;
- Сравнение указателей не на одну связную область памяти;
- Вызов функции с несоответствующими ей аргументами;
- Чтение из неинициализированной локальной переменной;
- Деление на ноль;
- Обращение за границы массива;
- Попытка изменения строкового литерала;
- Использование значения, посчитанного функцией, если в ней отсутствует `return`;
- И многое другое.

Стандарт C99 приводит 191 случай неопределённого поведения [16].

11.2.2 Unspecified behavior

В отличие от Undefined behavior, случай Unspecified behavior в стандарте языка даёт разные возможные сценарии того, как программа будет себя вести. Например, не регламентирован порядок вычисления подвыражений при вычислении большого выражения. В частности, в выражении `f(g(x), h(x))` подвыражения `g(x)` и `h(x)` будут посчитаны до вычисления функции `f`, но в каком порядке – стандарт не определяет.

11.2.3 Точки следования

Точки следования – те места в программе, когда состояние абстрактного вычислителя и конкретного «синхронизируются».

Можно думать об этом следующим образом: при отладке программы мы можем выполнить её по шагам. Но шагать можно по строчкам кода на C, а можно по ассемблерным инструкциям. Представим, что одновременно существуют две машины с похожей архитектурой: одна это абстрактный вычислитель языка C, другая – конкретный вычислитель, оперирующий ассемблерными инструкциями. Состояние абстрактной машины реализуется в конкретной, то есть её ячейкам соответствуют настоящие ячейки физической памяти. Каждый шаг программы на C соответствует многим шагам программы на ассемблере. Если мы оставновили выполнение предложения C «на полпути», то состояние конкретного вычислителя, конечно, всё еще полностью определено. Про состояние абстрактного вычислителя мы такого сказать не можем: его переменные могут принимать какие угодно значения. Но когда программа доходит до точки следования, эти вычислители приходят в согласованное состояние.

Другое определение звучит так: точка следования это место программы, где побочные эффекты предыдущих выражений уже применились, а следующих – ещё не наступили.

Точками следования являются:

- Точка с запятой;
- Запятая (когда она разделяет выражения, а не аргументы функций/элементы массива);
- Логические И/ИЛИ (не побитовые!);
- Момент запуска функции с аргументами (аргументы точно посчитаются до запуска функции, хотя и не определено, в каком порядке);

- Вопросительный знак в тернарном операторе.

Пример того, когда понятие точек следования всплывает:

```
1  int i = 0;
2  i = i ++ + 10;
```

Чему равно `i`? Единственный правильный неспекулятивный ответ – это неопределённое поведение. Причина в том, что непонятно, будет ли инкремент произведён до записи в `i` выражения `i+10` или после. Произошли два конфликтующих побочных эффекта (запись в одну и ту же переменную), для которых никак не определено, кто кому должен предшествовать.

11.3 Прагматика C

11.3.1 Выравнивание

С точки зрения абстрактного вычислителя, мы работаем с байтами памяти. Каждый байт имеет адрес. Аппаратно, однако, на уровне, скрытом от программиста, процессор умеет считывать из памяти только пачки байтов (например, по 8 штук), причем только с адресов, кратных какому-то числу. К примеру, только первую восьмерку, или вторую, ...

Если процессору нужен байт по адресу 2, он считывает число размером 8 байт с адресов 0-7, а затем «вырезает» оттуда один байт. Что же происходит, когда нам нужно считать не один байт, а единицу данных, состоящую из нескольких байт?

```
Вариант 1: выровненные данные
0x00 00 00 00 00 00 00 00 :    11 22 33 44 55 66 77 88

Вариант 2: невыровненные данные
0x00 00 00 00 00 00 00 00 :    .. .. .. 11 22 33 44 55
0x00 00 00 00 00 00 00 07 :    66 77 88 .. .. .. .. ..
```

На рисунке показаны две ситуации, в которых нам нужно считать большое число из памяти. Во втором случае нам нужно дважды об-

ратиться к памяти, а затем из двух кусков 8-байтных чисел «сшить» нужное число.

Памяти теперь у программистов много, а канал обмена данными с памятью по прежнему узкое место. Поэтому обычно используется **выравнивание**. Четырёхбайтовые числа, например, кладутся компилятором в память так, чтобы их можно было бы считать за одно обращение к памяти. *В таком случае говорят, что происходит выравнивание по границе 4.*

Пусть мы выравниваем по границе *align*, адрес начала данных *label*, а их размер *size*. Тогда если $label + size$ переходит через следующий после *label* адрес, кратный *align*, то мы отступим несколько байт и положим *label* равным следующему числу, кратному *align*.

Например, есть число `int v` размером 4 байта, а выравнивание идёт по границе 8 байт. Тогда адреса 0,1,2,3,4 считаются «хорошими» с точки зрения выравнивания, а вместо того, чтобы положить его по адресу, равному 5,6,7, его лучше разместить начиная с адреса 8.

Так как выравнивание не является, вообще говоря, специфичным для языка программирования, но затрагивает конкретный вычислитель, можно сказать, что это часть прагматики языка.

Если речь идёт о структурах, следует различать выравнивание самой структуры (рассматривая её как что-то неделимое, по какому адресу она начнётся?) и выравнивание её элементов.

Зачем нам может потребоваться изменять значение выравнивания? Пусть определена следующая структура:

```
1  typedef struct {
2    short a;
3    long b;
4  } mystr_t;
```

Нетрудно догадаться, что фактический размер структуры — 16 байт, так как между a и b будет вставлено шесть байтов для выравнивания, иначе для считывания b нам будет требоваться 2 обращения к памяти.

Типичная ситуация, когда нам это мешает – когда мы считываем некоторое количество данных из файла в память, а потом накладываем на них структуру, чтобы удобно получать доступ к соответствующим байтам через поля структуры. Считав в правильную структуру заголовок файла с изображением, наполненный служебной информацией, мы сразу можем обратиться к нужным байтам и узнать, например, ширину или высоту изображения. Однако в файле пустот между полями нет, а в структуре вполне могут быть.

Чтобы изменить значение выравнивания необходимо использовать специальные директивы компилятора – **прагмы** (да, они называются так, потому что связаны с прагматикой).

```
#pragma pack(2)
typedef struct {
  short a;
  long b;
} mystr_t;
```

Здесь `pack` — название прагмы, в скобках идут параметры. Двойка соответствует новому значению выравнивания. Выравнивание будет действовать начиная с данного места программы и до конца текущего файла с кодом. Однако такой подход чреват, так как многие программы (особенно системные, низкоуровневые) очень чувствительны к «непривычным» для них значениям выравнивания. Глобальное изменение выравнивания может привести к трудноуловимым ошибкам.

По этой причине выравнивание меняют только для конкретного куска кода (например, описания одной структуры).

```
#pragma pack(push, 2)
typedef struct {
  short a;
  long b;
} mystr_t;
#pragma pack(pop)
```

Здесь `#pragma pack(push, ...)` и `#pragma pack(pop)` — *команды компилятору записать двойку в его внутренний стек выравниваний и, соответственно, выкинуть из него последнее число.*

Стек выравниваний — структура данных, которую создаёт транслятор на время компиляции программы. На вершине этого стека хранится текущее значение выравнивания. Положив что-то в этот стек мы изменим текущее значение выравнивания, а сделав pop – вернём предыдущее.

11.3.2 Платформонезависимые типы данных

В C в стандартных заголовочных файлах многих компиляторов определены типы размера, который всегда одинаков. Они называются `int16_t`, `int32_t`, `uint32_t` и т.д. Их использование не считается плохим тоном, и особенно оно оправдано в случаях, когда необходимо использовать

число именно данной разрядности (например, поле для заголовка четко фиксированного размера).

Глава 12

Как писать на C

В этой главе собраны советы и рекомендации относительно того, как писать программы.

Важно осознавать то, как оценивать качество кода. При написании программы преследуются разные цели. Иногда во главу угла ставится производительность и оптимизации на самом низком уровне, что ухудшает читаемость кода; иногда – максимальная переиспользуемость или надежность. Разные цели могут вести к разным стилям написания программ, несовместимым между собой.

В нашем курсе мы хотим писать программы так:

1. Код хочется видеть максимально переиспользуемым (если это не идёт в сильный ущерб алгоритмической сложности).

   Это экономит время на написание логики программы **и на её отладку**. К сожалению, такие языки, как C, обладают очень скудными средствами контроля программиста (такими, как богатая система типов языков Haskell, Rust или Scala), так что писать корректный код становится сложным. Чем чаще мы используем уже многократно проверенную функцию, тем меньше ошибок окажется в программе.

   Пример когда переиспользование нарушает это правило — в задании, где необходимо создать функции работы со связным списком. Вполне корректно задать функцию `foreach` так: подсчитать длину массива, затем для всех меньших её i выполнить действие над i-ым элементом, причем каждый раз осуществлять доступ по индексу.

   Доступ по индексу в связном списке длины N тем дольше, чем

больше индекс (максимум – порядка N). Тогда при данной реализации алгоритма мы потратим порядка N операций на подсчёт длины, затем $1 + 2 + ... + N - 1 = \frac{N(N-1)}{2}$ операций.

2. Программу должно быть легко модифицировать.

   Этот пункт связан с предыдущим. Чем лучше переиспользуется код, тем меньше вероятность, что при добавлении функционала часть старого кода придется дублировать.

3. Код должен легко читаться. В частности, код с названиями переменных на английском языке читать гораздо легче, нежели транслитерированные русские слова (razmer, kolvo, massiv...).

4. Код должен быть лаконичным.

   Избыточность описания логики (через разбор большого числа частных случаев, например), ведёт к ухудшению читаемости, в таком коде легче допустить ошибку и его гораздо сложнее модифицировать.

5. Код должно быть легко тестировать.

   Тестирование – одна из важных частей написания кода. Оно повышает степень доверия к нему, демонстрируя, что на определенных входных значениях он ведёт себя так, как от него ожидается.

Подобрать правильные имена для типов, функций и переменных очень важно. Единообразное именование помогает легкому чтению кода. Используйте подчеркивания (_) для разделения частей имен. Например: `clients_count`. Имена типа `i`, `j` можно использовать для индексов внутри массивов.

В общем случае не стоит называть переменные одной-двумя буквами так, чтобы это не несло никакой смысловой нагрузки о её предназначении. Исключение – код, иллюстрирующий различные математические статьи, где имена переменных названы, обычно, в соответствии с описанием алгоритма на бумаге.

12.1 Типы

- **Отдавайте предпочтение типам из `stdint.h`: `uint64_t`, `uint8_t` и т.д.**

- Принято именовать типы с суффиксом `_t`, например, `size_t`;

- Можно создавать разные типы для данных разного предназначения (метры и килограммы);
- При объявлении структур упорядочивайте их поля так, чтобы минимизировать потери памяти на выравнивание, затем по размеру и в алфавитном порядке;
- Часто имена служебных полей структур, с которыми стоит работать только через специально написанные для этого функции, начинаются с символа подчеркивания;
- Поля перечислений называйте заглавными буквами, используя подчёркивание как разделитель (как и имена констант). Используйте в них общий префикс. Например:

```
1  enum colors_t {
2  COLOR_RED = 0,
3  COLOR_GREEN,
4  COLOR_BLUE
5  };
```

12.2 Переменные

- Стоит называть существительными
- Иногда в них включают размерность.

  Например: `uint32 delay_msecs;`
- Суффиксы полезны. Часто встречаются такие, как `cnt`, `max`...

  Например: `attempts_max` (максимальное количество попыток), `attempts_cnt` (количество совершенных попыток).
- Только строчные буквы в именах. Глобальным изменяемым переменным можно добавить префикс `g_`, глобальные константы именуются заглавными буквами.
- Не определяйте переменные в заголовочных файлах! Другие разработчики не будут их там искать, так как это считается плохим стилем. Определите переменную однажды в .c файле, а в соответствующем заголовочном файле упомяните её как `extern`.

- По возможности, не используйте глобальные *изменяемые* переменные. Они мешают вашей программе стать многопоточной (много потоков имеют по стеку на каждый из них, но общую секцию данных). Можно использовать неизменяемые глобальные переменные, но таковые часто можно заменить на глобальные константы, заданные с помощью `#define`. Кроме того, глобальные переменные не будут кэшироваться в регистрах, что плохо для производительности.

  Глобальные переменные также делают сложнее процесс автоматического тестирования, без которого нельзя представить написание коммерческого кода.

  Если без них не обойтись, старайтесь хотя бы ограничить их область видимости одним модулем с помощью `static`.

- Глобальные переменные, помеченные `const static` – вероятные кандидаты на то, чтобы быть встроенными в местах использования.

12.3 Функции

- Стоит называть глаголами (ведь они что-то *делают*). Например, `check`.

- Префикс `is` удобно использовать для функций, которые что-то проверяют.

  Например: `int is_prime( long num )`.

- Функциям, предназначенным для работы с определённым типом структуры, полезно давать имена, в которых префиксом выступает имя типа структуры. Например, `void screen_clear( struct screen_t* s` Т.к. в C нет пространств имён в привычном смысле, именование функций должно уменьшать хаос от определения большого количества имён в одном месте (фактически, глобально).

- Обязательно помечайте все функции, которые не должны использоваться извне файла, как `static`.

- Все аргументы, указывающие на данные, которые не изменяются функцией, стоит пометить `const`.

12.4 Модули

Один из основополагающих принципов мышления – абстракция. Если нам хочется реализовать некоторую логику (например, написать код для поворота изображения), мы хотели бы не думать ни о чём, кроме алгоритма поворота. Неважен формат входного файла, структура его служебных заголовков и т.д. – всё, что нужно, это уметь работать с точками изображения и знать его размеры. Однако где-то все детали реализации формата стоит оборабатывать.

Абстракция заключается в том, что мы разделяем программу на части, каждая из которых заточена под выполнение определённой логики. Эту логику можно вызывать через набор функций (в широком смысле это – **интерфейс** для данной части программы), или, возможно, обращаясь к тем или иным переменным, созданным внутри этой части программы. Однако для реализации логики требуется гораздо больше переменных и функций, доступ к которым извне необходимо в целях надежности ограничить.

В языках с ООП эту роль обычно выполняют классы. В экземплярах классов бывают приватные поля, обратиться к которым можно только из методов этих классов. В C же в структурах все поля доступны кому угодно.

Единственное, что может помочь с абстракцией – использование **модулей**. Напомним, что модуль – единица трансляции, один файл с расширением `.c`.

Функции и глобальные переменные, объявленные в файле с кодом, по умолчанию становятся глобальными символами, и потому доступны во всех других файлах с кодом. Модификатор `static` для функций и глобальных переменных, однако, позволяет ограничить их область видимости одним модулем, скрывая их от других файлов с кодом. При этом в соответствующем `.h` файле удобно описать интерфейс модуля, тогда включая `.h` файл в другие файлы с кодом мы будем давать доступ к функциональности модуля в других местах.

Пример 4. *Напишем модуль, в котором инкапсулируем логику работы стека. Стек реализуем с помощью связного списка: добавление элемента в стек это добавление элемента в начало связного списка.*

Заголовочный файл будет описывать структуру, соответствующую стеку, и функции для работы с ней.

Что мы хотели бы уметь делать со стеком из любого места программы?

- Создавать;

- Класть элемент на вершину, забирать оттуда элемент;
- Проверять, не пустой ли он;
- Запустить какую-то функцию для каждого элемента стека.

