C - Язык Программирования

C - Язык Программирования

Содержание

        * 4.3.1
        * 4.3.2
        * 4.3.3
        * 4.3.4
        * 4.3.5
        * 4.3.6
        * 4.3.7
        * 4.3.8
- 5. Assembly
    - 5.1. Регистры, базовые инструкции
        * 5.1.1
        * 5.1.2
        * 5.1.3
        * 5.1.4
        * 5.1.5
        * 5.1.6
        * 5.1.7
        * 5.1.8
        * 5.1.9
        * 5.1.10
        * 5.1.11
        * 5.1.12
        * 5.1.13
        * 5.1.14
    - 5.2. Application Binary Interface
        * 5.2.1
        * 5.2.2
        * 5.2.3
        * 5.2.4
        * 5.2.5
        * 5.2.6
        * 5.2.7
        * 5.2.8
        * 5.2.9
        * 5.2.10
        * 5.2.11
    - 5.3. Компоновка
        * 5.3.1
        * 5.3.2
        * 5.3.3
        * 5.3.4
        * 5.3.5
        * 5.3.6
        * 5.3.7
        * 5.3.8
        * 5.3.9

1. Введение

1.1. Об этом курсе

1.1.1

Здравствуй, уважаемый читатель. Добро пожаловать на курс по языку программирования C. Однако, скорее всего, если ты ещё не погружен в эту тему, этот курс окажется намного глубже, чем ты сейчас ожидаешь. Здесь мы не ограничимся только такими вещами, как типы данных, ветвления, функции и указатели, а лишь начнём с этого.

Попробуем подумать над такой идеей — а зачем нужен язык C? При поверхностном взгляде на него кажется, что у него нет таких сторон, с которых его бы не обходил какой-либо другой язык. Когда мы должны его использовать? Ответ — когда мы не можем развернуть среду для других языков. Язык C очень независимый. Полностью регулировать зависимости програм, написанных на нём, очень просто. То же самое касается и структуры результирующих программ. А это означает, что изучать язык C без понимания о том, как управлять процессом компиляции, не имеет смысла.

Немного о том, как следует работать с этим курсом. Если вы уже можете компилировать и запускать программы на C на какой-либо системе — можете сразу переходить к модулю Вид сверху на C, минуя пункт 1.2. Даже если вы уже знаете заявленный там контент, всё равно есть смысл пролистать текст, так как в нём есть полезные идеи. После завершения изучения этого модуля читайте пункт 1.2, который расскажет о том, как получить необходимую для нашей работы среду, и переходите к остальным модулям.

Мне было лень писать большое количество текста, поэтому информация в тексте очень концентрированная. Убедитесь, что вы понимаете каждую идею, которую я даю в тексте. То же самое касается и заданий — обычно это не просто вопросы к тексту параграфа, а задачи, требующие изучения и обдумывания какой-либо идеи. Иногда в качестве заданий будут даваться

проекты, которые дают большую свободу действий.

1.2. О чём нужно знать

1.2.1

Компилятор для C

На протяжении первого модуля допустимо использовать любые компиляторы и любые IDE, а также online компиляторы, например, godbolt (обратите внимание на количество доступных компиляторов там).

Для удобства мы будем работать с операционной системой Linux, так как она имеет более простой и сильный shell, более простое и понятное внутреннее устройство, которое мы также будем изучать, и более удобные необходимые нам инструменты.

Важно определить и понять, какие компиляторы мы будем использовать. Хоть и с верхней стороны язык C в них идентичен, с нижней стороны они могут сильно отличаться. Я выбрал компилятор gcc. Это стандартный компилятор в системах Linux, и в большинстве дистрибутивов он установлен изначально.

На Windows имеется компилятор mingw, который заявлен, как порт gcc на Windows. Его можно установить, например, вместе с IDE Codeblocks, а затем добавить исполняемый файл в переменную PATH.

Идея курса в "сквозном" изучении компилятора gcc, поэтому разрешается пользоваться IDE только в течении первого модуля. Далее мы будем запускать gcc только с помощью терминала. До начала второго модуля убедитесь, что gcc доступен в терминале.

Существуют альтернативные компиляторы, которые имеет смысл изучить, но которые не будут покрыты данным курсом.

- MSVC. О данном компиляторе мне известно мало. Он предназначен только для Windows, и с его установкой могут быть проблемы.
- Clang. Данный компилятор имеет другой "путь" компиляции, в процессе которого используется язык llvm. Предназначен для обеих ОС.
- Zig. Помимо того, что zig является отдельным языком, его компилятор является хорошей альтернативой другим компиляторам языка C.

1.2.2

Система

В случае, если ваша основная операционная система — Windows, прочитайте этот раздел.

Все дистрибутивы Linux без проблем позволяют поставить на один диск (HDD/SSD) несколько ОС Linux. В таком случае при запуске компьютера

будет загружаться сначала загрузчик (обычно, grub), который будет спрашивать у вас, какую ОС на этот раз загрузить. Однако установка к Linux-у на диск ОС Windows, насколько мне известно, является нестабильным способом, так как Windows иногда затирает загрузчик Linux-а.

Возможные способы:

- Если у вас имеется дополнительный диск (HDD/SSD), можете поставить на него вторую ОС. При такой установке Windows не будет трогать другой диск (а в некоторых вариантах установки даже и не сможет его видеть). Так как вы ставите Linux к уже рабочей Windows, либо выберите дистрибутив с хорошим GUI (например, Ubuntu), либо попросите помощи, чтобы случайно не затереть не тот диск.
- Если на вашем компьютере есть хотя бы 8 Gb RAM (оперативной памяти), вы можете поставить виртуальную машину. Я рекомендую программу VirtualBox. В ней выделите виртуальной машине примерно половину своей оперативной памяти, не менее двух процессорных ядер и не менее 30 Gb дисковой памяти (если вы знаете свой дистрибутив, этот размер может быть намного меньше). Лучше поставьте легковесный DE (desktop environment), например, XFCE.
- WSL, однако я им никогда не пользовался и ничего о нём не знаю.

Что не рекомендуется:

- Ставить Linux на один диск с Windows (но может, кто-то сможет убедить меня в обратном)
- Использовать Cygwin. У этого способа виртуализации большие проблемы с пакетами.
- Использовать не аппаратную виртуализацию (например, QEMU имеет аппаратную виртуализацию только на Linux).

Возможные проблемы с VirtualBox:

- Зависания машины намертво. Это происходит из-за того, как VirtualBox эмулирует диск. Вы можете заметить, что на гостевой ОС заканчивается Swap, а на хостовой ОС потребление диска возрастает до 100%. К сожалению, это не лечится.
- Ошибки запуска виртуальной ОС после неудачного завершения работы (например, вхождения в гибернацию). Оставьте файл диска (его расширение — .vdi), а остальное удалите. Создайте новую машину и добавьте в неё этот файл диска.
- Медленная работа. Это означает, что VirtualBox перешел в программную виртуализацию (в этом случае внизу-справа показывается иконка черепахи, вместо иконки чипа). Это может произойти из-за того, что вы поставили Hyper-V и он занял гипервизор процессора. Удалите Hyper-V.

2. Вид сверху на C

2.1. Введение

2.1.1

Рассмотрим простейшую однофайловую программу на C:

```
int main() {
    return 0;
}
```

Здесь определена функция под названием main. int main() — это сигнатура функции. По ней мы видим, что функция не принимает аргументов, так как круглые скобки ничего не содержат, и возвращает тип int — целое число. В фигурных скобках содержится тело функции, которое содержит лишь один statement return 0, который завершает выполнение функции, возвращая значение 0. Пока можно сильно не задумываться обо всем вышенаписанном, мы вернемся к функциям с большими подробностями позже.

2.1.2

Сложно анализировать выполнение программы не имея возможности ввода и вывода. Научимся выводить. Первой функцией для вывода будет int puts(const char *str). Давайте изучим её сигнатуру.

Во-первых, функция принимает один аргумент типа const char *str. Пока не будем расшифровывать этот сложный тип, и поверим, что это строка, известная на этапе компиляции (то есть, мы не сможем её сформировать интерактивно). Очевидно эта строка будет напечатана. Позже мы узнаем, что такое потоки streams, пока будем считать, что вывод идет в консоль.

Во-вторых, функция возвращает тип int, и, если мы посмотрим в документацию cppreference, мы увидим, что то, что она возвращает, описано с большой свободой.

> On success, returns a non-negative value.
>
> On failure, returns EOF and sets the error indicator (see ferror()) on stream.

В свою очередь:

EOF	integer constant expression of type int and negative value

Такую картину мы будем видеть во многих функциях. Можно посмотреть и ferror(), но нам это пока неинтересно.

Давайте, наконец, напечатаем что-либо.

```
int main() {
    puts("Hello");
    puts("Test");
    return 0;
}
```

Обратите внимание: в этой программе я не написал #include <stdio.h>, чтобы добавить файл stdio.h, несмотря на то, что так обычно делают. Дело в том, что мы пока не понимаем, что такое #include, но он и не нужен. Вы получите предупреждение о том, что putsне был объявлен, но исполняемый файл вы все равно получите.

При запуске этой программы будет выполнена функция main, в которой будет вызвана функция puts("Hello"), в результате чего будет выполнена печать строки Hello и перевод строки, а затем будет вызвана функция puts("Test"), в результате чего будет выполнена печать строки Test и перевод строки.

2.1.3

Научимся работать с локальными переменными, а заодно и выводить числа. Рассмотрим программу:

```
int main() {
    int a;
    int b = 2 + 3;
    b = a + 8;
    printf("%d %s %d\n", a, "Hello", b + 7);
    return 0;
}
```

В первой строке тела функции main мы объявляем declare переменную типа int с именем a. Мы не присваиваем define её. Её значение на этом моменте будет не определено. Какое значение получит переменная a мы узнаем позже.

Во второй строке мы объявляем declare и сразу определяем define переменную типа int с именем b. Мы присваиваем ей значение 5.

В третьей строке мы переопределяем значение переменной b. Так как при вычислении её значения используется неопределённая переменная a, теперь значение переменной b тоже не определено.

В четвертой строке вы используем новую, намного более сложную функцию — printf. Это функция с неопределённым количеством аргументов (variadic functions, vararg, ellipsis). Первый её аргумент — это строка, задающая формат. Он имеет достаточно богатые возможности, и мы будем ими пользоваться по мере необходимости. Пока остановимся на том, что подстроки %d будут заменены на число в следующем по счету аргументе, подстроки %s будут заменены на строку в следующем по счету аргументе (первый

%d заменится на аргумент a, %s заменится на аргумент "Hello", второй %d заменится на аргумент b + 7), а \n обозначает символ перехода строки line feed (другие подобные символы мы увидим позже).

Определить вывод этой программы наперед не получится, так как значения переменных зависят от компилятора. В будущем мы часто будем с этим сталкиваться.

2.1.4

Рассмотрим базовые типы данных, которые нам в ближайшее время могут пригодиться.

- int — целые числа, на хранение которых выделяется 4 байта
- short — целые числа, на хранение которых выделяется 2 байта
- char — целые числа, на хранение которых выделяется 1 байт
- long long — целые числа, на хранение которых выделяется 8 байт
- long — целые числа, на хранение которых выделяется ?? байт

Помимо этих типов, нам понадобится const char*.

Интересная ситуация с логическим типом _Bool — он был введён не сразу, и в многих программах вводили тип с названием bool самостоятельно с помощью препроцессора, поэтому и было выбрано такое странное название. Логический тип имеет лишь два значения: true и false. (На самом деле, размер этого типа 1 байт, и false соответствует числу 0, в то время как true всем остальным.)

Размер типа long в свою очередь зависит от системы. На Windows он равен 4, а на Linux он равен 8. В целом, встроенные типы не описывают нормально свою размерность, как в языках Rust и Zig, и это бывает проблемой.

Про указатели мы поговорим позже.

С помощью оператора sizeof(x) можно узнать размер типа в байтах, причём в качестве аргумента можно использовать как тип, так и переменную или значение.

```
int main() {
    printf("int: %d\n", sizeof(int));
    printf("short: %d\n", sizeof(short));
    printf("char: %d\n", sizeof(char));
    printf("long long: %d\n", sizeof(long long));
    printf("long: %d\n", sizeof(long));
    printf("const char*: %d\n", sizeof(const char*));

    const char *str = "Hello";
    printf("const char*: %d\n", sizeof(str));
    return 0;
}
```

2.1.5

Изучим несколько базовых операторов.

1. + — оператор сложения, складывает два числа. 2 + 3 равно 5.
2. - — оператор вычитания, вычитает второе число из первого. 5 - 3 равно 2.
3. * — оператор умножения, умножает два числа. 2 * 3 равно 6.
4. / — оператор деления, делит первое число на второе с округлением вниз. 5 / 2 равно 2. 6 / 2 равно 3.
5. % — оператор остатка от деления, делит первое число на второе и даёт остаток от деления. 5 % 2 равно 1. 6 % 2 равно 0.

Если второй аргумент у операторов / и % будет равен 0, то программа завершится с ошибкой. Какие при этом механизмы происходят, мы узнаем позже.

Также у этих операторов есть странная особенность: на самом деле они округляют не вниз, а к нулю, что может быть неприятным сюрпризом, если в ваших промежуточных вычислениях будут получаться отрицательные числа. Например -5 / 2 равно -1 и -5 % 3 равно -2.

Имейте ввиду, что значения чисел в C ограничены, и при их переполнении вы не получите никаких предупреждений или ошибок. (Когда такое происходит, процессор ставит флаг переполнения, но компилятор C для скорости даже не смотрит на него.) Например, 2000000000 + 2000000000 равно -294967296.

Помимо арифметических операторов существуют логические операторы для создания сложных условий.

1. && или and — оператор И, который возвращает 1, если оба аргумента не равны нулю, и возвращает 0 в противном случае.
2. || или or — оператор ИЛИ, который возвращает 1, если хотя бы один аргумент не равен нулю, и возвращает 0 в противном случае.
3. ! — унарный оператор НЕ, который возвращает 1, если аргумент равен нулю, и возвращает 0 в противном случае.
4. == — оператор равенства, который возвращает 1, если аргументы равны, и возвращает 0 в противном случае.

```
int main() {
    printf("%d ", 3 + 4);
    printf("%d ", 17 % 7);
    printf("%d ", 3 && 0);
    printf("%d ", 3 || 0);
    printf("%d ", !4);
    printf("%d ", 5 == 5);
    return 0;
}
```

Вывод этой программы: 7 3 0 1 0 1

2.1.6

Напишите программу, которая выведет следующий текст:

123
abracadabra
2 + 5 = 7
ababba

2.1.7

Напомним, что в C не сразу был введён логический тип данных _Bool. Однако при этом также есть возможность получить "тип" bool, добавив файл #include <stdbool.h>. (На самом деле, это не совсем тип.)

Как думаете, зачем этот файл был добавлен?

2.1.8

Пока эту задачу можно пропустить, и вернуться к ней намного позднее.

Я немного наврал, когда сказал, что размер типа int — 4 байта. Это не так при компиляции в разрядности ниже, чем 32 bit. Однако современные компиляторы не имеют возможности компилировать в такие разрядности, поэтому это не является проблемой. Можно попробовать найти древний компилятор (например, Open Watcom), который может компилировать в разрядность 16 bit, и проверить sizeof(int) в таком режиме.

2.2. Контроль потока

2.2.1

Начнём с введения нескольких терминов. Рассмотрим программу:

```
int main() {
    int a = 5 + 8;
    puts("123");
    int b = puts("456");
    return 0;
}
```

Тело функции main состоит из четырёх действий: объявление переменной a, вызов функции puts("123"), объявление переменной b, возвращение значения 0. Каждое из этих действий называется statement.

Посмотрим на первый statement: int a = 5 + 8. Часть 5 + 8 называется expression. Часть этого expression-а 5 тоже является expression-ом. В третьем statement-е вызов функции puts("456") также является expression-ом (в то время, как другой вызов этой же функции был statement-ом).

Есть проблема в терминологии: оба этих слова не имеют перевода на русский язык, и обычно их называют одним словом выражение. Позже, когда мы будем более подробно изучать синтаксис, мы увидим смысл такого разделения.

statement	expression
Исполняются сверху вниз	Порядок исполнения не определён
Не имеют конечного значения	Имеют конечное значение (а потому и тип)
Имеют side effect	По хорошему, не должны иметь side effect

Side effect означает, что исполнение фрагмента кода приведет к изменению чего либо за его пределами (изменятся внешние переменные; что-то запишется в память, доступную из внешних переменных; выполнится системный вызов и т.д.).

2.2.2

Control Flow (контроль потока) — это набор statement-ов в языке, которые определяют последовательность выполнения statement-ов.

Пока мы видели следующие типы statement-ов:

- Declaration (объявление): int a. Объявление также может быть с присваиванием/определением (по сути, это то же самое, но термин определение в других языках может налагать ограничения): int a = 5.
- Assignment/Definition (присваивание/определение): a = 5 + 2.
- Function Call (вызов функции): puts("123").
- Function Return (возврат из функции): return 2 + puts("456").

Конечно, хочется иметь вариативность в порядке исполнения statement-ов. Делать это можно с помощью ветвлений и циклов, которые являются control flow statements.

Посмотрим на if:

```
int main() {
    int a = 4;
    if (a - 4) {
        puts("1");
    }
    else if (a) puts("2");
    else {
        puts("3");
    }
    return 0;
}
```

Данный if statement состоит из трех ветвей. Эти ветви надо читать так:

1. Если a - 4 не 0, то выполнить puts("1") и завершить выполнение if
2. Если a не 0, то выполнить puts("2") и завершить выполнение if
3. Выполнить puts("3")

Обратите внимание на следующие вещи:

- Можно как обрамлять блок в if в фигурные скобки, так и не делать этого.
- В качестве условия в if должен быть expression, а его тип неважен. Проверка идет лишь на то, что он не 0. А в языке C по сути любой тип данных — число.
- Веток else if и else может не быть. Количество веток else if не ограничено.

В результате выполнения этой программы будет напечатано единственное число 2.

2.2.3

Вскоре нам потребуется начать решать input/output задачи для проверки знаний. Для этого нужно разобраться, как считывать данные. Для этого используется функция scanf. Посмотрим решение задачи A + B.

Напишите программу, которая считывает два целых числа a и b ($-100 \leq a, b \leq 100$), и выводит их сумму.

```
int main() {
    int a, b;
    scanf("%d%d", &a, &b);
    printf("%d\n", a + b);
    return 0;
}
```

Функция scanf имеет первым аргументом формат, по которому сопоставляются данные во входном потоке, а следующими аргументами адреса, по которым необходимо эти данные положить. scanf сам считает произвольное количество пробельных символов (пробел, табуляция, следующая строка, возврат каретки и т.д.) между идущими в формате подряд %d. (Принцип, по которому он это делает, мне не очень понятен, но это и не нужно.)

Пока мы не знаем, что такое адреса, но позже мы подробно их изучим. C не имеет возможности передать в функцию локальную переменную так, чтобы функция изменила её. Вместо этого мы передаем адрес локальной переменной, который мы берем оператором & (амперсанд). Интересно, что функция, получая информацию о местоположении локальной переменной, может работать не только с этой переменной, но и с тем что находится рядом с ней. (Она не получает этим права — права у неё от передачи аргументов не меняются. Она получает информацию, где искать данные.) Но об этом мы поговорим подробно потом.

Нетрудно догадаться, как считывать строки, но там есть неочевидные моменты. Мы поговорим о строках отдельно. Пример некорректной программы, которая может как выполниться успешно, так и завершиться с ошибкой. Ничего страшного, если вы пока не знаете, что такое массивы, и, соответственно, не понимаете программу.

```
int main() {
    char str[10];
    scanf("%s", str);
    return 0;
}
```

А эта программа гарантированно завершится с ошибкой, так как попытается считать данные в константу. Как функционирует такое ограничение, мы также позже узнаем.

```
int main() {
    const char *str = "Hello";
    scanf("%s", str);
    return 0;
}
```

2.2.4

Улитка ползет по столбу высотой h единиц и хочет забраться на самый его верх. Днем улитка поднимается вверх на a единиц, а ночью соскальзывает вниз на b единиц. На какой день улитка заберется на столб?

Формат ввода Единственная строка входных данных содержит три целых числа: a, b и h ($1 \leq b < a \leq h \leq 10^9$)

Формат вывода Выведите единственное число — номер дня, на который улитка достигнет верха столба.

2.2.5

В C присутствуют циклы while, do while и for. Выходить их циклов можно с помощью операторов break и continue. К сожалению, нет возможности выйти из нескольких циклов, кроме как с помощью оператора перехода к метке goto.

Цикл while выглядит так:

```
int main() {
    int i = 0;
    while (i < 10) {
        printf("%d", i);
        i++;
```

```
    }
    return 0;
}
```

Читать это надо так. Еслиi < 10, выполнить тело цикла, в котором необходимо вывести значение i и увеличить его на единицу, а затем прыгнуть в начало цикла. В противном случае, прервать выполнение цикла.

Вывод этой программы: 0123456789

Цикл do while выглядит так:

```
int main() {
    int i = 0;
    do {
        printf("%d", i);
        i++;
    } while (i < 10);
    return 0;
}
```

Читать это надо так. Выполнить тело цикла, в котором необходимо вывести значение i и увеличить его на единицу. Затем, еслиi < 10, прыгнуть в начало цикла. В противном случае, прервать выполнение цикла.

Вывод этой программы: 0123456789

Цикл do while отличается от цикла while тем, что он в любом случае сделает хотя бы одну итерацию.

2.2.6

Цикл for выглядит так:

```
int main() {
    int i;
    for (i = 0; i < 10; i++) {
        printf("%d", i);
    }
    i = 0;
    for (; i < 10; printf("%d", i) && i++);
    return 0;
}
```

Читать первый цикл for надо так. Сначала выполнить i = 0. Затем выполнить тело цикла, в котором необходимо вывести значение i, а затем выполнить i++. Затем, если i < 10, прыгнуть в начало цикла (не выполняя i = 0). В противном случае, прервать выполнение цикла.

Данный цикл на первый взгляд может выглядеть странным и запутанным, но на самом деле он удобен.

Часто вы будете видеть, что в первом "блоке" цикла for (там, где у нас написано i = 0) выполняют объявление переменной (например int i = 0). Однако, это работает не со всеми стандартами языка C. В некоторых стандартах это может считаться ошибкой.

Посмотрим на второй цикл for. Мы видим, что некоторые блоки могут быть пустыми. Все блоки должны являться expression-ами. Причина этого ясна для второго блока, ведь он проверяет условие, а значит его содержимое должно иметь значение. Но неужели i = 0 и i++ это тоже expression-ы, которые имеют какое-то значение? Да. i = 0 возвращает значение своего правого аргумента, а i++ возвращает значение i до увеличения. (Чуть позже мы изучим все эти операторы подробнее, но вы уже можете почитать про них.)

Оператор && — это оператор И. Он выполняет левый аргумент, и, если он не 0, выполняет правый аргумент и возвращает его, а иначе, возвращает 0. Попробуйте теперь самостоятельно понять логику работы второго цикла.

Вывод этой программы: 01234567890123456789

2.2.7

Немного изменим второй цикл из программы на предыдущем шаге.

```
int main() {
    int i = 0;
    for (; i < 10; i++ && printf("%d", i));
    return 0;
}
```

Какой будет вывод у этой программы и почему?

2.2.8

С помощью оператора break мы выходим из одного внутреннего цикла. Выглядит это так:

```
int main() {
    int i, j;
    for (i = 0; i < 5; i++) {
        for (j = 0; j < 5; j++) {
            printf("%d", j);
            if (j == i) break;
        }
    }
    return 0;
}
```

Вывод этой программы: 001012012301234

С помощью оператора continue мы переходим к следующей итерации внутреннего цикла, не выполняя оставшуюся часть тела цикла. Выглядит это так:

```
int main() {
    int i, j;
    for (i = 0; i < 5; i++) {
        for (j = 0; j < 5; j++) {
            if (j == i) continue;
            printf("%d", j);
        }
    }
    return 0;
}
```

Вывод этой программы: 12340234013401240123

С помощью оператора goto мы переходим к метке. Метка может быть объявлена в любом месте между statement-ов, и выглядит как идентификатор с двоеточием. С помощью оператора goto можно, например, выйти из двойного цикла:

```
int main() {
    int i, j;
    for (i = 0; i < 5; i++) {
        for (j = 0; j < 5; j++) {
            printf("%d", j);
            if (j == i) goto end;
        }
    }
    end:
    return 0;
}
```

Вывод этой программы: 0

Бывает, что необходимо по выполнению условия выйти из блока кода. Увы, сделать это так не получится:

```
int main() {
    {
        int i;
        if (i) break;
    }
    return 0;
}
```

Обычно, для этой цели используют цикл или оператор goto. (В языке программирования Zig это организовано получше.)

Старайтесь не использовать оператор goto там, где он может серьезно усложнить читаемость программы, с чем он отлично справляется. Обычно в языке C он используется для того, чтобы выйти из вложенного цикла, и для того, чтобы перейти к очистке в конце функции, если она что-то динамически создала, но в процессе её работы произошла ошибка, из-за которой она должна прерваться и вернуть ошибку. Выглядит это примерно так:

```
FOOSTATUS foo() {
    void *data = malloc(SIZE);
    int status = FOO_SUCCESS;
    if (!init_foo(data)) {
        status = FOO_ERROR_INIT;
        goto end
    }
    if (!process_foo(data)) {
        status = FOO_ERROR_PROCESS;
        goto end
    }
    end:
    free(data);
    return status;
}
```

2.2.9

Для более удобной организации ветвлений используются switch case statement-ы.

```
int main() {
    int x = 2;
    switch (x) {
        case 1: printf("1 "); break;
        case 2: printf("2 "); break;
    }
    switch (x) {
        case 1: printf("1 "); break;
        case 3: printf("3 "); break;
    }
    switch (x) {
        case 1: printf("1 "); break;
        case 2: printf("2 ");
        case 3: printf("3 "); break;
        case 4: printf("4 ");
    }
    switch (x) {
        case 1: printf("1 "); break;
        default: printf("- ");
```

```
    }
    printf("\n");
}
```

Здесь написаны четыре switch case statement-а. В своем теле они имеют ветви, каждая из которых состоит из ключевого слова case, выражения, при котором эта ветвь срабатывает, символа двоеточия и блока кода.

При выполнении switch case statement-а выполняется первая ветвь, выражение после слова case у которой равно выражению после слова switch. У первого switch это вторая ветвь, которая содержит блок printf("%d "); break;. (Ключевое слово break тоже часть блока.) У второго switch такой ветви нет, поэтому ничего не будет исполнено.

Как только срабатывает одна из ветвей, выполняются все следующие ветви вплоть до последней (что по моему мнению является довольно странной логикой), либо до встречи ключевого слова break. Так, у третьего switch будет выполнен блок второй ветви printf("2 ");, а затем продолжится выполнение блока третьей ветви, и, так как он завершается словом break, блок четвертой ветви не будет выполнен.

Четвертый switch statement имеет ветвь, которая начинается с ключевого слова default. Блок данной ветви будет выполнен в любом случае, если не было выполнено ни одного блока выше.

Вывод этой программы: 2 2 3 -

2.2.10

Успешный брокер Василий смотрит курс акций за последние n дней. Цена акции в день i равна a_i. Василий хочет найти два дня $p < q$, такие, что $a_q - a_p$ максимально возможное, ведь именно покупка акций в день p, а продажа в день q принесла бы ему максимальную прибыль, будь у него более развитые экстрасенсорные способности.

Формат ввода Первая строка входных данных содержит одно целое число n ($1 \le n \le 10^3$)

Вторая строка входных данных содержит n целых чисел $a_1, a_2, \ldots, a_n$ ($1 \le a_i \le 10^6$)

Формат вывода Выведите единственное число — наибольшую разность $a_q - a_p$.

2.3. Функции

2.3.1

Мы уже объявляли функции. int main() {} — это объявление функции, которая ничего не принимает. При использовании стандартного linker-скрипта

(что это такое, мы узнаем позже) наличие этой функции обязательно.

Сигнатура функции — это список типов аргументов, которые функция принимает, и тип того, что функция возвращает. По умолчанию, функция main возвращает int, и значение 0 означает успешное выполнение. Это значение называется кодом возврата программы, и в sh можно его посмотреть у запущенной в последний раз программы с помощью переменной $?: echo $?.

Попробуем написать свои функции:

```
int foo(int a) {
    return a + 2;
}

int main() {
    int a = 5;
    int b = foo(a);
    foo(a);
    printf("%d %d\n", a, b);
    return 0;
}
```

Функция foo принимает один аргумент типа int и возвращает его значение, увеличенное на 2. В результате выполнения этой программы будет выведено 5 7

Обратите внимание, что вызов функции может быть как expression-ом, как в первом случае, так и statement-ом, как во втором случае.

Функции используются для организации кода и улучшения читаемости.

Функции можно определять в отдельных файлах, но в C (и Assembly) это делается не так просто, как в остальных языках. Мы поговорим об этом отдельно.

В функции foo можно вызвать функцию foo. Это называется рекурсией. Например, функция вычисления наибольшего общего делителя (greatest common divisor) может быть написана так:

```
int gcd(int a, int b) {
    if (a == 0)
        return 0;
    else
        return gcd(b % a, a);
}
```

Естественно, можно допустить ошибку, при которой вызов функции будет происходить бесконечно. В таком случае, так как на хранение данных о предыдущих (и все ещё идущих) вызовах функции тратится память на стеке, стек вскоре закончится, и возникнет ошибка исполнения.

Функции могут ничего не возвращать. Для этого вместо типа возврата указывается void. В таком случае, нельзя использовать функцию, как expression.

2.3.2

Принято, что функция main принимает два аргумента, и её полная сигнатура на самом деле выглядит так: int main(int argc, char **argv). Второй аргумент — список строк, длина которого неизвестна, а первый аргумент — длина этого списка. Рассмотрим программу:

```
int main(int argc, char **argv) {
    int i;
    for (i = 0; i < argc; i++) {
        printf("%s\n", argv[i]);
    }
    return 0;
}
```

Что за список строк имеется ввиду? Это список аргументов программы. Когда мы запускаем программу в терминале, мы передаем их так: ./program ababba babba. При таком запуске вывод программы будет такой:

```
program
ababba
babba
```

Первый аргумент всегда должен быть названием исполняемого файла. Мы можем нарушить это правило, если сделаем системный вызов exec самостоятельно, и некоторые программы в таком случае могут отказаться работать. (Что такое системные вызовы мы изучим позже.)

Благодаря особенностям ABI (что это такое, мы узнаем позже) мы можем написать в сигнатуру не все аргументы. Однако мы не можем написать большее количество аргументов, иначе они попадут на данные функции, которая нас вызвала, и мы начнем оперировать ими. (main — не первая функция при стандартном linker-скрипте.) Обратите внимание: компилятор никак не помешает нам этого сделать.

2.3.3

Функции "видят" другие функции, объявленные выше. Но если мы хотим написать две вызывающие друг друга функции, то с помощью только определений функций сделать этого, возможно, не удастся. Такой код в C++ некорректен, но в C корректен (возможно, не во всех компиляторах):

```
void foo(int n) {
    boo(1);
}
```

```
void boo(int n) {
    foo(2);
}
```

Решить эту проблему можно с помощью объявления функции, или, прототипа функции. Выглядит он так:

```
void boo(int n);

void foo(int n) {
    boo(1);
}

void boo(int n) {
    foo(2);
}
```

Прототип функции объявляет лишь сигнатуру. Названия аргументов в нем не важны, и, если мы заменим первую строку на void boo(int);, то это будет по прежнему корректно.

Прототип функции должен соответствовать определению функции. Если прототип здесь будет void boo(long long n);, то это вызовет ошибку компиляции.

Очень важно, что если прототип функции, и её определение будут в разных модулях, то компилятор не сможет заметить несоответствия сигнатур функций. В таком случае во время выполнения программы при вызове функции стек "съедет", и произойдет stack corruption. (Более подробно этот механизм мы изучим позже.)

2.3.4

Реализуйте следующие функции:

1. int min(int, int)
2. int max(int, int)
3. int abs(int) — абсолютное значение (модуль)
4. int lcm(int, int) — наименьшее общее кратное (least common multiplier)
5. int powmod(int a, int n, int m) — возведение числа a в степень n по модулю m

Подставьте следующую функцию main:

```
int main() {
    assert(min(5, 8) == 5);
    assert(min(11, 4) == 4);
    assert(max(5, 8) == 8);
    assert(max(11, 4) == 4);
    assert(abs(43) == 43);
```

```
    assert(abs(-41) == 41);
    assert(lcm(14, 21) == 42);
    assert(lcm(1128960, 567000) == 254016000);
    assert(powmod(3, 4, 7) == 4);
    assert(powmod(43259, 64234, 784) == 505);
    return 0;
}
```

Функция assert завершает выполнение программы с кодом возврата 3, если её аргумент равен нулю (то есть, условие провалено). Возможно, для его использования потребуется добавить #include <assert.h>.

2.3.5

Функции printf и scanf — это функции с переменным количеством аргументов (variadic functions, vararg, ellipsis). Синтаксис C позволяет создавать такие функции, но по понятным (когда придет время) причинам, сам C такие функции поддерживать не может. Проблема в том, что при вызове функции на стек кладутся аргументы и адрес возврата, но ничего более. Поэтому по стеку нельзя понять, сколько аргументов было передано в функцию, и какие их типы или, хотя бы, размерности. Это может вызвать различные corruptions.

Пример функции, вычисляющей сумму аргументов:

```
#include <stdarg.h>

int sum(int n, ...) {
    va_list lst;
    va_start(lst, n);

    int sum = 0;
    int i;
    for (i = 0; i < n; i++) {
        sum += va_arg(lst, int);
    }

    va_end(lst);
    return sum;
}

int main() {
    printf("%d\n", sum(5, 2, 6, 4, 3, 7));
    return 0;
}
```

Обратим внимание на следующее:

- Нам необходим #include <stdarg.h> для того, чтобы иметь макросы va_list, va_start, va_arg, va_end. (Что такое макросы, мы узнаем позже. Пока считайте, что это функции.)
- Первый аргумент n — это длина массива. Напомню, что функция никак не может знать форму того, что ей передано. Количество аргументов надо указать. Обратите внимание, что если подать в функцию не n следующих аргументов, то она никак это не проверит — это приведет к corruption.
- Чтобы пробежать по аргументам, мы пользуемся переменной типа va_list. Эту переменную мы передаем всем функциям, связанным с аргументами. По сути, это не более, чем итератор.
- (Второй аргумент макроса va_start оказался мне неизвестен. Документация утверждает, что это аргумент, после которого начнут перебираться "переменные" аргументы, однако простой тест показал, что это не так.)
- Второй аргумент макроса va_arg это тип (обратите внимание: если бы это была функция, он не могла бы принимать тип), следующего аргумента (а также, его размер).

Вывод этой программы: 22

Функции с переменными аргументами следует использовать ограниченно и с большой осторожностью.

2.3.6

Реализуйте следующие функции:

1. int min(int n, ...)
2. int gcd(int n, ...)
3. int dot_product(int n, ...) — скалярное произведение двух векторов. Например, произведение векторов (1, 4) и (2, -3) должно вычисляться с помощью такого вызова функции: dot_product(2, 1, 4, 2, -3).

Подставьте следующую функцию main:

```
int main() {
    assert(min(2, 5, 8) == 5);
    assert(min(3, 11, 4, 8) == 4);
    assert(gcd(2, 12, 21) == 3);
    assert(gcd(3, 42, 91, 35) == 7);
    assert(dot_product(2, 1, 4, 2, -3) == -10);
    assert(dot_product(3, 1, 2, 3, 6, 5, 4) == 28);
    return 0;
}
```

2.3.7

Рассмотрим такую функцию set_args, которая записывает значение второго аргумента по всем остальным аргументам-адресам.

Пока необязательно понимать манипуляции с адресами, но кое-что проясню:

- int* — это адрес типа int. То есть va_arg(lst, int*) достает следующий адрес.
- *a = b выполняет запись значения b в адрес a.

```
#include <stdarg.h>

void set_args(int n, int val, ...) {
    va_list lst;
    va_start(lst, val);

    int i;
    for (i = 0; i < n; i++) {
        *va_arg(lst, int*) = val;
    }

    va_end(lst);
}

int main() {
    int a, b, c;
    set_args(3, 31, &a, &b, &c);
    printf("%d %d %d\n", a, b, c);
    return 0;
}
```

Представьте, что пользователь может вводить данные и передавать их в set_args без каких либо проверок. Какую атаку можно здесь произвести (потенциально нехорошее действие)?

2.4. Указатели

2.4.1

Переходим к самой интересной части языка C — указателям/адресам (pointers).

Указатель — это число, обозначающее ячейку в оперативной памяти. Любая локальная переменная имеет адрес, так как ей нужно где-то находиться. То же относится и к аргументам функций. (На самом деле иногда данные находятся в регистрах, но компилятор здесь будет подстраиваться под наши желания.)

Для взятия адреса используется унарный (с одним аргументом) оператор &, который мы уже видели, когда использовали функцию scanf.

Тип указателя обозначается символом * (asterisk). Например, указатель на int выглядит так: int*. При этом синтаксис объявления указателей неочевидный и часто вызывает ошибки.

- Так мы объявляем два указателя: int *a, *b;
- Так мы объявляем два указателя: int* a, *b;
- Так мы объявляем указатель и число: int* a, b;

Чтобы переместить данные по указателю (то есть положить их в ячейку оперативной памяти с тем же номером, что и значение указателя), используется такая запись: *a = b, где a это указатель, a b это данные.

Рассмотрим программу:

```
void set(int *a, int val) {
    *a = val;
}

int main() {
    int a;
    set(&a, 5);
    printf("%d\n", a);
    return 0;
}
```

Данная программа демонстрирует то, что я показывал выше. Функция set принимает аргумент типа int* (можно было также написать int* a). В функции set используется перемещение (этот термин не имеет отношения к тому, что с данными что-то происходит: они остаются в том же состоянии) по указателю *a = val. При вызове функции set в качестве аргумента передается адрес локальной переменной, который добывается оператором &.

Вывод этой программы: 5

2.4.2

В языке C присутствуют массивы, но работают они довольно странно и неочевидно. Рассмотрим программу:

```
int main() {
    int a[10];
    a[4] = 32;
    printf("%d %d\n", a[4], sizeof(a));
    return 0;
}
```

Первая строка тела функции main объявляет массив размера 10. Массив —

это занумерованная последовательность объектов одинакового типа. Размер массива должен быть константой, то есть известным до компиляции (4 + 6, например, тоже является константой).

На второй строке мы присваиваем четвертому элементу массива значение 32. Массивы нумеруются с нуля. Последний индекс массива в данном случае равен 9.

На третьей строке мы выводим четвертый элемент и размер массива. Вывод этой программы: 32 40. Обратите внимание: в качестве размера массива выводится не количество его элементов, а именно занимаемая им память в байтах (напомним, что размер типа int равен 4 байта).

Что произойдет, если мы выйдем за пределы массива, сделав, например a[10] = 32? В конкретно этом случае, на самом деле, ничего. Точнее, присвоение произойдет, но то, что оно происходит за пределами массива, никак не будет проверено. Массив, как и все локальные переменные, а также адреса возврата функций, лежит на стеке, а в этом месте стека ничего важного не будет. Однако, в некоторых случаях, можно все-таки, перезаписать что-либо важное. Эта особенность делает программирование на C коварным, так как можно создать "отложенные ошибки": пока некорректный код работает, так как ничего не затирает, а с другими исходными данными уже будет затирать.

Можно создавать многомерные массивы:

```
int main() {
    const int n = 4, m = 5;
    int a[n][m];
    a[2][3] = 5;
    a[0][5] = 3;
    printf("%d %d\n", a[2][3], a[1][0]);
    return 0;
}
```

Вывод этой программы: 5 3 80

Здесь мы создали двумерный массив размера 4x5. Оперативная память адресуется линейно, и данный двумерный массив расположен в ней следующим образом: a[0][0], a[0][1], a[0][2], a[0][3], a[0][4], a[1][0], a[1][1], a[1][2], a[1][3], a[1][4], a[2][5], ... Именно поэтому, так как C не обращает внимания на выходы за границы, присваивание некорректного индекса a[0][5] приводит к присваиванию индекса a[1][0].

Так как размер массива должен быть известен на момент компиляции, я должен объявить числа m и n, как константы. Значения констант нельзя менять.

Обратите внимания на то, что размер массива равен 80, или 20 int-ов. Этот факт нам пригодится позже.

2.4.3

Массивы практически являются указателями. Можно перевести массив в указатель. Рассмотрим программу:

```
#include <stdio.h>

int main() {
    int a[10];
    int *b = a;
    *(b + 4) = 32;
    printf("%d %d %d\n", b[4], sizeof(a), sizeof(b));
    return 0;
}
```

Вывод этой программы: 32 40 4 или 32 40 8, в зависимости от разрядности вашего компилятора.

Если вы добавляете к указателю типа T* число x, то на самом деле к указателю добавится число x * sizeof(T).

*(b + 4) сначала добавляет к b значение 4 * sizeof(int), а затем разыменовывает (берет значение в этой ячейке памяти) его. Комичный факт, что b[4] == *(b + 4) == *(4 + b) == 4[b].

Обратите внимание: вы можете делать операции с указателями так же, как и с массивами. Но есть одна разница: sizeof для указателя возвращает размер указателя, а не всего массива. Более того, имея только указатель, узнать размер массива невозможно. То есть, тип указатель, в отличии от типа массив, не знает размер.

Массивы нельзя передавать в функции. Поэтому, чтобы знать в функции размер массива, приходится передавать и размер:

```
int sum(int n, int *a) {
    int x = 0, i;
    for (i = 0; i < n; i++) {
        x += a[i];
    }
    return x;
}

int main() {
    int a[5] = {2, 5, 1, 4, 3};
    int n = sizeof(a) / sizeof(a[0]);
    printf("%d\n", sum(n, a));
    return 0;
}
```

Обратите внимание на синтаксис присваивания значений элементам масси-

ва int a[5] = {2, 5, 1, 4, 3};, и на то, как можно узнать количество элементов локального массива, разделив его размер на размер одного из его элементов (часто даже создают такой макрос).

2.4.4

Реализуйте функцию void sort(int n, int *a), которая сортирует массив a размера n.

Подставьте следующую функцию main:

```
int main() {
    const int n = 10;
    int a[n] = {6, 1, 8, 2, 10, 7, 4, 5, 9, 3};
    int b[n] = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10};
    sort(n, a);
    assert(memcmp(a, b, n * sizeof(int)) == 0);
    return 0;
}
```

2.4.5

Массивы можно создавать динамически. В C нет операторов для этого, но есть стандартные функции malloc и free.

Функция void *malloc(size_t size) создает массив размера size байт и возвращает указатель типа void на его начало. size_t здесь — это unsigned int, то есть, беззнаковое (неотрицательное) целое число. Все численные типы имеют соответствующий unsigned тип.

void* указатель обозначает указатель на "что-то", и мы должны будем определить, на что он указывает, самостоятельно. Кроме того, добавление числа x к void* указателю добавит к нему именно x, поэтому этот тип часто используется для создания структур данных.

В отличии от локальных массивов, динамические массивы выделяются не на стеке. Возможные их положения зависят от операционной системы. Как можно делать динамические выделения памяти, и как работает malloc, мы узнаем позже.

Функция void free(void *ptr) освобождает массив, на начало которого указывает ptr. Это значение должно быть ранее получено от malloc. В случае, если такое значение ранее не было получено от malloc, и, соответственно, такого массива не существует, будет ошибка исполнения.

```
int main() {
    int *a = (int*)malloc(10 * sizeof(int));
    a[5] = 23;
    printf("%d\n", a[5]);
    free(a);
```

```
    return 0;
}
```

Вывод этой программы: 23

На первой строке мы выделили массив int-ов, состоящий из 10 элементов (не забывает домножать на размер элемента). Чтобы не было предупреждений от компилятора, необходимо перевести тип указателя из void* в int* с помощью (int*). (При этом при переводе типа с любого указателя на любой указатель значение не изменится. Для других базовых типов это не всегда так.) Далее мы можем оперировать массивом, как любым другим. На четвертой строке мы освобождаем массив.

Что будет, если выйти за границы динамического массива? Обычно место, в котором выделился ваш массив, это куча — большой отрезок, на операции с которым у вашей программы есть права, а у других программ нет, и в котором выделяются куски на каждый ваш вызов функции malloc. Если вы попытаетесь выполнить операцию за пределами массива, но попадёте в ваш отрезок кучи, то операция будет успешной. При этом эта точка может принадлежать другому массиву, который используется в вашей программе.

Это ещё одна коварность языка C: вы промазали по массиву, но операция выполнилась успешно, и вы повредили память, которая позже будет использоваться. В таком случае, если вы попробуете использовать дебаггер для определения проблемы, он покажет тот, скорее всего корректный, код, память которого была затёрта, а не тот код, который затёр память.

Если же вы попадёте не в ваш отрезок кучи, то программа просто завершится из-за сигнала SIGSEGV.

Что произойдет, если вы не освободите память? На самом деле, ничего, и нет никакой проблемы не освобождать память в однопроходной программе. Вся выделенная память все равно освободится при завершении программы. Однако, если ваша программа работает длительное время, часто создавая и удаляя объекты, необходимо освобождать память, чтобы она не закончилась. Вызов функции malloc может вернуть 0, что означает отказ в выделении памяти, скорее всего, по причине превышения лимита.

2.4.6

Найдите все возможные ошибки и утечки памяти в следующем коде:

```
int* foo(int x) {
    int *a = malloc(N * sizeof(int));
    if (x < 0) return 0;
    int i;
    for (i = 0; i < N; i++) {
        a[i] = x;
    }
```

```
    return a;
}

void boo() {
    int *a = malloc(N * sizeof(int));
    int *b = a;
    int *c = malloc(N * sizeof(int));
    a = c;
    c = b;
    b = a;
    c = b;
    free(a);
    free(b);
}

int flghm_count(int mode, int *result) {
    int status;
    int *buffer;
    status = flghm_init(buffer);
    if (!status) return -1;
    status = flghm_fill(buffer, result);
    if (!status) return -2;
    status = flghm_free(buffer);
    if (!status) return -3;
    return 0;
}
```

2.4.7

Изучим алгоритм сортировки слиянием (Merge Sort).

```
void merge_sort(int *a, int l, int r) {
    if (r - l == 1) return;
    int m = (l + r) / 2;
    merge_sort(a, l, m);
    merge_sort(a, m, r);
    ...
}
```

Данный алгоритм является рекурсивным. Он принимает сам массив и границы подотрезка, который данному рекурсивному вызову следует отсортировать. Если длина подотрезка равна 1, то он уже отсортирован. Если это не так, то разделим подотрезок на два меньших подотрезка и вызовем merge_sort от каждого из них. В результате мы получим два отсортированных подотрезка. Их необходимо слить в один, после чего сортировка будет завершена.

Здесь я использовал полуинтервалы, то есть индекс r не входит в подотрезок, который необходимо отсортировать. При такой реализации нет необходимости добавлять +1 и -1 к различным индексам границ.

Реализуйте алгоритм слияния. Он должен по двум отсортированным подотрезкам [l, m) и [m, r) получить один отсортированный подотрезок [l, r). Здесь [] означают границы включительно, а () — не включительно.

Дана последовательность целых чисел $a_1, a_2, \ldots, a_n$ длины n. Отсортируйте заданную последовательность.

Формат ввода Первая строка входных данных содержит одно целое число n ($1 \leq n \leq 10^3$)

Вторая строка входных данных n целых чисел $a_1, a_2, \ldots, a_n$ ($1 \leq a_i \leq 10^6$)

Формат вывода Выведите n целых чисел $b_1, b_2, \ldots, b_n$ ($b_1 \leq b_2 \leq \ldots \leq b_n$) — отсортированную последовательность a.

2.4.8

Помимо многомерных массивов можно пользоваться также многомерными указателями. Однако, как и в одномерном случае, указатели не знают форму массива. Поэтому, например, двумерный массив размера n*m в помощью указателей представляется как массив размера n, состоящий из указателей на массивы размера m (на самом деле, размеры этих массивов могут различаться). Динамическое выделение двумерного массива выглядит так:

```
int main() {
    int n = 4, m = 5;
    int **a = malloc(n * sizeof(int*));
    int i;
    for (i = 0; i < n; i++) {
        a[i] = malloc(m * sizeof(int));
    }

    a[2][3] = 12;
    printf("%d\n", a[2][3]);

    for (i = 0; i < n; i++) {
        free(a[i]);
    }
    free(a);

    return 0;
}
```

Вывод этой программы: 12

Обратите внимание на следующее:

- Так как массив динамический, его размерности не должны быть константами.
- Выделение памяти для массива происходит в два этапа: сначала я создаю массив указателей (поэтому домножение идет на sizeof(int)) размера n, а затем для каждого указателя создаю массив int-ов размера m. То же самое касается освобождения памяти.
- Итоговый размер массива не n * m * sizeof(int), а n * sizeof(int*) + n * m * sizeof(int).

В отличии от локального массива, здесь отдельные ряды не идут друг за другом, а расположены так, как это сделал malloc. Мы можем расположить ряды массива подряд самостоятельно, но тогда нам нужно будет использовать не оператор индексации [], а свою функцию.

```
int offset(int i, int j, int m) {
    return i * m + j;
}

int main() {
    int n = 4, m = 5;
    int *a = malloc(n * m * sizeof(int));

    a[offset(2, 3, m)] = 12;
    printf("%d\n", a[offset(2, 3, m)]);

    free(a);

    return 0;
}
```

Вывод этой программы: 12

Попробуйте самостоятельно разобраться с тем, как здесь организован массив.

2.4.9

Задан граф, состоящий из n вершин и m ребер. i-е ребро задается парой вершин (v_i, u_i). Найдите количество компонент связности данного графа.

Компоненты связности — это разбиение множества вершин графа на попарно непересекающиеся подмножества такие, что для любых двух вершин в одном множестве (в одной компоненте связности) существует путь между ними, а для любых двух вершин в разных множествах не существует пути между ними.

Формат ввода Первая строка входных данных содержит два целых числа n и m ($1 \le n, m \le 10^3$)

Следующие m строк входных данных содержат по два целых числа v_i и u_i ($1 \le v_i, u_i \le n$)

Формат вывода Выведите единственное число — количество компонент связности в заданном графе.

2.4.10

Интересно, и не очень красиво, обстоят дела с указателями-константами. Рассмотрим примеры:

```
int main() {
    const int *n;
    int m;
    n = &m;

    return 0;
}
```

const int* объявляет указатель, у которого нельзя менять содержимое. Мы можем изменить сам указатель (то есть, адрес, на который он указывает), но не можем изменить его содержимое *n = 2.

```
int main() {
    int *const n;
    *n = 2;

    return 0;
}
```

Здесь обратная ситуация: int *const позволяет менять содержимое указателя, но не позволяет изменить сам указатель n += 10.

Рассмотрим немного комичный пример:

```
int main() {
    const int * *const *n;

    n += 2;
    // *n = 2;
    **n += 2;
    // ***n += 2;

    return 0;
}
```

Закомментированные строки нарушают правила константности (const correctness) и вызывают ошибку компиляции.

2.4.11

Мы теперь можем понять смысл типа "строки" — const char*. Это указатель на символ (и, соответственно, следующие символы), содержимое которого (то есть, ни первый символ, ни другие) нельзя менять.

Посмотрим на сигнатуру функции int execv(const char *path, char *const argv[]); (подробнее по ссылке). Что вы можете сказать про её второй аргумент, и как это влияет на использование функции? Попробуйте вызвать эту функцию (необязательно делать что-то осмысленное, главное добиться успешной компиляции).

2.5. Строки

2.5.1

Представим, что мы печатаем строку с помощью scanf("%s", str);. str — это не более, чем указатель. Как функция scanf понимает, где кончается строка? Она понимает это с помощью численно равного нулю символа \0, которым строка заканчивается. Когда мы объявляем строку, как const char *str = "aba";, то её длина будет составлять 4, и последним символом в ней будет \0.

Что же будет, если мы целенаправленно избавимся от нулевого символа в конце строки? Сделать это мы можем так:

```
int main() {
    char a[3];
    a[0] = 'a';
    a[1] = 'b';
    a[2] = 'c';
    printf("%s\n", a);
    return 0;
}
```

Вывод этой программы будет начинаться с символов abc. То, что пойдёт далее, зависит от компилятора и его режима. Здесь происходит довольно простая и очевидная вещь: функция scanf печатает символ за символом, пока не наткнется на нулевой символ где-то дальше за строкой в памяти. В зависимости от того, где расположена строка, это также может вызвать ошибку исполнения, если нулевого символа не найдется вплоть до конца доступной программе памяти.

Из-за такой особенности работа со строками в языке C довольно неприятна. Необходимо всегда помнить о лишнем символе.

Рассмотрим первую стандартную функцию для работы со строками: size_t strlen(const char* str);. С помощью этой функции мы определяем длину

строки (не включая нулевой символ). Написать эту функцию очень просто, вот её реализация:

```
size_t strlen( const char* str ) {
    const char *x = str;
    while (*x != '\0') {
        x++;
    }
    return x - str;
}
```

Чтобы проверить эту реализацию, функцию необходимо будет переименовать, либо убрать glibc.

Для использования стандартных функций для работы со строками часто необходимо добавить #include <string.h>.

2.5.2

При выполнении данной программы происходит ошибка исполнения, хотя строка не является константой, и компиляция проходит успешно. (Возможно, наблюдается не на всех компиляторах.) Почему?

```
int main() {
    char *a = "abacaba";
    a[3] = 'b';

    return 0;
}
```

2.5.3

Функция int sprintf(char* buffer, const char* format, ...); работает, как функция printf, но она принимает своим первых аргументом строку buffer, в которую и происходит печать. При этом содержимое строки buffer никак не проверяется на переполнение.

```
int main() {
    char str[100];
    sprintf(str, "%d %d", 32, 85);
    printf("%s\n", str);
}
```

Вывод этой программы: 32 85

Функция char *strcpy(char *dest, const char *src); выполняет копирование строки из второго аргумента в первый (поэтому тип первого аргумента лишён const). Копирование выполняется до обнаружения нулевого символа в src. При этом содержимое строки dest никак не проверяется на переполнение.

```
int main() {
    const char *a = "abacaba";
    char b[10];
    strcpy(b, a);
    printf("%s\n", b);
    return 0;
}
```

Вывод этой программы: abacaba

Обратите внимание, что функция strcpy закладывает нулевой символ в конец строки dest. Не забывайте оставлять под него место.

Часто необходимо ограничить длину копируемой строки. Вместо добавления if-а можно использовать стандартную функцию char *strncpy(char *dest, const char *src, size_t count);, которая копирует не более count символов, включая нулевой символ (то есть, может оставить строку без нуля на конце).

```
int main() {
    const char *a = "abacaba";
    char b[4];
    strncpy(b, a, 3);
    printf("%s\n", b);
    return 0;
}
```

Вывод этой программы: aba

Вообще, "n" версии есть у почти каждой строковой функции. Посмотрите функцию strnlen.

2.5.4

Функция int strcmp(const char* lhs, const char* rhs); сравнивает две строки лексикографически.

> Return value
>
> Negative value if lhs appears before rhs in lexicographical order.
>
> Zero if lhs and rhs compare equal.
>
> Positive value if lhs appears after rhs in lexicographical order.

Обратите внимание, что упоминается лишь знак возвращаемого значения. Само же значение не задокументировано.

Так мы проверяем две строки на равенство:

```
int main() {
    const char *a = "aba";
    const char *b = "baba";
```

```
    const char *c = "aba";
    printf("%d\n", strcmp(a, b));
    printf("%d\n", strcmp(b, a));
    printf("%d\n", strcmp(a, c));

    return 0;
}
```

Вывод этой программы: -1 1 0

Перевести строку в число можно с помощью функции int atoi(const char* str);.(В случае некорректной строки она возвращает 0.)

```
int main() {
    printf("%d\n", atoi("5328"));
    return 0;
}
```

Часто необходимо проводить копирование в динамическую строку (вообще, часто приходится работать и с динамическими строками). Вместо такого:

```
int main() {
    const char *a = "abacaba";
    int n = strlen(a);
    char *b = (char*)malloc((n + 1) * sizeof(char));
    strcpy(b, a);
    printf("%s\n", b);
    return 0;
}
```

, можно использовать функцию char * strdup(const char *str1);, которая делает то же самое.

```
int main() {
    const char *a = "abacaba";
    char *b = strdup(a);
    printf("%s\n", b);
    return 0;
}
```

2.5.5

Реализуйте функции strcpy, strncpy, strcmp и strncmp так, как ранее была реализована функция strlen. Добавьте в начало их названия символ _, чтобы не было конфликтов имен.

Подставьте следующую функцию main:

```
int main() {
    const char *a = "abacaba";
```

```
    char b[10];
    _strcpy(b, a);
    assert(strcmp(b, "abacaba") == 0);
    b[3] = '\0';
    _strncpy(b, a, 3);
    assert(strcmp(b, "aba") == 0);
    assert(_strcmp("aba", "aba") == 0);
    assert(_strcmp("aba", "abc") < 0);
    assert(_strcmp("abc", "aba") > 0);
    assert(_strncmp("aba", "abc", 2) == 0);
    assert(_strncmp("aba", "abc", 3) < 0);
    return 0;
}
```

2.5.6

Следующая функция char* strtok(char* str, const char* delim); более сложная в использовании. Эта функция используется для разбиения строки str на токены. Аргумент delim хранит символы-разделители токенов. Рассмотрим программу:

```
#include <string.h>

int main() {
    char str[] = "ab;cde,fghi";
    int n = strlen(str);

    char *a = strtok(str, ",;");
    printf("%d ", strlen(str));
    printf("%d ", strlen(a));

    char *b = strtok(NULL, ",;");
    printf("%d ", strlen(b));

    char *c = strtok(NULL, ",;");
    printf("%d ", strlen(c));

    char *d = strtok(NULL, ",;");
    printf("%d\n", d);

    int i;
    for (i = 0; i <= n; i++) {
        if (str[i] == 0) {
            printf(".");
        }
        else {
```

```
            printf("%c", str[i]);
        }
    }
    printf("\n");

    return 0;
}
```

Вывод этой программы:

```
2 2 3 4 0
ab.cde.fghi.
```

Пусть разделителями будут символы , и ;. Сначала выполним такой вызов функции: strtok(str, ",;");. В результате этого вызова вместо первого разделителя в изначальной строке ; будет поставлен нулевой символ, будет возвращен указатель на первый символ, не являющийся разделителем (а наша строка не начинается с разделителей, поэтому будет возвращен str) , а внутри функции strtok будет неким образом создана запись о том, какой последний символ был обработан.

Затем выполним вызов strtok(NULL, ",;");. В результате этого вызова повторятся те же действия, но не с аргумента (который теперь просто 0), а с последнего обработанного символа. Будет найден следующий разделительный символ , и заменен на нулевой символ. Затем будет возвращён указатель на символ c.

При выполнении третьего вызова разделительный символ не будет найден, но strtok всё равно вернёт указатель на начало последнего токена — символ f. Так как последний токен программы был считан, четвёртый вызов функции strtok вернёт ноль.

Для лучшего понимания посмотрите на конечное состояние массива str, в котором вместо каждого нулевого символа написан символ .. Все ненулевые значения, возвращённые функцией strtok указывают на этот массив.

Значение "переменной" (что это такое, узнаем чуть позже) NULL равно нулю. (Вы можете вместо NULL везде написать 0.) Также здесь нам необходимо добавить string.h.

2.5.7

Напишите функцию int strsum(const char *str);, которая принимает строку, состоящую из последовательности целых чисел и пробелов, и возвращает сумму чисел в этой строке.

Подставьте следующую функцию main:

```
int main() {
    assert(strsum("123") == 123);
```

```
    assert(strsum("-6") == -6);
    assert(strsum("  2  2    3 ") == 7);
    assert(strsum("  -6  24    -7 ") == 11);
    return 0;
}
```

2.5.8

Для выполнения таких же операций с массивами любого типа используются mem*** функции. Естественно, функции memlen нет, так как только строки завершаются нулём. Не забывайте о том, что размер массива принимается в байтах, поэтому нужно домножать количество элементов на размер одного элемента.

Функция void *memset(void *buf, int ch, size_t count) устанавливает len байт массива buf равными младшему байту числа ch (старшие байты при этом игнорируются).

```
int main() {
    int a[4];
    memset(a, 0x1234, 4 * sizeof(int));
    int i;
    for (i = 0; i < 4; i++) {
        printf("%d ", a[i]);
    }
    printf("%d", 0x34343434);

    return 0;
}
```

Вывод этой программы: 875836468 875836468 875836468 875836468 875836468

Функция void *memcpy(void *dest, const void *source, size_t count) копирует count байт массива source в массив dest.

```
int main() {
    int a[4] = {1, 2, 3, 4};
    int b[4];
    memcpy(b, a, 4 * sizeof(int));
    int i;
    for (i = 0; i < 4; i++) {
        printf("%d ", b[i]);
    }

    return 0;
}
```

Вывод этой программы будет состоять из четырёх чисел, первые три из которых: 1 2 3

Функция int memcmp(const void *buf1, const void *buf2, size_t count) сравнивает первые count элементов массивов buf1 и buf2 лексикографически так же, как и функция strcmp.

```
int main() {
    int a[4] = {1, 2, 3, 4};
    int b[4] = {1, 2, 3, 5};
    printf("%d ", memcmp(a, b, 3 * sizeof(int)));
    printf("%d ", memcmp(a, b, 4 * sizeof(int)));

    return 0;
}
```

Вывод этой программы: 0 -1

Следует отдавать предпочтение mem*** функциям, так как они используют специальные ассемблерные инструкции (которые мы ещё увидим), и потому более быстрые. (Хотя str*** функции в то же время тоже имеют некоторые хитрые оптимизации, связанные с итерированием не по одному символу, а по машинному слову. Но это не так сильно влияет.)

2.6. Структуры

2.6.1

Один байт состоит из восьми бит. Бит принимает одно из двух значений: 0 или 1. Поэтому один байт может принимать $2^8 = 256$ различных значений.

Рассмотрим, как переводить числа из двоичной в десятичную систему счисления. Мы будем указывать менее значимые цифры справа. (Хотя это и не важно, и часто для удобства менее значимые цифры размещают слева.)

Двоичная система счисления	Десятичная система счисления
0	0
1	1
10	2
11	3
100	4
101	5
110	6
111	7
1000	8

В общем случае, если i-й бит числа в двоичной записи равен a_i, в десятичной записи оно будет равно $a_0 \cdot 1 + a_1 \cdot 2 + a_2 \cdot 4 + a_3 \cdot 8 + \ldots = \sum a_i \cdot 2^i$

Целочисленные типы данных бывают знаковыми (signed) и беззнаковыми (unsigned). Беззнаковые типы принимают значение от 0 (когда все биты нулевые) до $2^{size} - 1$ (когда все биты единицы), где $size$ — это размер типа данных в битах. Так, максимальное значение типа unsigned int равно $2^{32} - 1 = 4294967295$.

Знаковые типы принимают значение от -2^{size-1} до $2^{size-1} - 1$. Как кодируются отрицательные числа покажем на примере типа char.

Двоичная система счисления	Десятичная система счисления
00000000	0
11111111	-1
11111110	-2
11111101	-3
11111100	-4

Переводить число из n-й системы счисления в десятичную можно с помощью формулы $\sum a_i \cdot n^i$. Чтобы писать в коде числа в двоичном, восмеричном и шестнадцатиричном формате, следует перед ними написать 0b, 0 и 0x соответственно. Формат %d выполнит печать в десятичном формате, но printf также имеет форматы и для других систем счисления.

```
int main() {
    printf("%d %d %d\n", 0b1010, 013, 0xC);
    return 0;
}
```

Вывод этой программы: 10 11 12

2.6.2

Для манипуляций с битами чисел используются битовые операторы. Они взаимодействуют на каждую позицию чисел-аргументов независимо и действуют аналогично одноименным логическим операторам.

- & — оператор И. Например, 0b10110 & 0b11010 = 0b10010.
- | — оператор ИЛИ. Например, 0b10101 | 0xb11001 = 0x11101.
- ^ — оператор исключающее ИЛИ. Например, 0x1001 ^ 0x1100 = 0x0101.
- ~ — унарный оператор НЕ. Например, для типа char, ~0b1010 = 0b11110101. (Обратите внимание на зависимость от размера типа данных.)
- << — оператор сдвига числа налево (в сторону увеличения числа). Например, 0b1001 << 2 = 0b100100. Сдвиг не циклический. Если никакие единичные биты не пропадают, то a << b= $a \cdot 2^b$.
- >> — оператор сдвига числа направо (в сторону уменьшения числа). Например, 0b1001 >> 2 = 0b10. Сдвиг не циклический. a >> b= $\lfloor \frac{a}{2^b} \rfloor$.

Проверить наличие i-го бита в числе можно так: if (a & (1 << i)).

Битовые операторы имеют неочевидные приоритеты. Например, если вы хотите проверить, что i-й бит отсутствует, то такая проверка if (a & (1 << i) == 0) некорректна, так как оператор сравнения выполнится до оператора битового И.

Если вы проверяете наличие i-го бита в числе типа long long, то следует делать сдвиг так: 1ll << i. Суффикс ll делает число 1 типа long long. Если же не приписать этот суффикс и проверить, например, 40-й бит, то сначала выполнится сдвиг числа 1 типа int на 40 позиций, в результате чего получится ноль, который затем будет переведен в тип long long.