Заголовочный файл мог бы выглядеть так:

Листинг 12.1: listings/examples/module/stack.h

```
#ifndef _STACK_H_
#define _STACK_H_

#include <stddef.h>
#include <stdint.h>

struct llist_t { int value; struct llist_t* next; };

struct stack_t {
   struct llist_t* first, *last;
   size_t count;
};

struct stack_t stack_init(void);

void stack_push( struct stack_t* s, int value );
int stack_pop ( struct stack_t* s );
int stack_is_empty( struct stack_t* s);


void stack_foreach( struct stack_t* s, void (f)(int));

#endif /* _STACK_H_ */
```

А логика работы стека могла бы быть описана так:

Листинг 12.2: listings/examples/module/stack.c

```
#include <malloc.h>
#include "stack.h"

static struct llist_t* item( int value, struct llist_t* next ) {
   struct llist_t* new_item;
```

```
6       new_item = (struct llist_t*)
7          malloc( sizeof( struct llist_t ) );
8       new_item-> next = next;
9       new_item-> value = value;
10      return new_item;
11  }
12
13  void stack_push( struct stack_t* s, int value ) {
14     s->first = item( value, s->first );
15     if (s->last == NULL) s->last = s-> first;
16     s->count++;
17  }
18
19  int stack_pop( struct stack_t* s ) {
20    struct llist_t* const head = s->first;
21    int value;
22    if ( head ) {
23       if ( head->next ) s->first = head->next;
24       value = head->value;
25       free( head );
26       if( -- s->count ) s->last = NULL;
27       return value;
28    }
29    return 0;
30  }
31
32  void stack_foreach( struct stack_t* s, void (f)(int)) {
33     struct llist_t* cur = s->first;
34     while ( cur ) {
35        f( cur->value );
36        cur = cur-> next;
37     }
38  }
39
40  int stack_is_empty( struct stack_t* s) {
41     return s->count == 0;
42  }
43
44  struct stack_t stack_init(void) {
45     struct stack_t empty = { NULL, NULL, 0 };
46     return empty;
```

```
47 }
```

Обратите внимание, что к некоторым функциям из этого файла (`item`) синтаксически никак не получить доступ извне из-за `static`. Они предназначены исключительно для удобства программиста. Компилятор вообще нередко может встраивать функции в местах их вызова, особенно небольшие; если же функция помечена `static`, она может даже не быть включенной в исполняемый файл (и отсутствовать в таблице символов).

12.4.1 Сокрытие содержимого структур

С помощью неполных типов можно скрыть содержимое структуры. При этом нельзя возвращать значение типа такой структуры из функции (только указатель на структуру) и создавать переменные такого типа (указатели можно). Разадресовывать указатели на такие структуры нельзя. Вот пример заголовочного файла и самого модуля:

Листинг 12.3: listings/hidden_type.h

```
1 struct hidden_type_t;
2
3 void f( struct hidden_type_t* h );
```

Листинг 12.4: listings/hidden_type.c

```
1 #include "hidden_type.h"
2
3 struct hidden_type_t {
4    int x, y;
5 }
6
7 void f( struct hidden_type_t* h ) {
8    printf( "%d␣%d\n", h->x, h->y);
9 }
```

12.5 Организация файлов

С самого начала проекта стоит позаботиться о том, чтобы его иерархия каталогов была удобна. Иначе при добавлении новых файлов в

проект будет всё сложнее ориентироваться в них.

Можно использовать следующий распространённый шаблон:

- `src/` – файлы с исходным кодом;
- `src/tests` – тесты для кода;
- `doc/` – файлы документации;
- `res/` – файлы ресурсов (например, изображения);
- `lib/` – скомпилированные библиотеки, необходимые для работы приложения;
- `build/` – в этой директории будут создаваться артефакты (исполняемые, объектные файлы, сборки);
- `Makefile` – файл, управляющий сборкой.

Иногда заголовочные файлы помещают в отдельную директорию `include/`, заголовочные файлы для сторонних библиотек часто выделяют в отдельную директорию. С помощью ключа `-I` можно подсказать `gcc`, где искать файлы, включенные с помощью `#include <file>`.

Вопрос 57. *Узнайте о системе документации Doxygen.*

Некоторые проекты, демонстрирующие хороший подход к организации кода на C:

- http://www.gnu.org/software/gsl/
- http://www.gnu.org/software/gsl/design/gsl-design.html
- http://www.kylheku.com/ kaz/kazlib.html

12.6 Иммутабельность или производительность

Часто в процессе описания различных преобразований необходимо выбрать между созданием измененной копии данных и изменением существующих экземпляров структур. Приведем некоторые аргументы за и против.

- Создание копии

- Обычно легче реализовать без ошибок;
- Легче отлаживать: в какой-то момент в программе живут и исходные данные, и модифицированные;
- Медленнее для больших данных.
- Может быть оптимизировано так, что копия создаваться не будет.

•

12.7 Задание: поворот картинки на 90 градусов

Необходимо написать программу на языке C. Программа должна принимать в качестве аргумента имя файла-изображения в формате BMP любого разрешения и поворачивать картинку на 90 градусов вправо или влево (реализовать необходимо только поворот в одну из сторон).

12.7.1 Формат BMP-файла

Формат файла BMP (сокращенно от BitMaP) - это формат растровой графики, т.е. такие файлы хранят информацию о точках и их цвете. В файлах BMP информация о цвете каждого пикселя кодируется 1, 4, 8, 16 или 24 бит (бит/пиксель). Числом бит/пиксель, называемым также глубиной представления цвета, определяется максимальное число цветов в изображении. Изображение при глубине 1 бит/пиксель может иметь всего два цвета, а при глубине 24 бит/пиксель - более 16 млн. различных цветов.

Ниже указана структура заголовка BMP-файла без палитры, который должна понимать ваша программа.

```
1  typedef struct {
2     uint16_t bfType;
3     uint32_t bfileSize;
4     uint32_t bfReserved;
5     uint32_t bOffBits;
6     uint32_t biSize;
7
8     uint32_t biWidth;
9     uint32_t biHeight;
10    uint16_t biPlanes;
```

```
11      uint16_t biBitCount;
12      uint32_t biCompression;
13      uint32_t biSizeImage;
14      uint32_t biXPelsPerMeter;
15      uint32_t biYPelsPerMeter;
16      uint32_t biClrUsed;
17      uint32_t biClrImportant;
18  } bmp_header_t;
```

Файл разбит на заголовок и растровый массив. Заголовок файла содержит информацию о файле, в том числе адрес, с которого начинается область данных растрового массива.

Вопрос 58. *Прочитайте, за что отвечают все эти поля.*

Формат данных растрового массива зависит от числа бит, используемых для кодирования данных о цвете каждого пикселя. Файлы с глубиной 16 и 24 бит/пиксель не имеют таблиц цветов; в этих файлах значения пикселей растрового массива непосредственно характеризуют значения цветов RGB.

Важно! каждая строчка пикселей дополняется «мусором» таким образом, чтобы её длина в байтах была кратна четырём. Например, ширина изображения 15 пикселей. Каждый пиксель кодируется тремя байтами и строчка занимает 15*3 = 45 байт. Ближайшее число, кратное четырём — 48, поэтому в конце строчки перед началом новой будут вставлены 3 байта исключительно чтобы выровнять следующую строчку. Поэтому фактический размер растрового массива **в байтах** отличается от произведения ширины на высоту в заголовке файла.

Замечание. *Открывать картинку в текстовом режиме – плохая идея; из большинства картинок после этого чтение будет происходить непредсказуемо!*

Оформите функции работы с изображением в отдельном модуле или модулях.

12.7.2 Архитектура

Мы бы хотели организовать код так, чтобы программу можно было легко расширить, а функции максимально изолировали кусочки логики. Для этого необходимо:

1. Описать структуру одного пикселя `pixel_t`, чтобы не работать с растровым массивом изображения как с абсолютно неразмеченными данными. Этого всегда стоит избегать!

2. Отделить формат представления картинки внутри программы от формата BMP или любого иного.

   Для этого создадим структуру `image_t`, в которой помимо указателя на массив пикселей будем хранить ту информацию, которая действительно важна в программе. Сигнатуру, например, хранить нет никакого смысла, как и большинство полей bmp-заголовка. Для нас хватит ширины и высоты.

3. Открытие bmp-файла от его чтения (данные можно было бы принять и иным путём, нежели открыв файл с диска);

4. Открытие файла и генерацию ошибок от их обработки;

   Для удовлетворения этих двух пунктов реализуем функцию `from_bmp`, которая попытается прочитать файл из потока, и вернет один из кодов, показывающих успешность выполнения операции.

Распространённая ошибка заключается в том, что *логику программы смешивают с выводом сообщений об ошибках*. Этот подход плох по следующим причинам:

- В идеале вашу логику можно было бы использовать и в приложениях с графическим интерфейсом, и в консольных, и, возможно, передавать диагностическую информацию по сети на удалённый компьютер и т.д. Однако привязываясь к конкретному способу вывода ошибок (вызывая прямо в месте возникновения ошибки, например, `fprintf`), вы лишаете себя возможности использовать этот же код с другой операцией вывода.

- Если коды ошибок описаны в одном месте, легко проконтролировать, что для каждого из них есть своё сообщение. Скорее всего, коды будут определены с помощью директив `#define` или перечисления. Сделав `switch` по коду ошибки мы легко можем проверить, что все возможные варианты из перечисления реализуются и обрабатываются.

Листинг 12.5: listings/image_rotation_example.c

```
1  struct pixel_t { char b,g,r; };
2
```

```
3  struct image_t {
4     uint32_t width, height;
5     struct pixel_t* data;
6  };
7
8  /* deserializer */
9  typedef enum {
10    READ_OK = 0,
11    READ_INVALID_SIGNATURE,
12    READ_INVALID_BITS,
13    READ_INVALID_HEADER
14       /* more codes */
15 } read_error_code_t;
16
17 read_error_code_t from_bmp( FILE* in, struct image_t* const read );
18
19 /* image_t from_jpg( FILE* );...
20  * and other deserializers are possible
21  * All information needed will be
22  * stored in image_t structure */
23
24 /* makes a rotated copy */
25 struct image_t rotate( struct image_t const source );
26
27
28 /* serializer */
29 typedef enum {
30    WRITE_OK = 0,
31    WRITE_ERROR
32       /* more codes */
33 } write_error_code_t;
34
35 write_error_code_t
36 to_bmp( FILE* out, struct image_t const* img );
```

При чтении bmp файла можно оставить мусор в конце строк на месте, обойдясь одним вызовом `fread` для чтения растрового массива, но тогда для доступа к тому или иному пикселю по координатам нужно использовать специально написанную функцию. Другой вариант в том, чтобы избавиться от пустот, пропуская мусор. Тогда для чтения растрового массива потребуется столько вызовов `fread`, какова высо-

та изображения, но доступ к пикселям станет чуть проще – просто по индексу.

12.7.3 Бонусные задания

- Реализовать gaussian blur;
- Реализовать поворот на произвольное количество градусов (лишние пиксели заполнить белым цветом);
- Реализовать преобразования dilate и erode.

12.8 Задание: аллокатор памяти

Теперь мы напишем свой аллокатор памяти. В этом задании нельзя пользоваться функциями `malloc`/`calloc`, `free`, `realloc`.

Куча строится там, где в адресном пространстве процесса есть свободное место. Для выделения новых страниц процессу служит вызов `mmap`. Выделенные страницы нужно разметить на блоки. Каждый блок состоит из заголовка и какого-то количества последовательно идущих байт. Эти байты – то место, которое можно использовать. В заголовке блока можно хранить ссылку на следующий блок, свободен или занят данный блок, а также его размер.

Имея запрос на количество байт `query` для каждого блока размера `capacity` нужно сравнить их.

- `query < capacity` – разрезать блок надвое и использовать первую часть (если во вторую часть уместится хотя бы заголовок, иначе использовать блок как есть);
- `query == capacity` – использовать блок как есть;
- `query > capacity` – блок не подходит.
  - Если этот блок не последний, рассматриваем кучу дальше;
  - Иначе пытаемся выделить недостающие байты после конца этото блока и увеличить его размер (в `mmap` нужен флаг `MAP_FIXED`). Если не получилось, пытаемся выделить достаточное количество байт в любом месте (первый аргумент `mmap` – `NULL`) и присоединить его к куче. Если и это не получилось, то вернём `NULL`.

При освобождении памяти нужно не допустить появления двух или более последовательно идущих свободных блоков: их нужно слить.

Оформите ваши версии `malloc`/`free` в отдельном модуле.

Это задание гораздо сложнее сделать, если не разбить `malloc`/`free` на много маленьких функций.

Вы можете ориентироваться на этот заголовочный файл:

Листинг 12.6: labs/7/stud/src/mem.h

```
#ifndef _MEM_H_
#define _MEM_H_

#define __USE_MISC

#include <stddef.h>
#include <stdint.h>
#include <stdio.h>

#include <sys/mman.h>

#define HEAP_START ((void*)0x04040000)

struct mem_t;

#pragma pack(push, 1)
struct mem_t {
   struct mem_t* next;
   size_t capacity;
   int is_free;
};
#pragma pack(pop)

void* _malloc( size_t query );
void _free( void* mem );
void* heap_init( size_t initial_size );

#define DEBUG_FIRST_BYTES 4


void memalloc_debug_struct_info( FILE* f,
      struct mem_t const* const address );

```

```
34  void memalloc_debug_heap( FILE* f, struct mem_t const* ptr );
35
36  #endif
```

Листинг 12.7: labs/7/stud/src/mem_debug.c

```
1   #include "mem.h"
2
3
4   void memalloc_debug_struct_info( FILE* f,
5         struct mem_t const* const address ) {
6
7      size_t i;
8      fprintf( f,
9            "start: %p\nsize: %lu\nis_free: %d\n",
10           (void*)address,
11           address-> capacity,
12           address-> is_free );
13     for ( i = 0;
14           i < DEBUG_FIRST_BYTES && i < address-> capacity;
15           ++i )
16        fprintf( f, "%hhX",
17              ((char*)address)[ sizeof( struct mem_t ) + i ] );
18     putc( '\n', f );
19  }
20
21  void memalloc_debug_heap( FILE* f, struct mem_t const* ptr ) {
22     for( ; ptr; ptr = ptr->next )
23        memalloc_debug_struct_info( f, ptr );
24  }
```

Примерный объём кода на C – 150 строк. Обязательно сделайте Makefile.

Часть III

C и ассемблер

Глава 13

Детали трансляции

13.1 Соглашения вызова

В разделе, посвящённом ассемблеру, мы уже познакомились с азами того, как передаются аргументы в функцию. Теперь мы рассмотрим этот процесс подробнее.

Полное описание можно найти в [13], оно крайне рекомендуется к прочтению.

13.1.1 Стековые фреймы

Передача аргументов

В процессорах на сегодняшний день существуют т.н. `xmm`-регистры (`xmm0`, `xmm1`, ..., `xmm15`). Они имеют размер 128 бит и используются преимущественно для SIMD команд и для работы с числами с плавающей точкой. Обмен данными между младшими 64-битными половинками `xmm`-регистров и РОН/памятью осуществляется с помощью команды `movq`.

Вопрос 59. *С и ассемблерПрочитайте документацию на команду `movq`.*

Соглашение вызова – набор правил, о которых программисты и инженеры явно договорились и которые описали как стандарт. По этим правилам производится вызов функций.

Как передать аргументы функции?

1. Из тех аргументов, что имеют тип указателя или целочисленный, первые шесть помещаются в регистры (`rdi`, `rsi`, `rdx`, `rcx`, `r8`, `r9`);

2. Аргументы типа `float`/`double` помещаются в `xmm`-регистры по порядку с `xmm0` до `xmm7`;

3. Оставшиеся аргументы помещаются в стек в порядке, обратном тому, в котором они были объявлены в функции вне зависимости от их типа.

   Например, если функция имеет 8 аргументов типа `long`, то сначала в стек будет помещен восьмой, а потом седьмой. Соответственно, адрес седьмого аргумента меньше, чем адрес восьмого.