Подобные суффиксы есть и у других типов. Дробные числа, записанные так — 4.3 — имеют тип double, а не float, что не позволит написать, например, так: const float a = 4.3;(возможно, не во всех компиляторах). Числа типа float следует завершать символом f.

2.6.3

Дана последовательность целых чисел $a_1, a_2, \ldots, a_n$ длины n. Найдите минимальный размер такого множества чисел $b_1, b_2, \ldots, b_m$, что $a_{b_1}|a_{b_2}|\ldots|a_{b_m} = a_1|a_2|\ldots|a_n$

Формат ввода Первая строка входных данных содержит одно целое число n $(1 \leq n \leq 20)$

Вторая строка входных данных содержит n целых чисел $a_1, a_2, \ldots, a_n$ $(1 \leq a_i \leq 2^{30} - 1)$

Формат вывода Выведите единственное целое число m — минимальный размер подходящего множества.

2.6.4

Рассмотрим перевод дробного числа из двоичной записи в десятичную. Здесь используется та же формула, но теперь она распространяется и на отрицательные индексы.$0b10.1011 = 1 \cdot 2^1 + 0 \cdot 2^0 + 1 \cdot 2^{-1} + 0 \cdot 2^{-2} + 1 \cdot 2^{-3} + 1 \cdot 2^{-4} = 2 + 0 + 0.5 + 0 + 0.125 + 0.0625 = 2.6875$

Рассмотрим структуру стандартного типа данных с плавающей точкой: float. Он состоит из 32-х бит:

- 1 бит — знак, как у целых чисел
- 2-9 биты — порядок (exponent)
- 10-32 биты — мантисса (fraction)

Порядок кодируется таким же образом, как и целое беззнаковое число. Обозначим его как A. Мантисса кодирует дробное число, меньшее единицы:

первый бит кодирует $\frac{1}{2}$, второй бит кодирует $\frac{1}{4}$ и так далее. Обозначим его как B. Тогда значение числа равно $2^{A-127} \cdot (1 + B)$

Посмотрим на такое значение типа float: 0 01111100 01000000000000000000000

- Первый бит равен 0, поэтому число неотрицательное.
- В порядке стоит число $A = 0 \cdot 2^7 + 1 \cdot 2^6 + 1 \cdot 2^5 + 1 \cdot 2^4 + 1 \cdot 2^3 + 1 \cdot 2^2 + 0 \cdot 2^1 + 0 \cdot 2^0 = 124$
- В мантиссе стоит число $B = 0 \cdot \frac{1}{2^1} + 1 \cdot \frac{1}{2^2} + 0 \cdot \frac{1}{2^3} + 0 \cdot \frac{1}{2^4} + \ldots = 0.25$
- Итого данное число равно $2^{124-127} \cdot (1 + 0.25) = 0.15625$

Есть отдельные два случая:

- Если $A = 0$, то формула выглядит так: $2^{A-127} \cdot B$
- Если $A = 255$, то это особое число, соответствующее, например, бесконечности или nan (not a number), которые возникают при некоторых некорректных математических операциях. Чтобы проверить число на nan следует сравнить его с собой: if (x != x) { "x is nan" }.

Чтобы использовать тип float в функциях printf и scanf, следует использовать формат %f.

Существуют и другие типы чисел с плавающей точкой:

- double — 8 байт
- long double — не определено (обычно 10 байт)

2.6.5

Ранее я рассказывал об отдельном способе кодирования числа типа float, когда его показатель равен нулю. Зачем это необходимо?

2.6.6

Перечисления (enumerations) — это способ задать числам идентификатор, привязав их при этом к контексту. Рассмотрим программу:

```
enum E {
    A,
    B,
    C = 10,
    D,
    E
};

int main() {
    enum E e = A;
    printf("%d %d %d %d %d %d\n", A, B, C, D, E, e);
    return 0;
}
```

Мы объявляем перечисление с помощью ключевого слова enum, затем мы даем ему название E и в фигурных скобках перечисляем значения. Сами значения в перечислении используются так же, как и переменные типа число-константа const int. Эти значения мы можем вывести без ограничений, как и если бы это были константы. Посмотрите на логику присваивания значений им. Мы можем также объявлять объекты или инстансы (instances) перечисления, указывая их тип как enum E. Этот тип — это всего лишь число.

Вывод этой программы: 0 1 10 11 12 0

Здесь может возникнуть вопрос: а где же привязка к контексту (названию перечисления), ведь их содержимое ведет себя как глобальные переменные? Этой привязки нет, что является недостатком перечислений в языке C. Тем не менее, в C++ были введены классы перечислений (enum classes), которые решают этот недостаток.

Перейдем к структурам (structures). Они необходимы для объединения группы переменных для того, чтобы манипулировать ими одновременно. В отличии от массивов, типы переменных в структурах разные, и их количество сильно ограничено.

```
struct S {
    int a;
    long long b;
};

int main() {
    struct S s1;
    s1.a = 4;
    s1.b = 7;
    printf("%d %d ", s1.a, s1.b);

    struct S *s2 = (struct S*)malloc(sizeof(struct S));
    (*s2).a = 12;
    s2->b = 17;
    printf("%d %d\n", s2->a, s2->b);
    free(s2);

    return 0;
}
```

Мы объявляем структуру с помощью ключевого слова struct, следом за которым указываем название структуры и её содержимое в фигурных скобках. Объект или инстанс структуры объявляются с указыванием типа struct S и названия структуры: struct S s1;. Как только мы объявили этот объект, у нас появились сразу две связанные переменные, к которым мы обращаемся через идентификатор struct. Хоть структуры и является сложным

типом, их можно передавать в функции и возвращать из них. К полям (field) структуры мы обращаемся через символ ..

Можно объявлять объекты структуры динамически. Тогда, чтобы получить доступ к полям структуры, необходимо выполнить разыменование. Так как это распространённое действие (вы даже чаще будете взаимодействовать именно с указателями на структуры), для него есть свой синтаксис ->.

2.6.7

Есть последовательность чисел, которая изначально состоит из единственного числа 0, указатель, который изначально указывает на первую позицию последовательности, и счетчик x, изначально равный 1.

Необходимо обработать q запросов четырех типов:

1. Сдвинуть указатель налево. Перед этим действием указатель находится не на первой позиции.
2. Сдвинуть указатель направо. Перед этим действием указатель находится не на последней позиции.
3. Вставить элемент x после указателя, а затем увеличить значение x на единицу.
4. Удалить элемент, на который указывает указатель, сдвинув его на предыдущий элемент. Перед этим действием указатель находится не на первой позиции.

Формат ввода Первая строка входных данных содержит одно целое число q ($1 \le q \le 1000$)

Вторая строка входных данных содержит q целых чисел $k_1, k_2, \ldots, k_q$ ($1 \le k_i \le 4$) — запросы, которые необходимо обработать. Гарантируется, что запросы удовлетворяют ограничениям, указанным в условии.

Формат вывода Выведите последовательность чисел после выполнения всех запросов.

2.6.8

Один из способов изменить битовую запись числа типа float напрямую следующий:

```
int main() {
    float a;
    int *b = (int*)(&a);
    *b = 0x3FFFFFFF;
    printf("%f\n", a);
    return 0;
}
```

(Попробуйте самостоятельно выяснить, какое число будет выведено.)

В C++ подобные манипуляции можно делать с помощью оператора reinterpret_cast. В C также можно сделать это с помощью объединений union.

Объединение реализует сумму типов (sum types), то есть её содержимое является одним из нескольких типов. Однако, в силу простоты языка C и того, как в нем объединения реализованы:

- Узнать, какой тип храниться в конкретном объекте объединения без дополнительной информации, невозможно.
- Само обращение к объединению является лишь приведением одного и того же адреса в тип, который мы запрашиваем.

Таким образом, объединения отличаются от просто приведения типа тем, что выбор типов ограничен, и у них есть псевдонимы.

```
union U {
    float a;
    int b;
};

int main() {
    union U u;
    u.b = 0x3FFFFFFF;
    printf("%f\n", u.a);
    return 0;
}
```

Можно типам данных давать псевдонимы. Например, если мы напишем typedef int i32;, то мы сможем, написав i32 создать переменную типа int: i32 a = 3;.

К перечислениям, структурам и объединениям можно применять такой синтаксис:

```
typedef int i32;

typedef struct {
    i32 a;
} S;

int main() {
    S s;
    s.a = 7;
    return 0;
}
```

При таком объявлении структуры мы не должны писать ключевое слово struct в типе.

2.7. Функции высшего порядка, макросы, шаблоны

2.7.1

В C можно передавать в функции указатели на другие функции, чтобы строить функции высших порядков.

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

int apply(int x, int(*f)(int)) {
    return f(x);
}

int main() {
    printf("%d\n", apply(5, square));
    return 0;
}
```

Функция square возвращает квадрат аргумента и интереса не представляет. Посмотрим на второй аргумент функции apply. Тип этого аргумента — int(*)(int) — указатель на функцию, которая принимает один аргумент типа int, и возвращает тип int. f — это название аргумента. Функция apply применяет свой второй аргумент к первому аргументу.

Указатели на функции можно объявлять и как обычные локальные переменные.

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

int main() {
    int(*foo)(int) = square;
    printf("%d\n", foo(5));
    return 0;
}
```

Выводы обеих программ — 25

Указатели на функции обычно используются для двух целей:

- Создание функций высшего порядка
- Создание наследования

2.7.2

В языке C нет синтаксиса для создания вложенных функций (объявления функции в теле другой функции). Как можно добиться похожего функционала?

2.7.3

До компиляции программы выполняется её препроцессирование, которое заключается в выполнении всех строк, которые начинаются с символа #. Помимо включения файлов с помощью #include, препроцессор даёт возможность писать макросы. Макросы могут напоминать функции, но у них есть и различия.

```
#define FOO 5
#define DOUBLE1(x) (x * 2)
#define DOUBLE2(x) ((x) * 2)

int main() {
    printf("%d ", FOO);
    printf("%d ", DOUBLE1(4 + 1));
    printf("%d\n", DOUBLE2(4 + 1));
    return 0;
}
```

Вывод этой программы — 5 6 10

#define выполняет подстановку своего третьего аргумента заместо второго в каждом его вхождении в программу. Например, все вхождения FOO заменятся на 5. Макросы могут иметь аргументы, которые будут также подставлены без каких либо изменений.

Здесь сразу обратите внимание на второе число в выводе и попробуйте самостоятельно понять, почему в результате получилось оно.

Чтобы это понять, раскроем макрос вручную: DOUBLE1(4 + 1) -> {x = "4 + 1"} -> (4 + 1 * 2) = 6. Это одна из главных проблем использования макросов — необходимость продумывать, как будет выполнено раскрытие. Конкретно эта проблема решается заключением аргумента макроса в скобки, чтобы сначала выполнить вычисление аргумента. Однако, не всегда решение подобных проблем простое и вообще возможное.

Чем макрос может быть хорош по сравнению с функцией? Он может позволить создавать подобие функции для обобщенного типа (то есть, без уточнения типа). Например, обычно с помощью макроса реализовывают функции min и max.

```
#define MIN(X, Y) \
    (X < Y) ? X : Y
```

```
int main() {
    printf("%d %d\n", MIN(3, 5), MIN('b', 'a'));
    return 0;
}
```

Вывод этой программы — 3 97

Символ \ продолжает действие конструкции препроцессора на следующую строку. Здесь мы с помощью тернарного оператора написали подобие функции, которое на самом деле будет просто подставлять в код выражение с тернарным оператором: MIN(3, 5) -> {X = 3, Y = 5} -> (3 < 5) ? 3 : 5.

Тернарный оператор A ? B : C похож на if-statement, но является expression. В случае, если A != 0, он вернёт B. В противном случае он вернёт C.

Если требуется в макросе выполнить последовательность действий, обычно делают так:

```
#define PRINT(X, Y)          \
    {                        \
        printf("%d ", X);  \
        printf("%d\n", Y); \
    }

int main() {
    PRINT(4, 7)
    return 0;
}
```

Вывод этой программы — 4 7

2.7.4

В языке C нет встроенных способов создания функций для обобщенных типов. Мы можем написать макрос, который будет объявлять функцию для заданного типа (ведь в качестве аргументов макроса можно использовать не только значение). Однако до определенного стандарта в языке C не было автоматического определения функции на основе типа. (В языке C++ для этого есть перегрузка функций. Как она работает, и почему она отсутствует в языке C, мы узнаем позже.)

Есть разные способы организации функции для обобщенных типов, и в разных проектах используются разные. Вот один из способов.

```
#include <stdio.h>
#include <stdlib.h>

#define SIZE(X) \
        sizeof(X) / sizeof(X[0])
```

```
typedef struct {
    int x;
} S;

typedef struct {
    int x;
} T;

T foo(S s) {
    T t;
    t.x = s.x * 2;
    return t;
}

int boo(int x) {
    return x + 1;
}

#define MAP(F, A, B)                    \
B* F(B(*f)(A), A* l1, int sz) {         \
    B* l2 = (B*)malloc(sz * sizeof(B)); \
    for (int i = 0; i < sz; i++) {      \
        l2[i] = f(l1[i]);               \
    }                                   \
    return l2;                          \
}

MAP(map_ii, int, int)
MAP(map_st, S, T)

int main() {
    int i;
    int lst1[4] = {1, 4, 6, 11};
    int *lst1_out = map_ii(boo, lst1, SIZE(lst1));
    for (i = 0; i < SIZE(lst1); i++) {
        printf("%d ", lst1_out[i]);
    }
    printf("\n");

    S lst2[4] = {{1}, {3}, {5}, {9}};
    T *lst2_out = map_st(foo, lst2, SIZE(lst2));
    for (i = 0; i < SIZE(lst2); i++) {
        printf("%d ", lst2_out[i].x);
    }
    printf("\n");
```

```
    return 0;
}
```

Реализуем функцию map из функциональных языков, которая принимает функцию и массив элементов, после чего создает новый массив, полученный поэлементным применением функции к исходному массиву. Посмотрим на сам макрос:

```
#define MAP(F, A, B)                    \
B* F(B(*f)(A), A* l1, int sz) {         \
    B* l2 = (B*)malloc(sz * sizeof(B)); \
    for (int i = 0; i < sz; i++) {      \
        l2[i] = f(l1[i]);               \
    }                                   \
    return l2;                          \
}
```

Макрос принимает название функции F, которую следует объявить, и входной и выходной типы A и B. Сигнатура функции выглядит так: B* F(B(*f)(A), A* l1, int sz), то есть она принимает функцию из типа A в тип B и массив типа A с его размером, а возвращает массив типа B. То есть, функция map параметризована двумя типами.

Мы создаем функцию map для типов int и int так: MAP(map_ii, int, int). После раскрытия макроса вручную мы получим такое:

```
int* map_ii(int(*f)(int), int* l1, int sz) {
    int* l2 = (int*)malloc(sz * sizeof(int));
    for (int i = 0; i < sz; i++) {
        l2[i] = f(l1[i]);
    }
    return l2;
}
```

Обратите внимание на то, что map не освобождает входной массив. Следует ли это делать зависит от вашего стиля создания функций обобщенных типов.

Вызвать функцию map для типов int и int мы можем так: map_ii(boo, lst1, SIZE(lst1));. К сожалению, нам необходимо самостоятельно подбирать правильное название функции на основе типа. В более современном языке C есть ключевое слово _Generic, которое выполняет эту работу за нас.

Вывод этой программы:

```
2 5 7 12
2 6 10 18
```

2.7.5

Реализуйте с помощью макроса функцию zip, которая принимает вторым и третьим аргументом массивы типа A и B, а первым аргументом [функцию, которая принимает два аргумента типа A и B, и возвращает тип C], и создает новый массив, полученный поиндексным применением функции к элементам массивов (первый к первому, второй ко второму и т.д.). Подставьте эту программу.

```
#include <stdio.h>
#include <stdlib.h>
#include <assert.h>

#define SIZE(X) \
        sizeof(X) / sizeof(X[0])

typedef struct {
    int x;
} S;

typedef struct {
    int x;
} T;

typedef struct {
    int x;
} U;

U doo(S s, T t) {
    U u;
    u.x = s.x * t.x;
    return u;
}

ZIP(zip_stu, S, T, U);

int main() {
    int i;
    S lst1[4] = {{1}, {3}, {5}, {9}};
    T lst2[4] = {{2}, {6}, {10}, {17}};
    U *lst3 = zip_stu(doo, lst1, lst2, SIZE(lst1));
    assert(lst3[0].x == 2);
    assert(lst3[1].x == 18);
    assert(lst3[2].x == 50);
    assert(lst3[3].x == 153);
```

```
    return 0;
}
```

2.7.6

Указатели на функции дают возможность сделать подобие наследования. Мы можем написать функции-конструкторы, которые будут подставлять в объекты структур указатели на необходимые функции. Однако, функции через указатели не могут видеть содержимое структур, поэтому придется передавать объект структуры вместе с каждым вызовом её функции вручную. (На самом деле, в языке C++ указатель на объект, функция которого вызывается, всегда передаётся в функцию и называется this.)

```
#include <stdlib.h>

const float pi = 3.14f;

struct Figure;

typedef struct {
    float x;
    float (*square)(struct Figure*);
} Figure;

float circle_square(Figure *f) {
    return pi * f->x * f->x;
}

float square_square(Figure *f) {
    return f->x * f->x;
}

Figure *build_circle(float x) {
    Figure *f = (Figure*)malloc(sizeof(Figure));
    f->x = x;
    f->square = circle_square;
    return f;
}

Figure *build_square(float x) {
    Figure *f = (Figure*)malloc(sizeof(Figure));
    f->x = x;
    f->square = square_square;
    return f;
}
```

```
int main() {
    Figure *circle = build_circle(3);
    Figure *square = build_square(3);
    printf("%f %f\n", circle->square(circle), square->square(square));

    return 0;
}
```

Функции circle_square и square_square — это соответствующие реализации функции square (площадь) для структур circle (окружность) и square (квадрат). Функции build_circle и build_square — это конструкторы, которые создают объект структуры и подставляют нужные реализации функций.

В результате этих манипуляций нам не нужно указывать вручную, какую функцию подсчета площади вызывать. На самом деле, в языке C++ выбор нужной функции организован похожим образом: объекты содержат указатели на нужные функции. Это называется виртуальными таблицами (virtual tables), и позже мы узнаем, как они устроены.

Тем не менее, несоответствие локальных переменных у структур в языке C решить нельзя. Если бы мы хотели ввести структуру rectangle (прямоугольник), которая задается двумя числами, сделать это красиво у нас не получится.

2.7.7

Подобно функциям для обобщенных типов можно с помощью макросов организовать структуры для обобщенных типов. Для создания объектов таких структур потребуется также генерация их конструкторов. Попробуйте реализовать это любым способом.

3. Компиляция, gcc

3.1. Фазы компиляции

3.1.1

Напомню, что мы работает с компилятором gcc.

Компиляция состоит из четырёх шагов:

1. Препроцессирование (preprocessing)
2. Компиляция
3. Ассемблирование
4. Компоновка (linking)

Мы запускаем компиляцию командой gcc <filename>, где filename — название файла с кодом. Его расширение должно быть .c. При таком запуске мы получим исполняемый файл с неким именем по умолчание. Имя выходного файла можно задать явно флагом -o <filename>.

- Чтобы выполнить только первый этап, необходимо добавить флаг -E.
- Чтобы выполнить первый и второй этап, необходимо добавить флаг -S.
- Чтобы выполнить первый, второй и третий этапы, необходимо ввести флаг -c.

Далее мы изучим каждый из этапов подробнее.

С этого момента наши программы будут состоять из нескольких файлов.

3.1.2

main.c

```
#include "foo.h"

int main() {
    printf("%d\n", foo(2));
    return 0;
}
```

foo.h

```
int foo(int x) {
    return x + 1;
}
```

Препроцессирование выполняет все строки, которые начинаются с символа #(и следующие за ними, если они заканчиваются символом \). Это могут быть инструкции #include, #define, #ifdef и некоторые другие.

Инструкция #include вставляет содержимое файла. Если название файла заключено в "", то путь файла указывается относительно текущего файла (в примере файл foo.h должен находиться в одной директории с файлом main.c). Если название файла заключено в <>, то компилятор попытается найти файл относительно стандартной глобальной директории заголовочных файлов (там, где лежат файлы stdio.h, stdlib.h и т.д.) и относительно переданных ему директорий заголовочных файлов с помощью флага -I (например, gcc -I. main.c, что означает, искать заголовочные файлы в директории . (то есть, в текущей)).

Выполните gcc main.c -E, чтобы выполнить препроцессирование. Вы получите такой вывод:

```
# 0 "main.c"
# 0 "<built-in>"
# 0 "<command-line>"
# 1 "/usr/include/stdc-predef.h" 1 3 4
# 0 "<command-line>" 2
# 1 "main.c"
# 1 "foo.h" 1
```

```
int foo(int x) {
    return x + 1;
}
# 2 "main.c" 2

int main() {
    printf("%d\n", foo(2));
    return 0;
}
```

Результатом препроцессирования всегда является один файл. Обратите внимание, что содержимое файла foo.h непосредственно вставилось в выходной файл без каких-либо изменений.

Рассмотрим такую программу.

main.c

```
#ifdef FOO
#define DOO 4
#elif BOO
#define DOO 5
#else
#define DOO 6
#endif

#define FOO 3

#ifdef FOO
#define BOO 4
#endif

int main() {
    printf("%d %d %d\n", FOO, BOO, DOO);
    return 0;
}
```

В результате её препроцессирования "переменные" FOO, BOO, DOO заменятся на, соответственно, значения 3, 4, 6. В целом, логика работы таких ветвлений очевидна. Обратим лишь внимание на то, что обрабатываются они просто сверху вниз, и поэтому в блоке, который определяет переменную DOO, переменные FOO и BOO ещё не определены, и переменная DOO получает значение 6.

Если вы не определите значение какой либо переменной, то её замены в коде не произойдет. При этом, полученный код может оказаться корректным.

Инструкции #define могут принимать аргументы. Такие инструкции называются макросами, и мы их уже видели.

3.1.3

main.c

```
#include "foo.h"

int main() {
    printf("%d\n", foo(2));
    return 0;
}
```

foo.h

```
int foo(int x) {
    return x + 1;
}
```

Компиляция переписывает код в язык ассемблера (Assembly). Язык ассемблера мы изучим позже.

Выполните gcc main.c -S, чтобы выполнить компиляцию. Вы получите такой вывод (содержимое обрезано):

```
   .file   "main.c"
    .text
    .globl  foo
    .type   foo, @function
foo:
.LFB0:
    .cfi_startproc
    endbr64
    pushq   %rbp
    .cfi_def_cfa_offset 16
    .cfi_offset 6, -16
    movq    %rsp, %rbp
    .cfi_def_cfa_register 6
    movl    %edi, -4(%rbp)
    movl    -4(%rbp), %eax
    addl    $1, %eax
    popq    %rbp
    .cfi_def_cfa 7, 8
    ret
    .cfi_endproc
...
```

Кратко охаректеризовать язык ассемблера можно так. У нас есть порядка десяти ячеек памяти, которые называются регистрами, и доступ к этим ячейкам самый быстрый. Кроме того у нас есть оперативная память (RAM), которая представляет собой большую ленту пронумерованных ячеек. Наша

программа является последовательностью инструкций в оперативной памяти, которые имеют следующий вид:

- Записать в регистр D значение, равное сумме значений в регистрах B и I.
- Записать в регистр E значение, записанное в оперативной памяти в ячейке под номером, который записан в регистре A.
- Если значение в регистре H равно нулю, то продолжить выполнение программы с инструкции, которая находится в оперативной памяти в ячейке под номером, который записан в регистре C.

Понятно, что с одной стороны такой язык легко выучить из-за его простоты, с другой стороны его очень тяжело читать из-за отсутствия конструкций, отвечающих за высокоуровневую организацию кода.

3.1.4

main.c

```
#include "foo.h"

int main() {
    printf("%d\n", foo(2));
    return 0;
}
```

foo.h

```
int foo(int x) {
    return x + 1;
}
```

Ассемблирование переписывает код, написанный на языке ассемблера, в объектный файл (имеет расширение .o или .obj). Объектный файл состоит из сегментов, которые содержат инструкции, из которых состоит наша программа, а также некоторую метаинформацию.

Выполните gcc main.c -c, чтобы выполнить ассемблирование. Выходной файл не состоит из читаемого текста. Выполним сначала hd main.o, чтобы посмотреть содержимое, как есть (содержимое обрезано):

```
00000000  7f 45 4c 46 02 01 01 00  00 00 00 00 00 00 00 00  |.ELF............|
00000010  01 00 3e 00 01 00 00 00  00 00 00 00 00 00 00 00  |..>.............|
00000020  00 00 00 00 00 00 00 00  e0 02 00 00 00 00 00 00  |................|
00000030  00 00 00 00 40 00 00 00  00 00 40 00 0e 00 0d 00  |....@.....@.....|
00000040  f3 0f 1e fa 55 48 89 e5  89 7d fc 8b 45 fc 83 c0  |....UH...}..E...|
00000050  01 5d c3 f3 0f 1e fa 55  48 89 e5 bf 02 00 00 00  |.].....UH.......|
00000060  e8 00 00 00 00 89 c6 48  8d 05 00 00 00 00 48 89  |.......H......H.|
00000070  c7 b8 00 00 00 00 e8 00  00 00 00 b8 00 00 00 00  |................|
00000080  5d c3 25 64 0a 00 00 47  43 43 3a 20 28 55 62 75  |].%d...GCC: (Ubu|
```

```
00000090  6e 74 75 20 31 31 2e 34  2e 30 2d 31 75 62 75 6e  |ntu 11.4.0-1ubun|
000000a0  74 75 31 7e 32 32 2e 30  34 29 20 31 31 2e 34 2e  |tu1~22.04) 11.4.|
000000b0  30 00 00 00 00 00 00 00  04 00 00 00 10 00 00 00  |0...............|
000000c0  05 00 00 00 47 4e 55 00  02 00 00 c0 04 00 00 00  |....GNU.........|
...
```

Это мало что нам говорит. Для анализа объектных файлов есть специальные программы.

readelf покажет нам структуру файла и содержимое некоторых секций (sections). Выполните readelf main.o -a.

```
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:                              REL (Relocatable file)
...
Section Headers:
  [Nr] Name              Type             Address           Offset
       Size              EntSize          Flags  Link  Info  Align
  [ 0]                   NULL             0000000000000000  00000000
       0000000000000000  0000000000000000           0     0     0
  [ 1] .text             PROGBITS         0000000000000000  00000040
       0000000000000042  0000000000000000  AX       0     0     1
  [ 2] .rela.text        RELA             0000000000000000  000001f0
       0000000000000048  0000000000000018   I      11     1     8
  [ 3] .data             PROGBITS         0000000000000000  00000082
       0000000000000000  0000000000000000  WA       0     0     1
...
Relocation section '.rela.text' at offset 0x1f0 contains 3 entries:
  Offset          Info           Type           Sym. Value    Sym. Name + Addend
000000000021  000400000004 R_X86_64_PLT32    0000000000000000 foo - 4
00000000002a  000300000002 R_X86_64_PC32     0000000000000000 .rodata - 4
000000000037  000600000004 R_X86_64_PLT32    0000000000000000 printf - 4
...
Symbol table '.symtab' contains 7 entries:
   Num:    Value          Size Type    Bind   Vis      Ndx Name
     0: 0000000000000000     0 NOTYPE  LOCAL  DEFAULT  UND
     1: 0000000000000000     0 FILE    LOCAL  DEFAULT  ABS main.c
     2: 0000000000000000     0 SECTION LOCAL  DEFAULT    1 .text
     3: 0000000000000000     0 SECTION LOCAL  DEFAULT    5 .rodata
     4: 0000000000000000    19 FUNC    GLOBAL DEFAULT    1 foo
     5: 0000000000000013    47 FUNC    GLOBAL DEFAULT    1 main
```

```
    6: 0000000000000000     0 NOTYPE  GLOBAL DEFAULT  UND printf
...
```

Мы изучим это содержимое позже.

objdump покажет нам машинный код на языке ассемблера. Выполните objdump main.o -d.

```
main.o:     file format elf64-x86-64


Disassembly of section .text:

0000000000000000 <foo>:
   0:   f3 0f 1e fa             endbr64
   4:   55                      push   %rbp
   5:   48 89 e5                mov    %rsp,%rbp
   8:   89 7d fc                mov    %edi,-0x4(%rbp)
   b:   8b 45 fc                mov    -0x4(%rbp),%eax
   e:   83 c0 01                add    $0x1,%eax
  11:   5d                      pop    %rbp
  12:   c3                      ret

0000000000000013 <main>:
  13:   f3 0f 1e fa             endbr64
  17:   55                      push   %rbp
  18:   48 89 e5                mov    %rsp,%rbp
  1b:   bf 02 00 00 00          mov    $0x2,%edi
  20:   e8 00 00 00 00          call   25 <main+0x12>
  25:   89 c6                   mov    %eax,%esi
  27:   48 8d 05 00 00 00 00    lea    0x0(%rip),%rax        # 2e <main+0x1b>
  2e:   48 89 c7                mov    %rax,%rdi
  31:   b8 00 00 00 00          mov    $0x0,%eax
  36:   e8 00 00 00 00          call   3b <main+0x28>
  3b:   b8 00 00 00 00          mov    $0x0,%eax
  40:   5d                      pop    %rbp
  41:   c3                      ret
```

Здесь объявлены две функции. В <> скобках показаны названия функций. Сравните их содержимое с результатом выполнения команды gcc main.c -S.

3.1.5

main.c

```
#include "foo.h"

int main() {
```

```
    printf("%d\n", foo(2));
    return 0;
}
```

foo.h

```
int foo(int x) {
    return x + 1;
}
```

Компоновка объединяет несколько объектных файлов, подставляет реализации функций, формирует сегменты (segments) из секций в объектных файлах. В результате мы получаем исполняемый файл (но можно получить и, например, shared object), который на ОС Linux не имеет расширения, а его формат называется elf; на ОС Windows имеет расширение .exe, а его формат называется PE (Portable Executable).

Выполните gcc main.c, чтобы выполнить компоновку. Выходной файл не состоит из читаемого текста. Рассматривать его мы не будем. Вместо этого выполним readelf a.out -a. (По умолчанию, выходной исполняемый файл называется a.out. Вы можете задать название явно с помощью флага -o: gcc main.c -o program.)

```
...
  Type:                              DYN (Position-Independent Executable file)
  Machine:                           Advanced Micro Devices X86-64
  Version:                           0x1
  Entry point address:               0x1060
...
Program Headers:
  Type           Offset             VirtAddr           PhysAddr
                 FileSiz            MemSiz             Flags  Align
  PHDR           0x0000000000000040 0x0000000000000040 0x0000000000000040
                 0x00000000000002d8 0x00000000000002d8  R      0x8
  INTERP         0x0000000000000318 0x0000000000000318 0x0000000000000318
                 0x000000000000001c 0x000000000000001c  R      0x1
      [Requesting program interpreter: /lib64/ld-linux-x86-64.so.2]
  LOAD           0x0000000000000000 0x0000000000000000 0x0000000000000000
                 0x0000000000000628 0x0000000000000628  R      0x1000
...
 Section to Segment mapping:
  Segment Sections...
   00
   01     .interp
   02     .interp .note.gnu.property .note.gnu.build-id .note.ABI-tag .gnu.hash .dynsym .dynstr .gnu.version .gnu.vers
   03     .init .plt .plt.got .plt.sec .text .fini
   04     .rodata .eh_frame_hdr .eh_frame
   05     .init_array .fini_array .dynamic .got .data .bss
```

```
06     .dynamic
07     .note.gnu.property
08     .note.gnu.build-id .note.ABI-tag
09     .note.gnu.property
10     .eh_frame_hdr
11
12     .init_array .fini_array .dynamic .got
```

Структура исполняемого файла похожа на структуру объектного файла. Однако здесь помимо секций у нас присутствуют и сегменты (по какой то причине, program headers это section headers). Обратите внимание на таблицу внизу, которая показывает, из каких секций состоит каждый сегмент. Например, сегмент под номером 01 (в первой таблице мы видим, что это сегмент с названием INTERP (по какой-то причине, они не пронумерованы, нумерация идет с 00)) состоит из одной секции .interp. То, как происходит это сопоставление, написано в linker script-е. gcc имеет стандартный linker script, но мы можем написать свой (и мы это сделаем).