Вы уже знаете, что значения целого типа (и указатели) возвращаются в регистрах `rax` и `rdx`. Значения типа `float`/`double` возвращаются в регистрах `xmm0`, `xmm1`.

После того, как аргументы были помещены в регистры/стек, мы вызываем инструкцию `call`, которая в качестве параметра получает адрес первой из команд функции. Кроме того, *она кладёт в стек адрес возврата*.

В каждый момент программы может быть запущено одновременно множество копий одной и той же функции, которые различаются своими аргументами и значениями локальных переменных. Для хранения таких **экземпляров функций** решили использовать стек. Его основной принцип – last in first out – хорошо согласуется с тем, как вызывают друг друга процедуры: последняя вызванная процедура первой завершит свою работу.

Стековым фреймом (кадром) называется кусочек стека, который хранит часть состояния программы, связанную с экземпляром процедуры.

В стековом фрейме хранятся значения локальных переменных (объявленных программистом или созданных компилятором) и сохранённых регистров.

Можно сказать, что содержательная часть функции заключена между её **прологом** и **эпилогом**, похожими для всех функций. Пролог помогает корректно инициализировать стековый фрейм, а эпилог – демонтировать.

В прологе происходят следующие действия:

```
1  func:
2    push rbp
```

```
3       mov rbp, rsp
4
5       sub rsp, 24 ; 24 -- total size of local variables
```

Для каждой функции поддерживается **инвариант** (условие, верное на протяжении всего выполнения функции): регистр `rbp` указывает на начало *текущего* стекового фрейма. Поэтому при создании нового стекового фрейма нужно сохранить старое значение `rbp`, а затем настроить `rbp` на начало данного фрейма, которое к этому времени хранится в `rsp`. Затем из `rsp` вычитается суммарный размер локальных переменных. Это и есть «выделение памяти в стеке», которое использовалось при написании библиотеки ввода-вывода на ассемблере в первом задании.

В конце функции находится стандартный код **эпилога**:

```
1       mov rsp, rbp
2       pop rbp
3       ret
```

Этот код эквивалентен:

```
1       leave
2       ret
```

Замечание. *Команда `leave` специально создана для демонтажа стекового кадра. Противоположная ей команда `enter` обычно не используется компиляторами, так как она более функциональна и нацелена на поддержку языков с вложенными функциями.*

Переместив в `rsp` адрес начала кадра, мы можем быть уверены в том, что вся выделенная в стеке память очищена и сохранённое старое значение `rbp` находится на вершине стека. Вторая инструкция восстанавливает `rbp` на начало предыдущего стекового кадра, затем `ret` выбрасывает из стека адрес возврата.

Остаётся лишь почистить стек от аргументов, которые там могли быть. *Этим занимается вызывающая функция.* Кто именно очищает стек от аргументов тоже явно обговаривается при создании соглашения вызова.

13.1.2 Поведение стека при вызове

Рассмотрим простую функцию для подсчёта максимума из двух чисел.

Листинг 13.1: listings/maximum.c

```
int maximum( int a, int b ) {
   char buffer[4096];
   if (a < b) return b;
   return a;
}

void main(void) {
   int x = maximum( 42, 999 );
}
```

Листинг 13.2: ./listings/maximum.lst

```
00000000004004b6 <maximum>:
  4004b6:   55                push rbp
  4004b7:   48 89 e5          mov  rbp,rsp
  4004ba:   48 81 ec 90 0f 00 00 sub rsp,0xf90
  4004c1:   89 bd fc ef ff ff mov  DWORD PTR [rbp-0x1004],edi
  4004c7:   89 b5 f8 ef ff ff mov  DWORD PTR [rbp-0x1008],esi
  4004cd:   8b 85 fc ef ff ff mov  eax,DWORD PTR [rbp-0x1004]
  4004d3:   3b 85 f8 ef ff ff cmp  eax,DWORD PTR [rbp-0x1008]
  4004d9:   7d 08             jge  4004e3 <maximum+0x2d>
  4004db:   8b 85 f8 ef ff ff mov  eax,DWORD PTR [rbp-0x1008]
  4004e1:   eb 06             jmp  4004e9 <maximum+0x33>
  4004e3:   8b 85 fc ef ff ff mov  eax,DWORD PTR [rbp-0x1004]
  4004e9:   c9                leave
  4004ea:   c3                ret

00000000004004eb <main>:
  4004eb:   55                push rbp
  4004ec:   48 89 e5          mov  rbp,rsp
  4004ef:   48 83 ec 10       sub  rsp,0x10
  4004f3:   be e7 03 00 00    mov  esi,0x3e7
  4004f8:   bf 2a 00 00 00    mov  edi,0x2a
  4004fd:   e8 b4 ff ff ff    call 4004b6 <maximum>
  400502:   89 45 fc          mov  DWORD PTR [rbp-0x4],eax
```

Очистив вывод `objdump` мы получим следующую последовательность команд.

```
mov rsi, 999
```

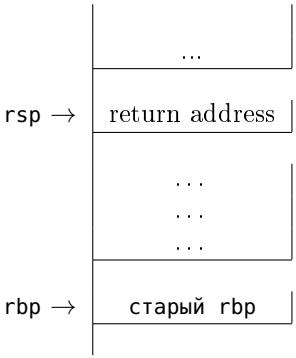
```
2   mov rdi, 42
3   call maximum
4   ...
5   maximum:
6   push rbp
7   mov rbp, rsp
8   sub rsp, 3984
9
10  mov [rbp-0x1004], edi
11  mov [rbp-0x1008], esi
12  mov eax, [rbp-0x1004]
13  ...
14
15  leave
16  ret
```

Замечание. *Почему перемещение данных в* `esi` *означает запись в весь регистр* `rsi` *см.3.4*

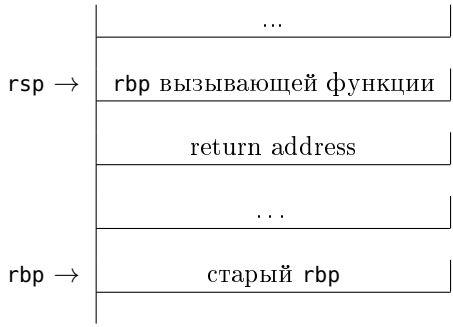
Для каждой команды нарисуем состояние стека сразу после её выполнения.

Сначала первые аргументы кладутся в регистры, остальные в стек в обратном порядке. Затем последовательно выполняются следующие команды:

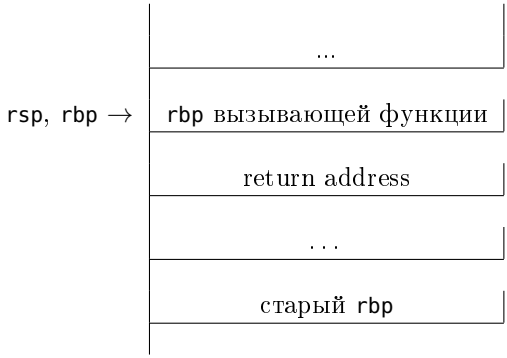
```
call maximum
```




```
push rbp
```




```
mov rbp, rsp
```




```
sub rsp, locals_size
```

13.1.3 Red zone

Красная зона — область размером 128 байт от вершины стека к младшим адресам. В ней безопасно выделять данные, они не затрутся при системных вызовах или прерываниях. Вызовы функций, однако, будут перезаписывать красную зону адресами возврата. Компиляторы используют её для того, чтобы не устанавливать полноценный стековый фрейм (регистр `rbp`). В этом случае локальные переменые адресуются относительно `rsp`. Очевидно, что для такой оптимизации необходимы как минимум следующие условия:

- Суммарный размер локальных переменных не больше 128 байт (но если больше, можно выделить в стеке меньшее количество);
- Функция не вызывает другие функции (кроме, возможно, системных вызовов);
- Функция не меняет значение `rsp` внутри себя, иначе адресовать аргументы относительно `rsp` нельзя.

При этом по прежнему можно выделять больше места для локальных переменных, отматывая `rsp`.

Вот пример того, как функция не создаёт стековый фрейм, если размер локальных переменных достаточно мал:

Листинг 13.3: ./listings/red_zone.c

```
1  typedef unsigned long size_t;
2  int no_stack_frame( void ) {
3     volatile char buffer[100];
4     size_t i;
5     for( i = 0; i < 100; i++ )
6        buffer[i] = 1;
7     i++;
8     return i;
9  }
10
11 int main( int argc, char** argv ) {
12    return no_stack_frame();
13 }
```

Листинг 13.4: ./listings/red_zone.lst

```
1  00000000004004b6 <no_stack_frame>:
2    4004b6: b8 00 00 00 00    mov  eax,0x0
3    4004bb: c6 44 04 90 01    mov  BYTE PTR [rsp+rax*1-0x70],0x1
4    4004c0: 48 83 c0 01       add  rax,0x1
5    4004c4: 48 83 f8 64       cmp  rax,0x64
6    4004c8: 75 f1             jne  4004bb <no_stack_frame+0x5>
7    4004ca: b0 65             mov  al,0x65
8    4004cc: c3                ret
```

Обращения к массиву `buffer` происходят относительно `rsp` (3). Регистр `rbp` никак не меняется.

13.1.4 Переменное количество аргументов

Используемое соглашение вызова позволяет функциям иметь переменное количество аргументов. Аргументы, положенные в стек, будут вычищены оттуда вызывающей функцией (после возврата из вызванной функции).

Функции с переменным количеством аргументом в C объявляются с помощью трёх точек (т.н. **ellipsis**). Типичный пример – функция `printf`.

```
1      void printf( char const* format, ... );
```

Как `printf` узнаёт точное количество аргументов? Он точно знает, что будет передан один аргумент – `format`. Анализируя эту строчку со спецификаторами он посчитает их количество и, таким образом, узнает, сколько аргументов брать из регистров.

Замечание. *Количество использованных для аргументов `xmm` регистров передаётся в регистре `al`.*

Поскольку доставать аргументы из регистров на C невозможно, такие функции нужно было бы писать с помощью ассемблера. К счастью, часть стандартной библиотеки языка, находящаяся в файле `stdarg.h`, определяет для каждой конкретной платформы интерфейс для доступа к аргументам, скрытым за ellipsis'ом. Она состоит из:

- `va_list` – структуры, которая хранит информацию об аргументах;
- `va_start` – **макроса**, который инициализирует `va_list`;
- `va_end` – **макроса**, который деинициализирует `va_list`;

- va_arg – **макроса**, который достаёт при каждом вызове следующий аргумент из списка типа va_list.

Рассмотрим пример:

Листинг 13.5: listings/vararg.c

```
1  #include <stdarg.h>
2  #include <stdio.h>
3
4  void printer( unsigned long argcount, ... ) {
5     va_list args;
6     unsigned long i;
7     va_start( args, argcount );
8    for (i = 0; i < argcount; i++ )
9      printf("␣%d\n", va_arg(args, int ) );
10
11    va_end( args );
12 }
13
14
15 int main () {
16    printer(10, 1, 2, 3, 4, 5, 6, 7, 8, 9, 0 );
17    return 0;
18 }
```

Сначала va_list инициализируется именем последнего обязательного аргумента (7). Затем при каждом следующем вызове макроса va_arg достаётся следующий из списка необязательных аргументов (9). Вторым параметром этот макрос получает тип аргумента.

В конце необходимо вызвать va_end от имени переменной va_list.

Выглядит этот пример непривычно. Макросы, очевидно, изменяют значение переменной args, но не принимают её адрес, а лишь имя. К тому же va_arg принимает имя типа. Функции не могут изменять значение чужой переменной, если только не получили указатель на неё. Они также не могут принимать имя типа, это нонсенс.

13.1.5 vprintf

У функций printf, sprintf и т.д. есть специальные версии, которые последним аргументом принимают экземпляр типа va_list.

```
int vprintf(const char *format, va_list ap);
```

Их используют внутри собственных реализаций функций с переменным количеством аргументов. Например:

Листинг 13.6: ./listings/vfprintf.c

```
#include <stdarg.h>
#include <stdio.h>

void logmsg( int client_id, const char* const str, ... ) {
   va_list args;
   char buffer[1024];
   char* bufptr = buffer;

   va_start( args, str );

   bufptr += sprintf(bufptr, "from␣client␣%d␣:", client_id );
   vsprintf( bufptr, str, args );
   fprintf( stderr, "%s", buffer );

   va_end( args );
}
```

13.2 volatile

Существует важное ключевое слово `volatile`. Его использование влияет на то, как компилятор оптимизирует код.

Модель вычислений языка C – фон Неймановский вычислитель. Она не предполагает параллельного или псевдопараллельного исполнения программ, и потому компилятор обычно вправе делать агрессивные оптимизации, например, удалять код, который *не приведёт к изменению наблюдаемого состояния программы*. Например, это чтение значения из памяти без его дальнейшей записи куда-либо.

Чтение и запись из `volatile` областей памяти всегда происходят в тот момент, когда это нужно по логике программы. В отличие от них, значения из других переменных могут кешироваться в регистрах.

Прежде всего это необходимо для:

- **memory mapped IO**, при котором взаимодействие с устройствами осуществляется через их отображение на области памяти. Например, запись по адресу может означать вывод определённого символа на экран.

- Разделения данных между потоками. Один поток может в любой момент изменить содержимое ячейки памяти, поэтому некорректно было бы кэшировать её значение в регистре.

В случае с указателями, `volatile`, как и `const`, может применяться к самому указателю или к данным, на которые он указывает. Правило то же: `volatile` слева от звёздочки относится к типу данных, на который мы указываем, а справа – к самому указателю.

Отложенное выделение памяти Многие ОС выделяют физическую память процессу не при запросе страниц (`mmap`), а при первом к ним обращении. Если программист хочет выделить страницы сразу, чтобы избежать задержек в дальнейшем, ему придётся обратиться к каждой странице, например, так:

```
1       char* ptr;
2       for( ptr = start; ptr < start + size; ptr += pagesize )
3               *ptr;
```

Этот код, однако, с точки зрения компилятора не делает ничего полезного, а потому может быть выброшен. Если же пометить указатель `volatile`, то этого не произойдёт.

```
1       volatile char* ptr;
2       for( ptr = start; ptr < start + size; ptr += pagesize )
3               *ptr;
```

Замечание. *Если указатель на не-`volatile` область памяти помечен `volatile`, по стандарту языка это ничего не гарантирует во время обращений по ним! Гарантии существуют только для массивов, помеченных `volatile`. Таким образом, пример выше с точки зрения стандарта* ***некорректен****. Программисты, однако, используют `volatile` указатели именно для таких целей, поэтому распространённые реализации языка (msvc, gcc, clang) не оптимизируют обращения к памяти по `volatile`-указателям. Правильного, соответствующего стандарту способа это сделать не существует.*

Пример 5. *Генерация кода для работы с `volatile` переменными.*

Листинг 13.7: ./listings/volatile.c

```
1   #include <stdio.h>
```

```
2
3  int main( int argc, char** argv ) {
4     int ordinary = 0;
5     volatile int vol = 4;
6     ordinary++;
7     vol++;
8     printf( "%d\n", ordinary );
9     printf( "%d\n", vol );
10    return 0;
11 }
```

Объявлены две переменные (одна из них – `volatile`), затем обе увеличиваются на единицу и передаются аргументами в функцию `printf`. При компиляции с флагом `-O2 gcc` сгенерирует следующий код:

Листинг 13.8: ./listings/volatile.asm

```
1  0000000000400410 <main>:
2    sub   rsp,0x18
3
4    ; Это два аргумента для первого вызова printf.
5    mov   esi,0x1
6    mov   edi,0x4005d4
7
8    ; Инициализируем vol числом 4
9    mov   DWORD PTR [rsp+0xc],0x4
10
11   ; Увеличим на 1 содержимое vol
12   mov   eax,DWORD PTR [rsp+0xc]
13   add   eax,0x1
14   mov   DWORD PTR [rsp+0xc],eax
15
16   xor   eax,eax
17
18   ; первый вызов printf
19   call  4003e0 <printf@plt>
20
21   ; второй аргумент берется из памяти!
22   mov   esi,DWORD PTR [rsp+0xc]
23   ; первый аргумент -- адрес строчки "%d\n"
24   mov   edi,0x4005d4
25   xor   eax,eax
```

```
26      ; второй вызов printf
27      call 4003e0 <printf@plt>
28      xor  eax,eax
29      add  rsp,0x18
30      ret
```

Как видно, только содержимое `volatile` переменной действительно будет прочитано (7) и записано в память (9). Место под переменную `ordinary` даже не будет выделено. Зачем создавать переменную, если явно понятно, что единственное её предназначение – быть увеличенной на единицу и переданной в функцию `printf`? Можно сразу увеличить её значение (оно станет равно 1) и передать получившееся число в регистре `rsi` как второй аргумент (3).