Выполним objdump a.out -d.

```
...
Disassembly of section .init:

0000000000001000 <_init>:
    1000:   f3 0f 1e fa             endbr64
    1004:   48 83 ec 08               sub    $0x8,%rsp
   1008:  48 8b 05 d9 2f 00 00   mov    0x2fd9(%rip),%rax      # 3fe8 <__gmon_start__@Base>
    100f:   48 85 c0                  test   %rax,%rax
    1012:   74 02                     je     1016 <_init+0x16>
    1014:   ff d0                     call   *%rax
    1016:   48 83 c4 08               add    $0x8,%rsp
    101a:   c3                        ret
...
Disassembly of section .text:

0000000000001060 <_start>:
    1060:   f3 0f 1e fa             endbr64
    1064:   31 ed                     xor    %ebp,%ebp
    1066:   49 89 d1                  mov    %rdx,%r9
    1069:   5e                        pop    %rsi
    106a:   48 89 e2                  mov    %rsp,%rdx
    106d:   48 83 e4 f0               and    $0xfffffffffffffff0,%rsp
    1071:   50                        push   %rax
    1072:   54                        push   %rsp
    1073:   45 31 c0                  xor    %r8d,%r8d
    1076:   31 c9                     xor    %ecx,%ecx
    1078:   48 8d 3d dd 00 00 00     lea    0xdd(%rip),%rdi        # 115c <main>
```

```
  107f:  ff 15 53 2f 00 00      call  *0x2f53(%rip)      # 3fd8 <__libc_start_main@GLIBC_2.34>
  1085:  f4                     hlt
  1086:  66 2e 0f 1f 84 00 00   cs nopw 0x0(%rax,%rax,1)
  108d:  00 00 00
...
0000000000001149 <foo>:
  1149:  f3 0f 1e fa            endbr64
  114d:  55                     push   %rbp
  114e:  48 89 e5               mov    %rsp,%rbp
  1151:  89 7d fc               mov    %edi,-0x4(%rbp)
  1154:  8b 45 fc               mov    -0x4(%rbp),%eax
  1157:  83 c0 01               add    $0x1,%eax
  115a:  5d                     pop    %rbp
  115b:  c3                     ret

000000000000115c <main>:
  115c:  f3 0f 1e fa            endbr64
  1160:  55                     push   %rbp
  1161:  48 89 e5               mov    %rsp,%rbp
  1164:  bf 02 00 00 00         mov    $0x2,%edi
  1169:  e8 db ff ff ff         call   1149 <foo>
  116e:  89 c6                  mov    %eax,%esi
  1170:  48 8d 05 8d 0e 00 00   lea    0xe8d(%rip),%rax      # 2004 <_IO_stdin_used+0x4>
  1177:  48 89 c7               mov    %rax,%rdi
  117a:  b8 00 00 00 00         mov    $0x0,%eax
  117f:  e8 cc fe ff ff         call   1050 <printf@plt>
  1184:  b8 00 00 00 00         mov    $0x0,%eax
  1189:  5d                     pop    %rbp
  118a:  c3                     ret
...
```

Количество кода резко увеличилось. Даже в секции .text, в которой раньше находились наши две функции, теперь присутствует множество новых функций. Все они были сгенерированы в результате выполнения linker script-а.

Обратите внимание на значение Entry point address: 0x1060 в начале вывода программы readelf. Это адрес инструкции, с которой начнется выполнение программы. Попобуйте найти соответствующий адрес в выводе программы objdump. Это не функция main!

3.1.6

Вспомним строку Entry point address: 0x1060 из вывода программы readelf на исполняемый файл. Здесь имеется ввиду адрес в оперативной памяти (RAM). Значит ли это, что программа этой строкой уже решила, что она будет находится именно в этом месте памяти? А что будет, если мы запустим её дважды — как удовлетворить условие для обоих процессов?

3.2. Многомодульные программы

3.2.1

main.c

```
int foo(int);

int main() {
    printf("%d\n", foo(2));
    return 0;
}
```

В этой программе объявлена функция foo. Но её реализация отсутствует. Попробуем выполнить компиляцию без компоновки: gcc main.c -c. Компиляция завершена успешно и мы получили объектный файл main.o!

Если же мы попытаемся выполнить компоновку gcc main.c, то получим ошибку:

```
/usr/bin/ld: /tmp/cchwCA2L.o: in function ‘main’:
main.c:(.text+0xe): undefined reference to ‘foo’
collect2: error: ld returned 1 exit status
```

Задача компоновщика подставить реализации функций из других модулей, которые мы ему подали. Мы подали компоновщику единственный модуль, в котором он не нашел реализацию функции foo.

Как же выглядит функция foo в объектном файле, когда её реализация отсутствует? В инструкциях вызова этой функции не указывается её адрес (ведь он неизвестен). А информация о том, что этот адрес должен быть подставлен на этапе компоновки, хранится в секциях, в названии которых есть подстрока rel (проверьте, есть ли такие секции в вашем объектном файле).

Реализуем функцию foo в другом модуле.

foo.c

```
int foo(int x) {
    return x + 1;
}
```

Если мы попытаемся скомпилировать это в исполняемый файл, то получим ошибку undefined reference to `main' . Скомпилируем в объектный файл gcc foo.c -c.

Чтобы выполнить компоновку обоих объектных файлов, просто напишем их всех при вызове gcc: gcc main.o foo.o. Полученный исполняемый файл можно запустить.

3.2.2

В предыдущем примере мы, чтобы иметь возможность вызывать функцию foo в модуле main.c, объявили её: int foo(int);. Делать это таким способом не удобно. Для каждого .c файла создают заголовочный (header) .h файл, в котором записаны объявления функций, и который мы добавляем с помощью #include, чтобы получить эти объявления.

main.c

```
#include <stdio.h>
#include "foo.h"

int main() {
    printf("%d\n", foo(2));
    return 0;
}
```

foo.c

```
int foo(int x) {
    return x + 1;
}
```

foo.h

```
int foo(int x);
```

Скомпилировать такой проект можно так:

```
gcc main.c -c
gcc foo.c -c
gcc main.o foo.o
```

, либо так: gcc main.c foo.c

Обычно, внешние библиотеки распространяются только в виде заголовочных и объектных файлов; без исходного кода реализаций функций. (При этом объектные файлы могут быть объединены в архивы и/или быть shared.) При этом сами реализации функций могут быть и не на языке C (мы это сделаем позже).

Но как компилятор может удостовериться в том, что сигнатура функции в заголовочном файле соответствует её сигнатуре в реализации, когда компилятор видит только объектный файл? (Вам придется пока поверить мне, что общих способов узнать сигнатуру функции по объектному файлу нет.) Ответ простой — никак!

Изменим только файл foo.c

```
void foo(int x) {
```

}

Это успешно скомпилируется и скомпонуется, а значение, которое printf выведет, не имеет отношения к функции foo.

3.2.3

Разрешается объявлять функции в программе многократно, а определять их только единожды. По хорошему, реализации функций должны быть только в .c файлах (хотя в C++ есть способы делать это и в .h файлах).

Однако, в заголовочных файлах мы также объявляем структуры (ведь, если они используются другими модулями, они должны знать содержимое структуры). Структуры можно объявлять лишь единожды, что может вызвать проблемы.

foo.c

```
#include "boo.h"

int foo(int x) {
    if (x == 0)
        return x + 1;
    else
        return x + boo(x - 1);
}
```

foo.h

```
struct foo {
    int x;
};
```

boo.c

```
#include "boo.h"

void add(struct boo *b) {
    b->f.x++;
}
```

boo.h

```
#include "foo.h"

struct boo {
    struct foo f;
};

void add(struct boo *b);
```

main.c

```
#include "foo.h"
#include "boo.h"

int main() {
    struct foo f;
    struct boo b;
    b.f.x = 3;
    add(&b);
    return 0;
}
```

В этой программе в модуле main.c структура foo объявлена дважды, что приводит к ошибке. Проблема в том, что и модулю main.c, и модулю boo.c необходима эта структура. Мы не можем убрать #include "foo.h" из boo.h, так как тогда в модуле boo.c не будет определена эта структура.

Данную проблему решают с помощью header guard-a (или include guard-a). Это простая конструкция для препроцессора.

```
#ifndef FOO
#define FOO

struct foo {
    int x;
};

#endif
```

Напомню, что препроцессинг происходит независимо для каждого модуля, поэтому содержимое этого if-a будет добавлено лишь по одному разу в каждый из модулей.

Ещё один случай, при котором пригодится header guard: структура foo содержит указатель на boo, а структура boo содержит указатель на foo. В таком случае у нас будет два header файла, которые include друг-друга. Попробуйте проверить, что произойдет в таком случае при отсутствии header guard-a.

Иногда, вместо такой конструкции в самое начало файла вставляют такую строку: #pragma once, однако она есть не во всех компиляторах.

3.2.4

Пусть мы хотим написать библиотеку, которая будет состоять из нескольких файлов, а соответственно, будет компилироваться в несколько объектных файлов. Хоть мы и можем написать под них общий заголовочный файл, пользователю всё равно придется добавить множество наших объектных

файлов. Эту проблему можно решить, объединив наши объектные файлы в .a архив с помощью программы ar, что часто называют статической библиотекой.

Пусть наша библиотека выглядит так:

foo.c

```
int foo(int x) {
    return x + 1;
}
```

foo.h

```
int foo(int x);
```

boo.c

```
int boo(int x) {
    return x + 2;
}
```

boo.h

```
int boo(int x);
```

Скомпилируем файлы нашей библиотеки в объектные файлы, а затем создадим архив с помощью программы ar:

```
gcc foo.c -c
gcc boo.c -c
ar -rc libfboo.a foo.o boo.o
```

Флаг -r означает добавить указанные файлы в архив, заместив ими файлы с теми же названиями. Флаг -c означает создать архив, если его нет.

Файл библиотеки обязан начинаться с lib.

Пусть теперь пользователь хочет воспользоваться нашей библиотекой.

main.c

```
#include <stdio.h>
#include "foo.h"
#include "boo.h"

int main() {
    printf("%d\n", foo(2) + boo(2));
    return 0;
}
```

Тогда для выполнения компиляции ему необходимо будет упомянуть только архив:

```
gcc main.c -L. -lfboo
```

-L. означает искать файлы статических библиотек в директории . (то есть, в текущей). -lfboo означает прикомпоновать эту библиотеку. Обратите внимание на странный формат названий: libfboo.a -> lfboo.

IDE Visual Studio имеет свой формат для статических библиотек — .lib, в то время как MinGW имеет формат .a.

3.2.5

Shared object — это объектные файлы, которые прикомпоновываются к программам при их запуске. Они используются для оптимизации места на диске, так как их могут использовать различные программы, не повторяя реализации тех же функций. Часто shared objects называют динамическими библиотеками.

Оформим пример из предыдущего шага в виде динамической библиотеки. Скомпилируем библиотеку:

```
gcc foo.c boo.c -c
gcc -shared -o libfboo.so foo.c boo.c
```

Во втором вызове gcc мы, с помощью флага -shared, создаём shared object.

Выполним readelf libfboo.so -a.

```
...
  Type:                              DYN (Shared object file)
...
```

Полученный объектный файл пользователь может положить в директорию со своей программой и попросить gcc искать его там. Но это неправильно. Для того, чтобы библиотека была shared, она должна находиться в месте, где глобально хранятся все shared objects (в директории проекта должны быть только заголовочные файлы). Скопируем файл туда:

```
cp libfboo.so /usr/lib
```

Теперь пользователь может скомпилировать свою программу:

```
gcc main.c -lfboo
```

Чтобы посмотреть, какие shared objects требует исполняемый файл, введём ldd a.out.

```
    linux-vdso.so.1 (0x00007ffc58396000)
    libfboo.so => /lib/libfboo.so (0x00007171575e0000)
    libc.so.6 => /lib/x86_64-linux-gnu/libc.so.6 (0x0000717157200000)
    /lib64/ld-linux-x86-64.so.2 (0x0000717157606000)
```

Помимо libfboo.so здесь присутствует, например, libc.so.6.

Когда мы запускаем программу, загрузчик ищет shared objects:

- В директориях /lib* и /usr/lib*
- В директориях в environmental variable LD_LIBRARY_PATH
- В директориях в rpath, который пишется в исполняемый файл в процессе компиляции с помощью флага -rpath

В ОС Windows формат динамических библиотек — .dll, и его использование выглядит значительно менее приятно, чем использование .so на Unix, так как требует использование директив компилятора. (Можете изучить пример: https://github.com/TransmissionZero/MinGW-DLL-Example)

3.2.6

В современных системных языках отсутствуют заголовочные файлы. Импорт модуля в них обозначает обычно лишь добавление сигнатур функций. Есть ли реальная необходимость в заголовочных файлах?

3.3. GNU make

3.3.1

foo.c

```
int foo(int x) {
    return x + 1;
}
```

foo.h

```
int foo(int x);
```

boo.c

```
int boo(int x) {
    return x + 2;
}
```

boo.h

```
int boo(int x);
```

main.c

```
#include <stdio.h>
#include "foo.h"
#include "boo.h"

int main() {
    printf("%d\n", foo(2) + boo(2));
    return 0;
}
```

Продолжим работать с той же программой. Можно сделать вывод, что процесс сборки такой программы уже нетривиальный, и мы тратим много времени на ввод повторяющегося набора команд. Чтобы решить эту проблему, мы можем написать все команды в один shell скрипт, и запускать его.

build.sh

```
rm foo.o boo.o main.o a.out
gcc foo.c -c
gcc boo.c -c
gcc main.c -c
gcc foo.o boo.o main.o
```

Все команды, кроме первой, можно объединить в одну, но в целях наглядности проблемы я намеренно напишу их так.

Обратите внимание на первую команду: удаление всех файлов, полученных в результате компиляции. Зачем это необходимо? Если мы допустим ошибку компиляции в одном из модулей, то соответствующий объектный файл не сформируется. Однако старый не будет удален, и скрипт продолжит выполняться. В результате мы получим либо ошибку компоновки, либо скомпонованную программу, один из модулей в которой устаревший.

Обратите внимание на такую проблему. Пусть мы изменили реализацию функции foo (но не её сигнатуру). Конечно, модуль foo следует перекомпилировать. Но нет смысла в перекомпиляции других модулей, ведь они никак не изменились.

Обратите внимание на строки 2-4 скрипта — их выполнение является независимым, и мы можем их выполнить одновременно. Сделать это с помощью shell скрипта непросто.

Пусть у нас программа является составной. Например, у нас есть компилятор и его стандартная библиотека. Скорее всего компилятору понадобиться стандартная библиотека, но в то же время, если мы хотим собрать стандартную библиотеку, нам не требуется собирать компилятор.

Итак, мы сформировали несколько проблем, которые решает программа make:

1. Выполнение этапа сборки только в том случае, если её результат устарел.
2. Автоматическое распараллеливание этапов сборки.
3. Разделение сборки на этапы и автоматическое рекурсивное определение необходимых этапов для сборки текущего этапа.

3.3.2

В большинстве дистрибутивов программа make отсутствует. Поставьте её. Из-за особенности её названия, более разумным будет использовать GNU make как ключевую строку.

Начнём с простого make-файла. Для наглядности будем компилировать модули foo и boo с статическую библиотеку.

Makefile

```
lib:
    gcc foo.c -c
    gcc boo.c -c
    ar -rc libfboo.a foo.o boo.o


program:
    gcc foo.c -c
    gcc boo.c -c
    ar -rc libfboo.a foo.o boo.o
    gcc main.c -L. -lfboo
```

Здесь определены два рецепта: lib и program. Чтобы выполнить их, необходимо ввести make lib или make program соответственно. Первый рецепт собирает только библиотеку (пусть, например, пользователю может пригодиться только она), а второй и библиотеку, и саму программу.

Важно: в make-файлах требуется использовать символы табуляции, вместо пробелов. Добавьте соответствующую настройку в ваш text editor, если требуется (VS Code, например, сам ставит табуляции в make-файлах).

Содержимое рецептов — это почти обычные shell команды. Однако они запускаются не в одной shell-сессии, а в разных, что иногда важно.

Мы можем упростить наш make-файл: рецепт сборки программы включает в себя содержимое рецепта сборки библиотеки. Мы можем в требование одного рецепта вписать название другого рецепта.

```
lib:
    gcc foo.c -c
    gcc boo.c -c
    ar -rc libfboo.a foo.o boo.o


program: lib
    gcc main.c -L. -lfboo
```

В этом случае, перед выполнением рецепта program будет выполнен рецепт lib. Важно отметить, что рецепт lib будет выполнен в любом случае и полностью, а значит, экономии ресурсов мы пока не добились.

Добавим рецепт на удаление временных файлов:

...
clean:
    rm libfboo.a foo.o boo.o main.o a.out

Его не следует добавлять в зависимости другим рецептам, так как удаление всех файлов сделает необходимой полную перекомпиляцию.

Обратите внимание, что make является ленивым языком. (Можете теперь рассказывать программистам на haskell-е, что тоже в некотором смысле пишете на ленивом языке.)

3.3.3

Заменим название рецепта lib на libfboo.a.

```
libfboo.a:
    gcc foo.c -c
    gcc boo.c -c
    ar -rc libfboo.a foo.o boo.o

program: libfboo.a
    gcc main.c -L. -lfboo
...
```

На самом деле, lib и libfboo.a в этом месте обозначает не просто название рецепта, а файл, который эти рецепты производят. Конечно, на прошлом шаге рецепт lib не производил файл lib, и так делать разрешено. Теперь же, когда файл в процессе выполнения рецепта производится, логика его выполнения усложняется. Рецепт выполняется только если его целевой файл отсутствует, либо один из файлов, от которых рецепт зависит, стал новее его целевого файла (это мы увидим чуть позже).

Выполните make program несколько раз. Вы увидите, что инструкции в рецепте libfboo.a были выполнены лишь при первой сборке, так как уже после неё файл libfboo.a был создан.

Таким образом мы впервые добились того, что перекомпиляция происходит только для обновленных модулей. Сделаем то же самое и для итогового файла a.out, так как он по прежнему собирается в любом случае.

```
libfboo.a:
    gcc foo.c -c
    gcc boo.c -c
    ar -rc libfboo.a foo.o boo.o

a.out: libfboo.a
    gcc main.c -L. -lfboo

program: a.out
```

...

Теперь, при втором вызове make program вы получите сообщение: make: Nothing to be done for 'program'.. Обратите внимание, что инструкции в рецепте program исполняются, ведь файла program нет (можете это проверить, вставив в рецепт любые инструкции). Но рецепт a.out не был выполнен, так как файл a.out моложе всех зависимостей — единственного файла libfboo.a.

Изменим файл libfboo.a. Здесь есть проблема: его пересборка на make libfboo.a будет выполнена только в том случае, если файла libfboo.a нет, ведь мы не написали для этого рецепта никаких зависимостей (сейчас мы это исправим). Поэтому просто удалим libfboo.a и сделаем make libfboo.a. Теперь при вызове make program будет выполнена перекомпиляция a.out.

Добавим зависимости для рецепта libfboo.a.

```
libfboo.a: foo.c boo.c
    gcc foo.c -c
    gcc boo.c -c
    ar -rc libfboo.a foo.o boo.o
...
```

Теперь при изменения файла foo.c или boo.c будет выполнена пересборка libfboo.a на команду make libfboo.a. Но, более того, будет выполнена пересборка libfboo.a и на команду make program. Это означает, что make проверяет необходимость пересборки не только в текущей точке, но и по всему дереву.

3.3.4

Попробуем добиться распараллеливания. Мы хотим, чтобы компиляция модулей foo и boo велась параллельно. Сделать это легко.

```
foo.o: foo.c
    gcc foo.c -c

boo.o: boo.c
    gcc boo.c -c

libfboo.a: foo.o boo.o
    ar -rc libfboo.a foo.o boo.o

a.out: libfboo.a
    gcc main.c -L. -lfboo

program: a.out
...
```

Сделаем новые рецепты на компиляцию каждого их этих модулей. Пусть

сборка библиотеки зависит от объектных файлов этих модулей. Теперь make будет распараллеливать выполнение рецептов foo.o и boo.o, так как для них обоих удовлетворены условия. Чтобы выполнить сборку с помощью двух процессов, введите make program -j 2.

Для того, чтобы удостовериться, что сборка действительно идёт параллельно, напишем такой make-файл.

```
foo.o: foo.c
    sleep 3
    gcc foo.c -c

boo.o: boo.c
    sleep 3
    gcc boo.c -c
...
```

Обе паузы при запуске sleep пройдут одновременно.

Представьте теперь, что у вас много модулей. Писать для каждого из них такой рецепт долго и усложняет чтение make-файла. make имеет способ решения этого.

```
%.o: %.c
    gcc $< -c

libfboo.a: foo.o boo.o
    ar -rc libfboo.a $^

a.out: libfboo.a
    gcc main.c -L. -lfboo

program: a.out
...
```

Посмотрите на первый "универсальный" рецепт. Он означает следующее: для того, чтобы собрать файл с названием xxx.o необходим файл xxx.c. Чтобы иметь доступ к необходимому файлу xxx.c в самом скрипте, необходимо использовать оператор $<.

Посмотрите на рецепт libfboo.a. Вместо оператора $^, будут подставлены все требования, то есть, foo.o boo.o. Если бы мы написали оператор $^, то было бы подставлено только первое требование: foo.o.

Существует ещё один оператор: $@, вместо которого подставляется файл-цель. В примере с рецептом xxx.o вместо оператора $@ будет подставлено xxx.o.

3.3.5

Мы добились поставленныъ целей с помощью make. Тем не менее продолжим улучшать читаемость и организацию make-скриптов.

```
OBJS=foo.o boo.o

%.o: %.c
    gcc $< -c

libfboo.a: $(OBJS)
    ar -rc libfboo.a $^

a.out: libfboo.a
    gcc main.c -L. -lfboo

program: a.out
...
```

Мы можем вынести список целевых объектных файлов в отдельную переменную, и писать её, вместо файлов. (Важно: это не environmental variables, хоть и их синтаксис похож. Они не будут проталкиваться в программы, которые вы запускаете.)

Пойдем дальше. Если мы перенесем файлы foo.c и boo.c в отдельную директорию, то мы можем определять целевые файлы автоматически по контенту этой директории. Тогда, когда мы добавим в директорию (и, соответственно, в библиотеку) новый файл, нам не придется изменять make-файл. Перенесем foo.c и boo.c в директорию lib.

Для определения списка файлов мы можем написать что-то вроде SRCS=$(shell ls lib). (Оператор shell запускает программу и возвращает её вывод. Присутствует не во всех программах make.) Вместо этого воспользуемся оператором wildcard.

```
SRCS=$(wildcard lib/*.c)
```

Смысл этого оператора очевиден. В результате его выполнения, переменная SRCS будет равна lib/foo.c lib/boo.c. Для удобства вывода значений переменных можно воспользоваться оператором info.

```
$(info $(SRCS))
```

Однако, нам необходимы не исходные .c файлы, а объектные файлы. Можно снова пошаманить со строками, а можно воспользоваться оператором patsubst.

```
OBJS=$(patsubst lib/%.c, %.o, $(SRCS))
```

Смысл этого оператора тоже почти очевиден. Вы пробегаемся по каждому слову в третьем аргументе $(SRCS), и преобразовываем каждое слово из

формата lib/xxx.c в xxx.o.

Обратите внимание, что в результате такого вызова OBJS=$(patsubst %.c, %.o, $(SRCS)) значение OBJS будет lib/boo.o lib/foo.o, что немного не то, что мы хотим.

Обратите внимание на то, что переменная SRCS, кроме формирования переменной OBJS, больше ни на что не влияет, и в рецепте %.o в требованиях теперь находятся несуществующие файлы (например, foo.o требует foo.c, когда его настоящее имя теперь lib/foo.c). Заменим требование на lib/%.c.

Наш итоговый make-файл на текущий момент:

```
SRCS=$(wildcard lib/*.c)
OBJS=$(patsubst lib/%.c, %.o, $(SRCS))

%.o: lib/%.c
    gcc $< -c

libfboo.a: $(OBJS)
    ar -rc libfboo.a $^

a.out: libfboo.a
    gcc main.c -L. -lfboo

program: a.out
...
```

3.3.6

На предыдущем шаге мы начали изменять организацию нашего проекта в лучшую сторону, когда переместили файлы библиотеки lfboo в отдельную директорию. Выполним теперь правильную организацию проекта полностью. К сожалению, в языке C нет стандарта организации проекта. Я покажу один из способов. Для большей наглядности в некоторых местах я добавлю больше файлов и модулей.

Начнем с основной программы. Её .c файлы будут лежать в директории src, а .h файлы будут лежать в директории include. Мы добавим туда ещё один модуль goo.

src/goo.c

```
int goo(int x) {
    return x + 4;
}
```

include/goo.h

```
int goo(int x);
```

src/main.c

```
#include <stdio.h>
#include <fboo/fboo.h>
#include <doo/doo.h>
#include <goo.h>

int main() {
    printf("%d\n", foo(2) + boo(2) + doo(2) + goo(2));
    return 0;
}
```

Создадим директорию build, в которую будем складировать объектные файлы и финальный исполняемый файл. В результате компиляции файла xxx/src/yyy.c будет получаться файл build/xxx/yyy.o. (Например, src/main.c -> build/main.o, lib/fboo/src/foo.c -> build/fboo/foo.o.)

Библиотеки мы будем хранить в директории lib. В директории каждой библиотеки также будут директории src и include. Кроме того, библиотеки будут иметь публичный заголовочный файл (в данном случае lib/fboo/fboo.h) в корне директории библиотеки, на который мы и будем ссылаться из основной программы.

lib/fboo/src/foo.c

```
int foo(int x) {
    return x + 1;
}
```

lib/fboo/src/boo.c

```
#include <box.h>

int boo(int x) {
    return x + 2;
}
```

lib/fboo/include/box.h

```
struct box {
    int x;
};
```

lib/fboo/fboo.h

```
int foo(int x);
int boo(int x);
```

Для удобства в директории каждой библиотеки будет свой Makefile.

lib/fboo/Makefile

```
SRCS=$(wildcard src/*.c)
OBJS=$(patsubst src/%.c, $(BUILD_DIR)/fboo/%.o, $(SRCS))
CFLAGS+=-Iinclude

$(BUILD_DIR)/fboo/%.o: src/%.c prepare
    gcc $(CFLAGS) $< -c -o $@

$(BUILD_DIR)/fboo/libfboo.a: $(OBJS)
    ar -rc $@ $^

fboo: $(BUILD_DIR)/fboo/libfboo.a

prepare:
    mkdir -p $(BUILD_DIR)/fboo

clean:
    rm -rf $(BUILD_DIR)/fboo
```

Здесь переменная $(BUILD_DIR) равна абсолютному пути директории build, и будет нам передана "верхним" make-файлом. Для того, чтобы уже сейчас проверить этот файл, мы можем добавить BUILD_DIR=../../build и вызвать make fboo из директории lib/fboo. Флаг -Iinclude нам необходим, чтобы файлы в директории lib/fboo/include были видимы. С помощью рецепта prepare мы создаём директорию build/fboo, так как gcc не создаёт её самостоятельно.

Организуем таким же образом библиотеку doo.

lib/doo/src/doo.c

```
int doo(int x) {
    return x + 3;
}
```

lib/doo/doo.h

```
int doo(int x);
```

lib/doo/Makefile

```
SRCS=$(wildcard src/*.c)
OBJS=$(patsubst src/%.c, $(BUILD_DIR)/doo/%.o, $(SRCS))
CFLAGS+=-Iinclude

$(BUILD_DIR)/doo/%.o: src/%.c prepare
    gcc $(CFLAGS) $< -c -o $@

$(BUILD_DIR)/doo/libdoo.a: $(OBJS)
    ar -rc $@ $^
```

```
doo: $(BUILD_DIR)/doo/libdoo.a
	echo $(BUILD_DIR)

prepare:
	mkdir -p $(BUILD_DIR)/doo

clean:
	rm -rf $(BUILD_DIR)/doo
```

Здесь также можно абстрагировать название модуля doo.

Напишем, наконец, главный Makefile

Makefile

```
BUILD_DIR=$(abspath build)
SRCS=$(wildcard src/*.c)
OBJS=$(patsubst src/%.c, $(BUILD_DIR)/%.o, $(SRCS))
LIBS=$(BUILD_DIR)/fboo/libfboo.a $(BUILD_DIR)/doo/libdoo.a
CFLAGS+=-Iinclude -Ilib

$(BUILD_DIR)/%.o: src/%.c prepare
	gcc $(CFLAGS) $< -c -o $@

$(BUILD_DIR)/fboo/libfboo.a:
	$(MAKE) -C lib/fboo fboo BUILD_DIR=$(BUILD_DIR)

$(BUILD_DIR)/doo/libdoo.a:
	$(MAKE) -C lib/doo doo BUILD_DIR=$(BUILD_DIR)

$(BUILD_DIR)/program: $(OBJS) $(LIBS)
	gcc $(OBJS) $(LIBS) -o $@

program: $(BUILD_DIR)/program

prepare:
	mkdir -p $(BUILD_DIR)

clean:
	rm -rf $(BUILD_DIR)
```

Для начала, мы создаём переменную BUILD_DIR, в которую записываем абсолютный путь директории build, который мы добываем с помощью функции abspath. К флагам мы добавляем флаг -Ilib, чтобы мы могли добавлять файлы из библиотек.

С помощью $(MAKE) -C dir recipe мы рекурсивно вызываем make в директории dir с рецептом recipe. Мы проталкиваем к дочернему make переменную BUILD_DIR с помощью BUILD_DIR=$(BUILD_DIR).

Для того, чтобы получить исполняемый файл build/program необходимо ввести make program (либо make dir, где dir это абсолютный путь до исполняемого файла, но так менее удобно).

В этом make-файле есть недостатки. Например, названия библиотек недостаточно абстрагированы. Я намеренно не стал этим усложнять пример.

3.3.7

Попробуйте найти недостатки в организации проекта, который показан на предыдущем шаге, и исправить их.

3.3.8

До сих пор мы рассматривали только зависимости от .c файлов. Пусть теперь мы изменили .h файл. В таком случае, необходима перекомпиляция всех модулей, которые его добавляют. К сожалению, нет адекватных общих способов определить это множество модулей. Исследуйте способы, как эта проблема может быть решена.

3.3.9

Проект: Программа со структурой

Напишите любую программу (полная свобода мысли) вместе с несколькими её библиотеками и систему сборки для неё. Используйте контент шестого шага в качестве образца.

4. Unix

4.1. Shell

4.1.1

Разберемся подробнее в том, что такое "терминал". Запустите терминал. У вас начнется сессия с командной оболочкой (shell). В зависимости от вашего дистрибутива, запускаемая оболочка может быть различной. Однако, скорее всего, в вашем дистрибутиве по умолчанию будет запущен bash. Существуют другие оболочки: например, древняя и более примитивная sh, или более продвинутая zsh. Начнем с sh. Введите sh в терминале, чтобы запустить его. Какие при этом происходят действия, мы узнаем немного позже.

Prompt (приглашение) в sh очень простое — $. Чтобы запустить локальную программу в любом shell, необходимо ввести ./program, где program — название исполняемого файла. Чтобы запустить глобальную программу, необходимо просто ввести название её исполняемого файла. Shell будет искать этот исполняемый файл во всех директориях, которые записаны в переменной PATH (об этом позже).

Чтобы узнать текущую директорию, в которой мы находимся, можно воспользоваться программой pwd.

```
$ pwd
/home/igor/test
```

Чтобы создать папку, можно воспользоваться программой mkdir.

```
$ mkdir dir
```

Чтобы посмотреть все файлы в текущей директории, можно воспользоваться программой ls.

```
$ ls
dir
```

Чтобы сменить директорию, воспользуйтесь командой cd.

```
$ cd dir
$ pwd
/home/igor/test/dir
```

Чтобы создать файл, можно воспользоваться программой touch.

```
$ touch file
$ ls
file
```

Чтобы редактировать файл в терминале, можно воспользоваться одним из текстовых редакторов: nano, vim, etc. (Их необходимо осваивать отдельно.)

Чтобы вывести содержимое файла в текстовом, можно воспользоваться программой cat.

```
$ cat file
A dog
```

Чтобы вывести содержимое файла в hex, можно воспользоваться программой hd.

```
00000000  41 20 64 6f 67 0a                              |A dog.|
00000006
```

Чтобы узнать расположение исполняемого файла программы, можно воспользоваться программой which.

```
$ which ls
/usr/bin/ls
$ which which
/usr/bin/which
$ which cd
```

Чтобы удалить файл, можно воспользоваться программой rm.

```
$ rm file
$ ls
```

Чтобы перейти не папку наверх, необходимо аргументом к команде cd написать ...

```
$ cd ..
```

Чтобы удалить папку, можно воспользоваться программой rmdir.

```
$ rmdir dir
$ ls
```

4.1.2

Чтобы "зависнуть" на некоторое время, можно воспользоваться программой sleep.

```
$ sleep 5
```

Можно выполнить процесс "на фоне" (background), то есть, shell не будет ждать завершения процесса, а продолжит выполнять команды. Для этого в конце команды следует написать &.

```
$ sleep 5 &
$ pwd
/home/igor/test
$
[1] + Done                    sleep 5
```

Здесь, после ввода первой команды я сразу же ввёл вторую команду. После ввода очередной команды shell уведомляет о завершенных фоновых командах.