13.3 setjmp

Стандартная библиотека C содержит заголовочный файл `setjmp.h`. Он определяет набор функций для работы с **контекстом** вычислений. Контекст описывает текущее состояние программы *за вычетом состояния вычислений с плавающей точкой и сущностей «внешнего мира» (дескрипторов и пр.), а также данных, выделенных не автоматически (в стеке).*

На практике это означает возможность совершения нелокальных переходов (как goto, но не ограниченных одной функцией).

Контекст можно сохранить в переменную типа `jmpbuf` с помощью функции `setjmp`, а затем к нему вернуться с помощью `longjmp`.

```
1   int setjmp(jmp_buf env);
```

`setjmp` возвращает не ноль только если вызывается изнутри вызова `longjmp`.

Все локальные не-`volatile` переменные после совершения `longjmp` могут иметь какие угодно значения.

Этот механизм является частым источником ошибок (помимо очевидного – возвращения к выполнению программы в другой точке после выделения памяти `malloc`'ом без очистки оной).

В принципе, возможно вызывать `setjmp` в составе сложного выражения, но лучше этого не делать. Стандарт определяет только малое количество контекстов в программе, в которых корректно вызывать `setjmp`. Мы рекомендуем не вызывать `setjmp` в составе сложных выражений.

```
1  void longjmp(jmp_buf env, int val);
```

После выполнения `longjmp` программа вернётся к исполнению в том месте, где был вызван `setjmp`. Этот `setjmp` вернёт значение `val`, переданное в `longjmp`.

Важно понимать, что механизм сохранения контекста завязан на стековые фреймы. Нельзя выполнить `longjmp` в функцию с демонтированным стековым фреймом (т.е. такую, которая уже завершила свою работу). Например, поведение этого кода **не определено**:

```
1  jmp_buf jb;
2  void f(void) {
3     setjmp( jb );
4  }
5
6  void g(void) {
7     f();
8     longjmp(jb);
9  }
```

Функция `f` завершила свою работу, но мы производим `longjmp` внутрь неё. Поведение этой программы неопределено, т.к. мы пытаемся восстановить значения переменных в уже уничтоженном стековом фрейме.

Итак, прыгнуть так мы можем с точки зрения кода в любую функцию, но с точки зрения выполнения программы можно сделать только прыжок в функцию, чей фрейм выше по стеку, нежели текущий.

13.3.1 `volatile` и `setjmp`

С точки зрения компилятора, вызов `setjmp` – обычный вызов функции. Однако в реальности это необычная функция: программа может внезапно начать исполнение с того места, где был вызван `setjmp`. При этом в «нормальных условиях» компилятор мог закешировать какие-то локальные переменные в регистрах до обращения к `setjmp`, что приведет к неприятным последствиям тогда, когда после выполнения `longjmp` мы вернемся к месту `setjmp` с попорченными регистрами.

Компилятор может и вообще не создавать переменную в стеке. Из-за этого её значение *может потеряться*.

По этим причинам иногда случается, что оптимизированная программа работает иначе, нежели неоптимизированная!

Чтобы избежать этого, нужно действовать в соответствии со стандартом языка, который постулирует: *только `volatile`-локальные переменные после `longjmp` сохраняют своё значение.*

Листинг 13.9: ./listings/setjmp_volatile.c

```
#include <stdio.h>
#include <setjmp.h>

jmp_buf buf;

int main( int argc, char** argv ) {
   int var = 0;
   volatile int b = 0;
   setjmp( buf );
   if (b < 3) {
      b++;
      var ++;
      printf( "\n\n%d\n", var );
      longjmp( buf, 1 );
   }

   return 0;
}
```

Посмотрим на код, получающийся в результате компиляции этой программы без оптимизаций (`-O0`) и с ними (`-O2`).

Без оптимизаций

Листинг 13.10: ./listings/setjmp_volatile_o0.asm

```
main:
push rbp
mov  rbp,rsp
sub  rsp,0x20

; argc и argv сохраняются в стеке, чтобы освободить rdi/rsi
mov  DWORD PTR [rbp-0x14],edi
mov  QWORD PTR [rbp-0x20],rsi

; var = 0
```

```
11  mov   DWORD PTR [rbp-0x4],0x0
12
13  ; b = 0
14  mov   DWORD PTR [rbp-0x8],0x0
15
16  ; 0x600a40 это адрес buf (типа jmp_buf, глобальная переменная)
17  mov   edi,0x600a40
18  call  400470 <_setjmp@plt>
19
20  ; по условию if выполняется если b < 3.
21  ; Значит при b > 2 нужно пропустить его
22  mov   eax,DWORD PTR [rbp-0x8]
23  cmp   eax,0x2
24  jg    .endlabel
25
26  ; Здесь всё честно
27  ; b++
28  mov   eax,DWORD PTR [rbp-0x8]
29  add   eax,0x1
30  mov   DWORD PTR [rbp-0x8],eax
31
32  ;var++
33  add   DWORD PTR [rbp-0x4],0x1
34
35  ; вызов printf. Оба аргумента берутся из памяти
36  mov   eax,DWORD PTR [rbp-0x4]
37  mov   esi,eax
38  mov   edi,0x400684
39  ; у нас 0 аргументов с плавающей точкой
40  mov   eax,0x0
41  call  400450 <printf@plt>
42
43  ; вызов longjmp
44  mov   esi,0x1
45  mov   edi,0x600a40
46  call  400490 <longjmp@plt>
47
48  .endlabel:
49  mov   eax,0x0
50  leave
51  ret
```

При запуске будет выведено:

```
1
2
3
```

С оптимизациями

Листинг 13.11: ./listings/setjmp_volatile_o2.asm

```
1  main:
2
3  ; выделяем память в стеке
4  sub   rsp,0x18
5
6  ; аргумент для setjmp -- адрес buf
7  mov   edi,0x600a40
8
9  ; b = 0
10 mov   DWORD PTR [rsp+0xc],0x0
11 ; инструкции перемешаны без потери общего смысла с целью
12 ; более эффективного использования конвейера и других
13 ; внутренних механизмов процессора
14 call  400470 <_setjmp@plt>
15
16 ; честное чтение b из памяти с проверкой
17 mov   eax,DWORD PTR [rsp+0xc]
18 cmp   eax,0x2
19 jle   .branch
20
21 ; return 0
22 xor   eax,eax
23 add   rsp,0x18
24 ret
25
26 .branch:
27
```

```
28  mov  eax,DWORD PTR [rsp+0xc]
29
30  ; второй аргумент printf -- значение var+1
31  ; оно не было считано из памяти, для var
32  ; даже не было выделено памяти
33  ; подсчет произошёл во время компиляции, чтобы
34  ; не тратить время во время работы программы
35  mov  esi,0x1
36
37  ; первый аргумент printf
38  mov  edi,0x400674
39
40  ; запись b = b + 1
41  add  eax,0x1
42  mov  DWORD PTR [rsp+0xc],eax
43
44  xor  eax,eax
45  call 400450 <printf@plt>
46
47  ; longjmp( buf, 1 )
48  mov  esi,0x1
49  mov  edi,0x600a40
50  call 400490 <longjmp@plt>
```

При запуске будет выведено:

```
1
1
1
```

Вопрос 60. *Придумайте, как можно реализовать систему, похожую на обработку исключений в языках высокого уровня, с помощью макросов и* `setjmp`.

13.4 Классические уязвимости

Язык C не создавался как язык для написания надёжных приложений. Он позволяет работать с памятью напрямую и при этом не имеет

никакой «разметки» памяти с целью предотвращения записи «куда не следует» (механизмы защиты памяти существуют, например, в Rust).

13.4.1 Переполнение стекового буфера

Предположим, что в программе используется функция `f` с локальным буфером:

Листинг 13.12: ./listings/buffer_overrun.c

```
#include <stdio.h>

void f( void ) {
   char buffer[16];
   gets( buffer );
}

int main( int argc, char** argv ) {
   f();
   return 0;
}
```

После инициализации стекового кадра буфер будет расположен в стеке таким образом:

Функция `gets` считывает строку с `stdin` и помещает в буфер, адрес которого получает в аргументе. Проблема в том, что она никак не контролирует, насколько вместителен буфер. Она может считать

сколь угодно длинную строку и записать её в память начиная с указанного адреса.

Если строка слишком длинная, она перезапишет сначала буфер, затем сохранённый `rbp`, а потом и адрес возврата. Этого уже достаточно, чтобы программа аварийно завершила свою работу.

Разумеется, `fgets` не единственная функция, которая может привести к проблемам.

13.4.2 return-to-libc

Предположим, что злоумышленник может перезаписать адрес возврата. Тогда в качестве адреса он может поставить адрес начала какой-то функции из самой программы или из `libc`.

Когда текущая функция завершит свою работу, запустится соответствующий адресу возврата код. При этом новая функций будет считать, что её аргументы, конечно, в `rdi`, `rsi` и т.д.

Для этого, конечно, надо знать адрес функции из `libc`. Если включен **ASLR**, то это сделать совсем нетривиально. К тому же существует вопрос того, как именно сформировать аргументы и передать их в регистры.

Наибольший интерес для злоумышленника, конечно, представляет функция `system`.

13.4.3 shellcode

Если же злоумышленник передаст в сообщении небольшой исполняемый код и затрёт адрес возврата так, чтобы новый указывал непосредственно в буфер, куда записалось сообщение, то он сможет выполнить произвольный код в системе. Такой код часто называют **shellcode**, потому что он очень маленький и обычно лишь открывает удалённый shell для злоумышленника на определенном порту.

13.5 Меры противодействия

Перезапись адреса возврата ведёт к двум возможным вариантам:

- Злоумышленник выполняет произвольный код;
- Программа просто завершает свою работу.

Во втором случае происходит т.н. DoS (Denial of service) атака, когда злоумышленник может убить программу, предоставляющую определенный сервис. Это неприятно, но не так плохо, как выполнение произвольного кода.

13.5.1 Использование security cookie

Этот механизм не защищает от DoS атак, но мешает выполнять произвольный код. Суть в том, чтобы добавить специальное секретное значение (называемое **security cookie** в стековый фрейм непосредственно рядом с сохранённым `rbp` и адресом возврата:

Тогда при последовательной записи начиная с какого-то из адресов локальных переменных мы сначала перезапишем security cookie, а потом уже дойдём до `rbp` и адреса возврата. Перед выходом из функции (инструкцией `ret`) компилятор вставит код, проверяющий, был ли изменён security cookie или нет. Если он не сохранился в неизменном виде, то программа аварийно завершает работу, не выполняя инструкцию `ret`, потенциально способную совершить прыжок на shellcode.

И `msvc`, и `gcc` имеют такие механизмы. В `msvc` он по-умолчанию включен. В *nix мире есть разные вариации этого механизма.

Замечание. *Помимо security cookie, это значение также называют stack guard и canary.*

13.5.2 Address space layout randomization

Если загружать каждую секцию программы по случайному адресу, то злоумышленник не сможет корректно сформировать адрес возврата ни для какого разумного прыжка.

Распространённые ОС поддерживают этот механизм. Однако необходимо, чтобы приложения были скомпилированы с поддержкой этой технологии. Тогда при создании исполняемого файла в него будет добавлена информация для загрузчика о поддержке ASLR. Чтобы все прыжки и обращения в память были валидными, загрузчику нужно будет произвести релокацию. Или же нужно генерировать **Position Independent Code**

13.5.3 DEP

Мы уже упоминали DEP. Эта технология, которая защищает определённые страницы виртуальной памяти от выполнения кода с них. Если DEP включен, то загрузчик должен пометить страницы, не принадлежащие секции кода, битом EX. DEP защищает от атак, которые свзяаны с выполнением кода в области данных (или кучи). Однако он не подходит для защиты в тех случаях, когда выполняемый код формируется в процессе выполнения программы, например, при работе любого Just-in-time-компилятора.

Это совсем не редкий случай. Все распространённые браузеры используют движки Javascript с jit-компиляцией (иначе Javascript выполнялся бы гораздо медленнее).

Чтобы включить DEP необходимо скомпилировать программу с его поддержкой. Тогда в заголовок исполняемого файла будет включена информация для загрузчика о том, что приложение поддерживает DEP.

Глава 14

Производительность

> Premature optimization is the root of all evil
>
> D. Knuth

В этой главе мы больше узнаем о том, как писать производительный код. Очень многое зависит от конкретной машины и системы, и единственный по-настоящему универсальный совет – обязательно измерять производительность, причем *воспроизводимым* способом. Иными словами, в таком эксперименте стоит явно описать систему, на которой производились измерения, настолько полно, чтобы любой другой человек смог собрать такую же систему и повторить этот эксперимент, получив схожие результаты. Сегодня процессоры настолько сложны, что предсказать их поведение крайне сложно.

14.1 SSE и AVX

Фон Неймановская модель вычислений изначально не предполагала параллельного исполнения команд. С течением времени, однако, стало ясно, что многие вычисления можно производить параллельно, так как их части друг от друга не зависят. Например, посчитать сумму тысячи чисел можно, посчитав сумму 10 частей этого массива по 100 чисел на 10 процессорах, а потом просуммировав результаты (типичный пример использования методики **map-reduce** [1]).

Параллелизм можно, однако, вводить не только на уровне последовательности команд, но и на уровне отдельных инструкций. На-

пример, можно производить несколько арифметических действий с парами операндов в памяти. Для этого, конечно, в процессоре должны быть дополнительные арифметико-логические устройства, но нет необходимости, например, в одновременной выборке нескольких разных команд. Такие команды называют **SIMD**-командами (Single Instruction, Multiple Data).

На идее использования SIMD-инструкций основаны расширения системы команд под названием **SSE (Streaming SIMD Extension)** и, более нового, **AVX (Advanced Vector Extensions)**. Команды из этих расширений используются не только для манипуляции с несколькими числами с плавающей точкой одновременно, но рекомендованы для любых вычислений с плавающей точкой.

По умолчанию `gcc` будет генерировать SSE-инструкции для работы с числами с плавающей точкой. Операнды для них обычно хранятся в `xmm`-регистрах или в памяти.

Замечание. *Если вы собираете несколько частей проекта разными компиляторами, нужно следить, чтобы они все использовали единый способ работы с числами с плавающей точкой. Устаревший способ с использованием сопроцессора плохо совместим с использованием SSE-инструкций.*

Инструкций, которые могут оперировать с несколькими парами операндов одновременно, достаточно много. Для начала рассмотрим такой пример.

Листинг 14.1: ./listings/examples/simd/src/main.c

```
1  #include <stdlib.h>
2  #include <stdio.h>
3
4  void sse( float*, float[4] );
5
6  int main() {
7     float x[4] = {1.0f, 2.0f, 3.0f, 4.0f };
8     float y[4] = {5.0f, 6.0f, 7.0f, 8.0f };
9
10    sse( x, y );
11
12    printf( "%f␣%f␣%f␣%f\n", x[0], x[1], x[2], x[3] );
13    return 0;
14 }
```

Здесь вызывается функция `sse`, определённая где-то в другом файле, которая принимает два массива чисел типа `float` по 4 элемента каждый. Она делает с ними какие-то действия и перезаписывает первый из них результатом. Затем будет выведен изменённый первый массив.

Функция `sse` определена в другом файле.

Листинг 14.2: ./listings/examples/simd/src/sse.asm

```
section .text
global sse
; rdi = x, rsi = y
sse:
   movdqa xmm0, [rdi]
   mulps xmm0, [rsi]
   addps xmm0, [rsi]
   movdqa [rdi], xmm0
   ret
```

Чтобы правильно собрать исполняемый файл, нужно сначала сгенерировать объектные файлы из C-кода и ассемблерного кода с помощью `gcc` и `nasm` соответстенно. Затем можно с помощью `gcc` вызвать компоновщик, который соберёт из объектных файлов исполняемый.

```
CC=gcc
CFLAGS=-ggdb -Wall -ansi -pedantic
ASM=nasm
ASMFLAGS=-f elf64

all: main.o sse.o
  $(CC) $(CFLAGS) build/main.o build/sse.o -o build/main

main.o:
  $(CC) $(CFLAGS) -c src/main.c -o build/main.o

sse.o:
  $(ASM) $(ASMFLAGS) src/sse.asm -o build/sse.o
```

Будем называть **упакованными** значения, если ими заполнен `xmm`-регистр или связная область памяти такого же размера.

Например, `float x[4]` в примере – упакованная четвёрка значений.

Были использованы следующие команды из расширения SSE:

- `movdqa` – **MOV**e **D**ouble **Q**word **A**ligned – для перемещения 16-байтных данных между памятью и `xmm`-регистрами; природа данных неважна, но они должны быть выровнены по границе 16.
- `mulps` – **MUL**tiply **P**acked **S**ingle precision floating point values – для умножения четырёх упакованных значений типа `float` на четыре других значения в `xmm`-регистре или в памяти.
- `addps` – **ADD P**acked **S**ingled precision floating point – для сложения четырёх упакованных значений типа `float` с четырьмя другими значениями в `xmm`-регистре или в памяти.

Обратите внимание: практически все SSE-инструкции требуют, чтобы операнд в памяти был **выровнен по границе 16**. Некоторые инструкции позволяют работать с невыровненными данными, но они медленнее, так что лучше всегда выравнивать данные.

Общий паттерн наименования похож: помимо семантики действия (`mov`, `add`, `mul` ...) добавляются суффиксы, чтобы обозначить тип операндов. Первым идёт **P** (packed) или **S** (scalar, для одиночных значений). Вторым – **D** для значений типа `double` (double precision) или **S** для значений типа `float` (single precision).

Чтобы получить некоторый кругозор, изучите эту выборку команд по [7]:

- `movsd`
- `movdqa`
- `movdqu`
- `mulps`
- `mulpd`
- `addps`
- `haddps`
- `shufps`
- `unpcklps`
- `packswb`
- `cvtdq2pd`
- `mulsd`

Эти инструкции принадлежат набору SSE. В 2008 Intel представил новое расширение – AVX, в котором были введены дополнительные команды, а регистры `xmmN` были расширены в два раза (до 256 бит), став младшими половинками регистров `ymmN`.

Новые инструкции предваряются префиксом v, например, `vbroadcastss`.

Не всегда новые инструкции приносят существенный прирост в производительности. Часто более дорогие модели процессоров отличаются от более дешёвых (той же линейки) схемной поддержкой тех или иных сложных команд. Иными словами, команда, перемножающая 8 пар значений типа `float` может использовать одновременно 8 схем для перемножения чисел или же интерпретироваться как использование четырёх схем умножения дважды.

14.1.1 Задание: сепия-фильтр для изображения

В одном из предыдущих заданий вы реализовывали программу для поворота изображения. Если вы хорошо продумали архитектуру, вам не составит труда добавить в неё фильтр сепия.

Фактически, смысл фильтра в том, чтобы посчитать новое значение яркости каждого из трёх цветов в точке на основании старых значений. С точки зрения математики это умножение вектора из трёх компонент на матрицу 3x3.

Пусть новое значение пикселя выражается как $(B\ G\ R)^T$.

В векторной форме:

$$\begin{pmatrix} B \\ G \\ R \end{pmatrix} = \begin{pmatrix} b \\ g \\ r \end{pmatrix} \times \begin{pmatrix} c_{11} & c_{12} & c_{13} \\ c_{21} & c_{22} & c_{23} \\ c_{31} & c_{32} & c_{33} \end{pmatrix}$$

В скалярной форме:

$$\begin{aligned} B &= bc_{11} + gc_{12} + rc_{13} \\ G &= bc_{21} + gc_{22} + rc_{23} \\ R &= bc_{31} + gc_{32} + rc_{33} \end{aligned}$$

Иногда компилятор генерирует специализированные версии функций, если один из параметров при вызове – константа.

Обратите внимание: используется арифметика с насыщением, т.е. если при сложении мы получили результат, больший 255, то мы положим результат равным 255.

Сделать это на C легко:

Листинг 14.3: ./listings/image_sepia_example.c

```
struct image_t {
   uint32_t width, height;
   struct pixel_t* array;
};

void sepia_c_inplace( struct image_t* img ) {
   uint32_t x,y;
   for( y = 0; y < img->height; y++ )
      for( x = 0; x < img->width; x++ )
         sepia_one( pixel_of( *img, x, y ) );
}


static unsigned char sat( uint64_t x) {
   if (x < 256) return x; return 255;
}
static void sepia_one( struct pixel_t* const pixel ) {
   static const float c[3][3] = {
      { .393f, .769f, .189f },
      { .349f, .686f, .168f },
      { .272f, .543f, .131f } };
   struct pixel_t const old = *pixel;

   pixel->r = sat(
         old.r * c[0][0] + old.g * c[0][1] + old.b * c[0][2]
         );
   pixel->g = sat(
         old.r * c[1][0] + old.g * c[1][1] + old.b * c[1][2]
         );
   pixel->b = sat(
         old.r * c[2][0] + old.g * c[2][1] + old.b * c[2][2]
         );
}
```

В задании необходимо:

- реализовать в отдельном файле ассемблерную процедуру для применения фильтра к большей части изображения (кроме, возможно, нескольких последних пикселей);
- эта ассемблерная процедура должна использовать xmm-регистры для того, чтобы работать с несколькими пикселями одновремен-

но;

- удостовериться, что сепия правильно работает как в случае C-кода, так и в случае ASM-кода;
- сравнить на большом количестве итераций время работы сепии на C и с помощью SSE-инструкций ИЛИ использовать очень большое изображение (порядка сотен мегабайт).

Чтобы добиться реального выигрыша в производительности, лучше осуществлять как можно больше операций одновременно. Каждый пиксель это три байта, после перевода во float каждый пиксель займёт 12 байт. В `xmm`-регистры помещается 16 байт. Соответственно наименьшее число байт, делящееся и на 12 и на 16 и есть наша цель. Очевидно, это 48, что соответствует трём `xmm`-регистрам.

Пусть нижний индекс обозначает номер пикселя. Тогда изображение выглядит так:

$$b_1 g_1 r_1 b_2 g_2 r_2 b_3 g_3 r_3 b_4 g_4 r_4 \dots$$

Пускай мы хотим обработать первые четыре компоненты. Три из них соответствуют первому пикселю, одна – второму.

Чтобы умножить вектор на матрицу с помощью SSE инструкций, удобно сформировать следующие значения в регистрах:

$xmm_0 = b_1 b_1 b_1 b_2$

$xmm_1 = g_1 g_1 g_1 g_2$

$xmm_2 = r_1 r_1 r_1 r_2$

Пусть в памяти или в xmm-регистрах у нас хранятся коэффициенты. Но хранить мы будем не строчки матрицы коэффициентов, а её столбцы.

Для демонстрации идеи сформируем следующие значения:

$xmm_3 = c_{11}|c_{21}|c_{31}|c_{11}$

$xmm_4 = c_{12}|c_{22}|c_{32}|c_{12}$

$xmm_5 = c_{13}|c_{23}|c_{33}|c_{13}$

Перемножив их с помощью `mulps` с `xmm0 - xmm2` получим:

$xmm_3 = b_1 c_{11}|b_1 c_{21}|b_1 c_{31}|b_2 c_{11}$

$xmm_4 = g_1 c_{12}|g_1 c_{22}|g_1 c_{32}|g_2 c_{12}$

$xmm_5 = r_1 c_{13}|r_1 c_{23}|r_1 c_{33}|r_2 c_{13}$

Остаётся пакетно сложить их с помощью инструкций `addps`.

Аналогичным образом можно поступить с двумя другими четвёрками компонент.

Такая техника умножения матрицы на вектор с транспонированием матрицы коэффициентов, чтобы можно было использовать обычные команды пакетного сложения и умножения, а не горизонтального сложения, использовалась Intel в [9].

Обратите внимание на то, что перекодировать `char` в `float` быстрее всего с помощью таблицы. Она будет выглядеть примерно так:

```
1  float const byte_to_float[] = {
2      0.0f, 1.0f, 2.0f, ..., 255.0f };
```

Дополнительные задания

1. Посчитать доверительные интервалы с помощью коэффициента Стьюдента для доверительной вероятности 0.95

14.2 Оптимизации

В этом разделе мы рассмотрим некоторые важные оптимизации, которые используются в языках высокого уровня.

14.2.1 Миф о скорости C

Существует миф о том, что если написать программу на C, она будет быстрее, чем, скажем, программа на Java. На самом деле это далеко не всегда правда.

Разнообразные тесты производительности обычно тем адекватнее, чем более узконаправлены. Но это лишает нас права делать какие-то обобщения и говорить о производительности вообще, а не в конкретных сценариях использования.

Поэтому разговор о производительности имеет смысл только для конкретных сценариев и всегда должен подкрепляться тестами производительности. Они должны быть описаны достаточно подробно, чтобы их можно было повторить.

Когда программа на C бывает медленнее?

Куча в стандартной библиотеке C работает существенно иначе, чем в JVM.

В начале 2000-х годов реализация `malloc` в `libc` была построена схожим образом с тем, как она описана в задании 12.8. Она обладает определёнными недостатками, в частности, заранее сложно предугадать, сколько времени будет работать `malloc`, так как ему приходится проходить по связному списку блоков в поиске свободного.

В Java выделение памяти происходит обычно очень быстро. Очень грубо говоря, куча представляет из себя связную область памяти, в которой отмечена граница свободного пространства. При выделении памяти мы просто двигаем этот указатель на размер свежевыделенной области.

Цена за это – необходимость в уплотнении кучи, которое происходит при сборке мусора. Однако если сборки мусора не происходит, то производительность такого решения выше. (Разумеется, такой аллокатор можно написать и использовать на C).

Предположим, что программа, которая использовалась для тестирования, выделяет некоторое количество памяти, а затем её освобождает при завершении работы. Если сборщик мусора так и не начал деятельность по уплотнению кучи, то программа на Java может отработать быстрее в сравнении с оной на C, которая аккуратно будет освобождать участки кучи.

Кроме того, в отличие от компилятора C, JVM имеет доступ к оптимизациям, которые основаны на том, как именно программа обычно выполняется. Например, оттранслированный код часто использующихся последовательно методов можно поместить рядом в памяти, чтобы они вместе попадали в кэш. Для долго работающих программ, о поведении которых можно собрать статистику, это иногда приводит к существенному приросту производительности.

Реальная особенность C не в высокой производительности, а в том, что у него очень простая модель стоимостей операций. В сравнении с языками, которые заточены под виртуальные машины (JVM), или предоставляющими абстракции более высокого уровня за счёт дополнительного машинного кода (C++), оценить, где в программе вообще возникают накладные расходы, просто. Фактически, единственные две абстракции, которые предоставляет C – процедуры и составные типы данных (структуры и объединения).

Сам по себе код на C если наивно оттранслировать его в машинные коды будет работать достаточно медленно – гораздо медленее, чем код на высокоуровневом языке, пропущенный через оптимизатор. Чтобы ускорить его, компилятор использует знания об архитектуре компьютера. **Очень непросто самому написать на ассемблере программу более эффективную, нежели сгенерированную компилятором с учетом поведения конвейера команд, branch-predicting'а и других скрытых от программиста механизмов.** Если же у программиста есть большой опыт и понимание того, что он делает, то иногда он может немного переписать программу на C в ущерб читаемости и поддерживаемости, но в угоду производитель-

ности. Однако и здесь всякое изменение должно пройти через тесты, чтобы с определенной степенью надёжности говорить о его эффективности.

14.2.2 Как писать быстрый код?

При написании программ почти всегда **не стоит сразу задумываться о том, чтобы оптимизировать их на низком уровне**.

Замечание. *Premature optimization is the root of all evil. (D. Knuth)*

В отношении большинства программ выполняется эмпирическое правило: почти всё время выполняется очень небольшая часть кода. Этот код и является узким местом программы и именно его нужно оптимизировать. Наиболее надежный способ понять, что именно нужно оптимизировать – использовать **профилировщик** – утилиту, которая определяет, какие участки кода выполняются дольше.

Главную роль играют алгоритмические оптимизации. Если можно найти более эффективный алгоритм, то, обычно, никакие низкоуровневые оптимизации не повысят быстродействие в той же мере. Важно правильно выбрать структуры данных, так как каждая из них позволяет делать какие-то операции быстро, а какие-то – нет. Например, в связный список удобно добавлять элементы, а в массиве быстро обращаться по индексу.

Часто простой и ясный код является и самым эффективным. Это включает в себя написание максимально маленьких функций и использование простых выражений.

Обычно оптимизациями управляют с помощью флагов `-O0`, `-O1`, `-O2`, `-O3`, `-Os` (optimize space usage, исполняемый файл минимального размера). Чем больше индекс при `O`, тем большее количество оптимизаций включено. Их можно включать (вдобавок к уже определённым в составе соответствующего O-флага) и отключать по отдельности. Для каждого типа оптимизации есть включающие и отключающие флаги, например, `-fforward-propagate` и `-fno-forward-propagate`.

Мы будем обращать внимание на флаги, связанные с оптимизациями (включающие их полностью или частично).

14.2.3 Пропуск инициализации стековых фреймов

Связанные флаги `gcc`: `-fomit-frame-pointer`

Иногда нам на самом деле не нужен регистр `rbp`, указывающий на начало текущего стекового фрейма. Его задача в программе – служить базой для адресации локальных переменных и аргументов. А

если локальных переменных нет (или функция ничего не вызывает и может обойтись **red zone** для складирования аргументов, то можно не создавать полноценный стековый фрейм. У этого, однако, есть и негативная сторона: благодаря связному списку из адресов начал стековых фреймов в любой момент программы можно легко узнать состояние её call-стека, то есть определить начала стековых фреймов, а на основании них – адреса возврата и локальные переменные.

Этот код показывает, как можно пройтись по стеку и отобразить адреса начал стековых фреймов для всех функций, запущенных в момент вызова `unwind`. Для его правильной работы нужно, чтобы `stack_unwind.c` был скомпилирован с `-O 0`.