Когда у вас выполняется программа не на фоне, вы можете нажать комбинацию Ctrl + Z. Данная комбинация ставит процесс на паузу и помещает его наверх специального буфера. Чтобы возобновить процесс с вершины буфера, введите команду fg (foreground).

```
$ sleep 5
^Z[1] + Stopped                 sleep 5
$ fg
sleep 5
```

Обратите внимание (при личном тесте), что между нажатием комбинации Ctrl + Z и введением команды fg, таймер продолжает идти (то есть, программа sleep игнорирует просьбу о паузе). Пока таймер не истёк, вы не можете вводить команды, так как команда выполняется не на фоне.

Чтобы возобновить программу из буфера на фоне, вместо команды fg введите команду bg (background).

```
$ sleep 5
^Z[1] + Stopped                 sleep 5
$ bg
[1] sleep 5
$
[1] + Done                      sleep 5
```

Обратите внимание (хотя это и не так важно), что буфер имеет форму стека (то есть берётся последний пришедший).

```
$ sleep 5
^Z[1] + Stopped                 sleep 5
$ sleep 6
^Z[2] + Stopped                 sleep 6
$ bg
[2] sleep 6
$ bg
[1] sleep 5
```

4.1.3

Запустим программу grep которая принимает как аргумент слово и ищет его в своём стандартном вводе (stdin). Как только она на очередной строке, ограниченной символом \n находит слово, как подстроку, она выводит всю строку.

Как только вы запустим программу, которая ожидает данные в стандартном вводе, мы увидем в терминале не prompt, а пустую строку. Каждая наша строка, которую мы завершаем нажатием enter-a, передаётся программе в её стандартный ввод. В свою очередь, программа может что-то выводить в свой стандартный вывод (stdout), и мы также его будем видеть.

```
$ grep a
aaa
aaa
bbb
aba
aba
```

Первую, третью и четвёртую строку ввёл я. Вторую и пятую строку вывода вывела программа grep. Всё, что мы ввели будет передаваться программе grep, пока она не закроет свой stdin, однако она его никогда не закроет. Чтобы нам закрыть его, мы должны нажать комбинацию Ctrl + D.

Мы пожем связать стандартый вывод программы с файлом. Тогда всё, что она выводит, окажется в файле. Для этого, следует к команде добавить > file.

```
$ grep a > file
```

```
aaa
bbb
aba
$ cat file
aaa
aba
```

При такой команде всё, что было до этого в файле, удаляется. Чтобы вместо этого выполнить дозапись в файл, следует вместо > написать >>.

Можно также связать стандартный ввод программы с файлом с помощью оператора <. Программа echo выводит в стандартный вывод единственный свой аргумент.

```
$ echo aaa > file
$ echo bbb >> file
$ echo aba >> file
$ grep a < file
aaa
aba
```

Можно запустить две программы и связать стандартный вывод одной со стандартным вводом в другой. Для этого необходимо воспользоваться оператором | (pipe).

```
$ cat file | grep a
aaa
aba
```

Stdin и stdout — это streams. (Данное слово, как и слова thread и flow переведены на русский язык, как поток. Причём, по смыслу эти три слова совсем не близки.) Есть ещё один частоиспользуемый stream — стандартный вывод ошибок (stderr). В него обычно пишут программы при нарушении правил общения с ними.

В терминале мы видим и stdout и stderr от выполняемой программы, и отличить их нельзя. Попробуем перенаправить stdout у этой команды.

```
$ ls dirrr > out
ls: cannot access 'dirrr': No such file or directory
$ cat out
```

Не перенаправилось. Значит, эту строку ls вывел в stderr. Чтобы перенаправить его, следует воспользоваться оператором 2>.

```
$ ls dirrr 2> out
$ cat out
ls: cannot access 'dirrr': No such file or directory
```

Чтобы перенаправить оба потока в один файл, следует написать так.

```
$ ls > out 2>&1
```

Чтобы в программе на C выполнить вывод в stderr, можно воспользоваться функцией fprintf, которая первым аргументом принимет дескриптор stream-a. Дескриптор stderr-a храниться в глобальной переменной stderr.

```
fprintf(stderr, "%d\n", 42);
```

Теперь мы можем понять, что когда мы в самом начале запустили sh, то мы начали с ним интерактивную сессию, такую же, как и с grep здесь. Мы можем её прервать комбинацией Ctrl + D.

4.1.4

Когда вы вводите команду cat file, вы запускаете программу cat и передаёте ей аргумент file. Посмотрим, как эти аргументы можно читать из программы на C.

```
#include <stdio.h>

int main(int argc, char **argv) {
    int i;
    for (i = 0; i < argc; i++) {
        printf("%s\n", argv[i]);
    }
    return 0;
}
```

На самом деле, в функцию main передаётся два аргумента — количество аргументов программы argc, и массив этих аргументов argv, которые являются строками. (На ещё более самом деле, в функцию main может передаваться аргумент envp, но он есть не на всех системах.) Благодаря том, как устроен application binary interface, мы можем просто не писать суффикс аргументов, и это не вызовет никаких проблем. Подробнее об этом мы узнаем позже.

Данная программа просто выводит аргументы. Потестируем её.

```
$ ./main
./main
$ /home/igor/test/main
/home/igor/test/main
$ ./main a -b c
./main
a
-b
c
```

Аргументы вида --help и -h называются флагами. Используются также аргументы вида --output=file. Сам shell никак не задаёт формат аргументов: вам необходимо обработать его самостоятельно.

Обратите внимание, что первым аргументом программы получают путь к своему исполняемому файлу. Когда мы будем запускать программы с помощью системного вызова exec, мы сможем нарушить это правило. В таком случае, некоторые программы могут отказаться работать.

Чтобы считать переменные среды (environmental variables), можно воспользоваться третьим аргументом envp.

```
#include <stdio.h>

int main(int argc, char **argv, char **envp) {
    int i;
    for (i = 0; envp[i] != NULL; i++) {
        printf("%s\n", envp[i]);
    }
    return 0;
}
```

Массив envp, в отличии от argv, является ноль-терминированным. (Напомню, что это по сути два способа хранения массивов: либо хранить отдельно длину массива, либо завершать его нулем.)

```
$ ./main
LESSOPEN=| /usr/bin/lesspipe %s
no_proxy=localhost,127.0.0.0/8,::1
USER=igor
LC_TIME=ru_RU.UTF-8
all_proxy=socks://127.0.0.1:1080/
XDG_SESSION_TYPE=x11
SHLVL=1
HOME=/home/igor
...
```

Мы можем установить значение переменной в sh с помощью оператора export. Чтобы получить значение переменной, нужно перед её названием написать символ $.

```
$ export DOG=Shepherd
$ echo $DOG
```

Переменные среды видны всем процессам-потомкам, то есть, в том числе тем, которые запущены в данном sh.

```
$ ./main | grep DOG
DOG=Shepherd
```

Один из других способов вывести переменные среды — это воспользоваться глобальной переменной environ.

```
#include <stdio.h>
```

```
extern char **environ;

int main(int argc, char **argv) {
    int i;
    for (i = 0; environ[i] != NULL; i++) {
        printf("%s\n", environ[i]);
    }
    return 0;
}
```

Ключевое слово extern достаёт на этапе компоновки переменную environ из другого модуля, так как эта переменная не определена в заголовочных файлах. Переменна environ определена где-то в файлах libc. Конечно, использование extern-а считается плохой практикой.

Чтобы установить значение переменной, можно воспользоваться функцией int setenv(const char *name, const char *value, int overwrite);. Если overwrite равен нулю, то она записывает value в name, только если name не определено. В противном случае, записывает всегда.

Чтобы прочитать значение переменной, можно воспользоваться функцией char *getenv(const char *name);. В случае, если переменная name отсутствует, функция возвратит NULL.

```
#include <stdio.h>
#include <stdlib.h>

int main(int argc, char **argv) {
    printf("%s\n", getenv("HOME"));
    setenv("HOME", "/bin", 0);
    printf("%s\n", getenv("HOME"));
    setenv("HOME", "/bin", 1);
    printf("%s\n", getenv("HOME"));
    printf("%d\n", getenv("HOM"));
    return 0;
}
```

```
$ ./main
/home/igor
/home/igor
/bin
0
```

4.1.5

Когда мы запускаем программу, запускается процесс, выполняющий эту программу. После завершения каждый процесс возвращает число. По хорошему, при успешном завершении процесс должен возвращать ноль, а при

неуспешном — не ноль. Узнать, какое число вернул последний процесс, мы можем, напечатав значение переменной $?.

```
$ pwd
/home/igor/test
$ echo $?
0
$ gcc
gcc: fatal error: no input files
compilation terminated.
$ echo $?
1
$ gcc main.c -o main
$ echo $?
0
```

Когда мы пишем shell-скрипты, мы так проверяем, был ли запуск программы успешен.

Чтобы выполнить несколько программ друг за другом, мы можем написать их на одной строке, разделив символом ;.

```
$ pwd ; ls
/home/igor/test
main  main.c
```

Чтобы выполнять программы из списка, пока одна из них не завершится успешно (вернёт 0), мы можем написать их на одной строке, разделив оператором ||.

```
$ pwd || ls
/home/igor/test
$ gcc || ls
gcc: fatal error: no input files
compilation terminated.
main  main.c
```

В первом случае программа pwd завершилась успешно, поэтому программа ls не запускалась.

Во втором случае программа gcc завершилась неуспешно, поэтому программа ls запустилась.

Чтобы выполнять программы из списка, пока они выполняются успешно, мы можем написать их на одной строке, разделив оператором &&. (Звучит, как оператор, который часто будет пригождаться.)

```
$ pwd && ls
/home/igor/test
main  main.c
$ gcc && ls
```

```
gcc: fatal error: no input files
compilation terminated.
```

В первом случае программа pwd завершилась успешно, поэтому программа ls запускалась.

Во втором случае программа gcc завершилась неуспешно, поэтому программа ls не запустилась.

Каждый процесс при запуске получает уникальный PID (process identifier). Когда мы запускаем процесс на фоне, в некоторых shell-ах нам выводится его PID (но не в sh). Его же мы можем прочитать из переменной $!. Пока мы мало что умеем делать с процессами. Мы можем завершить процесс с помощью программы kill.

```
$ sleep 100 &
$ echo $!
18865
$ kill $!
$
[1] + Terminated                sleep 100
```

Нетрудно догадаться, что код возврата — это то, что мы возвращаем из функции main.

```
int main() {
    return 123;
}
```

```
$ ./main
$ echo $?
123
```

4.1.6

Напишите простую программу, работающую аналогично программе grep. Выберите и реализуйте 1-3 её флага. Не следует реализовывать весь её функционал, так как он очень большой. Чтобы узнать флаги, которые принимает программа, введите флаг --help.

```
grep --help
```

4.1.7

В sh есть более сложные синтаксические конструкции. Например, так мы можем написать if-statement.

```
$ if gcc
> then
> echo 1
> else
```

```
> echo 2
> fi
gcc: fatal error: no input files
compilation terminated.
2
```

Заметьте, что пока мы не завершили if-statement ключевым словом fi, sh выводит особый prompt, который подсказывает нам, что мы вводим сейчас вводим сложную синтаксическую конструкцию. Так как gcc вернул не ноль, выполнилась ветка else.

Очевидно, что можно сохранить такой скрипт в отдельном файле, и просто подать этот файл на stdin к sh.

script

```
if gcc
then
    echo 1
else
    echo 2
fi
```

```
$ sh < script
gcc: fatal error: no input files
compilation terminated.
2
```

Но такой способ требует, чтобы мы сами указали обработчик скрипта. В Unix есть возможность указать обработчик скрипта прямо в файле. В таком случае, мы можем просто запустить скрипт, как исполняемый файл, и загрузчик программ запустит указанный в файле обработчик скрипта и передаст ему путь к скрипту.

script

```
#!/bin/sh
if gcc
then
    echo 1
else
    echo 2
fi
```

На первой строке написан Shebang. Если файл начинается с текстовых символов #! то загрузчик программ (не shell) запустит программу, путь к которой указан следом (в данном случае, /bin/sh), и передаст ей вторым аргументом путь к скрипту.

```
$ chmod +x script
$ ./script
```

```
gcc: fatal error: no input files
compilation terminated.
2
```

Нам необходимо разрешить запускать скрипт. Сделать это можно с помощью программы chmod.

Давайте напишем свой обработчик скриптов.

main.c

```
#include <stdio.h>

int main(int argc, char **argv) {
    int i;
    for (i = 0; i < argc; i++) {
        printf("%s\n", argv[i]);
    }
    return 0;
}
```

script

```
#!./main
```

Укажем путь к исполняемому файлу нашего обработчика: ./main.

```
$ ./script
./main
./script
```

Видим, что вторым аргументом указан путь к скрипту. Теперь мы можем в нашем обработчике открыть этот файл и что-то сделать.

4.1.8

Мы можем посмотреть список процессов-потомков с помощью программы ps.

```
$ sleep 5 &
$ ps
   PID TTY          TIME CMD
 30872 pts/1    00:00:00 bash
 71987 pts/1    00:00:00 sh
 72063 pts/1    00:00:00 sleep
 72070 pts/1    00:00:00 ps
```

Здесь мы видем bash, который был запущен у меня изначально; sh, который я запустил в нём; ps, который вывел этот текст; и sleep, запущенный ранее на фоне.

Мы можем проверить время работы процесса с помощью программы time.

Напишем вспомогательную программу.

```
int main() {
    int i;
    for (i = 0; i < 1000000000; i++);

    return 0;
}
```

```
$ time sleep 2
0.00user 0.00system 0:02.00elapsed 0%CPU (0avgtext+0avgdata 2048maxresident)k
0inputs+0outputs (0major+87minor)pagefaults 0swaps
$ time ./main
1.80user 0.00system 0:01.80elapsed 99%CPU (0avgtext+0avgdata 1152maxresident)k
0inputs+0outputs (0major+63minor)pagefaults 0swaps
```

По какой-то причине, вывод в sh не отформатирован. Обратите внимание, что всё время работы программы sleep приходится на system, в то время как всё время работы программы main приходится на user. Это связано с тем, что программа sleep просит процессор не выполнять её некоторое время, в то время как main всё это время полностью использует процессор.

Мы можем посмотреть информацию обо всех процессах с помощью программы top. Данная программа является интерактивной, то есть, она меняет текст в терминале. Чтобы выйти из программы, нажмите q. Мы можем получить мгновенный статус с помощью флага -n со значением 1. Проверим статус процесса sh. Нам понадобится флаг -b, чтобы top не выводил символы форматирования.

```
$ top -n 1 -b | grep sh
...
  30872 igor      20   0   11556   4776   3788 S   0,0   0,1   0:00.45 bash
...
  71987 igor      20   0    2892   1664   1664 S   0,0   0,0   0:00.00 sh
...
```

4.1.9

На sh-скриптах можно решать задачи. Так мы можем решить задачу A + B.

```
read a
read b
c=$(($a + $b))
echo $c
```

Изучите синтаксические конструкции в sh (они простые, хоть и очень странные). В sh есть только один тип данных — строка.

Напишите решение любой более сложной задачи. Например, задачи про брокера Василия из главы "Контроль потока".

4.1.10

Проект: Judge System

Напишите локальную тестирующую систему, работающую по аналогии с https://codeforces.com/ и подобными сайтами.

- В директории, отведенной под задачу должны быть:
  - Директория с тестами
  - Директория с ответами на тесты
- Пользователь запускает bash-скрипт, передавая ему исходный код своей программы-решения (выбор поддерживаемых языков неважен) и путь к директории задачи. bash-скрипт проходит по всем парам <тест, ответ на тест> и запускает решение (предварительно скомпилированное), подавая ему в stdin тест и сравнивая вывод в stdout с ответом на тест.
  - Если программа не скомпилировалась, пользователь должен получить вердикт "Compilation error"
  - Если программа использует больше X памяти, она должна прерваться, и пользователь должен получить вердикт "Memory limit exceeded".
  - Если программа использует больше Y времени, она должна быть прерванное, и пользователь должен получить вердикт "Time limit exceeded"
  - Если программа вернула не 0, пользователь должен получить вердикт "Runtime error"
  - Если вывод программы не совпал с ответом на тест, пользователь должен получить вердикт "Wrong answer"
  - В противном случае, пользовать должен получить вердикт "Accepted"

По желанию, можете изучить, что такое checkers (чекеры) и добавить их поддержку. Но это почти не усложнит логику.

4.2. Системные вызовы

4.2.1

Возможности наших программ сильно ограничены. Многие вещи, которые наши программы в прошлом делали, например, считывали ввод или создавали файлы, на самом деле они выполняли не самостоятельно. Программы могут лишь делать что-либо в своих сегментах (если у них есть на это права) и выполнять системные вызовы.

Во время существования программы её оперативная память сильно изолирована. Программа не может получить доступ к памяти других программ. Эта изоляция достигается за счет механизма paging-а. Каждый процесс имеет свою page directory, и, когда ОС переключает процесс, она также переключает и page directory. Подробнее об этом мы узнаем позже.

Системные вызовы — это примерно то же самое, что и функции. Их реализует ОС. Именно с помощью них мы и делаем такие действия, как взаимодействие с stream-ами, файлами, процессами. Самостоятельно вызвать системные вызовы из кода на C мы не можем — на нём просто нет соответствующих конструкций. Мы сможем самостоятельно вызывать их из кода на Assembly, но позже. Пока мы будем пользоваться функциями-обёртками.

Здесь указан список системных вызовов в ОС Linux на разных архитектурах. Набор системных вызовов во всех случаях почти одинаковый, меняется лишь порядок. https://chromium.googlesource.com/chromiumos/docs/+/master/constants/syscalls.md

Обратите внимание, что все ссылки в этой таблице ведут на документацию функций-обёрток в Linux-е. Многие из них вы скорее всего уже видели.

Важно понять, что механизм системных вызовов предоставляет архитектура (причём, все распространённые на данный момент архитектуры), но то, какие будут системные вызовы и как они будут реализованы, определяет ОС. Сравните системные вызовы в Linux с системными вызовами в Windows: https://j00ru.vexillium.org/syscalls/nt/64/

Начнём с простого системного вызова (СВ) fork. Данный СВ:

- не принимает никаких аргументов,
- создаёт новый процесс, полностью идентичный родительскому,
- возвращает родителю pid ребёнка, а ребёнку 0.

```
#include <stdio.h>
#include <unistd.h>

int main() {
    int x = fork();
    printf("%d\n", x);
}
```

В моём случае вывод:

```
19025
0
```

Ребёнок наследует stream-ы, и поэтому выводит в тот же терминал, что и родитель. Порядок вывода случайный. Попробуйте позапускать эту программу многократно.

Рассмотрим пример сложнее.

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>

int main() {
```

```
    int t = 2, *v = (int*)malloc(sizeof(int));
    *v = 2;
    int x = fork();
    t++;
    (*v)++;
    printf("%d %d %d\n", x, t, *v);
}
```

В моём случае вывод:

```
20561 3 3
0 3 3
```

Обратите внимание, что и переменная на стеке, и переменная на куче, в обоих процессах независимы. (Если бы это было не так, процесс, выполнивший вывод вторым, вывел бы 4.) Это показывает, что и стек, и куча, копируются в новый процесс. Также это показывает, что кучи не являются общими для всех процессов. (О том, как работают кучи, и что делает malloc, мы узнаем позже. Обратите внимание, что системного вызова malloc или подобного нет.)

4.2.2

Данная программа принимает вторым аргументом целое число и выводит некоторое количество чисел. Выведете количество выведенных чисел, как функцию от значения аргумента.

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>

int main(int argc, char **argv) {
    int n = atoi(argv[1]);
    int i;
    while (n > 0) {
        printf("%d\n", n);
        fflush(stdout);
        if (fork() == 0) {
            n -= 1;
        }
        else if(fork() == 0) {
            n -= 2;
        }
        else {
            break;
        }
    }
```

```
    return 0;
}
```

Функция fflush необходима, если вы будете выводить в файл. Функция printf выполняет буферизацию (то есть, копит текст, и только затем выполняет вывод), поэтому необходимо потребовать её выполнить вывод сейчас с помощью функции fflush.

4.2.3

СВ fork создаёт полностью идентичный процесс. Однако гораздо чаще мы хотим запускать новый процесс другой программы. С помощью системного вызова execve мы можем "загрузить" в процесс другую программу. Изучите аргументы этой функции в документации: https://man7.org/linux/man-pages/man2/execve.2.html . Нам необходимо передать путь к исполняемому файлу, переменные запуска и переменные среды. Пока не будем сильно задумываться об этом и рассмотрим такой пример.

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>

int main() {
    char *argv[] = {"/usr/bin/pwd", NULL};
    char *envp[] = {NULL};
    execve("/usr/bin/pwd", argv, envp);
    printf("Check\n");
    return 0;
}
```

В моём случае вывод:

/home/igor/test

В результате выполнения функции execve выполнение нашей программы прервалось, и вместо этого начала выполняться программа pwd. Проверьте, что если в первый аргумент ввести pwd, то функция не найдет программу и вернёт ошибку.

Обратите внимание, что мы передаём функции в качестве первого аргумента путь к ней. Мы здесь можем нарушить правило о том, что первым аргументом должен быть пусть к программе. На pwd это не влияет, но, например, программа rustc выводит такое:

error: unknown proxy name: 'xxx'; valid proxy names are 'rustc', 'rustdoc', 'cargo', 'rust-lldb', 'rust-gdb', 'rust-gdbgui', 'rls', 'cargo-clippy', 'clippy-driver', 'cargo-miri', 'rust-analyzer', 'rustfmt', 'cargo-fmt'

Напишем вспомогательную программу, которая просто будет выводить аргументы и переменные среды.

test.c

```
#include <stdio.h>

int main(int argc, char **argv, char **envp) {
    int i;
    for (i = 0; i < argc; i++) {
        printf("%s\n", argv[i]);
    }
    printf("__________________\n");
    for (i = 0; envp[i] != NULL; i++) {
        printf("%s\n", envp[i]);
    }
    return 0;
}
```

main.c

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>

int main() {
    char *argv[] = {"xxx", "yyy", NULL};
    char *envp[] = {"A=a", "B=b", "C=c", NULL};
    execve("./test", argv, envp);
    printf("Check\n");
    return 0;
}
```

Вывод:

```
xxx
yyy
__________________
A=a
B=b
C=c
```

Запущенная программа не получила переменные среды своего родителя. Нам необходимо передавать их самостоятельно. Для удобства существуют дополнительные функции-обёртки для execve. Здесь можно посмотреть их: https://www.opennet.ru/docs/RUS/linux_parallel/node8.html . Например, чтобы передать переменные среды, можно воспользоваться execvp.

Наконец, если мы хотим начать новый процесс, но при этом продолжить выполнение старого, нам необходимо скомбинировать execve с fork-ом. Напомню, что ребёнку fork возвращает ноль, а родителю pid ребёнка, который больше нуля.

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>

int main() {
    int pid = fork();
    if (pid == 0) {
        char *argv[] = {"/usr/bin/pwd", NULL};
        char *envp[] = {NULL};
        execve("/usr/bin/pwd", argv, envp);
    }
    else {
        printf("Check\n");
    }
    return 0;
}
```

В таком случае будет присутствовать и вывод программы pwd, и строка Check.

При завершении родительского процесса, дочерний продолжит выполняться. (При этом, в иерархии процессов будут интересные действия, но мы их обсуждать не будем.) Однако, вы можете остановить родительский процесс до завершения дочернего с помощью системного вызова wait4. Это довольно сложный СВ, так как он позволяет также прочитать состояние интересующего нас процесса в специальной структуре. Я же просто здесь покажу, как подождать завершения процесса по его pid.

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/wait.h>

int main() {
    int pid = fork();
    if (pid == 0) {
        char *argv[] = {"/usr/bin/sleep", "5", NULL};
        char *envp[] = {NULL};
        execve("/usr/bin/sleep", argv, envp);
    }
    else {
        wait4(pid, NULL, 0, NULL);
        printf("Check\n");
    }
    return 0;
}
```

Строка Check будет выведена после завершения программы sleep.

4.2.4

Посмотрите на системные вызовы read и write: https://chromium.googlesource.com/chromiumos/docs/+/master/constants/syscalls.md . В качестве первого аргумента они принимают некий файловый дескриптор fd, который является числом. В Linux работа с stream-ами реализована весьма красиво, любой stream, будь он stdin, файлом, pipe-ом, определяется единственным числом — файловым дескриптором. Посмотрим на номера файловых дескрипторов для stdin, stdout и stderr.

```
#include <stdio.h>
#include <unistd.h>

int main() {
  printf("%d %d %d\n", STDIN_FILENO, STDOUT_FILENO, STDERR_FILENO);
   return 0;
}
```

Вывод этой программы: 0 1 2

Попробуем вывести текст с помощью функции write в файловый дескриптор 1, то есть, stdout.

```
#include <stdio.h>
#include <unistd.h>

int main() {
    write(1, "Abacaba\n", 8);
    return 0;
}
```

Вывод этой программы: Abacaba

Мы можем также считать данные с помощью функции read.

```
#include <stdio.h>
#include <unistd.h>

int main() {
    char buffer[1024];
    int cnt = read(0, buffer, 1024);
    printf("%.*s", cnt, buffer);
    return 0;
}
```

Обратите внимание, что функция read не добавляет нулевой символ после считанной строки. Поэтому нам необходимо считать длину считанной строки и попросить функции printf вывести ровно столько символов, а не до первого нуля.

Вспомним, что ранее вы выводили текст в stderr с помощью функции fprintf, в которую мы передавали первым аргументом переменную FILE *stderr. Структура FILE является высокоуровневой абстракцией над файловыми дескрипторами, которая предоставляет дополнительную информацию. FILE можно получить по дескриптору с помощью функции fdopen. Я не буду здесь рассказывать про эти функции, так как после изучения дескрипторов читатель без проблем сможет изучить это самостоятельно. (Думаю, на этом этапе вы уже согласитесь, что C является высокоуровневым языком.)

Как вывести текст в файл с помощью системных вызовов? Очень просто. Нам понадобится (внезапно) системный вызов open. Я особо не буду рассказывать о его аргументов. Отмечу лишь, что нам необходимо наличие флага O_WRONLY или O_RDWR во втором аргументе.

```
#include <stdio.h>
#include <unistd.h>
#include <fcntl.h>

int main() {
    int fd = open("file", O_WRONLY, 0);
    printf("%d\n", fd);
    write(fd, "Abacaba\n", 8);
    return 0;
}
```

Чтобы вызов функции open завершился успешно, файл "file" должен существовать. Вы можете также сами посмотреть в документации, что нужно передать в функцию, чтобы она создала файл сама.

В моём случае вывод этой программы: 3

Можно перечислить несколько флагов с помощью оператора |. Например, O_WRONLY | O_APPEND. Часто передачу флагов делают так. Чтобы такое работало, необходимо, чтобы у всех флагов множество битов не пересекалось.

Обратите внимание на номер дескриптора: 3. Это означает, что, в отличии от, например, pid, номера дескрипторов не являются глобальными.

4.2.5

Изучите, с помощью каких системных вызовов можно манипулировать файлами (open может только создавать файлы).

https://chromium.googlesource.com/chromiumos/docs/+/master/constants/syscalls.md

Реализуйте любую программу, использующую эти системные вызовы.

4.2.6

С помощью системного вызова pipe мы можем получить сразу два таких дескриптора, что при записи данных в один из них, мы сможем считать эти данные из другого. Эта связка дескрипторов называется pipe (канал). Функция pipe принимает указатель на массив двух чисел и записывает в первый элемент принимающего, а во второй отправителя.

```
#include <stdio.h>
#include <unistd.h>

int main() {
    int fds[2];
    pipe(fds);

    write(fds[1], "Hello\n", 6);

    char buffer[1024];
    read(fds[0], buffer, 1024);
    printf("%s", buffer);
    return 0;
}
```

```
3 4
Hello
```

Использование двух полученных дескрипторов никак не отличается от использования stdin/stdout. Нам просто необходимо подставлять нужный дескриптор. Обратите внимание, что если мы поменяем местами write и read, то на вызове write наша программа зависнет в ожидании ввода, который никогда не получит.

Напомню, что все стандартные функции вывода выполняют вывод в stdout, то есть конкретный файловый дескриптор. Пусть мы хотим, чтобы эти функции выполняли вывод в наш pipe, а не в терминал. (Который, кстати, скорее всего тоже связан с нашей программой с помощью своего pipe-а.) С помощью системного вызова dup2 мы можем закрыть stdout stream (не путайте с stdout-дескриптором, то есть, с числом 1) и связать его дескриптор с отправляющей стороной pipe-а. В результате и при выводе в stdout, и при выводе в fds[1] будет выполнен вывод в pipe.

```
#include <stdio.h>
#include <unistd.h>
#include <fcntl.h>

int main() {
    int fds[2];
    pipe(fds);
```

```
    dup2(fds[1], STDOUT_FILENO);
    write(STDOUT_FILENO, "Hello\n", 6);
    // write(fds[1], "Hello\n", 6);
    // printf("Hello\n");
    // fflush(stdout);

    char buffer[1024];
    int cnt = read(fds[0], buffer, 1024);
    int fd = open("file", O_WRONLY, 0);
    write(fd, buffer, cnt);

    return 0;
}
```

После вызова функции dup2 и при выводе в 1, и при выводе в fds[1], будет выполнен вывод в pipe, который мы позже сможем считать через fds[0]. По желанию можно закрыть дескриптор fds[1] с помощью функции close: close(fds[1]). В таком случае вывод в fds[1] станет вызывать ошибку.

Так как теперь дескриптор 1 связан с pipe-ом, функция printf будет выводить в него. Только так как она выполняет буферизацию (то есть, копит текст, и только затем выполняет СВ write), вам необходимо будет потребовать её выполнить write сейчас с помощью функции fflush.

Так как stdout больше не связан с терминалом (и снова его связать простым способом невозможно), мы посмотрим на результат, выполним вывод в файл.

До сих пор мы с помощью pipe-а отправляли данные из процесса в него же. Но такое применение довольно бессмысленное. Обычно мы хотим с помощью pipe-ов передавать данные между процессами. Когда мы выполняем fork, оба процесса наследуют общий pipe, у которого теперь два дескриптора отправителя и два принимающих дескриптора. Имейте ввиду, что когда у вас два процесса считывают данные из одного pipe, то сложность системы сильно увеличивается. Это называется состоянием гонки (race condition).

```
#include <stdio.h>
#include <unistd.h>
#include <fcntl.h>

int main() {
    int fds[2];
    pipe(fds);

    if (fork() != 0) {
        char buffer[1024];
        write(fds[1], "Cat\n", 4);
        read(fds[0], buffer, 1024);
```

```
    }
    else {
        char buffer[1024];
        read(fds[0], buffer, 1024);
        write(fds[1], "Dog\n", 4);
    }

    return 0;
}
```

Если бы отправители и/или принимающие были связаны с разными pipe-ами, какой-то из процессов бы завис в ожидании ввода. Однако, этого не происходит.

При некоторых конфигурациях кода мне необходимо "потыркать" stream-ы, чтобы они немедленно передали данные. Иногда они могут это не сделать сразу, из-за чего программа зависнет. Например, у меня это происходит, если заменить fork() != 0 на fork() == 0.

В процессах детях также можно переоткрыть pipe-ы в дескрипторы стандартных потоков с помощью dup2. Вам может это понадобиться, если вы хотите запустить в дочернем процессе другую программу с помощью execve и общаться с ней через её стандартные потоки.

```
#include <stdio.h>
#include <unistd.h>
#include <fcntl.h>

int main() {
    int fds[2];
    pipe(fds);

    if (fork() != 0) {
        char buffer[1024];
        write(fds[1], "Cat\n", 4);
        write(fds[1], "Dog\n", 4);
        write(fds[1], "Crab\n", 5);
    }
    else {
        dup2(fds[0], STDIN_FILENO);
        char *argv[] = {"/usr/bin/grep", "a", NULL};
        char *envp[] = {NULL};
        execve("/usr/bin/grep", argv, envp);
    }

    return 0;
}
```

Cat
Crab

Здесь мы передаём дочернему процессу, выполняющему программу grep, данные в стандартный ввод через наш pipe.

Когда вы будете сами пробовать работать с pipe-ами, у вас часто будут проблемы с недосброшенными буферами, из-за чего ваши программы будут зависать. Я не знаю, как нормально решать эту проблему. Здесь я не использовал СВ close, который закрывает дескриптор, но иногда он может быть необходим.

4.2.7

Вы, наверное, слышали про thread (поток), который является альтернативой создания нового процесса. Чем thread отличается от процесса? Тем, насколько дочерний процесс/поток отделён от родительского. Мы видели, что новый процесс имеет свой стек (что логично, ведь блуждание по функциям меняет стек) и свою кучу. Новый поток же своей кучи не имеет. После создания нового потока, основную программу мы будем называть основным потоком.

Посмотрите на список СВ: https://chromium.googlesource.com/chromiumos/docs/+/master/constants/syscalls.md — в нём нет СВ, имеющих какое-то отношение к потоку. Функции создания потоков в libc используют СВ clone, который выполняет то же самое, что и fork, но имеет возможность настройки нового процесса.

Я не буду объяснять большинство функций здесь. Информация о потоках здесь лишь для ознакомления.

```
#include <stdio.h>
#include <unistd.h>
#include <pthread.h>

void *thread_main() {
    printf("Thread begin\n");
    sleep(2);
    printf("Thread end\n");
    return NULL;
}

int main() {
    pthread_t thr;
    pthread_create(&thr, NULL, thread_main, NULL);
    sleep(1);
    printf("Main end\n");

    return 0;
```

```
}
```

```
Thread begin
Main end
```

Мы создаём поток с помощью функции pthread_create. В первый аргумент функция записывает дескриптор потока, по которому мы будем этот поток упоминать. Третий аргумент — это функция, которая будет вызвана при старте потока (это сделано для удобства, так как обычно в потоках мы не запускаем другие программы с помощью execve).

Обратите внимание, что завершение основного потока прервало дочерний поток, и он не вывел строку Thread end. (Дочерние процессы продолжают выполнение и вывод.)

```
#include <stdio.h>
#include <unistd.h>
#include <pthread.h>

void *thread_main() {
    printf("Thread begin\n");
    sleep(2);
    printf("Thread end\n");
    return NULL;
}

int main() {
    pthread_t thr;
    pthread_create(&thr, NULL, thread_main, NULL);
    sleep(1);
    pthread_join(thr, NULL);
    printf("Main end\n");

    return 0;
}
```

```
Thread begin
Thread end
Main end
```

Функция pthread_join останавливает текущий поток до завершения потока, дескриптор которого мы передали в первый аргумент. Это аналог функции wait4 для процессов.

```
#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>

int *x;
```

```
void *thread_main() {
    (*x)++;
    return NULL;
}

int main() {
    x = (int*)malloc(sizeof(int));
    *x = 2;
    pthread_t thr;
    pthread_create(&thr, NULL, thread_main, NULL);
    pthread_join(thr, NULL);
    printf("%d\n", *x);

    return 0;
}
```

3

В данном примере мы видим, что куча у обоих потоков общая.

4.2.8

Напишите любую программу, в которой несколько потоков объединены в цикл из pipe-ов и по кругу передают друг-другу сообщение.

4.2.9

Посмотрим на такую программу:

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <pthread.h>

int *x;

void *thread_main() {
    if (*x > 0) {
        sleep(1);
        (*x)--;
    }
    return NULL;
}

int main() {
    x = (int*)malloc(sizeof(int));
    *x = 1;
```

```
    pthread_t thr1, thr2;
    pthread_create(&thr1, NULL, thread_main, NULL);
    pthread_create(&thr2, NULL, thread_main, NULL);
    pthread_join(thr1, NULL);
    pthread_join(thr2, NULL);
    printf("%d\n", *x);

    return 0;
}
```

Программа немного искусственная, но она покажет нам проблему. Мы создаём два потока. Каждый из них уменьшает счётчик на единицу, если он больше нуля. Однако вывод прогаммы:

-1

Чтобы лучше понять, что произошло, я покажу порядок, в котором исполнялись statement-ы потоков.

```
Thread 1: if (*x > 0)
Thread 1: sleep(1);
Thread 2: if (*x > 0)
Thread 2: sleep(1);
Thread x: (*x)--;
Thread x: return NULL;
Thread y: (*x)--;
Thread y: return NULL;
```

Здесь из-за блокировки функции sleep, после которой процессор переходит к другому потоку, сначала выполнились две проверки, а только потом два уменьшения. Это называется состоянием гонки (race condition), и исправить проблему можно с помощью использования семафора (semaphore).

Мы воспользуемся posix semaphore-ами, функции которых имеют вид sem_xxx. Существует много других реализаций, но я не вижу в них смысла. Как и в случае с thread-ами, эта реализация семафоров основана на не очень дружелюбных системных вызовах semxxx (можете найти их в таблице CB). Я так же покажу лишь пример использования этого.

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <pthread.h>
#include <semaphore.h>

int *x;
sem_t sem;

void *thread_main() {
```

```
    sem_wait(&sem);
    if (*x > 0) {
        sleep(1);
        (*x)--;
    }
    sem_post(&sem);
    return NULL;
}

int main() {
    x = (int*)malloc(sizeof(int));
    *x = 1;
    sem_init(&sem, 0, 1);
    pthread_t thr1, thr2;
    pthread_create(&thr1, NULL, thread_main, NULL);
    pthread_create(&thr2, NULL, thread_main, NULL);
    pthread_join(thr1, NULL);
    pthread_join(thr2, NULL);
    printf("%d\n", *x);

    return 0;
}
```

Функция sem_init записывает дескриптор семафора в первый аргумент, а третий аргумент — это начальное значение семафора. Пусть сначала оно будет равно единице.

Функция sem_wait уменьшает значение семафора на один, но если оно сейчас равно нулю, блокирует поток до момента, когда оно станет больше нуля.

Функция sem_post увеличивает значение семафора на один.

Нам необходимо обрамить блок, использующий общую переменную в sem_wait и sem_post. Теперь порядок выполнения statement-ов такой:

```
Thread 1: sem_wait(&sem);
Thread 1: if (*x > 0)
Thread 1: sleep(1);
Thread 2: sem_wait(&sem);
Thread 1: (*x)--;
Thread 1: sem_post(&sem);
Thread 1: return NULL;
Thread 2: if (*x > 0)
Thread 2: sem_post(&sem);
Thread 2: return NULL;
```

Если мы заменим третий аргумент функции sem_init на 2, то это можно понимать, как редактирование переменной одновременно не более чем двумя

потоками.

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <pthread.h>
#include <semaphore.h>

int *x;
sem_t sem;

void *thread_main() {
    sem_wait(&sem);
    if (*x > 0) {
        sleep(1);
        (*x)--;
    }
    sem_post(&sem);
    return NULL;
}

int main() {
    x = (int*)malloc(sizeof(int));
    *x = 1;
    sem_init(&sem, 0, 2);
    pthread_t thr1, thr2, thr3;
    pthread_create(&thr1, NULL, thread_main, NULL);
    pthread_create(&thr2, NULL, thread_main, NULL);
    pthread_create(&thr3, NULL, thread_main, NULL);
    pthread_join(thr1, NULL);
    pthread_join(thr2, NULL);
    pthread_join(thr3, NULL);
    printf("%d\n", *x);

    return 0;
}
```

-1

4.2.10

Рассмотрим программу.

```
#include <stdio.h>

int main() {
    int *a = 0;
```

```
    printf("%d\n", *a);

    return 0;
}
```

```
Segmentation fault (core dumped)
```

Когда наша программа нарушает правила доступа, ей поступает сигнал SIGSEGV. По умолчанию при получении этого сигнала программа завершается. Мы можем сами указать, что следует сделать при получении этого сигнала. Функция signal принимает номер сигнала и функцию, которая будет вызвана при получении этого сигнала. Попробуем перехватит SIGSEGV.

```
#include <stdio.h>
#include <signal.h>

void handler(int s) {
    printf("%d\n", s);
}

int main() {
    signal(SIGSEGV, handler);
    int *a = 0;
    printf("%d\n", *a);

    return 0;
}
```

```
11
11
11
...
```

Число 11 — это номер SIGSEGV-а. Мы не завершаем программу в нашем обработчике. После возвращения из функции handler процессор снова выполняет инструкцию, которая привела к SIGSEGV. На уровне ОС есть способы "передумать" и не выполнять инструкцию, которая приводит к ошибке. Однако у нас, как у прикладной программы, таких способов нет. Поэтому обработчик SIGSEGV-а должен завершать программу.

А вот и квест: программа в таком состоянии не будет слушать сигналов. Как её завершить?

Попробуем поиграться с разными сигналами.

```
#include <stdio.h>
#include <signal.h>

void handler(int s) {
    printf("%d\n", s);
```

```
}

int main() {
    signal(SIGSEGV, handler);
    signal(SIGINT,  handler);
    signal(SIGQUIT, handler);
    signal(SIGSTOP, handler);
    signal(SIGCONT, handler);
    signal(SIGTERM, handler);
    signal(SIGKILL, handler);
    signal(SIGPIPE, handler);
    signal(SIGCHLD, handler);

    while(1);

    return 0;
}
```

Пойдем по порядку.

Сигнал SIGINT вызывается при нажатии комбинации Ctrl + C в терминале и по умолчанию прерывает процесс. Нажмите эту комбинацию, и увидете число 2.

Сигнал SIGQUIT вызывается при нажатии комбинации Ctrl + \ в терминале и по умолчанию прерывает процесс. Нажмите эту комбинацию, и увидете число 3.

Сигнал SIGSTOP вызывается при нажатии комбинации Ctrl + Z в терминале, то есть, при переводе процесса на фон. Нажмите эту комбинацию, однако число 19(номер этого сигнала) вы не увидете. Дело в том, что этот сигнал особенный — его нельзя перехватить.

Сигнал SIGCONT вызывается при возвращении фонового процесса командой fg. Введите эту команду, и увидете число 18.

Сигнал SIGTERM вызывается при вызове программы kill с pid нашей программы. Выполните это и увидете число 15.

Сигнал SIGKILL вызывается при вызове программы kill с дополнительным аргументом -9. При выполнении этого программа завершится и не выведет число, так как этот сигнал нельзя перехватить.

Для генерации следующих сигналов потребуется внести изменения в программу.

Сигнал SIGPIPE вызывается при записи в закрытый stream.

```
...
    int fds[2];
    pipe(fds);
```

```
    close(fds[0]);
    write(fds[1], "Hello\n", 6);
...
```

Сигнал SIGCHLD вызывается при завершении дочернего процесса.

```
...
    if (fork() == 0) {
        return 0;
    }
...
```

Здесь возникнет интересная ситуация. Посмотрите на таблицу процессов любым способом (например, через ps или top). Вы увидете, что дочерний процесс ещё есть, и, более того, он не будет пропадать при сигнале SIGKILL. Дочерний процесс находиться в состоянии зомби. В этом состоянии процесс уже завершен, но ОС хранит некоторую метаинформацию, которая может быть полезна родительскому процессу. ОС уничтожит эти данные после того, как родительский процесс вызовет СВ wait. Обработчик по умолчанию делает этот вызов, но наш — нет.

```
...
void handler(int s) {
    printf("%d\n", s);
    wait(NULL);
}
...
```

4.2.11

Проект: Shell

Напишите программу, аналогичную sh, но без скриптов (так как их написание довольно сложно). Список, на который можно опираться:

- Принимать команды, которые состоят из нескольких слов и отделены друг от друга переводами строк.
- Первое слово в команде определяет путь к исполняемому файлу. Остальные слова — аргументы.
- Может быть команда cd, которая меняет текущую директорию. Для её реализации необходимо воспользоваться системным вызовом chdir.
- В команде могут быть операторы $<$, $>$ после которых идут пути к файлам. Для реализации этих операторов необходимы pipe-ы.
- Команды могут быть разделены символов |. Тогда нужно запускать программы сразу во всех таких командах и связывать их pipe-ами.
- В команде может быть оператор &. Также могут быть команды fd и bg. Это также реализуется манипуляциями с pipe-ами.
- При получении сигналов, которые могут быть предназначены текущему дочернему процессу, передавать эти сигналы ему. Для реализации этого

необходимо воспользоваться системным вызовом kill.

4.3. Файловая система

4.3.1

Внимание! В данной главе мы будем работать с потенциально разрушительными командами. Внимательно читайте команды перед тем, как выполнять их.

Создадим для удобства папку mnt (это не более, чем обычная папка).

```
mkdir mnt
```

Посмотрите содержимое директории /dev. Среди файлов вы увидете файлы с названиями sda, sda1, sda2, и т. д. sda — это ваш загрузочный диск, на котором располагается ОС. Диск разбит на несколько partitions (разделов), которые являются обычными, но независимыми друг от друга, директориями. Каждый из файлов sdaX — это соответствующий partition.

В Unix есть возможность "подвесить" partition к любой пустой директории. В результате этого действия вы сможете видеть директорию partition-а при в входе в эту пустую директорию. Это дествие называется mount (монтирование) и выполняется программой mount.

Монтрирование производится централизованно, и список всех можно увидеть, выполнив mount -l.

```
...
/dev/sda1 on /boot type ext4 (rw,relatime)
...
```

Посмотрите на мой вывод. Здесь указано, что первый partition моего системного диска уже подвешен к директории /boot. Этот partition отвечает за первичную загрузку нашей ОС.

Давайте примонтируем этот partition к нашей директории mnt с помощью команды sudo mount /dev/sda1 mnt.

Теперь в выводе программы mount -l я получаю такую строку в конце:

```
...
/dev/sda1 on /home/igor/test/mnt type ext4 (rw,relatime)
```

Мы можем зайти в директорию и увидеть её содержимое.

```
$ ls mnt
config-6.5.0-44-generic         initrd.img-6.8.0-40-generic  System.map-6.5.0-44-
generic
config-6.8.0-40-generic         initrd.img.old               System.map-6.8.0-40-generic
efi                             lost+found                   vmlinuz
grub                            memtest86+.bin               vmlinuz-6.5.0-44-generic
```

initrd.img memtest86+.elf vmlinuz-6.8.0-40-generic
initrd.img-6.5.0-44-generic memtest86+_multiboot.bin vmlinuz.old

Директорию можно редактировать, что является отличным способом сломать свою ОС.

Чтобы размонтировать директорию, необходимо воспользоваться программой umount: sudo umount mnt

Давайте теперь примонтируем /dev/sda3. В моём случае в этом partition-е хранится основная файловая система вместе с директорией /home. В вашем случае, это может быть другой partition, или даже директория /home может находится в отдельном от основной файловой системы partition-е. Найдите partition с /home самостоятельно и работайте с ним.

При такой файловой системе вы можете вновь прийти к директории mnt, однако на этот раз она будет пуста.

$ ls mnt/home/igor/test/mnt

4.3.2

В Linux есть возможность создать файловую систему прямо на обычном файле, а затем этот файл примонтировать и пользоваться им. Для этого используются loop devices.

Посмотрите содержимое директории /dev: во многих дистрибутивах у вас там будут использоваться loop devices (на Ubuntu их особенно много).

Создадим файл, состоящий только из нулей.

dd if=/dev/zero of=dsk count=1024

Программа dd берет префикс файла if размера bs * count и пишет его в файл of. Файл /dev/zero это особенный файл, который состоит только из нулей, а его размер бесконечен.

Примонтировать сам файл нельзя, зато можно примонтировать loop device. Введите losetup -l, чтобы увидеть список loop devices, которые сейчас у вас присутствуют. В зависимости от дистрибутива, их может быть разное количество. Все их названия имеют форму /dev/loopX. Введите losetup -f, чтобы определить свободное название.

Я предположу, что название /dev/loop100 у вас свободно. Создадим loop device:

sudo losetup /dev/loop100 dsk

Введите теперь losetup -l: вы увидете ваш loop device в списке.

/dev/loop100 0 0 0 0 /home/igor/test/dsk 0 512

Попробуем его примонтировать.

```
$ sudo mount /dev/loop100 mnt
mount: /home/igor/test/mnt: wrong fs type, bad option, bad superblock on /dev/loop100, missing codepage or hel
```

Монтировать можно только корректные файловые системы, а у нас просто последовательность нулей. Создадим файловую систему ext4 (об этом чуть позже).

```
$ mkfs.ext4 dsk
mke2fs 1.46.5 (30-Dec-2021)

Filesystem too small for a journal
Discarding device blocks: done
Creating filesystem with 128 4k blocks and 64 inodes

Allocating group tables: done
Writing inode tables: done
Writing superblocks and filesystem accounting information: done

$ sudo mkfs.ext4 /dev/loop100
mke2fs 1.46.5 (30-Dec-2021)
/dev/loop100 contains a ext4 file system
    created on Wed Sep 11 13:38:27 2024
Proceed anyway? (y,N)
```

Обратите внимание: вы можете упомянуть как сам файл, так и loop device.

Теперь мы можем выполнить монтирование.

```
$ sudo mount /dev/loop100 mnt
$ ls mnt
lost+found
```

В этой файловой системе у вас изначально есть директория lost+found.

Чтобы удалить loop device, необходимо выполнить sudo losetup -d /dev/loop100 .

4.3.3

Разберемся, как установить на диске несколько partition-ов и примонтировать их.

Есть несколько утилит для редактирования partition-ов диска. Мы воспользуемся утилитой fdisk. Введите fdisk dsk, чтобы начать редактирование.

Ранее я говорил о том, что мы создали диск с единственным partition-ом. Это не совсем так, так как диск с partition-ами должен содержать специальную метаинформацию, которой у нас не было. Когда мы запустим fdisk, он нас предупредит о том, что наш диск является чистой ext4 файловой системой, которую он просто перезапишет.

Интерфейс этой программы не совсем тривиальный. Внимательно читайте, что она выводит.

Для начала, познакомимся с форматами таблиц partition-ов. Выведите help с помощью m и посмотрите на последний блок, в котором написаны форматы GPT, SGI, DOS и Sun. Мы поговорим о первом и третьем.

```
...
  Create a new label
   g   create a new empty GPT partition table
   G   create a new empty SGI (IRIX) partition table
   o   create a new empty DOS partition table
   s   create a new empty Sun partition table
```

Когда вы запускаете компьютер, на нём выполняется "некий" код, который делает некоторую подготовку, а затем начинает выполнять код, написанный где-то на диске. Этот "некий" код, который ответственнен за первичную подготовку, бывает двух типов: BIOS и EFI.

BIOS действует очень просто: он загружает первый сектор диска и начинает его выполнять. Поэтому в первом секторе мы обязательно должны написать код, который поймет, где находится нужный нам partition, и загрузит его. (Один сектор — это, кстати, 512 байт. А вы сможете вместить такую логику в этот объем?)

BIOS сейчас является устаревшим, и маловероятно, что на вашем компьютере используется он. Однако, вы можете запустить виртуальную машину с BIOS (почему-то, это фукнция по умолчанию в VirtualBox).

EFI значительно умнее и его "некий" код изучает таблицу partition-ов и загружает нужный самостоятельно.

Обратите внимание, что термин "таблица partition-ов" не имеет отношения к термину "файловая система". Вы можете иметь на диске partition-ы с разными файловыми системами.

Итак, EFI использует формат GPT, а BIOS использует формат DOS (на самом деле, формат MBR, к которому относится DOS).

Посмотрим на используемый сейчас формат с помощью p.

```
Command (m for help): p
Disk dsk: 512 KiB, 524288 bytes, 1024 sectors
Units: sectors of 1 * 512 = 512 bytes
Sector size (logical/physical): 512 bytes / 512 bytes
I/O size (minimum/optimal): 512 bytes / 512 bytes
Disklabel type: dos
Disk identifier: 0xaf3f4246
```

По умолчанию fdisk выбрал формат DOS. Давайте проверим, как он отформатирует наш диск. Введите n для создания нового partition-а и, пока

что, прокликайте всё без изменения. Теперь введите w, чтобы выполнить форматирование.

Посмотрим на первый сектор, то есть, первые 512 байт получившегося файла dsk.

Посмотрите на таблицу из http://wiki.osdev.org/Partition_Table . Согласно таблице, информация о нашем первом partition-е должна находиться на байте 0x01BE.

Partition number	Offset
Partition 1	0x01BE (446)
Partition 2	0x01CE (462)
Partition 3	0x01DE (478)
Partition 4	0x01EE (494)

Проверим это с помощью hd: hd dsk.

```
00000000  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  |................|
*
000001b0  00 00 00 00 00 00 00 00  e2 79 d9 ef 00 00 00 00  |.........y......|
000001c0  02 00 83 01 04 15 01 00  00 00 ff 03 00 00 00 00  |................|
000001d0  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  |................|
*
000001f0  00 00 00 00 00 00 00 00  00 00 00 00 00 00 55 aa  |..............U.|
...
```

Мы действительно видим ненулевые биты на отрезке 0x1be - 0x1cd. (Что написано на отрезке 0x1b8 - 0x1ba можете изучить самостоятельно по той же ссылке.) Кроме того, мы здесь видим в конце первого сектора байты 0x55 и 0xaa, которые обязательно должны быть в MBR.

4.3.4

Когда вы создаёте первый partition в DOS-диске, вы можете разместить его только начиная с 1-го сектора (везде нумерация идёт с нуля), так как первый сектор является хратит код и таблицу разделов MBR. Однако, когда вы создаёте первый partition в GPT-диске, вы можете разместить его только начиная с 34-сектора.

- Что находится в первых 34-х секторах?
- Зачем нужет первый сектор?

Можете прочитать о GPT на OSDev Wiki.

4.3.5

Создадим диск с несколькими partition-ами с помощью fdisk. Прочитайте самостоятельно help в утилите и добейтесь похожей картины:

```
Command (m for help): p
Disk dsk: 512 KiB, 524288 bytes, 1024 sectors
Units: sectors of 1 * 512 = 512 bytes
Sector size (logical/physical): 512 bytes / 512 bytes
I/O size (minimum/optimal): 512 bytes / 512 bytes
Disklabel type: gpt
Disk identifier: 4A705392-B1C6-E34B-B7B9-39BFA385DB58

Device Start   End Sectors  Size Type
dsk1      34   200     167 83,5K Linux filesystem
dsk2     201   990     790  395K Linux filesystem
```

Формат диска не важен, так как мы планируем использовать loop devices, и они поддерживают большинство форматов.

Выполните losetup с флагом -P: sudo losetup -P /dev/loop100 dsk . Вывод losetup -l ничего нового не покажет, но если вы выполните ls /dev, то увидете интересную картину.

```
...
loop100
loop100p1
loop100p2
...
```

У нас появилось по одному дополнительному loop device на каждый partition. Мы можем в каждом из них выполнить форматирование. Для интереса выполним форматирование в разные файловые системы.

```
$ sudo mkfs.fat /dev/loop100p1
$ sudo mkfs.ext4 /dev/loop100p2
```

Теперь мы можем примонтировать их.

```
$ mkdir mnt1 mnt2
$ sudo mount /dev/loop100p1 mnt1
$ sudo mount /dev/loop100p2 mnt2
$ ls mnt1
$ ls mnt2
lost+found
```

Расскажу ещё немного о файловых системах. Есть три основных часто используемых файловых систем:

- ext4 — сложная файловая система, часто используемая на Linux
- NTFS — сложная файловая система, используемая на Windows

- fat — простая файловая система, драйвер к которой можно написать самостоятельно

4.3.6

Выполните установку любого дистрибутива с ручной установкой (я рекомендую Arch) на виртуальную машину в VirtualBox. Нас интересует часть, связанная с разметкой диска. Попробуйте выполнить установку и на машину с BIOS, и на машину с EFI (меняется в настройках машины).

4.3.7

Предположим, что мы хотим запустить процесс так, чтобы он видел лишь некое поддерево файловой системы. Есть два способа достичь этого: chroot и системный вызов clone с флагом CLONE_NEWNS. Второй способ использует namespaces, которые были введены в Linux не очень давно для контейнеризации (изоляции процессов), и именно его сейчас используют утилиты LXC и Docker. Namespaces способны изолировать не только файловую систему и значительно сложнее chroot-а, поэтому будем здесь работать с ним.

При использовании любого способа, так как программа не будет видеть ничего вне выбранного поддерева, необходимо будет иметь как минимум, необходимые shared objects, а, желательно, почти всю ОС в этом поддереве.

Попробуем запустить pwd в изолированной директории. Создайте директорию dir и скопируйте туда pwd.

```
$ mkdir dir
$ cp /usr/bin/pwd dir
```

Попробуем запустить его с помощью программы chroot.

```
$ sudo chroot dir /pwd
chroot: failed to run command '/pwd': No such file or directory
```

Первым аргументом мы вводим поддерево, которое будет видеть процесс, а вторым — путь к исполняемому файлу относительно корня этого поддерева.

Ошибка очень неочевидная и она говорит о том, что загрузчик процесса не смог найти необходимые shared objects. (Как это понять, история умалчивает.) Мы можем посмотреть необходимые shared objects с помощью программы ldd, а затем скопировать их в dir.

```
$ cd dir
$ ldd pwd
    linux-vdso.so.1 (0x00007ffd1aed3000)
    libc.so.6 => /lib/x86_64-linux-gnu/libc.so.6 (0x00007d59e6000000)
    /lib64/ld-linux-x86-64.so.2 (0x00007d59e637a000)
$ mkdir -p lib/x86_64-linux-gnu lib64
```

```
$ cp /lib/x86_64-linux-gnu/libc.so.6 lib/x86_64-linux-gnu/
$ cp /lib64/ld-linux-x86-64.so.2 lib64
$ cd ..
$ sudo chroot dir /pwd
/
```

linux-vdso.so.1 — особый shared object, который импортируется в из ядра в любую программу. Обратите внимание, что pwd вывел свою позицию относительно поддерева.

Для запуска sh скопируем в dsk образ ОС. (Ведь, чтобы запускать другие программы в sh, необходимо будет их всех также скопировать вместе с их зависимостями.) Наверное, легче всего скачивать образы с помощью утилиты Docker. Очистите директорию dsk и выполните docker export $(docker create ubuntu) | tar -C dsk -xvf - , чтобы скачать и образ ОС Ubuntu и разархивировать его в dsk.

Выполните sudo chroot dsk /bin/sh , чтобы запустить sh. Попробуйте походить по папкам, чтобы удостовериться, что вы не имеете доступа к основной файловой системе.

chroot имеет некоторые уязвимости, которые устраняют namespaces. Можете изучить самостоятельно, что нехорошего может сделать программа, запущенная из chroot-а.

4.3.8

Проект: Containerization

Напишите программу для контейнеризации, которая позволяет:

- Создавать файлы-диски и монтировать их (создавать контейнеры)
- Автоматически устанавливать на контейнер ОС (можно загружать их с помощью docker-а)
- Монтировать несколько дисков
- Запускать программы в контейнерах в chroot

По желанию, можете использовать namespaces, но это намного сложнее.

5. Assembly

5.1. Регистры, базовые инструкции

5.1.1

Начнём изучать язык ассемблера (assembly language). Мы уже знаем, что в процессе компиляции код на C переписывается на язык ассемблера, а затем он переписывается ассемблером в объектный файл с машинным кодом.

Для чего может быть нужно знание ассемблера? На самом деле, ответ такой же, как и на вопрос о том, зачем знать C. Мы очень не хотим писать на

языке ассемблера, так как это очень сложно (чуть позже увидим почему). Обычно мы прибегаем к его использованию в следующих случаях:

- Мы хотим организовать среду, в которой сможет работать код, написанный на языке C. Это касается embedded programming (встраимоевого программирования).
- Мы хотим изучить результат работы компилятора на предмет возможных недостатков. Иногда специфические математические алгоритмы лучше написать на языке ассемблера, так как компилятор может не догадываться, как их писать правильно.
- Мы пишем непосредственно генерацию кода для ассемблера.

Первый случай отличается от двух других гараздо сильнее, чем может показаться на первый взгляд. Остальные случаи относятся к написанию кода на прикладном уровне. Возможно, вы уже слышали такую мудрость: Для написания кода на ассемблере достаточно знать random_range(5, 10) инструкций. Для прикладного уровня это так, и с него мы и начнём изучение. Организация же среды в embedded program требует взаимодействия с состояниями процессора с помощью инструкций, которые мы, по понятной причине, не можем применять на прикладном уровне.

Существует много различных архитектур процессоров. Все они работают по одной модели, но отличаются множеством состояний и набором инструкций. Самые часто используемые на практике процессоры: x86, arm, risc-v. Подобно тому, как мы можем использовать другие операционные системы с помощью виртуальной машины, мы можем выполнить эмуляцию других процессоров (понятно, что не аппаратно, а программно). Однако, мы будем в основном работать с архитектурой x86, так как, скорее всего, эта архитектура и стоит на вашем компьютере.

5.1.2

Сама архитектура задаёт только инструкции, которые мы будем видить при дизассемблировании. Однако, даже ассемблеры имеют конструкции для структуризации кода, которые называются директивами. Проблема в том, что хоть все ассемблеры имеют одинаковые инструкции (ведь их задаёт архитектура), директивы в них отличаются.

Мы будем пользоваться ассемблером GNU Assembler, который и используется в процессе компиляции программы на C компилятором gcc. Существует два основных синтаксиса языков ассемблера: AT&T и Intel. По умолчанию, gcc использует синтаксис AT&T, с которого мы и начнём, но позже посмотрим и на второй. Чтобы gcc использовал синтаксис Intel, следует добавить флаг -masm=intel.

Когда мы пишем на ассемблере, следует соблюдать некоторые протоколы взаимодействия. ОС Linux использует специальные объектные файлы, поэтому, когда мы пишем программу на Linux, следует генерировать именно

объектный файл. Другие ОС имеют другие структуры у подобных файлов. Если мы хотим на своей ОС иметь свой формат, аналогичный объектным файлам, нам придётся написать свой ассемблер. Также ассемблеры могут генерировать flat binary файлы, то есть файлы, состоящие непосредственно из секций, без какой-либо метаинформации. Естественно, мы их не сможем запустить на Linux-е, но они могут быть неплохой отправной точкой, когда мы пишем загрузчик программ для своей ОС.

Итак, перейдем к языку ассемблера.

main.s

```
    .text
    .globl  main
main:
    mov     $2, %rax # comment
    mov     $3, %rbx
    add     %rax, %rbx
    ret
```

Первые три строки являются директивами. То есть, они говорят ассемблеру, что делать, но сами в результирующий объектный файл не попадают.

.text говорит о том, что все следующие инструкции следует положить в секцию text. Компоновщик gcc требует наличие этой секции. Но позже мы научимся регулировать это.

.globl говорит о том, что label (метка) main должна быть видна извне данного файла. В данном случае она нужно, чтобы компоновщик gcc увидел эту метку и записал, что выполнять программу следует с этого адреса.

main — это метка, на которую мы можем ссылаться в других местах нашей программы. Каждое использование слова main будет заменено её адресом (каким адресом, мы узнаем позже).

Комментарии в gnu assembler начинаются с символа #.

Далее идут четыре инструкции. В языке ассемблера у нас есть регистры, в которых мы храним результаты провежуточных вычислений. Это ячейки памяти с самым быстрым доступом. Обычно регистров общего назначения около десяти, но помимо них могут быть десятки более специфических регистров. rax и rbx — это регистры общего назначения. Мы изучим полный их список позже. Можно воспринимать их, как очень глобальные переменные.

Что делают инструкции mov и add догадаться просто. Инструкция mov кладёт значение в первом аргументе (source — источник) во второй аргумент (destination — цель, назначение) (что происходит, когда второй аргумент не является регистром, узнаем чуть позже). Инструкция add складывает оба аргумента и кладёт результат во второй аргумент. Инструкция ret более сложная, мы её изучим позже.

Скомпилируем: gcc main.s -o main и запустим ./main. Программа ничего не выведет. Написать ввод и вывод в языке ассемблера несколько сложнее, чем в C, и мы научимся этому позже.

5.1.3

Научимся отлаживать (debug) код на языке ассемблера. Сама отладка, по моему мнению, по большей части бессмысленна, но конкретно сейчас она хорошо покажет, как выполняется программа на уровне регистров и памяти.

Установите GNU Debugger — gdb. Запустите его на нашей программе из прошлого шага: gdb main. Начнётся интерактивная сессия, подобная тем, которые используются в текстовых редакторах.

Я хочу выполнить четыре инструкции в нашей программе и посмотреть на то, что происходит с регистрами.

- Чтобы видеть наши инструкции, следует написать layout asm. Это выполнит дизассемблирование, поэтому директивы мы не увидим.
- Чтобы видет регистры, следует написать layout regs.
- На самом деле, до и после выполнения функции main выполняется немало другого кода. Поэтому начинать с самого начала нам пока не следует (хотя позже мы изучим, что это за код, и как он генерируется). Поставьте точку останова (breakpoint) на метке main: b main
- Наконец, запустите программу: r. Чтобы перейти к следующей инструкции, напишите ni. Наблюдайте за подсвеченной командой в коде и значениями регистров rax и rbx.

Что вы должны видеть в процессе ввода команд ni.