Листинг 14.4: ./listings/stack_unwind.c

```
void unwind();
void f( int count ) {
   if ( count ) f( count-1 ); else unwind();
}
int main(int argc, char** argv) {
   f( 10 ); return 0;
}
```

Листинг 14.5: ./listings/stack_unwind.asm

```
extern printf
global unwind

section .rodata
format : db "%x ", 10, 0

section .code
unwind:
   push rbx

  ; while (rbx != 0) {
  ;    print rbx; rbx = [rbx];
  ; }
   mov rbx, rbp
.loop:
   test rbx, rbx
   jz .end
```

```
18      mov rdi, format
19      mov rsi, rbx
20      call printf
21      mov rbx, [rbx]
22      jmp .loop
23
24  .end:
25      pop rbx
26      ret
```

rsi Отношение к этому методу оптимизации неоднозначное. Есть мнение, что на современных архитектурах эта оптимизация не приводит к хоть сколько-нибудь заметному выигрышу в производительности, а отладку может затруднить изрядно [14]. Действительно, при сохранении текущего `rbp` в каждом стековом фрейме мы можем легко очертить верхнюю границу каждого фрейма (**адрес возврата** соответствующей ему функции).

Кроме того, если последняя инструкция функции – `call P`, то её можно заменить на `jmp`, если стекового фрейма не было установлено. Тогда данная функция воспринимается как некоторая прелюдия к функции P, которая выполняется, а затем передаёт ей управления, не создавая лишний адрес возврата в стеке.

14.2.4 Концевая рекурсия

Связанные флаги gcc: `-fomit-frame-pointer -foptimize-sibling-calls`
Рассмотрим такую функцию:

Листинг 14.6: ./listings/factorial_tailrec.c

```
1  __attribute__(( noinline ))
2  int factorial( int acc, int arg ) {
3      if ( arg == 0 ) return acc;
4      return factorial( acc * arg, arg-1 );
5  }
6
7  int main(int argc, char** argv) { return factorial(1, argc); }
```

Она рекурсивно вызывает сама себя, но этот вызов особенный.

Говорят, что в функции рекурсия **концевая** (или хвостовая), если функция или возвращает какое-то значение, полученное без рекурсии, или запускает себя рекурсивно с другими аргументами.

Рекурсия не концевая, если с полученным после рекурсивного вызова значением надо еще что-то сделать.

В частности, такая реализация факториала нерекурсивна, потому что после рекурсивного вызова нужно еще умножить его результат на аргумент.

Листинг 14.7: ./listings/factorial_non_tailrec.c

```
__attribute__(( noinline ))
int factorial( int arg ) {
   if ( arg == 0 ) return acc;
   return arg * factorial( arg-1 );
}

int main(int argc, char** argv) { return factorial(argc); }
```

Замечание. *__attribute__((param1, param2, ...)) это директива gcc, которая указывает детали трансляции. Например, чтобы предотвратить встраивание функции factorial, мы пометили её атрибутом noinline.*

Современные компиляторы C обычно умеют оптимизировать концевую рекурсию. Вот код, который сгенерируется после ассемблирования `factorial_tailrec.c`:

Листинг 14.8: ./listings/factorial_tailrec.asm

```
00000000004004c6 <factorial>:
  4004c6: 89 f8            mov  eax,edi
  4004c8: 85 f6            test esi,esi
  4004ca: 74 07            je   4004d3 <factorial+0xd>
  4004cc: 0f af c6         imul eax,esi
  4004cf: ff ce            dec  esi
  4004d1: eb f5            jmp  4004c8 <factorial+0x2>
  4004d3: c3               ret
```

На уровне ассемблера концевую рекурсию можно объяснить так. Представим, что мы хотели бы вызвать `factorial` следущим образом:

```
    ;rdi <- rdi * rsi
    mov rax, rsi
    mul rdi
    mov rdi, rax
```

```
5
6       ; rsi <- rsi - 1
7       dec rsi
8
9       call factorial
10      ret
```

Здесь видно, что последняя инструкция – рекурсивный вызов. Всё, что она фактически делает – кладёт в стек адрес возврата, который по завершении нового экземпляра factorial будет оттуда вынут инструкцией `ret`, выходящей из вложенного вызова. После этого мы окажемся опять на инструкции `ret`, но уже текущего экземпляра функции. Если такая функция, скажем, вызвалась пять раз, то последними пятью командами, которые выполнит процессор, будут пять инструкций `ret`, очищающих стек от одинаковых адресов возврата.

Легко заметить, что при концевой рекурсии мы на самом деле хотим положить в регистры новые аргументы и прыгнуть на начало функции, не загрязняя стек новым адресом возврата:

```
1       ;rdi <- rdi * rsi
2       mov rax, rsi
3       mul rdi
4       mov rdi, rax
5
6       ; rsi <- rsi - 1
7       dec rsi
8
9       jmp factorial
```

Вообще пара инструкций `call` – `ret` может быть заменена на `jmp` на тот же адрес, на который производился вызов. В самом деле, это лишь вызов той же функции с другими аргументами, а значит нужно установить регистры для аргументов и прыгнуть на адрес её начала.

Зачем вообще концевая рекурсия? Известно, что в наивном варианте рекурсия медленнее цикла; к тому же необходимо дополнительное место в стеке, что может привести к его переполнению. Концевую рекурсию компилятор может автоматически преобразовать в цикл. Почему не писать сразу цикл? Некоторые алгоритмы просто слишком красиво выглядят в рекурсивном виде, чтобы этим можно было пренебречь.

Листинг 14.9: ./listings/tail_rec_example_list.c

```
1   #include <stdio.h>
```

```
2  #include <malloc.h>
3  struct llist_t {
4     struct llist_t* next;
5     int value;
6  };
7
8  struct llist_t* llist_at(
9        struct llist_t* lst,
10       size_t idx ) {
11    if ( lst && idx ) return llist_at( lst->next, idx-1 );
12    return lst;
13 }
14 struct llist_t* c( int value, struct llist_t* next) {
15    struct llist_t* lst = malloc( sizeof(struct llist_t*) );
16    lst->next = next;
17    lst->value = value;
18    return lst;
19 }
20
21 int main( int argc, char** argv ) {
22    struct llist_t* lst = c( 1, c( 2, c( 3, NULL )));
23    printf("%d\n", llist_at( lst, 2 )->value );
24    return 0;
25 }
```

После компиляции с `-Os` будет сгенерирован нерекурсивный код:

Листинг 14.10: ./listings/tail_rec_example_list.asm

```
1  0000000000400596 <llist_at>:
2    400596:    48 89 f8          mov   rax,rdi
3    400599:    48 85 f6          test  rsi,rsi
4    40059c:    74 0d             je    4005ab <llist_at+0x15>
5    40059e:    48 85 c0          test  rax,rax
6    4005a1:    74 08             je    4005ab <llist_at+0x15>
7    4005a3:    48 ff ce          dec   rsi
8    4005a6:    48 8b 00          mov   rax,QWORD PTR [rax]
9    4005a9:    eb ee             jmp   400599 <llist_at+0x3>
10   4005ab:    c3                ret
```

Как это использовать? Не бояться использовать концевую рекурсию, если это ведёт к более понятному коду.

14.2.5 Common Subexpressions Elimination

Связанные флаги gcc: `-fgcse` и другие, содержащие в названии `cse`.

Если необходимо подсчитать два выражения, части которых совпадают, необязательно предварительно подсчитывать их общую часть, вынося её вручную в отдельную переменную. Вот пример, где подвыражение x^2+2x подсчитывается только один раз, хотя наивно должно было бы посчитаться дважды:

Листинг 14.11: ./listings/common_subexpressions.c

```
#include <stdio.h>

__attribute__((noinline))
void test(int x) {
   printf("%d␣%d",
         x*x + 2*x + 1,
         x*x + 2*x - 1 );
}

int main(int argc, char** argv) {
   test( argc );
   return 0;
}
```

Листинг 14.12: ./listings/common_subexpressions.asm

```
0000000000400516 <test>:
; rsi = x + 2
 400516:    8d 77 02          lea  esi,[rdi+0x2]
 400519:    31 c0             xor  eax,eax
 40051b:    0f af f7          imul esi,edi
; rsi = x*(x+2)
 40051e:    bf b4 05 40 00    mov  edi,0x4005b4
; rdx = rsi-1 = x*(x+2) - 1
 400523:    8d 56 ff          lea  edx,[rsi-0x1]
; rsi = rsi + 1 = x*(x+2) - 1
```

```
11    400526:    ff c6             inc   esi
12    400528:    e9 b3 fe ff ff    jmp   4003e0 <printf@plt>
```

Как это использовать? Если программа лучше читается, когда в ней формулы написаны целиком, то нет смысла пытаться добиться увеличения производительности, самому подсчитав значения общих подвыражений.

14.2.6 Constant propagation

Связанные флаги gcc: `-fipa-cp`, `-fgcse`, `-fipa-cp-clone` и др.

Речь идёт о подстановке непосредственно заданных значений вместо переменных и выражений, если компилятор может доказать, что семантика программы не изменится.

Иногда компилятор генерирует специализированные версии функций, если один из параметров при вызове – константа (`-fipa-cp-clone`. Например, здесь будет создана версия функции, в которой первого аргумента нет: вместо него повсюду проставлено число 42.

Листинг 14.13: ./listings/constant_propagation.c

```
1  __attribute__((noinline))
2  static int sum(int x, int y) { return x + y; }
3
4  int main( int argc, char** argv ) {
5
6     return sum( 42, argc );
7  }
```

Листинг 14.14: ./listings/constant_propagation.asm

```
1  00000000004004c0 <sum.constprop.0>:
2    4004c0:    8d 47 2a          lea   eax,[rdi+0x2a]
3    4004c3:    c3                ret
```

Как это использовать? Нет опасности в именных константах. Кроме того, всё, что компилятор сможет подсчитать на этапе компиляции, будет подсчитано (включая, например, значение функции `factorial` на аргументе 6. Много копий функций в редких случаях могут увеличить размер кода очень сильно, что плохо для локальности.

14.2.7 Return value optimization

Это важная оптимизация, позволяющая уменьшить количество копирований.

Вспомним, что наивно локальные переменные создаются внутри стекового кадра функции. Соответственно, если функция возвращает экземпляр структуры, она должна будет сначала создать его у себя в локальных переменных, а перед выходом скопировать во внешний мир (если структура не вмещается в пару регистров `rax`, `rdx`).

Листинг 14.15: ./listings/nrvo.c

```
struct p {
   long x;
   long y;
   long z;
};

__attribute__((noinline))
struct p f(void) {
   struct p copy;
   copy.x = 1;
   copy.y = 2;
   copy.z = 3;
   return copy;
}

int main(int argc, char** argv) {
   volatile struct p inst = f();
   return 0;
}
```

Листинг 14.16: ./listings/nrvo_off.asm

```
00000000004004b6 <f>:
; Пролог
  4004b6: 55                push  rbp
  4004b7: 48 89 e5          mov   rbp,rsp
; Аргументом (скрытым) будет адрес, по которому надо записать
; содержимое структуры
; Сохраняем в стеке этот аргумент
  4004ba: 48 89 7d d8       mov   QWORD PTR [rbp-0x28],rdi
```

```
9   ; Формируем структуру в стеке
10    4004be: 48 c7 45 e0 01 00 00  mov QWORD PTR [rbp-0x20],0x1
11    4004c5: 00
12    4004c6: 48 c7 45 e8 02 00 00  mov QWORD PTR [rbp-0x18],0x2
13    4004cd: 00
14    4004ce: 48 c7 45 f0 03 00 00  mov QWORD PTR [rbp-0x10],0x3
15    4004d5: 00
16  ; rax = адрес, куда запишем содержимое структуры
17    4004d6: 48 8b 45 d8       mov   rax,QWORD PTR [rbp-0x28]
18  ; [rax] = 1
19    4004da: 48 8b 55 e0       mov   rdx,QWORD PTR [rbp-0x20]
20    4004de: 48 89 10          mov   QWORD PTR [rax],rdx
21  ; [rax + 8] = 2
22    4004e1: 48 8b 55 e8       mov   rdx,QWORD PTR [rbp-0x18]
23    4004e5: 48 89 50 08       mov   QWORD PTR [rax+0x8],rdx
24  ; [rax + 10] = 3
25    4004e9: 48 8b 55 f0       mov   rdx,QWORD PTR [rbp-0x10]
26    4004ed: 48 89 50 10       mov   QWORD PTR [rax+0x10],rdx
27  ; rax = адрес, по которому записывали содержимое структуры
28    4004f1: 48 8b 45 d8       mov   rax,QWORD PTR [rbp-0x28]
29    4004f5: 5d                pop   rbp
30    4004f6: c3                ret
31
32  00000000004004f7 <main>:
33    4004f7: 55                push  rbp
34    4004f8: 48 89 e5          mov   rbp,rsp
35    4004fb: 48 83 ec 30       sub   rsp,0x30
36    4004ff: 89 7d dc          mov   DWORD PTR [rbp-0x24],edi
37    400502: 48 89 75 d0       mov   QWORD PTR [rbp-0x30],rsi
38    400506: 48 8d 45 e0       lea   rax,[rbp-0x20]
39    40050a: 48 89 c7          mov   rdi,rax
40    40050d: e8 a4 ff ff ff    call  4004b6 <f>
41    400512: b8 00 00 00 00    mov   eax,0x0
42    400517: c9                leave
43    400518: c3                ret
44    400519: 0f 1f 80 00 00 00 00  nop DWORD PTR [rax+0x0]
```

Оказывается, компиляторы умеют действовать более эффективно.

Листинг 14.17: ./listings/nrvo_on.asm

```
1   00000000004004b6 <f>:
```

```
4004b6: 48 89 f8          mov  rax,rdi
4004b9: 48 c7 07 01 00 00 00  mov QWORD PTR [rdi],0x1
4004c0: 48 c7 47 08 02 00 00  mov QWORD PTR [rdi+0x8],0x2
4004c7: 00
4004c8: 48 c7 47 10 03 00 00  mov QWORD PTR [rdi+0x10],0x3
4004cf: 00
4004d0: c3                ret

00000000004004d1 <main>:
4004d1: 48 83 ec 20       sub  rsp,0x20
4004d5: 48 89 e7          mov  rdi,rsp
4004d8: e8 d9 ff ff ff    call 4004b6 <f>
4004dd: b8 00 00 00 00    mov  eax,0x0
4004e2: 48 83 c4 20       add  rsp,0x20
4004e6: c3                ret
4004e7: 66 0f 1f 84 00 00 00 00  nop WORD PTR [rax+rax*1+0x0]
4004ee: 00 00
```

Мы более не выделяем место в локальных переменных, чтобы сохранить там переменную, единственная роль которой – быть скопированной во внешний мир. Вместо этого мы сразу переписываем её поля в том месте, куда она будет записана перед завершением функции.

Как это использовать? Если ваша задача – сделать функцию, которая заполняет определённую структуру и возвращает его, не думайте, что вам обязательно передавать в неё явно указатель на область памяти (что не очень красиво) или выделять память с помощью `malloc` (что медленно, как вы убедились, написав свой аллокатор памяти).

Branch prediction

Когда присутствует условный переход в машинных инструкциях (например, `jg`), процессор может начать исполнять одну или обе ветки программы еще до того, как проверка пройдёт, спекулятивно. Если он ошибается, то откатывается и исполняет правильную ветку. Если нет, то этот механизм повышает производительность, причем на современных процессорах это один из важнейших факторов, влияющих на производительность, поскольку позволяет не простаивать частям процессора, которые иначе бы ждали полного завершения всех команд, необходимых для проверки условия перехода.

Ошибки предсказателя негативно сказываются на производительности, но должны случаться редко. Компиляторы используют знание о том, как обычно работает этот механизм, чтобы создавать более быстрый код.

Существует два типа ?? предсказаний: **статические** и **динамические**.

- Если процессор в момент перехода не собрал никакой релевантной информации (обычно, когда этот переход происходит впервые), то используется статический алгоритм предсказания. Распространён простой алгоритм:
  - Если это переход вперед, процессор предполагает, что он произойдёт.
  - Если это переход назад, он, скорее всего, не произойдёт.