```
Register group: general
rax            0x2                2
rbx            0x0                0
rcx            0x555555557df8     93824992247288
rdx            0x7fffffffdc98     140737488346264
rsi            0x7fffffffdc88     140737488346248
rdi            0x1                1
rbp            0x1                0x1
rsp            0x7fffffffdb78     0x7fffffffdb78
r8             0x7ffff7e1bf10     140737352154896
r9             0x7ffff7fc9040     140737353912384
r10            0x7ffff7fc3908     140737353890056
r11            0x7ffff7fde660     140737353999968
r12            0x7fffffffdc88     140737488346248
r13            0x555555555129     93824992235817
r14            0x555555557df8     93824992247288

B+ 0x555555555129 <main>          mov    $0x2,%rax
```

```
> 0x555555555130 <main+7>       mov   $0x3,%rbx
  0x555555555137 <main+14>      add   %rax,%rbx
  0x55555555513a <main+17>      ret
  0x55555555513b                add   %dh,%bl
  0x55555555513d <_fini+1>      nop   %edx
  0x555555555140 <_fini+4>      sub   $0x8,%rsp
  0x555555555144 <_fini+8>      add   $0x8,%rsp
  0x555555555148 <_fini+12>     ret
  0x555555555149                add   %al,(%rax)
  0x55555555514b                add   %al,(%rax)
  0x55555555514d                add   %al,(%rax)
  0x55555555514f                add   %al,(%rax)
  0x555555555151                add   %al,(%rax)
  0x555555555153                add   %al,(%rax)
  0x555555555155                add   %al,(%rax)

multi-thre Thread 0x7ffff7f9c7 In: main                                L??   PC: 0x555555555130
(gdb) layout regs
(gdb) b main
Breakpoint 1 at 0x1129
(gdb) r
Starting program: /home/igor/test/main
[Thread debugging using libthread_db enabled]
Using host libthread_db library "/lib/x86_64-linux-gnu/libthread_db.so.1".

Breakpoint 1, 0x0000555555555129 in main ()
(gdb) ni
0x0000555555555130 in main ()
(gdb)
```

Чтобы не вводить постоянно эти команды, можно написать скрипт.

.gdbinit

```
layout asm
layout regs
b main
r
```

После того, как вы запустите в следующий раз gdb, среди текста вам будет выведено такое сообщение с вашими путями.

```
warning: File "/home/igor/test/.gdbinit" auto-loading has been declined by your `auto-
load safe-path' set to "$debugdir:$datadir/auto-load".
To enable execution of this file add
    add-auto-load-safe-path /home/igor/test/.gdbinit
line to your configuration file "/home/igor/.config/gdb/gdbinit".
To completely disable this security protection add
```

```
    set auto-load safe-path /
```
line to your configuration file "/home/igor/.config/gdb/gdbinit".

Удовлетворите это условие, и при следующих запусках gdb будет выполнять скрипт.

5.1.4

Инструкция mov имеет несколько вариантов. (Мне здесь не особо известна терминология. Возможно, правильно говорить, что есть неколько инструкций с названием mov.) Вариант, который мы использовали, имеет в качестве destination (назначения) регистр. Но назначение может быть также и адрес.

В языке C обычно мы начинаем изучение адресов с адресов локальных переменных. Однако, понятия локальной переменной в языке ассемблера нет — нам нужно организовать стек самостоятельно. Вызывать функции динамического выделения памяти мы тоже пока не можем. Придётся использовать глобальные переменные.

Здесь мы выполним более серьёзный анализ. Начнём с gnu assembly 64-bit.

```
    .bss
arr:
    .zero   8

    .text
    .globl  main
main:
    leaq    arr(%rip), %rax
    movq    $1, (%rax)
    ret
```

Мы объявляем новую стандартную секцию bss, которая изначально заполнена нулями. С помощью директивы .zero мы объявляем последовательность из восьми байт, на начало которой указывает метка arr.

Посмотрим сначала на инструкцию movq. Мы взяли второй аргумент в скобки. Это означает, что мы хотим записать значение не в сам регистр, а по адресу регистра. Но какая размерность числа, которое мы записываем? (От этого зависит то, сколько старших байт регистра мы занулим.) Это мы явно сообщаем суффиксом q в слове movq.

Посмотрим на строку с инструкцией leaq. Её первый аргумент упоминает метку arr. Может показаться, что здесь упоминается регистр rip, который является адресом текущей инструкции. Но это не так, и это лишь директива ассемблеру, сообщающая о том, что адрес метки arr должен вычисляться относительно регистра rip. (Мы поговорим об этом отдельно. Дело в том, что в 64-bit мы хотим, чтобы все адреса в нашей программе были относительными. Это называется position independent code (PIC).)

Что делает инструкция lea? Она загружает не значение по адресу arr (что сделала бы инструкция mov), а сам адрес arr. (Здесь у знающих ассемблеры может быть вопрос — точно ли mov загружает значение? Да, и я сам не понимаю, почему в gnu assembly это работает так.)

Проверим программу. Запустите дебаггер и выведите память около адреса, равного значению регистра rax, с помощью команды x. (Эта команда имеет много параметров для форматирования вывода. Ищите "gdb cheet sheet".)

```
(gdb) x $rax
0x555555558011: 0x00000001
```

Действительно, три нуля и единица.

Замените теперь инструкцию leaq на movq. Убедитесь, что программа получает SIGSEGV и дизассемблируйте её с помощью objdump: objdump -d main.

```
...
0000000000001129 <main>:
    1129:   48 8b 05 e1 2e 00 00     mov    0x2ee1(%rip),%rax        # 4011 <arr>
    1130:   48 c7 00 01 00 00 00     movq   $0x1,(%rax)
    1137:   c3                       ret
...
```

Посмотрите на opcode этой инструкции: 0x8b (число 48, очевидно, не может быть opcode-ом, ведь эти две инструкции явно разные). Эта информация нам скоро понадобится.

Чтобы получить значение из ячейки по адресу, следует так же использовать скобки.

```
    .bss
arr:
    .zero   8

    .text
    .globl  main
main:
    leaq    arr(%rip), %rax
    movq    $1, (%rax)
    movq    arr(%rip), %rbx
    movq    (%rax), %rcx
    ret
```

Так мы запишем значение по адресу arr в регистры rbx и rcx. (Третья и четвёртая инструкция эквивалентны, так как rax хранит адрес arr.)

Так мы можем работать с сдвинутыми адресами (например, обратиться к ненулевому индексу массива).

```
    .bss
arr:
    .zero   16

    .text
    .globl  main
main:
    leaq    arr(%rip), %rax
    movq    $1, 8(%rax)
    movq    8+arr(%rip), %rbx
    ret
```

Здесь мы создали массив для двух 64-bit чисел, и обращаемся ко второму, добавляя 8 к адресам. (Синтаксис этого, конечно, очень странный. В синтаксисе Intel это выглядит получше.)

5.1.5

Инструкция lea (load effective address) загружает адрес, который может быть представлен в форме специфического арифметического выражения. Изучите эту форму (ищите по запросу "effective address"), а затем напишите одну инструкцию, которая сложит значения двух регистров и сохранит результат в третий (инструкция add так не может, так как принимает только два аргумента).

5.1.6

Рассмотрим, как писать 32-bit код.

```
    .bss
arr:
    .zero   4

    .text
    .globl  main
main:
    leal    arr, %eax
    movl    $1, (%eax)
    ret
```

Здесь произошли следующие изменения:

- Использованы 32-битные регисты. Их названия начинаются с буквы e, вместо буквы r.
- Использованы команды с размерностью четыре байта, которые определяются суффиксом l, так как восьмибайтных команд нет.
- Отсутствует директива (%rip), так как теперь мы можем написать position independent executable (PIE) (для этого нужен дополнительный

флаг -no-pie).

Для компиляции в 32-bit необходимы дополнительные зависимости в системе, которые в некоторых дистрибутивах может быть непросто поставить. Выполнять компиляцию в 32-bit необязательно. Я просто хочу показать, что это такое.

Скомпилируем программу: gcc main.s -o main -m32 -no-pie. Флаг -m32 выполняет компиляцию в 32-bit.

Посмотрим теперь на суффиксы размерностей инструкций.

	AT&T	Intel
1	b	byte
2	w	word
4	l	dword
8	q	qword

Теперь посмотрим на регистр rax. Когда мы писали 32-bit код, мы использовали регистр eax. Однако этот же регистр есть и в 64-bit коде и он является младшими четырьмя байтами регистра rax.

Рассмотрим программу:

```
    .bss
arr:
    .zero   8

    .text
    .globl  main
main:
    movq    $0x1122334455667788, %rax
    movl    $0xeeff, %eax
    ret
```

С помощью дебаггера убедитесь, что обе инструкции действительно меняют значение регистра rax. Обратите также внимание на то, что инструкция movl также зануляет старшие четыре байта регистра.

Регистр eax имеет более маленькие составные части. Его младшие два байта покрываются регистром ax, первый байт покрывается регистром al, а второй байт покрывается регистром ah.

Рассмотрим теперь такую программу:

```
    .bss
arr:
    .zero   8
```

```
    .text
    .globl  main
main:
    movl    $0x12345678, %eax
    movw    $0xabcd, %ax
    movb    $0xee, %al
    movb    $0xff, %ah
    ret
```

Выполните её в дебаггере и обратите внимание на то, что инструкция movw занулила старшие два байта регистра eax. То же самое происходит и при использовании инструкции movb с регистрами al и ah.

5.1.7

Контент этого шага сложный и необязательный для полного понимания. Здесь я покажу, как читать машинный код.

Информацию обо всех инструкциях мы можем найти в instruction set architecture (ISA), который предоставляется производителем процессора. Научимся их читать. Откройте ISA AMD. (Это один из производителей процессоров архитектуры x86. Возможно, на вашем компьютере процессор AMD.) https://www.amd.com/content/dam/amd/en/documents/processor-tech-docs/programmer-references/24594.pdf

Давайте откроем страницу с описанием инструкции mov.

Mnemonic	Opcode	Description
MOV reg/mem8, reg8	88 /r	Move the contents of an 8-bit register to an 8-bit destination register or memory operand.
MOV reg/mem16, reg16	89 /r	Move the contents of a 16-bit register to a 16-bit destination register or memory operand.
MOV reg/mem32, reg32	89 /r	Move the contents of a 32-bit register to a 32-bit destination register or memory operand.
MOV reg/mem64, reg64	89 /r	Move the contents of a 64-bit register to a 64-bit destination register or memory operand.

MOV reg8, reg/mem8	8A /r	Move the contents of an 8-bit register or memory operand to an 8-bit destination register.
MOV reg16, reg/mem16	8B /r	Move the contents of a 16-bit register or memory operand to a 16-bit destination register.
MOV reg32, reg/mem32	8B /r	Move the contents of a 32-bit register or memory operand to a 32-bit destination register.
MOV reg64, reg/mem64	8B /r	Move the contents of a 64-bit register or memory operand to a 64-bit destination register.
MOV reg16/32/64/mem16, segReg	8C /r	Move the contents of a segment register to a 16-bit, 32-bit, or 64-bit destination register or to a 16-bit memory operand.
MOV segReg, reg/mem16	8E /r	Move the contents of a 16-bit register or memory operand to a segment register.
MOV AL, moffset8	A0	Move 8-bit data at a specified memory offset to the AL register.
MOV AX, moffset16	A1	Move 16-bit data at a specified memory offset to the AX register.
MOV EAX, moffset32	A1	Move 32-bit data at a specified memory offset to the EAX register.

MOV RAX, moffset64	A1	Move 64-bit data at a specified memory offset to the RAX register.
MOV moffset8, AL	A2	Move the contents of the AL register to an 8-bit memory offset.
MOV moffset16, AX	A3	Move the contents of the AX register to a 16-bit memory offset.
MOV moffset32, EAX	A3	Move the contents of the EAX register to a 32-bit memory offset.
MOV moffset64, RAX	A3	Move the contents of the RAX register to a 64-bit memory offset.
MOV reg8, imm8	B0 +rb ib	Move an 8-bit immediate value into an 8-bit register.
MOV reg16, imm16	B8 +rw iw	Move a 16-bit immediate value into a 16-bit register.
MOV reg32, imm32	B8 +rd id	Move an 32-bit immediate value into a 32-bit register.
MOV reg64, imm64	B8 +rq iq	Move an 64-bit immediate value into a 64-bit register.
MOV reg/mem8, imm8	C6 /0 ib	Move an 8-bit immediate value to an 8-bit register or memory operand.
MOV reg/mem16, imm16	C7 /0 iw	Move a 16-bit immediate value to a 16-bit register or memory operand.
MOV reg/mem32, imm32	C7 /0 id	Move a 32-bit immediate value to a 32-bit register or memory operand.

MOV reg/mem64, imm32	C7 /0 id	Move a 32-bit signed immediate value to a 64-bit register or memory operand.

Вспомним пример:

```
    .bss
arr:
    .zero   8

    .text
    .globl  main
main:
    movq    arr(%rip), %rax
    movq    $1, (%rax)
    ret

...
0000000000001129 <main>:
    1129:   48 8b 05 e1 2e 00 00    mov    0x2ee1(%rip),%rax        # 4011 <arr>
    1130:   48 c7 00 01 00 00 00    movq   $0x1,(%rax)
    1137:   c3                      ret
...
```

Первая инструкция mov записывает значение, хранящееся по адресу (не сам адрес), в регистр. Найдем opcode 0x8b в таблице. В первой колонке мы видим:

MOV regXX, reg/memXX

Здесь мы видим, что первых аргументом должен быть регистр, а вторым регистр или адрес в памяти (который трактуется, как содержимое по этому адресу). Разве не наоборот? Да. Дело в том, что в данном ISA используется синтаксис Intel, в котором первым аргументом стоит destination. Поэтому нам придётся визуально поменять эти аргументы местами. Мы посмотрим синтаксис Intel чуть позже.

Посмотрим чуть внимательнее на инструкцию:

48 8b 05 e1 2e 00 00

Обратите внимание: последние четыре байта — это число 0x00002ee1. Оно хранится байтами наоборот (как и всё в архитектуре x86) как есть. Что же говорит о том, что это адрес, а не регистр, и что перемещение идёт в регистр rax? Третий байт, равный 0x05. Это ModR/M байт, о наличии которого сообщает приписка /r во втором столбце. Можете найти таблицу

кодирования этого байта, она общая для всех инструкций. Я же просто скажу, что число 0x05 находится на пересечении регистра rax и disp32.

Байт 0x48 является префиксом инструкции и, полагаю, говорит о том, что число в инструкции имеет размер четыре байта.

Напомню, что метка — это не более, чем число. Поэтому, и число, и метка относятся к immXX.

5.1.8

Выполните анализ второй инструкции.

```
...
0000000000001129 <main>:
    1129:   48 8b 05 e1 2e 00 00     mov    0x2ee1(%rip),%rax        # 4011 <arr>
    1130:   48 c7 00 01 00 00 00     movq   $0x1,(%rax)
    1137:   c3                       ret
...
```

5.1.9

Мы уже увидели некоторую проблему в синтаксисе AT&T, поэтому не будет ошибкой попробовать перейти на синтаксис Intel.

```
    .intel_syntax   noprefix

    .bss
arr:
    .zero   8

    .text
    .globl  main
main:
    lea     rax, arr[rip]
    movq    [rax], 1
    ret
```

Что изменилось:

- Операнды в инструкциях поменялись местами.
- Вместо круглых скобок используются квадратные.
- Перед названиями регистров и константами больше нет знаков % и $.

Есть проблема в том, что обычно на gnu assembly не используют синтаксис intel, и в разных ассемблерах директивы разные. Мы будем использовать ассемблер nasm, так как он более известен среди ассемблеров с синтаксисом intel.

Итак, установите nasm. Обычно, его файлы имеют расширение .asm.

```
section .bss
arr:
    resb    8

section .text
global _start
_start:
    lea     rax, [rel arr] ; comment
    mov     qword [rax], 1
    ret
```

Это та же самая программа на nasm-е. Обратите внимание: директивы теперь выглядят совсем по другому.

- При объявлении секции присутствует ключевое слово section
- Вместо .globl теперь global
- Другой синтаксис обозначения размерности инструкции: вместо movq теперь mov qword
- Другой синтаксис обозначения того, что адрес относительно rip: вместо arr[rip] теперь [rel arr]
- Комментарии начинаются с символа ;.

В общем, хоть синтаксис и одинаковый, код для этих ассемблеров плохо совместим.

Обратите внимание на метку: теперь используется метка _start. Это стартовая метка для стандартного linker script-а ассемблера nasm. В отличии от gnu assembly исполнение здесь начинается непосредственно с нашего кода.

Выполним компиляцию и компоновку.

```
$ nasm -felf64 main.asm -o main.o
$ ld main.o -o main
```

При запуске программы вы получите SIGSEGV. Это происходит из-за инструкции ret, так как она переходит к внешней функции с помощью стека, а на данный момент у нас стек пустой.

Проверим entry point.

```
$ objdump -d main

main:     file format elf64-x86-64


Disassembly of section .text:

0000000000401000 <_start>:
  401000:   48 8d 05 f9 0f 00 00    lea   0xff9(%rip),%rax      # 402000 <__bss_start>
  401007:   48 c7 00 01 00 00 00    movq    $0x1,(%rax)
```

```
  40100e:   c3                      ret
$ readelf -h main
...
  Entry point address:               0x401000
...
```

Entry point (то есть, первая исполняемая инструкция) действительно метка _start.

5.1.10

Далее для удобства мы будем использовать ассемблер nasm.

Изучим арифметические инструкции. Инструкции add и sub для сложения и вычитания являются очевидными. Посмотрим на инструкцию mul для умножения.

В ISA AMD написано:

> Multiplies the unsigned byte, word, doubleword, or quadword value in the specified register or
> memory location by the value in AL, AX, EAX, or RAX and stores the result in AX, DX:AX,
> EDX:EAX, or RDX:RAX (depending on the operand size). It puts the high-order bits of the product in
> AH, DX, EDX, or RDX.
>
> If the upper half of the product is non-zero, the instruction sets the carry flag (CF) and overflow flag
> (OF) both to 1. Otherwise, it clears CF and OF to 0. The other arithmetic flags (SF, ZF, AF, PF) are
> undefined.

Данная инструкция принимает только один операнд (причём, не imm), а в качестве второго использует регистр *A*. При размерности операндов равной X младшие X байт результата помещается в регистр *A*, а старшие в регистр *D*. Во втором абзаце упоминаются некие флаги.

В x86 есть специальный регистр FLAGS/EFLAGS/RFLAGS, к которому у нас нет свободного доступа, но на который влияют многие инструкции. В данном случае при переполнения младших X байт результата нас дополнительно уведомляют об этом, включая биты, соответствующие флагам carry (перевод в следующий разряд) и overflow (переполнение). (Эти флаги не эквивалентны при знаковых вычислениях.) Здесь есть странность терминологии: часто под "включает флаг, если X" подразумевается также "выключает флаг, если не X".

```
section .text
global _start
_start:
```

```
    mov     eax, 0xffffffff
    mov     ebx, 0x2
    mul     ebx
    ret
```

Посмотрим на это в gdb (не забудьте поставить точку останова на метку _start). Мы можем вывести флаги с помощью команды p $eflags.

```
(gdb) p $eflags
$4 = [ CF IF OF ]
```

Помимо carry flag и overflow flag у нас также есть следующие арифметические флаги (это их примерные словесные описания, так как инструкции, работающие с ними, разные):

- Parity flag — чётность количества единичных битов в результате (когда это может быть полезно?).
- Zero flag — равен ли результат нулю.
- Sign flag — отрицателен ли результат.
- Direction flag — в какое направление будет идти итерирование по массиву при выполнении инструкций циклов. Мы сами управляем этим флагом.

На остальные флаги мы влиять не можем или они для нас бесполезны. IF — это interrupt flag, который выключается инструкцией cli. Что будет, если мы её выполним?

```
section .text
global _start
_start:
    cli
    ret
```

При выполнении этой инструкции мы получим SIGSEGV. Некоторые инструкции, например, cli, требуют большие права для выполнения. Условно, на текущий уровень прав ссылаются сегментные регистры, и мы на них влиять не можем (иначе в чём смысл такой защиты?).

5.1.11

Control flow выполняется с помощью условных прыжков, которые совершают переход к адресу при условии, что какой-то флаг включен. Например, инструкция jz выполняет прыжок, если включен ZF.

Мы можем написать if (rax == rbx) такой последовательностью инструкций:

```
    sub     rax, rbx
    jz      if_true
```

Если при выполнении инструкции вычитания получается ноль, то включается ZF. Если он включен, выполняется переход не к инструкции, следующей

за инструкцией jz, а к метке if_true.

Рассмотрим более полный пример. Так мы пишем if statement на языке ассемблера.

```
section .text
global _start
_start:
    mov     rax, 1
    mov     rbx, 2

    sub     rax, rbx
    jz      if_true
if_else:
    mov     rcx, 0
    jmp     if_end
if_true:
    mov     rcx, 1
if_end:

    ret
```

Проследите за логикой прыжков когда регистры rax и rbx равны, и когда не равны.

Не много ли меток (метка if_else присутствует для наглядности и не используется)? Давайте посмотрим, как скомпилирует if statement компилятор gcc.

```
int main() {
    int a = 1, b = 2, c;
    if (a == b) {
        c = 1;
    }
    else {
        c = 0;
    }
    return 0;
}
```

```
$ gcc main.c -S -masm=intel -o main.s
```

```
...
    cmp eax, DWORD PTR -8[rbp]
    jne .L2
    mov DWORD PTR -4[rbp], 1
    jmp .L3
.L2:
    mov DWORD PTR -4[rbp], 0
```

```
.L3:
...
```

Здесь есть следующие отличия:

- Вместо регистров используется стек. Конечно, я не могу заставить компилятор использовать регистры, ведь это смысл компилятора — использовать регистры эффективно. Поэтому он обращается к стеку, адрес которого содержится в регистре rbp. Чуть позже мы это изучим.
- Вместо инструкции sub используется инструкция cmp. Она выставляет флаги так же, как и инструкция sub, но сам destination регист не меняет.
- Вместо инструкции jz используется инструкция jne — jump not equal, которая выполняет прыжок, если ZF выключен.

В остальном, сама структура меток и прыжков полностью идентична нашей.

Изучите самостоятельно все conditional jump инструкции, например, jl (jump less), jg (jump greater) и т.д.

Помимо обычных меток есть и локальные метки, названия которых начинаются с символа ..

```
section .text
global _start

_start:
    jmp     .aba
.1:
    jmp     foo
.aba:
    jmp     .1

foo:
    jmp     .1
.1:
    jmp     $
```

В данном примере nasm транслирует название локальной метки .1 под меткой _start в _start.1, название локальной метки .aba под меткой _start в _start.aba, а название локальной метки .1 под меткой .foo в foo.1. Попробуйте самостоятельно проследить за порядком выполнения инструкций в этой программе.

Оператор $ транслируется в адрес начала текущей инструкции, поэтому в результате выполнения инструкции jmp $ мы перейдем в неё же, создав бесконечный цикл.

Существует также оператор $$, который транслируется в адрес начала текущей секции. Однако область его применения достаточно специфична,

поэтому мы на него здесь не посмотрим.

5.1.12

Напишите циклы while, do while и for.

5.1.13

В x86_64 есть следующие регистры общего назначения: rax, rbx, rcx, rdx, rdi, rsi, rbp, rsp, r8, r9, r10, r11, r12, r13, r14, r15. Изучить их составные регистры можно по этой картинке: https://upload.wikimedia.org/wikipedia/commons/1/15/Table_of_x86_Registers_svg.svg

У этих регистров есть роли. Некоторые из этих ролей задаются самой архитектурой (например, инструкция mul выполняла перемножение строго с регистром rax), некоторые задаются "стилем" написания кода, который называется Application Binary Interface (ABI).

Посмотрим на роли со стороны архитектуры. Для работы с массивами мы используем инструкции stos и lods. Первая инструкция копирует данные в массив, в вторая — их массива. Прочитаем их описание в ISA.

> STOS
> STOSB
> STOSW
> STOSD
> STOSQ
>
> Copies a byte, word, doubleword, or quadword from the AL, AX, EAX, or RAX registers to the
> memory location pointed to by ES:rDI and increments or decrements the rDI register according to the
> state of the DF flag in the rFLAGS register.
> If the DF flag is 0, the instruction increments the pointer; otherwise, it decrements the pointer. It
> increments or decrements the pointer by 1, 2, 4, or 8, depending on the size of the value being copied.

Рассмотрим код, выполняющий эту инструкцию.

```
section .bss
arr:
    resb 32

section .text
global _start
_start:
    mov     rax, 0x1122334455667788
    lea     rdi, arr
```

```
    stosq
    stosq
    stosq
    stosq
    ret
```

Здесь мы четыре раза кладём значение регистра rax в очередные восемь байт массива. Посмотрим содержимое массива после выполнения этих инструкций. (Здесь 32 — это количество блоков, x — это формат вывода (шестнадцатеричный), b — это размерность блока.)

```
...
(gdb) x/32xb &arr
0x402000:       0x88    0x77    0x66    0x55    0x44    0x33    0x22    0x11
0x402008:       0x88    0x77    0x66    0x55    0x44    0x33    0x22    0x11
0x402010:       0x88    0x77    0x66    0x55    0x44    0x33    0x22    0x11
0x402018:       0x88    0x77    0x66    0x55    0x44    0x33    0x22    0x11
...
```

Обратите внимание на второй абзац описания инструкции — там упоминается о том, что есть мы установим direction flag, то итерирование будет в обратную сторону. Чтобы включить и выключить DF, следует использовать инструкции std и cld. Помните, что он глобальный, что может поломать функции, которые вы вызываете.

```
section .bss
arr:
    resb    32

section .text
global _start
_start:
    std
    mov     rax, 0x1122334455667788
    lea     rdi, arr
    stosq
    stosq
    stosq
    stosq
    ret
```

После выполнения первой инструкции stosq у вас будет такая картина:

```
...
(gdb) x/32xb &arr
0x402000:       0x88    0x77    0x66    0x55    0x44    0x33    0x22    0x11
0x402008:       0x00    0x00    0x00    0x00    0x00    0x00    0x00    0x00
0x402010:       0x00    0x00    0x00    0x00    0x00    0x00    0x00    0x00
0x402018:       0x00    0x00    0x00    0x00    0x00    0x00    0x00    0x00
```

...

При выполнении второй инструкции вы получите SIGSEGV, так как значение регистра edi будет на 0x402000, а 0x401ff8.

Самостоятельно прочитайте описание инструкции lods. Рассмотрим программу:

```
section .data
arr:
    dw      0xaaaa, 0xbbbb, 0xcccc

section .text
global _start
_start:
    lea     rsi, arr
    lodsw
    lodsw
    lodsw
    ret
```

Чтобы было, что копировать из памяти, я создал инициализированный массив. Он должен находиться в секции data (так как секция bss хранит только нули). Директива dw означает, что мы хотим положить в секцию words (2 байта), которые перечислены далее (аналогично, есть директивы db, dd, и dq).

Выполните программу в дебаггере и убедитесь, что значние регистра al меняется после каждого выполнения инструкции lodsw на очередной элемент массива.

5.1.14

Напишите программу, которая выполнит копирование из одного нольтерминированного массива в другой с помощью функций lods и stos.

5.2. Application Binary Interface

5.2.1

Научимся писать функции.

```
section .text

set_1:
    mov     rax, 1
    ret

global _start
_start:
```

```
    mov     rax, 0
    call    set_1
    ret
```

Здесь мы объявили метку set_1, после которой идёт реализация нашей функции (понятия функции в языке ассемблера нет).

Мы вызываем функцию с помощью инструкции call. Эта инструкция кладёт на вершину стека адрес следующей инструкции (в данном случае, инструкции ret в функции _start) и выполняет прыжок к адресу, равному значению аргумента. Мы выходим из функции с помощью инструкции ret. Эта инструкция вытаскивает адрес с вершины стека и прыгает в него.

Стек — это массив в отдельном сегменте, на который мы можем записывать данные. Мы оперируем с ним так же, как и со структурой данных "стек". Адрес текущей вершины стека хранится в регистре rsp (sp = stack pointer). В отличии от других регистров общего назначения, из-за обилия инструкций, которые влияют на этот регистр, мы не можем его комфортно использовать так, как мы захотим. (Это же часто касается и регистра rbp (bp = base pointer), но о нём позже.)

Проверим это. Запустим программу в дебаггере и выведем стек, находясь в функции set_1. Если мы выполним команду ni, находясь на инструкции call, то мы не войдём в функцию. Чтобы в неё войти, следует использовать команду si.

```
Register group: general
rax            0x0                 0
rbx            0x0                 0
rcx            0x0                 0
rdx            0x0                 0
rsi            0x0                 0
rdi            0x0                 0
rbp            0x0                 0x0
rsp            0x7fffffffdc78      0x7fffffffdc78
r8             0x0                 0
r9             0x0                 0
r10            0x0                 0
r11            0x0                 0
r12            0x0                 0
r13            0x0                 0
r14            0x0                 0

 > 0x401000 <set_1>       mov    $0x1,%eax
   0x401005 <set_1+5>     ret
B+ 0x401006 <_start>      mov    $0x0,%eax
   0x40100b <_start+5>    call   0x401000 <set_1>
   0x401010 <_start+10>   ret
```

```
   0x401011              add    %al,(%rax)
   0x401013              add    %al,(%rax)
   0x401015              add    %al,(%rax)
   0x401017              add    %al,(%rax)
   0x401019              add    %al,(%rax)
   0x40101b              add    %al,(%rax)
   0x40101d              add    %al,(%rax)
   0x40101f              add    %al,(%rax)
   0x401021              add    %al,(%rax)
   0x401023              add    %al,(%rax)
   0x401025              add    %al,(%rax)

native process 16786 In: set_1                                L??   PC: 0x401000
Function "main" not defined.
--Type <RET> for more, q to quit, c to continue without paging--
Make breakpoint pending on future shared
library load? (y or [n]) [answered N; input not from terminal]
Breakpoint 1 at 0x401006

Breakpoint 1, 0x0000000000401006 in _start ()
(gdb) ni
0x000000000040100b in _start ()
(gdb) si
0x0000000000401000 in set_1 ()
(gdb) x/32xb $rsp
0x7fffffffdc78: 0x10    0x10    0x40    0x00    0x00    0x00    0x00    0x00
0x7fffffffdc80: 0x01    0x00    0x00    0x00    0x00    0x00    0x00    0x00
0x7fffffffdc88: 0x34    0xe0    0xff    0xff    0xff    0x7f    0x00    0x00
0x7fffffffdc90: 0x00    0x00    0x00    0x00    0x00    0x00    0x00    0x00
```

Верхние четыре байта на стеке — 0x00401010. Видно, что это же является адресом инструкции ret в функции _start. Также мы видим, что следующие четыре байта на стеке — 0x00000000, в которые и прыгнет инструкция ret в функции _start.

Чтобы передать в функцию аргументы, мы должны перед её вызовом положить их в стек. (Если вы ещё не заметили по выводу в дебаггере: когда мы кладём что-либо на стек, адрес его вершины уменьшается, а когда мы вытаскиваем что-либо из стека, адрес его вершины увеличивается.) Тогда эти аргументы будут под адресом возврата, и последний аргумент, который мы положили на стек, в вызванной функции будет находится по адресу [rsp + 8] (или не 8, если у вас другая разрядность программы).