  Это имеет смысл, поскольку переходы для организации цикла более вероятно будут происходить, нежели нет (т.е. циклы обычно выполняются много раз).

- Если переход уже происходил, процессор использует более сложные алгоритмы. Какие – зависит от модели процессора.

  Например, можно использовать кольцевой буфер, в который при условном переходе будет записываться бит, означающий, был ли переход произведен. Таким образом, этот буфер будет хранить историю переходов, что можно использовать для предсказания. При таком подходе небольшие циклы длины, делящей буфер нацело, будут хорошо предсказываться.

  Актуальная информация об организации конвеера и предиктора может быть найдена в ??. К сожалению, часто информация о внутреннем устройстве процессора получается лишь путём его тестирования как чёрного ящика, т.к, производители считают её коммерческой тайной.

Как это использовать? При использовании if-then-else и switch сначала обрабатывайте более вероятные случаи. Кроме того существуют специальные подсказки для предсказателя в виде префиксов к инструкциям перехода, см.??.

Execution unit

Процессор на самом деле состоит из многих частей, разные части задействуются разными командами на разных этапах их исполнения. Эти части и называются **execution unit** Некоторые части дублируются, возможно, несколько раз, что позволяет использовать их параллельно на уровне микропрограммы. Например, уже на процессоре Pentium IV возможно было исполнять четыре инструкции, задействующие арифметико-логический блок, если правильно описать этот процесс.

Рассмотрим следующий код:

Листинг 14.18: ./listings/examples/cycle_nonparallelized_arith.asm

```
looper:
      mov   rax,[rsi]
; эта строчка зависит от предыдущей, т.е.
; нельзя поменять эти инструкции местами без
; потери смысла кода
      xor   rax, 0x1
; еще одна зависимость
      add   [rdi],rax
      add   rsi,8
      add   rdi,8
      dec   rcx
      jnz   looper
```

Как можно было бы ускорить его? Мы видим зависимости между командами, которые ограничивают внутренний оптимизатор самого процессора, работающий на уровне микропрограммы. Развернём цикл так, чтобы две итерации старого цикла стали одной итерацией нового цикла.

Листинг 14.19: ./listings/examples/cycle_parallelized_arith.asm

```
looper:
      mov   rax, [rsi]
      mov   rdx, [rsi + 8]
      xor   rax, 0x1
      xor   rdx, 0x1
      add   [rdi],rax
      add   [rdi+8],rdx
      add   rsi, 16
```

```
9           add   rdi, 16
10          sub   rcx, 2
11          jnz   looper
```

Теперь зависимости строчек друг от друга убрались. Строчки двух итераций оказались перемешаны в одном цикле. В таком порядке работать они будут значительно быстрее, нежели если бы они выполнились последовательно.

Узнавать, какие в процессоре существуют execution unit'ы нужно для каждой модели процессора специально. Эта информация подсказывает, как оптимизировать программу, чтобы выжать максимум ресурсов из процессора.

Например, для процессоров Haswell хорошее описание можно найти в [8]

Группировка чтений и записи

По схемотехническим причинам лучше если чтения и записи будут сгруппированы вместе. Пример менее производительного кода:

```
1           mov rax,[rsi]
2           mov [rdi],rax
3           mov rax,[rsi+8]
4           mov [edi+4],eax
5           mov rax,[rsi+16]
6           mov [rdi+16],rax
7           mov rax,[esi+24]
8           mov [rdi+24],eax
```

В нём чтения и записи чередуются. А так можно его улучшить, сгруппировав чтения вместе:

```
1           mov rax, [rsi]
2           mov rbx, [rsi+8]
3           mov rcx, [rsi+16]
4           mov rdx, [rsi+24]
5           mov [rdi], rax
6           mov [rdi+8], rbx
7           mov [rdi+16], rcx
8           mov [rdi+24], rdx
```

14.3 Кэширование

14.3.1 Эффективное использование кэшей

Кэширование — один из важнейших механизмов увеличения производительности.

Прежде всего заметим, что вопреки фон Нейману для данных и инструкций в распространённых процессорах уже почти 25 лет используют *разные* кэши. Инструкции и код почти всегда располагаются в разных областях памяти, что обуславливает большую эффективность раздельных кэшей. Сейчас нас интересует кэш данных.

По умолчанию все операции с памятью происходят не напрямую, а с участием кэша (не считая страниц, особо помеченных ОС как некэшируемые).

В кэш попадают участки памяти фиксированного размера (cache-line) размера 64 байта, **выровненные** по границе 64.

Схемотехнически кэш-память устроена особо: каждому кусочку сопоставляется **тэг** – адрес этого кусочка в основной памяти. С помощью специальных схем можно сделать поиск элемента кэша по адресу его начала в памяти очень быстрым (но только для небольших кэшей, например, 4 МБ на процессор, иначе это очень дорого). При попытке чтения из памяти процессор сначала попытается достать значение из кэша.

Если его в кэше нет, то сооветствующий кусочек памяти подгрузится в кэш, а затем оттуда прочитан. Такая ситуация называется **cache-miss** и является одним из главных тормозящих программы факторов.

Сейчас кэшей, обычно, несколько уровней, каждый следующий больше и медленнее. **LL-кэшем** называется последний уровень кэша перед оперативной памятью.

Кэширование работает хорошо если выполняется свойство локальности. Поэтому если локальность нарушается, может иметь смысл не использовать кэш, например, при единократной записи в память, если мы уверены, что чтения с этих адресов долго не будет.

Предзагрузка данных в кэш

Если вы хотите подсказать процессору, что вскоре будет доступ к определенной области памяти, можно использовать инструкцию `prefetch`. Она принимает адрес из куска памяти, который процессор постарается поместить в кеш в ближайшее время.

Использование `prefetch` может быть эффективной микрооптимизацией, но для этого необходимо обязательно испытывать код на ре-

альной машине. Если эту инструкцию поместить слишком близко к обращению к данным, рекомендованным к кэшированию, процессор не успеет её асинхронно выполнить, и при обращении к этим данным всё равно произойдёт cache-miss.

Кроме того, следует понимать, что имеет значение не то, в каком месте программы структурно был использован `prefetch`, а то, в какой момент времени она вызвалась относительно собственно обращения в память. Это означает, что очень часто вставлять эту инструкцию нужно будет в совершенно логически не связанную с чтением данных часть программы, просто потому что какая-то функция оттуда вызывает код из другого модуля, ответственный за чтение. Это очень плохо отражается на возможности переиспользования кода, внося неочевидные зависимости между модулями.

Чтобы использовать возможности инструкции `prefetch` в C, можно использовать одну из встроенных функций-заглушек `gcc`:

```
void __builtin_prefetch (const void *addr, ...)
```

На её место будет вставлена правильная для целевой архитектуры команда.

Помимо адреса, к окрестностям которого скоро будут обращения, она принимает также два параметра, которые должны быть константными числами:

1. Доступ к адресам будет на чтение (0) или на запись (1)? По умолчанию считается равным нулю.

2. Насколько сильная локальность? Число от нуля до трёх, три соответствует максимальной локальности. Ноль подсказывает, что можно не оставлять значение в кэше после использования, три – что нужно оставить его в кэшах всех уровней.

Пример: двоичный поиск

Рассмотрим такой пример:

Листинг 14.20: ./listings/examples/prefetch/prefetch.c

```
1  #include <time.h>
2  #include <stdio.h>
3  #include <stdlib.h>
4
5  #define SIZE 1024*512*16
6
7  int binarySearch(int *array, size_t number_of_elements, int key) {
```

```
8       size_t low = 0, high = number_of_elements-1, mid;
9       while(low <= high) {
10         mid = (low + high)/2;
11  #ifdef DO_PREFETCH
12         // low path
13         __builtin_prefetch (&array[(mid + 1 + high)/2], 0, 1);
14         // high path
15         __builtin_prefetch (&array[(low + mid - 1)/2], 0, 1);
16  #endif
17
18         if(array[mid] < key)
19            low = mid + 1;
20         else if(array[mid] == key)
21            return mid;
22         else if(array[mid] > key)
23            high = mid-1;
24      }
25      return -1;
26  }
27
28  int main() {
29      size_t i = 0;
30      int NUM_LOOKUPS = SIZE;
31      int *array;
32      int *lookups;
33
34
35      srand(time(NULL));
36      array = malloc(SIZE*sizeof(int));
37      lookups = malloc(NUM_LOOKUPS * sizeof(int));
38
39      for (i=0;i<SIZE;i++) array[i] = i;
40      for (i=0;i<NUM_LOOKUPS;i++) lookups[i] = rand() % SIZE;
41
42      for (i=0;i<NUM_LOOKUPS;i++)
43        binarySearch(array, SIZE, lookups[i]);
44      free(array);
45      free(lookups);
46  }
```

Встроенный в процессор механизм предсказания того, какие данные будут необходимы (и что нужно загрузить в кэш заранее), не может совладать с двоичным поиском. Для него паттерн обращений к памяти выглядит случайным образом. Благодаря нашим подсказкам, однако, процессор может отработать более эффективно.

Без prefetch:

```
> gcc -O3 prefetch.c -o prefetch_off && /usr/bin/time -v ./prefetch_off

        Command being timed: "./prefetch_off"
        User time (seconds): 7.56
        System time (seconds): 0.02
        Percent of CPU this job got: 100%
        Elapsed (wall clock) time (h:mm:ss or m:ss): 0:07.58
        Average shared text size (kbytes): 0
        Average unshared data size (kbytes): 0
        Average stack size (kbytes): 0
        Average total size (kbytes): 0
        Maximum resident set size (kbytes): 66432
        Average resident set size (kbytes): 0
        Major (requiring I/O) page faults: 0
        Minor (reclaiming a frame) page faults: 16444
        Voluntary context switches: 1
        Involuntary context switches: 51
        Swaps: 0
        File system inputs: 0
        File system outputs: 0
        Socket messages sent: 0
        Socket messages received: 0
        Signals delivered: 0
        Page size (bytes): 4096
        Exit status: 0
```

С prefetch:

```
> gcc -O3 prefetch.c -o prefetch_off && /usr/bin/time -v ./prefetch_off

        Command being timed: "./prefetch_on"
        User time (seconds): 6.56
        System time (seconds): 0.01
        Percent of CPU this job got: 100%
        Elapsed (wall clock) time (h:mm:ss or m:ss): 0:06.57
        Average shared text size (kbytes): 0
        Average unshared data size (kbytes): 0
        Average stack size (kbytes): 0
        Average total size (kbytes): 0
        Maximum resident set size (kbytes): 66512
        Average resident set size (kbytes): 0
        Major (requiring I/O) page faults: 0
        Minor (reclaiming a frame) page faults: 16443
        Voluntary context switches: 1
        Involuntary context switches: 42
        Swaps: 0
        File system inputs: 0
        File system outputs: 0
        Socket messages sent: 0
        Socket messages received: 0
        Signals delivered: 0
        Page size (bytes): 4096
        Exit status: 0
```

Запись в память минуя кэш

Команда `movntps` позволяет осуществить запись в память минуя кэши. ОС также может управлять кэшированием с помощью бита cache-write-through в таблице страниц, что происходит обычно при memory mapped IO, т.е. когда устройства ввода-вывода отображаются в адресное пространство.

Кроме того, последовательный доступ к памяти намного быстрее, чем «прыжки» от адреса к адресу с большими промежутками между ними. Поэтому старайтесь организовать обращения к памяти так, чтобы они были последовательными.

14.3.2 Пример: инициализация матрицы

Рассмотрим две похожие программы. Они инициализируют большую глобальную матрицу числами 42. Матрица в плоской памяти хранится строчка за строчкой.

Одна программа делает это путем обращений к последовательным ячейкам памяти (для каждой строки обращаемся к каждому элементу), другая – наоборот, для каждого столбца обращаемся к каждому элементу. Разница в скорости оказывается весьма ощутимой.

Приведем результат тестирования.

Инициализация построчно:

Листинг 14.21: ./listings/examples/cache-matrix-init/matrix_init_linear.c

```
1  #include <stdio.h>
2  #include <malloc.h>
3  #define DIM (16*1024)
4
5  int main( int argc, char** argv ) {
6     size_t i, j;
7     int* mat = (int*)malloc( DIM * DIM * sizeof( int ) );
8     for( i = 0; i < DIM; ++i )
9        for( j = 0; j < DIM; ++j )
10          mat[i*DIM+j] = 42;
11    puts("TEST␣DONE");
12    return 0;
13 }
```

Инициализация по столбцам:

Листинг 14.22: ./listings/examples/cache-matrix-init/matrix_init_ra.c

```
1  #include <stdio.h>
2  #include <malloc.h>
3  #define DIM (16*1024)
4
5  int main( int argc, char** argv ) {
6     size_t i, j;
7     int* mat = (int*)malloc( DIM * DIM * sizeof( int ) );
8     for( i = 0; i < DIM; ++i )
9        for( j = 0; j < DIM; ++j )
```

```
10            mat[j*DIM+i] = 42;
11        puts("TEST␣DONE");
12        return 0;
13    }
```

Тестируем время с помощью утилиты `time`:

```
> /usr/bin/time -v ./matrix_init_ra
        Command being timed: "./matrix_init_ra"
        User time (seconds): 2.40
        System time (seconds): 1.01
        Percent of CPU this job got: 86%
        Elapsed (wall clock) time (h:mm:ss or m:ss): 0:03.94
        Average shared text size (kbytes): 0
        Average unshared data size (kbytes): 0
        Average stack size (kbytes): 0
        Average total size (kbytes): 0
        Maximum resident set size (kbytes): 889808
        Average resident set size (kbytes): 0
        Major (requiring I/O) page faults: 2655
        Minor (reclaiming a frame) page faults: 275963
        Voluntary context switches: 2694
        Involuntary context switches: 548
        Swaps: 0
        File system inputs: 132368
        File system outputs: 0
        Socket messages sent: 0
        Socket messages received: 0
        Signals delivered: 0
        Page size (bytes): 4096
        Exit status: 0


> /usr/bin/time -v ./matrix_init_linear

        Command being timed: "./matrix_init_linear"
        User time (seconds): 0.12
        System time (seconds): 0.83
        Percent of CPU this job got: 92%
```

```
Elapsed (wall clock) time (h:mm:ss or m:ss): 0:01.04
Average shared text size (kbytes): 0
Average unshared data size (kbytes): 0
Average stack size (kbytes): 0
Average total size (kbytes): 0
Maximum resident set size (kbytes): 900280
Average resident set size (kbytes): 0
Major (requiring I/O) page faults: 4
Minor (reclaiming a frame) page faults: 262222
Voluntary context switches: 29
Involuntary context switches: 449
Swaps: 0
File system inputs: 176
File system outputs: 0
Socket messages sent: 0
Socket messages received: 0
Signals delivered: 0
Page size (bytes): 4096
Exit status: 0
```

Такая разница как раз обусловлена огромным количеством кэш-промахов, что можно проверить с помощью утилиты `valgrind` с модулем `cachegrind`:

```
> valgrind --tool=cachegrind ./matrix_init_ra

==17022== Command: ./matrix_init_ra
==17022==
--17022-- warning: L3 cache found, using its data for the LL simulation.
==17022==
==17022== I   refs:      268,623,230
==17022== I1  misses:            809
==17022== LLi misses:            804
==17022== I1  miss rate:        0.00%
==17022== LLi miss rate:        0.00%
==17022==
==17022== D   refs:       67,163,682  (40,974 rd   + 67,122,708 wr)
==17022== D1  misses:     67,111,793  ( 2,384 rd   + 67,109,409 wr)
```

```
==17022== LLd misses:     67,111,408  ( 2,034 rd   + 67,109,374 wr)
==17022== D1  miss rate:        99.9% (   5.8%     +      100.0%  )
==17022== LLd miss rate:        99.9% (   5.0%     +      100.0%  )
==17022==
==17022== LL refs:        67,112,602  ( 3,193 rd   + 67,109,409 wr)
==17022== LL misses:      67,112,212  ( 2,838 rd   + 67,109,374 wr)
==17022== LL miss rate:         20.0% (   0.0%     +      100.0%  )
```

I соответствует кэшу инструкций, D – кэшу данных, LL – **кэшу последнего уровня (last level cache)**.

Видно, что в кэше данных количество промахов почти 100%, что очень плохо.

Напротив, при последовательной записи количество промахов радикально уменьшилось:

```
==17023== Command: ./matrix_init_linear
==17023==
--17023-- warning: L3 cache found, using its data for the LL simulation.
==17023==
==17023== I   refs:      336,117,093
==17023== I1  misses:            813
==17023== LLi misses:            808
==17023== I1  miss rate:        0.00%
==17023== LLi miss rate:        0.00%
==17023==
==17023== D   refs:       67,163,675  (40,970 rd   + 67,122,705 wr)
==17023== D1  misses:     16,780,146  ( 2,384 rd   + 16,777,762 wr)
==17023== LLd misses:     16,779,760  ( 2,033 rd   + 16,777,727 wr)
==17023== D1  miss rate:        25.0% (   5.8%     +       25.0%  )
==17023== LLd miss rate:        25.0% (   5.0%     +       25.0%  )
==17023==
==17023== LL refs:        16,780,959  ( 3,197 rd   + 16,777,762 wr)
==17023== LL misses:      16,780,568  ( 2,841 rd   + 16,777,727 wr)
==17023== LL miss rate:          4.2% (   0.0%     +       25.0%  )
```

Вопрос 61. *Обзорно изучите* `man gcc optimizations`

Часть IV

Приложение

Глава 15

Отладка в gdb

gdb – мощный инструмент отладки. Мы лишь дадим обзорную справку по нему, которая поможет сделать первые шаги.

Отладка это процесс поиска ошибок и изучения работы программы. В процессе отладки программа, обычно, выполняется по шагам. Также можно запустить программу с тем, чтобы она выполнялась пока не произойдёт доступа к какому-то адресу кода или данных, в этот момент мы остановимся. Каждая остановка означает, что мы можем посмотреть на состояние программы, а именно содержимое памяти и регистров.

15.1 Отладка на уровне ассемблера

Для примера рассмотрим уже знакомую программу:

Листинг 15.1: ./listings/print_rax.asm

```
1  section .data
2  codes:
3     db    '0123456789ABCDEF'
4
5  section .text
6  global _start
7  _start:
8     ; число 1122... в 16-ричной системе счисления
9     mov rax, 0x1122334455667788
10
```

```
11      mov rdi, 1
12      mov rdx, 1
13      mov rcx, 64
14      ; Каждые 4 бита нужно вывести на экран как одну 16-ричную цифру.
15      ; Для этого с помощью сдвига и побитового И с маской 0xf выделим
16      ; каждую тетраду, и используем её как смещение относительно
17      ; метки codes
18    .loop:
19      push rax
20      sub rcx, 4
21      ; cl это однобайтовый регистр, часть rcx
22      ; rax -- eax -- ax -- ah + al
23      ; rcx -- ecx -- cx -- ch + cl
24      sar rax, cl
25      and rax, 0xf
26      lea rsi, [codes + rax]
27      mov rax, 1
28
29      ; при выполнении syscall регистр rcx не сохраняется
30      push rcx
31      syscall
32      pop rcx
33
34      pop rax
35      ; проверка на 0 быстрее с помощью команды test, нежели cmp
36      test rcx, rcx
37      jnz .loop
38
39      mov   rax, 60       ; для вызова close
40      xor   rdi, rdi
41      syscall
```

Скомпилируем её в файл `print_rax` и запустим отладчик:

```
> nasm -o print_rax.o -f elf64 print_rax.asm
> ld -o print_rax print_rax.o
> gdb print_rax
...
(gdb)
```

Общение с `gdb` осуществляется через его собственную систему команд. Загрузить исполняемый файл для отладки можно передав его аргументом самому `gdb` или через команду `file` внутри него.

```
(gdb) file print_rax
Reading symbols from print_rax...(no debugging symbols found)...done.
```

В командной строке `gdb` действует клавиша `tab`, выдающая подсказки команд на основании уже введённых символов.

- `quit` выходит из отладчика;
- `help` выводит справку, `help cmd` выводит справку по команде.

Файл `/.gdbinit` хранит команды, которые будут исполняться автоматически как только запускается `gdb`. Дополнительный файл настроек может быть и в текущей директории, где находится ваш исполняемый файл, но чтобы он тоже выполнялся нужно включить его загрузку в `/.gdbinit`.

Вопрос 62. *Попытайтесь создать `.gdbinit` в текущей директории (она не должна быть домашней) и запустить `gdb`. Прочитайте сообщение от `gdb`. Как включить загрузку `.gdbinit` из текущей директории?*

Почему нас интересует файл `.gdbinit`? Мы будем отлаживать код на ассемблере. У ассемблера, используемого нами, есть два синтаксиса: AT&T, который используется `gdb` по умолчанию, и Intel, который включается командой:

```
(gdb) set disassembly-flavor intel
```

Эту команду разумно добавить в `.gdbinit`.

- `run` запускает программу;
- `break название_метки` создаст точку останова (breakpoint). Можно и по адресу: `break *адрес`. При выполнении команды `run` или `continue` мы остановимся, если наткнемся на точку останова. В программе их может быть много.
- `continue` возобновляет выполнение программы после остановки.
- `stepi` продвинет программу на одну инструкцию;
- `ni` или `nexti` продвинет программу на одну инструкцию. Однако если мы должны будем вызвать процедуру, мы не зайдём внутрь с остановкой, а остановимся только после её выполнения (или если наткнёмся на точку останова).

Выполним следующие команды:

```
(gdb) break _start
Breakpoint 1 at 0x4000b0
(gdb) start
Function "main" not defined.
Make breakpoint pending on future shared library load? (y or [n]) n
Starting program: /home/stud/test/print_rax

Breakpoint 1, 0x00000000004000b0 in _start ()
```

Мы остановились на breakpoint, который сами и создали. Переключимся в более удобный вид с помощью команд `layout asm` и `layout regs`.

- Стрелки позволяют прокручивать вверх и вниз текущее окно. С помощью последовательного нажатия Ctrl-`x` и `o` можно переключаться между окнами.
- print позволяет печатать регистры. Имена регистров предваряются символом $, например, `$rax`.
- `x` используется для печати содержимого памяти.

Вопрос 63. *Изучите команду `x` с помощью команды `help`.*

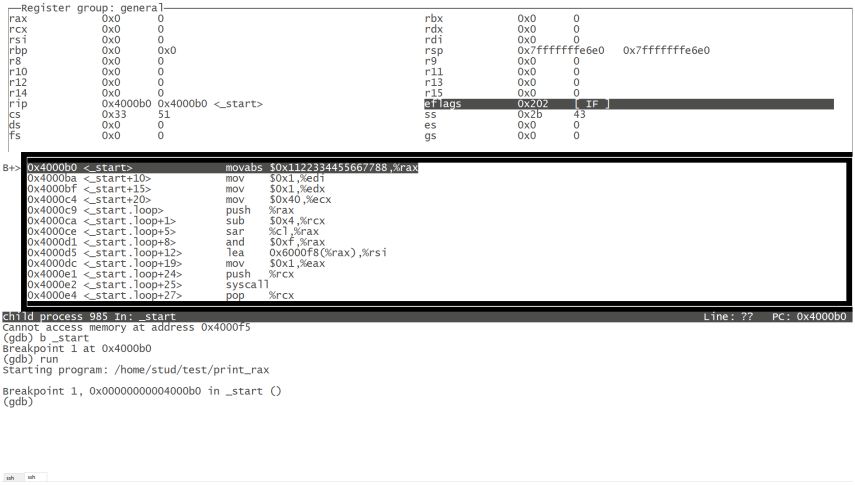


Рис. 15.1: `gdb` и псевдографика

Глава 16

Сборка `make`

Когда для получения какого-то конечного артефакта необходимо произвести много действий, некоторые из которых взаимосвязаны, логично подумать об автоматизации этого процесса. Для этого существуют разные системы сборки, одной из которых – `make` – мы будем пользоваться.

Действия, которые должен совершить `make`, описываются в специальном файле. Если вы просто запустите `make`, он попытается найти файл с именем `Makefile` в текущей директории и выполнить его. С помощью ключа `-f` можно указать какой-то конкретный файл для выполнения.

Содержимое `Makefile` можно описать так:

```
цель: зависимость1 зависимость2
```

Глава 17

Некоторые системные вызовы

Мы приведем точные коды для флагов и т.д. В программах на C никогда не используйте точные коды, только соответствующие `#define`. Они собраны в соответствующих заголовочных файлах.

sys_open

```
1  #include <sys/types.h>
2  #include <sys/stat.h>
3  #include <fcntl.h>
```

Открывает файл с заданным именем. Имя – нуль-терминированная строка.

rax	rdi	rsi	rdx	r10	r8	r9
2	const char* filename	int flags	int mode			

Флаги:

- O_APPEND
- O_ASYNC
- O_CLOEXEC
- O_CREAT

Глава 18

Информация о тестах производительности

Все тесты производились на следующей системе:

```
> uname -a

Linux perseus 3.16-2-amd64 #1 SMP Debian 3.16.3-2 (2014-09-20) x86_64 GNU/Lin

> cat /proc/cpuinfo

processor       : 0
vendor_id       : GenuineIntel
cpu family      : 6
model           : 69
model name      : Intel(R) Core(TM) i5-4210U CPU @ 1.70GHz
stepping        : 1
microcode       : 0x1d
cpu MHz         : 2394.458
cache size      : 3072 KB
physical id     : 0
siblings        : 1
core id         : 0
cpu cores       : 1
apicid          : 0
```

```
initial apicid  : 0
fpu             : yes
fpu_exception   : yes
cpuid level     : 13
wp              : yes
flags           : fpu vme de pse tsc msr pae mce cx8 apic
sep mtrr pge mca cmov pat pse36 clflush dts mmx fxsr sse
sse2 ss syscall nx pdpe1gb rdtscp lm constant_tsc arch_perfmon
pebs bts nopl xtopology tsc_reliable nonstop_tsc aperfmperf
pni pclmulqdq ssse3 fma cx16 pcid sse4_1 sse4_2 x2apic movbe
popcnt aes xsave avx f16c rdrand hypervisor lahf_lm ida arat
epb pln pts dtherm fsgsbase smep
bogomips        : 4788.91
clflush size    : 64
cache_alignment : 64
address sizes   : 40 bits physical, 48 bits virtual
power management:

processor       : 1
vendor_id       : GenuineIntel
cpu family      : 6
model           : 69
model name      : Intel(R) Core(TM) i5-4210U CPU @ 1.70GHz
stepping        : 1
microcode       : 0x1d
cpu MHz         : 2394.458
cache size      : 3072 KB
physical id     : 2
siblings        : 1
core id         : 0
cpu cores       : 1
apicid          : 2
initial apicid  : 2
fpu             : yes
fpu_exception   : yes
cpuid level     : 13
wp              : yes
flags           : fpu vme de pse tsc msr pae mce cx8 apic
sep mtrr pge mca cmov pat pse36 clflush dts mmx fxsr sse
sse2 ss syscall nx pdpe1gb rdtscp lm constant_tsc arch_perfmon
pebs bts nopl xtopology tsc_reliable nonstop_tsc aperfmperf
```

```
pni pclmulqdq ssse3 fma cx16 pcid sse4_1 sse4_2 x2apic movbe
popcnt aes xsave avx f16c rdrand hypervisor lahf_lm ida arat
epb pln pts dtherm fsgsbase smep
bogomips        : 4788.91
clflush size    : 64
cache_alignment : 64
address sizes   : 40 bits physical, 48 bits virtual
power management:

> cat /proc/meminfo

MemTotal:        1017348 kB
MemFree:          516672 kB
MemAvailable:     565600 kB
Buffers:           32756 kB
Cached:           114944 kB
SwapCached:        10044 kB
Active:           376288 kB
Inactive:          49624 kB
Active(anon):     266428 kB
Inactive(anon):    12440 kB
Active(file):     109860 kB
Inactive(file):    37184 kB
Unevictable:           0 kB
Mlocked:               0 kB
SwapTotal:        901116 kB
SwapFree:         868356 kB
Dirty:                44 kB
Writeback:             0 kB
AnonPages:        270964 kB
Mapped:            43852 kB
Shmem:               648 kB
Slab:              45980 kB
SReclaimable:      29016 kB
SUnreclaim:        16964 kB
KernelStack:        4192 kB
PageTables:         6100 kB
NFS_Unstable:          0 kB
Bounce:                0 kB
WritebackTmp:          0 kB
CommitLimit:     1409788 kB
```

```
Committed_AS:    1212356 kB
VmallocTotal:   34359738367 kB
VmallocUsed:      145144 kB
VmallocChunk:   34359590172 kB
HardwareCorrupted:     0 kB
AnonHugePages:         0 kB
HugePages_Total:       0
HugePages_Free:        0
HugePages_Rsvd:        0
HugePages_Surp:        0
Hugepagesize:       2048 kB
DirectMap4k:       49024 kB
DirectMap2M:      999424 kB
DirectMap1G:           0 kB
```

Литература

[1] What is mapreduce? hadoop documentation. https://www-01.ibm.com/software/data/infosphere/hadoop/mapreduce/.

[2] Jeff Andrews. Branch and loop reorganization to prevent mispredicts. https://software.intel.com/en-us/articles/branch-and-loop-reorganization-to-prevent-mispredicts, May 2011.

[3] C language standard – committee draft, 2007.

[4] Luca Cardelli and Peter Wegner. On understanding types, data abstraction, and polymorphism. *ACM Comput. Surv.*, 17(4):471–523, December 1985.

[5] Ryan A. Chapman. Таблица системных вызовов linux 3.2.0-33, x86_64. https://www.cs.utexas.edu/~bismith/test/syscalls/syscalls64_orig.html.

[6] Agner Fog. *The microarchitecture of Intel, AMD and VIA CPUs: An optimization guide for assembly programmers and compiler makers*, 2016.

[7] Intel Corporation. Intel® 64 and IA-32 Architectures Software Developer's Manual. http://www.intel.com/content/dam/www/public/us/en/documents/manuals/64-ia-32-architectures-software-developer-manual-325462.pdf, September 2014.

[8] David Kanter. Intel's haswell cpu microarchitecture. http://www.realworldtech.com/haswell-cpu/1.

[9] Petter Larsson and Eric Palmer. Image processing acceleration techniques using intel streaming simd extensions and intel advanced vector extensions. January 2010.

[10] Doug Lea. A memory allocator.

[11] Michael E. Lee. Optimization of computer programs in c. `http://leto.net/docs/C-optimization.php`.

[12] Chris Lomont. Fast inverse square root. February 2003.

[13] Michael Matz, Jan Hubicka, Andreas Jaeger, and Mark Mitchell. *System V Application Binary Interface. AMD64 Architecture Processor Supplement. Draft version 0.99.6*, 2013.

[14] Pawell Moll. How do debuggers (really) work? In *Embedded Linux Conference Europe*, October 2015. `http://events.linuxfoundation.org/sites/events/files/slides/slides_16.pdf`.

[15] The netwide assembler: Nasm. manual. `http://www.nasm.us/doc/`.

[16] John Regehr. A guide to undefined behavior in c and c++, part 1. `http://blog.regehr.org/archives/213`, July 2010.

[17] Юреш Вахалия. *Unix изнутри*. Питер, 2003.

[18] Н.Н. Непейвода and И.Н. Скопин. *Основания программирования*. РХД, 2003.