```
section .text

sum:
    mov     rax, [rsp + 8]
```

```
    add     rax, [rsp + 16]
    ret

global _start
_start:
    push    2
    push    3
    call    sum
    add     rsp, 16
    ret
```

Назначение функции push очевидно — она кладёт свой аргумент на вершину стека. Мы можем имитировать её поведение комбинацией из двух инструкций:

```
    sub     rsp, 8
    mov     dword [rsp], 2
```

Чтобы вытащить аргументы, можно воспользоваться инструкцией pop. Но нам не нужно их вытаскивать — вместо этого мы можем просто сдвинуть адрес вершины стека.

5.2.2

Попробуем вызвать функцию printf. В 32-bit правила такие же: все аргументы должны быть положены на стек в обратном порядке их перечисления в сигнатуре (правда, мы видели, что практически сигнатура не налагает никаких ограничений).

Сделать это несколько проблематично: при комбинациях для компиляции и компоновки gcc + gcc и nasm + ld у меня были нетривиальные проблемы. (В первом случае gcc упорно добавлял PIC, а во втором не получалось нормально прикомпоновать libc.) При комбинации nasm + gcc получилось проще всего.

main.asm

```
section .data
format:
    db      "Check... %d", 0xA, 0x0

section .text

extern printf

global main
main:
    push    dword 123
    push    format
```

```
    call    printf
    add     esp, 8
    ret
```

```
$ nasm -felf32 main.asm -o main.o
$ gcc -m32 -no-pie main.o -o main
$ ./main
Check... 123
```

Внимание! При выполнении операций со стеком будет поддерживаться инвариант "значение rsp делится на 8". Однако, фунцкия printf требует, чтобы значение rsp делилось на 16. Простой способ этого добиться — добавлять 8 байт на стек, когда нужно. Необходимость таких сдвигов носит случайный характер, поэтому на вашей системе код может быть нерабочим, и вам понадобится выполнить сдвиг стека самостоятельно.

Чтобы nasm поверил, что метка printf появится на этапе компоновки, мы пользуемся директивой extern.

Под меткой format написана строка Check... %d\n\0. (Помните, что ассемблеры обычно сами не добавляют нули в конце строки. Например, в gnu assembly для создания строки без нуля и с нулём на конце используются директивы .ascii и .asciz соответственно.)

Обратите внимание — программа перестала завершаться SIGSEGV-ом. Это произошло потому, что gcc использовал свой скрипт для компоновки (поэтому же я также использовал метку main).

Сейчас мы впервые прикомпоновали нашу программу с функцией во внешнем объектном файле. Функция успешно вызвалась и выполнилась, потому что мы положили аргументы на стек именно так, как она ожидала. Есть некие стандарты о том, как следует передавать аргументы функциям, и как функции должны их считывать. Это необходимо, так как на таком уровне у нас нет никаких автоматических способов это контролировать (подобно тому, как в цельной программе на высокоуровневом языке C компилятор не позволит нам передать что-либо неправильно). Эти стандарты называются Application Binary Interface (ABI).

5.2.3

Напишите решение любой задачи (например, задачи про брокера Василия из главы "Контроль потока") на языке ассемблера, используя функции libc. Попробуйте хранить данные как в секции .bss/.data, так и в динамическом массиве, создав его с помощью функции malloc.

5.2.4

C ABI всё довольно запутано. По хорошему, мы будем стремиться использовать System V. В основном, нас интересуют только две вещи в очередном

ABI:

- Как передаются аргументы в функции.
- Какие регистры являются scratch/caller saved и какие регистры являются preserved/callee saved.

Посмотрим на таблицу с System V для 32-bit и 64-bit. (На самом деле, cdecl, который используется в C — это не совсем System V.) https://wiki.osdev.org/Calling_Conventions

Platform	Return Value	Parameter Registers	Additional Parameters	Stack Alignment	S
System V i386	eax, edx	none	stack (right to left)[1]		e
System V X86_64[2]	rax, rdx	rdi, rsi, rdx, rcx, r8, r9	stack (right to left)[1]	16-byte at call[3]	r
Microsoft x64	rax	rcx, rdx, r8, r9	stack (right to left)[1]	16-byte at call[3]	r
ARM (32-bit)	r0, r1	r0, r1, r2, r3	stack	8 byte[4]	r

Разберёмся для начала с 32-bit (i386 (история x86 полна сюрпризов)).

- Ранее я говорил, что результат функции возвращается в регистре *ax. Напомню также, что результат умножения хранится в регистрах *dx:*ax.
- Аргументы в функции не передаются в регистрах (мы этого и не наблюдали).
- Дополнительные (а, так как аргументов в регистрах и нет, то все) аргументы передаются на стеке, и кладутся вызываемой функцией справа налево (есть ABI, где наоборот) в порядке, указанном в сигнатуре функции.

Обратим внимание на scratch и preserved registers. Напомню, что регистры являются глобальными, поэтому за ними нужно очень внимательно наблюдать. Если вы выполняете какое-либо вычисление в регистрах, и в процессе этого вызываете функцию, значения регистров могут быть изменены. Для того, чтобы не потерять промежуточные вычисления, следует либо сохранить эти регистры на стек перед вызовом функции, либо поверить, что вызываемая функция их не изменит.

Итак, регистры, которые функция обязана не менять называются preserved (callee (вызываемый) saved), а регистры, которые функция может изменить (а поэтому их должна сохранить на стек вызывающая функция) называются scratch (caller (вызывающий) saved).

Зачем нужны scratch регистры, если без проблем можно сохранять все регистры? Для экономии времени выполнения, так как мы можем намеренно не хранить промежуточные вычисления в этих регистрах и не сохранять их вообще.

Помните, что "недобросовестная" функция без каких либо проблем может не сохранить preserved регистры (может быть, она была написана под особым

ABI).

```
foo:
    push    rbx
    mov     rcx, 1
    mov     rbx, 1
    mov     rax, rbx
    add     rax, rcx
    pop     rbx
    ret
```

Посмотрите на эту функцию-пример. Она использует для своих вычислений регистры rbx и rcx. При этом, так как регистр rbx является preserved, она его сохраняет на стеке, а затем восстанавливает.

Обратите внимание, что некоторые регистры в 32-bit и 64-bit имеют разную принадлежность. Например, esi в 32-bit является preserved, а rdi в 64-bit является scratch.

5.2.5

Обратим теперь внимание на 64-bit ABI. Помимо добавления новых регистров r8-15 (которые ничем не отличаются от первых восьми) и перераспределения их по группам scratch/preserved, теперь первые аргументы функций передаются в регистрах (так как операции с регистрами быстрее, а раньше регистров не хватало).

main.asm

```
section .data
format:
    db      "Check... %d", 0xA, 0x0

section .text

extern printf

global main
main:
    sub     rsp, 8 ; for printf
    mov     rdi, format
    mov     rsi, 123
    call    printf
    add     rsp, 8 ; for printf
    ret
```

```
$ nasm -felf64 main.asm -o main.o
$ gcc main.o -o main -no-pie
$ ./main
```

```
Check... 123
```

Посмотрев на таблицу, мы видим, что первый аргумент передаётся в регистре rdi, а второй — в регистре rsi.

(Здесь я добавил сдвиг стека для удовлетворения printf-а.)

Откроем таблицу системных вызовов: https://chromium.googlesource.com/chromiumos/docs/+/master/constants/syscalls.md

Мы готовы вызывать системные вызовы самостоятельно!

Начнём с простейшего СВ — fork. Посмотрите на таблицу. Функция кодируется значеним регистра rax. У СВ fork это значение — 0x39. Аргументы в СВ передаются в регистрах почти также, как и в функции в System V (единственное отличие — третий аргумент передаётся в регистре r10, вместо rcx).

```
section .data
format:
    db      "Check... %d", 0xA, 0x0

section .text

extern printf

global main
main:
    mov     rax, 0x39
    syscall

    push    rbp
    mov     rdi, format
    mov     rsi, 123
    call    printf
    pop     rbp
    ret
```

Чтобы выполнить СВ, мы заполняем регистры нужными значениями, а затем выполняем инструкцию syscall.

```
$ ./main
Check... 123
Check... 123
```

Для 32-bit дейтвия чуть другие.

```
section .data
format:
    db      "Check... %d", 0xA, 0x0
```

```
section .text

extern printf

global main
main:
    mov     eax, 0x2
    int     0x80

    push    dword 123
    push    format
    call    printf
    add     esp, 8
    ret
```

```
$ nasm -felf32 main.asm -o main.o
$ gcc -m32 -no-pie main.o -o main
$ ./main
Check... 123
Check... 123
```

Во-первых, номер СВ теперь 0x2 (другая таблица), а во-вторых, нам необходимо использовать инструкцию int с аргументов 0x80 (об этой инструкции позже).

5.2.6

Изучите системный вызов exit и с помощью неё завершите программу не SIGSEGV-ом.

5.2.7

Научимся выводить текст без помощи libc. Для этого нужно воспользоваться СВ write, которые, в отличии от СВ fork, имеет несколько аргументов: дескриптор stream-а (в нашем случае, stdout, то есть, 1), буфер для вывода (никакое форматирование не поддерживается) и длина буфера (буфер не обязан заканчиваться нулём).

```
section .data
buffer:
    db      "Check...", 0xA
length:
    dd      length - buffer

section .text
global _start
_start:
    mov     rax, 0x39
```

```
    syscall

    mov     rax, 0x1
    mov     rdi, 0x1
    mov     rsi, buffer
    mov     rdx, [length]
    syscall
    ret
```

```
$ nasm -felf64 main.asm -o main.o
$ ld main.o -o main
```

Чтобы добыть длину буфера, мы вычитаем из метки, находящейся после буфера метку, находящеюся до буфера. Длина числа, согласно документации — четыре байта.

Так мы можем написать программу, которая считывает строку и выводит её же.

```
%define LENGTH 1024

section .bss
buffer:
    resb    LENGTH
length:
    resb    4

section .text
global _start
_start:
    mov     rax, 0x0
    mov     rdi, 0x0
    mov     rsi, buffer
    mov     rdx, LENGTH
    syscall
    mov     [length], rax

    mov     rax, 0x1
    mov     rdi, 0x1
    mov     rsi, buffer
    mov     rdx, [length]
    syscall
    ret
```

Здесь я для удобства использовал макрос %define. Макросы работают в nasm-е так же, как и в gcc.

Если подать CB write длину, равную LENGTH, то оставшиеся в буфере нули мы никак в терминале не увидим. Тем не менее, следует вывести ровно

столько байт, сколько мы считали. СВ возвращают значение в регистре rax. СВ read возвращает количество символов, которые он считал. Сохраним это число и используем как длину сообщения, которое следует вывести.

5.2.8

Напишите решение любой задачи (например, задачи про брокера Василия из главы "Контроль потока") на языке ассемблера, используя системные вызовы.

5.2.9

Чтобы получить лучшее представление о том, как следует складировать локальные переменные и аргументы функций на стеке, обратимся к компилятору и посмотрим, как он это делает. Здесь же мы наконец поймем, для чего нужен регистр rbp.

main.c

```
int boo(int x, int y) {
    return x * y;
}

int foo(int x, int y) {
    int a = x + y;
    int b = x - y;
    return boo(a, b);
}

int main() {
    int x = 2;
    int y = 3;
    int z = foo(x, y);
}
```

При обычной компиляции вы будете получать такие cfi* директивы, которые используют дебаггеры, чтобы понять, где в стеке какие функции. Для удобства уберём их флагом -fno-asynchronous-unwind-tables .

Начнём в 64-bit.

$ gcc main.c -o main.S -S -masm=intel -fno-asynchronous-unwind-tables

Пойдем по каждой функции.

```
boo:
    endbr64
    push    rbp
    mov rbp, rsp
    mov DWORD PTR -4[rbp], edi
```

```
    mov DWORD PTR -8[rbp], esi
    mov eax, DWORD PTR -4[rbp]
    imul    eax, DWORD PTR -8[rbp]
    pop rbp
    ret
```

Инструкция (не директива) endbr64 является защитной: при определенной комбинации флагов в контрольных регистрах процессор будет производить прерывание, если вы совершите инструкцию call не в адрес, на котором стоит endbr64. (Это сделано для предотвращения атаки, при которой вы подменяете адрес и прыгаете не в начало функции.)

Манипуляции с регистром rbp пока не особо понятны. Этот регистр зачем-то временно сохраняется на стек и в него записывается значение регистра rsp.

После выполнения push rbp и mov rbp, rsp по адресу rbp хранится старый rbp, а по адресу rbp + 8 хранится адрес возврата. Аргументы передаются в регистрах edi и esi (так как имеют размерность четыре байта) и сразу же сохраняются на стек (обратите внимание, что это делается не инструкцией push, поэтому регистр rsp не меняется, и поэтому выполнение инструкции call здесь затрёт аргументы). После этого выполняется умножение об эти аргументы, сохранённые на стеке.

Почему аргументы были сначала помещены на стек, если можно перемножить сразу регистры? Из-за низкого уровня оптимизации. (Его можно установить флагами -O1, -O2 и т.д.) (На самом деле, эта задача весьма сложна, и я плохо понимаю, как её решать.)

```
foo:
    endbr64
    push    rbp
    mov rbp, rsp
    sub rsp, 24

    mov DWORD PTR -20[rbp], edi
    mov DWORD PTR -24[rbp], esi

    mov edx, DWORD PTR -20[rbp]
    mov eax, DWORD PTR -24[rbp]
    add eax, edx
    mov DWORD PTR -8[rbp], eax

    mov eax, DWORD PTR -20[rbp]
    sub eax, DWORD PTR -24[rbp]
    mov DWORD PTR -4[rbp], eax

    mov edx, DWORD PTR -4[rbp]
    mov eax, DWORD PTR -8[rbp]
```

```
    mov esi, edx
    mov edi, eax
    call    boo

    leave
    ret
```

Здесь я разделил программу на шесть блоков. Я в предыдущей функции говорил: "обратите внимание, что это делается не инструкцией push, поэтому регистр rsp не меняется, и поэтому выполнение инструкции call здесь затрёт аргументы". Теперь компилятор эту проблему исправил: он сдвинул регистр rsp на то расстояние, которое использовал в функции для локальных вычислений, а при выходе использовал инструкцию leave (прочитайте самостоятельно в ISA, что она делает).

Во втором блоке компилятор просто сохранил переданные аргументы на стек, но несколько дальше, чем в прошлый раз.

В третьем блоке компилятор выполнил вычисление int a = x + y;. Для этого он достал аргументы из стека в регистры edx и eax, сложил, и сохранил назад на стек. То же самое он сделал и в четвёртой строке для вычисления int b = x - y.

В пятом блоке компилятор достал a и b в регистры edx и eax, переместил их в регистры edi и esi и выполнил вызов функции boo.

Обратите внимание, что ячейки rbp - 12 и rbx - 16 не были использованы. Полагаю, это лишь из-за недостаточного уровня оптимизации.

Функцию main можете разобрать самостоятельно.

5.2.10

Посмотрим, как gcc компилирует в 32-bit ту же программу. Из-за того, что на стеке теперь хранятся и аргументы функций, манипуляции с ним стали чуть сложнее.

$ gcc main.c -o main.S -S -m32 -masm=intel -fno-asynchronous-unwind-tables -fno-pie

```
boo:
    push    ebp
    mov ebp, esp
    mov eax, DWORD PTR [ebp+8]
    imul    eax, DWORD PTR [ebp+12]
    pop ebp
    ret
```

Посмотрите, где теперь функция boo берёт аргументы. Напомню, что в 32-bit мы передаём все аргументы на стеке, причем кладём их справа налево

в порядке, который указан в сигнатуре функции. То есть, левые аргументы будут ближе к верху стека (иметь меньшие адреса). Мы знаем, что в ebp хранится старый ebp, а в ebp + 4 хранится адрес возврата. Тогда в ebp + 8 хранится первый аргумент, а в ebp + 12 хранится второй аргумент.

```
foo:
    push    ebp
    mov ebp, esp
    sub esp, 16

    mov edx, DWORD PTR [ebp+8]
    mov eax, DWORD PTR [ebp+12]
    add eax, edx
    mov DWORD PTR [ebp-8], eax

    mov eax, DWORD PTR [ebp+8]
    sub eax, DWORD PTR [ebp+12]
    mov DWORD PTR [ebp-4], eax

    push    DWORD PTR [ebp-4]
    push    DWORD PTR [ebp-8]
    call    boo
    add esp, 8

    leave
    ret
```

Здесь фаза сохранения аргументов из регистров на стек отсутствует (так как они изначально на стеке). Здесь стек использован несколько оптимальнее: локальная переменная a хранится в ebp - 8, а локальная переменная b хранится в ebp - 4. При этом стек всё равно сдвигается на 16: sub esp, 16. Полагаю, чтобы значение esp по прежнему делилось на 16.

Я заметил пару забаных вещей по сравнению с 32-bit компиляцией.

- Здесь появилась инструкция add esp, 8. Её смысл понятен — она возвращает вершину стека на место после того, как мы положили на него аргументы функции boo. Но из-за логики работы инструкции leave, в этой инструкции add нет смысла.
- Почему-то в 64-bit адреса пишутся в формате -4[rbp], а в 32-bit в формате [ebp-4]. (Насколько я знаю, они ничем не отличаются.)

5.2.11

Проект: Компилятор

Это масштабный, сложный и интересный проект.

Напишите компилятор для простейшего языка C подобного языка, который

будет переписывать код на язык ассемблера. Ключевым здесь является наличие функций, локальных переменных и ветвлений.

Примерный план, как это можно сделать. Для простоты, язык будет иметь только statement-ы.

- Считать текст и разбить его на statement-ы.
- Создать структуры для statement-ов.
- Один из statement-ов — блок, содержащий список statement-ов.
- Остальные statement-ы:
  - if — if (a == 0) BLOCK (по хорошему, аргумент должен быть expression, но можно проверять и только на ноль)
  - definition — def a, b
  - assignment — a = 5
  - addition — a = b + 4 (по хорошему, это должен быть expression)
  - function definition — func foo(a, b) BLOCK
  - function call — foo(a, b)
- Statement-ы должны быть организованы в дерево, и компиляция выполняется обходом по этому дереву.
- Компиляция каждого из statement-ов, после изучения вывода компилятора gcc, должна быть очевидной.

5.3. Компоновка

5.3.1

В коде на языке C мы без ограничений можем вызывать функции, которые определены в отдельном объектном файле, который был ассемблирован nasm-ом или любым другим ассемблером. Необходимо лишь, чтобы названия меток были одинаковыми.

main.c

```
#include <stdio.h>

int my_fork();

int main() {
    int x = my_fork();
    printf("%d\n", x);
}
```

fork.asm

```
section .text
global my_fork
my_fork:
    mov     rax, 0x39
    syscall
```

```
$ nasm -felf64 fork.asm -o fork.o
$ gcc main.c -c -o main.o
$ gcc main.o fork.o -o main
$ ./main
51517
0
```

Так ли реализована функция fork, которую мы вызываем в C? Проверим это с помощью дебаггера.

```
#include <unistd.h>

int main() {
    fork();
}
```

```
   0x7ffff7ceab30 <__GI__Fork>    endbr64
   0x7ffff7ceab34 <__GI__Fork+4>  mov   %fs:0x10,%rax
   0x7ffff7ceab3d <__GI__Fork+13> xor   %r8d,%r8d
   0x7ffff7ceab40 <__GI__Fork+16> xor   %edx,%edx
   0x7ffff7ceab42 <__GI__Fork+18> xor   %esi,%esi
   0x7ffff7ceab44 <__GI__Fork+20> mov   $0x1200011,%edi
   0x7ffff7ceab49 <__GI__Fork+25> lea   0x2d0(%rax),%r10
   0x7ffff7ceab50 <__GI__Fork+32> mov   $0x38,%eax
 > 0x7ffff7ceab55 <__GI__Fork+37> syscall
```

В моём случае функция сделала некоторую подготовительную работу и вызвала системный вызов 0x38 (что видно по инструкции перед syscall). Номер 0x38 имеет CB clone, частным случаем которого является fork.

Сделаем еще один эксперимент и проверим, какой системный вызов использует функция malloc.

```
#include <stdlib.h>

int main() {
    void *a = malloc(1);
}
```

```
Register group: general
rax            0xc                12
rbx            0x21000            135168
rcx            0x2b0              688
rdx            0x0                0
rsi            0x7ffff7e1ac80     140737352150144
rdi            0x0                0
rbp            0x7ffff7e22218     0x7ffff7e22218 <__curbrk>
rsp            0x7fffffffd998     0x7fffffffd998
r8             0x2                2
```

```
r9             0x7ffff7fc9040     140737353912384
r10            0x7ffff7e1ace0     140737352150240
r11            0x7ffff7e1ace0     140737352150240
r12              0x0              0
r13            0x7ffff7e1ace0     140737352150240
r14             0x1000             4096
```

```
   0x7ffff7d1a820 <__brk>         endbr64
   0x7ffff7d1a824 <__brk+4>       mov    $0xc,%eax
 > 0x7ffff7d1a829 <__brk+9>       syscall
   0x7ffff7d1a82b <__brk+11>      mov    0xff62e(%rip),%rdx        # 0x7ffff7e19e60
   0x7ffff7d1a832 <__brk+18>      mov    %rax,(%rdx)
   0x7ffff7d1a835 <__brk+21>      cmp    %rdi,%rax
   0x7ffff7d1a838 <__brk+24>      jb     0x7ffff7d1a840 <__brk+32>
   0x7ffff7d1a83a <__brk+26>      xor    %eax,%eax
   0x7ffff7d1a83c <__brk+28>      ret
   0x7ffff7d1a83d <__brk+29>      nopl   (%rax)
   0x7ffff7d1a840 <__brk+32>      mov    0xff5c9(%rip),%rax        # 0x7ffff7e19e10
   0x7ffff7d1a847 <__brk+39>      movl   $0xc,%fs:(%rax)
   0x7ffff7d1a84e <__brk+46>      mov    $0xffffffff,%eax
   0x7ffff7d1a853 <__brk+51>      ret
   0x7ffff7d1a854                 cs nopw 0x0(%rax,%rax,1)
   0x7ffff7d1a85e                 xchg   %ax,%ax
```

На этот раз подготовительного кода было намного больше, и вам скорее всего потребуется несколько попыток, чтобы разобраться в нём и не потеряться. В конечном итоге будет вызван СВ с номером 0xC, то есть, brk.

Скажу просто как факт: malloc использует два системного вызова в зависимости от ситуации: brk и mmap/mmunmap.

СВ brk изменяет размер сегмента .data (данный СВ считается устаревшим и информацию о нём найте непросто), а mmap/munmap работают примерно как malloc/free, но дают больший контроль над адресами, которые мы можем получить.

В любом случае, мы не можем делать системный вызов на каждый вызов malloc-а, поскольку выполнение любого системного вызова довольно медленное (поэтому, например, printf выполняет буферизацию). Вместо этого для реализации malloc-а мы должны один раз выделить большой отрезок памяти а затем построить на нём структуру данных "куча/heap" (не путать с кучей из теории алгоритмов (здесь терминология совпала и на английском языке)).

Сейчас мы выполнили reverse engineering. Будьте аккуратны — это в общем случае незаконно.

5.3.2

В любой ранее написанной программе замените как можно больше libc-функций (fork, open, printf), выполняющих системные вызовы, на свои функции, выполняющие системные вызовы и написанные на языке ассемблера.

5.3.3

Попробуем прикомпоновать функцию, написанную на языке ассемблера, к программе, написанной на языке C++. Воспользуемся GNU GCC компилятором — g++. Его использование ничем не отличается, от использования компилятора gcc.

main.cpp

```
#include <stdio.h>

int foo(int x);

int main() {
    printf("%d\n", foo(2));
    return 0;
}
```

foo.asm

```
section .text
global foo
foo:
    mov     rax, rdi
    add     rax, 1
    ret
```

```
$ nasm -felf64 foo.asm -o foo.o
$ g++ main.cpp -c -o main.o
$ g++ main.o foo.o -o main
/usr/bin/ld: main.o: in function ‘main’:
main.cpp:(.text+0xe): undefined reference to ‘foo(int)’
collect2: error: ld returned 1 exit status
```

Хм, компоновщик сообщает, что функция foo не определена. На самом деле, он немного лжёт и говорит о другой функции. Посмотрим на файл main.o подробнее с помощью readelf.

```
$ readelf -a main.o
...
Symbol table '.symtab' contains 7 entries:
   Num:    Value          Size Type    Bind   Vis      Ndx Name
     0: 0000000000000000     0 NOTYPE  LOCAL  DEFAULT  UND
```

```
    1: 0000000000000000     0 FILE    LOCAL  DEFAULT  ABS main.cpp
    2: 0000000000000000     0 SECTION LOCAL  DEFAULT    1 .text
    3: 0000000000000000     0 SECTION LOCAL  DEFAULT    5 .rodata
    4: 0000000000000000    47 FUNC    GLOBAL DEFAULT    1 main
    5: 0000000000000000     0 NOTYPE  GLOBAL DEFAULT  UND _Z3fooi
    6: 0000000000000000     0 NOTYPE  GLOBAL DEFAULT  UND printf
...
```

Секция .symtab, которую автоматически генерирует ассемблер, хранит метки, используемые в файле. Посмотрите на список: в нём нет метки foo, но есть некая метка _Z3fooi. Это и есть наша функция foo.

Если вы ещё не знали, в C++ существует перегрузка функций (function overloading). Это означает, что вы можете определить несколько функций, которые имеют одинаковое название, на разную сигнатуру, и компилятор будет подставлять нужную функцию, исходя из типов аргументов.

main.cpp

```
#include <stdio.h>
#include <string.h>

int foo(int x) {
    return x + 1;
}

int foo(const char *str) {
    return strlen(str);
}

int main() {
    printf("%d %d\n", foo(2), foo("Hello"));
    return 0;
}
```

```
$ g++ main.cpp -o main
$ ./main
3 5
```

Компоновщик не знает о перегрузке функций, поэтому компилятор языка C++ просто кодирует сигнутуру функции прямо в название её метки. Это называется mangling (на русском, полагаю, можно назвать экранированием). Чтобы расшифровать метку, можно использовать программу c++filt.

```
$ c++filt
_Z3fooi
foo(int)
```

Чтобы g++ не выполнил mangle, следует в объявлении функции написать директиву extern "C". (Либо же можно в файле ассемблера объявить метку

_Z3fooi, что, конечно, неудобно.)

```
#include <stdio.h>

extern "C" int foo(int x);

int main() {
    printf("%d\n", foo(2));
    return 0;
}
```

```
$ g++ main.cpp -c -o main.o
$ g++ main.o foo.o -o main
$ ./main
3
```

5.3.4

Разберёмся с тем, какие секции присутствуют в объектном файле, и как компоновщих ищет и подставляет адреса меток из разных модулей.

main.asm

```
section .data
buffer:
    db      "Check...", 0xA
length:
    dd      length - buffer

section .text
extern exit
global _start
_start:
    mov     rax, 0x1
    mov     rdi, 0x1
    mov     rsi, buffer
    mov     rdx, [length]
    syscall

    call    exit
```

Выполним ассемблирование, а затем дизассемблирование объектного файла.

```
$ nasm -felf64 main.asm -o main.o
$ objdump -d main.o

main.o:     file format elf64-x86-64
```

```
Disassembly of section .text:

0000000000000000 <_start>:
   0:   b8 01 00 00 00          mov    $0x1,%eax
   5:   bf 01 00 00 00          mov    $0x1,%edi
   a:   48 be 00 00 00 00 00    movabs $0x0,%rsi
  11:   00 00 00
  14:   48 8b 14 25 00 00 00    mov    0x0,%rdx
  1b:   00
  1c:   0f 05                  syscall
  1e:   e8 00 00 00 00          call   23 <length+0x1a>
```

Посмотрите на инструкцию call. Если вы посмотрите в ISA, то увидете, что после такого opcode-а идёт адрес, в который следует прыгнуть. Но здесь записано число ноль, как временное значение, так как адреса метки exit мы пока не знаем.

Здесь идёт речь не об адресе в файле, а об адресе в оперативной памяти во время исполнения (возможно, относительно регистра rip, если код position independent). А проектировать то, как будет лежать программа в оперативной памяти, будет компоновщик с использованием linker script-а.

Посмотрим на структуру объектного файла.

```
$ readelf -a main.o
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:                              REL (Relocatable file)
  Machine:                           Advanced Micro Devices X86-64
  Version:                           0x1
  Entry point address:               0x0
  Start of program headers:          0 (bytes into file)
  Start of section headers:          64 (bytes into file)
  Flags:                             0x0
  Size of this header:               64 (bytes)
  Size of program headers:           0 (bytes)
  Number of program headers:         0
  Size of section headers:           64 (bytes)
  Number of section headers:         8
  Section header string table index: 4

Section Headers:
  [Nr] Name              Type            Address          Offset
```

```
     Size              EntSize          Flags  Link  Info  Align
  [ 0]                   NULL             0000000000000000  00000000
     0000000000000000  0000000000000000           0     0     0
  [ 1] .data             PROGBITS         0000000000000000  00000240
     000000000000000d  0000000000000000  WA       0     0     4
  [ 2] .bss              NOBITS           0000000000000000  00000250
     0000000000000010  0000000000000000  WA       0     0     4
  [ 3] .text             PROGBITS         0000000000000000  00000250
     0000000000000023  0000000000000000  AX       0     0     16
  [ 4] .shstrtab         STRTAB           0000000000000000  00000280
     0000000000000037  0000000000000000           0     0     1
  [ 5] .symtab           SYMTAB           0000000000000000  000002c0
     00000000000000f0  0000000000000018           6     8     8
  [ 6] .strtab           STRTAB           0000000000000000  000003b0
     000000000000002a  0000000000000000           0     0     1
  [ 7] .rela.text        RELA             0000000000000000  000003e0
     0000000000000048  0000000000000018           5     3     8
Key to Flags:
  W (write), A (alloc), X (execute), M (merge), S (strings), I (info),
  L (link order), O (extra OS processing required), G (group), T (TLS),
  C (compressed), x (unknown), o (OS specific), E (exclude),
  D (mbind), l (large), p (processor specific)

There are no section groups in this file.

There are no program headers in this file.

There is no dynamic section in this file.

Relocation section '.rela.text' at offset 0x3e0 contains 3 entries:
  Offset          Info           Type           Sym. Value    Sym. Name + Addend
00000000000c  000200000001 R_X86_64_64       0000000000000000 .data + 0
000000000018  00020000000b R_X86_64_32S      0000000000000000 .data + 9
00000000001f  000800000002 R_X86_64_PC32     0000000000000000 exit - 4
No processor specific unwind information to decode

Symbol table '.symtab' contains 10 entries:
   Num:    Value          Size Type    Bind   Vis      Ndx Name
     0: 0000000000000000     0 NOTYPE  LOCAL  DEFAULT  UND
     1: 0000000000000000     0 FILE    LOCAL  DEFAULT  ABS main.asm
     2: 0000000000000000     0 SECTION LOCAL  DEFAULT    1 .data
     3: 0000000000000000     0 SECTION LOCAL  DEFAULT    2 .bss
     4: 0000000000000000     0 SECTION LOCAL  DEFAULT    3 .text
     5: 0000000000000000     0 NOTYPE  LOCAL  DEFAULT    1 buffer
     6: 0000000000000009     0 NOTYPE  LOCAL  DEFAULT    1 length
     7: 0000000000000000     0 NOTYPE  LOCAL  DEFAULT    2 zeros
```

```
8: 0000000000000000     0 NOTYPE  GLOBAL DEFAULT  UND exit
9: 0000000000000000     0 NOTYPE  GLOBAL DEFAULT    3 _start
```

No version information found in this file.

Мы видим восемь секций: , .data, .bss, .text, .shstrtab, .symtab, .strtab, .rela.text.

В нашем файле на языке ассемблера мы написали только три секции: .data, .bss и .text. Также, мы часто сами пишем секцию .rodata (read-only data), которая во время исполнения будет неизменяемой. Обычно в неё попадают const char*.

Остальные секции являются вспомогательными и генерируются ассемблером. На секцию .symtab мы уже смотрели. Секции .shstrtab и .strtab в данном файле являются пустыми, и мы их разбирать не будем.

Посмотрим на таблицу внимательнее.

```
Section Headers:
  [Nr] Name              Type            Address          Offset
       Size              EntSize         Flags  Link  Info  Align
  [ 0]                   NULL            0000000000000000  00000000
       0000000000000000  0000000000000000           0     0     0
  [ 1] .data             PROGBITS        0000000000000000  00000240
       000000000000000d  0000000000000000  WA       0     0     4
...
```

Поле Address получит смысл только после выполнения компоновки. Поле Offset — это позиция секции в самом файле. Например, позиция секции .data — 0x240 байт. Проверим это.

```
$ hd main.o
...
00000240  43 68 65 63 6b 2e 2e 2e  0a 09 00 00 00 00 00 00  |Check...........|
...
```

Действительно, в этом месте начинается строка Check....

Поле Flags хранит, помимо не особо интересных нам флагов, флаги прав (permissions). Например, мы видим, что секция .data имеет флаг W, что означает, что она writable (изменяемая).

Посмотрим на секцию .rela.text.

```
Relocation section '.rela.text' at offset 0x3e0 contains 3 entries:
  Offset          Info           Type           Sym. Value    Sym. Name + Addend
00000000000c  000200000001 R_X86_64_64       0000000000000000 .data + 0
000000000018  00020000000b R_X86_64_32S      0000000000000000 .data + 9
00000000001f  000800000002 R_X86_64_PC32     0000000000000000 exit - 4
```

Посмотрите на третью строку таблицы. В ней записана метка exit, и адрес 0x1f. Посмотрите на инструкцию call:

```
...
  1e:   e8 00 00 00 00          call   23 <length+0x1a>
```

0x1f — это как раз место, в которое следует положить адрес метки. Вот и вся цепочка действий для подстановки адресов.

5.3.5

Добавим в нашу программу модуль с реализацией функции exit, скомпонуем их вместе и посмотрим на структуру полученного исполняемого файла.

exit.asm

```
global exit
exit:
    mov     rax, 0x3C
    mov     rdi, 0x0
    syscall
```

```
$ nasm -felf64 exit.asm -o exit.o
$ ld exit.o -o exit
ld: warning: cannot find entry symbol _start; defaulting to 0000000000401000
$ ./exit
$ objdump -d exit

exit:     file format elf64-x86-64


Disassembly of section .text:

0000000000401000 <exit>:
  401000:   b8 3c 00 00 00          mov    $0x3c,%eax
  401005:   bf 00 00 00 00          mov    $0x0,%edi
  40100a:   0f 05                   syscall
```

Если мы скомпонуем только этот файл, то компоновщик не найдет метку _start, и сделать entry point-ом начало секции (в данном случае, функцию exit).

```
$ ld main.o exit.o -o main
$ readelf -a main
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:               0x401000
  Start of program headers:          64 (bytes into file)
  Start of section headers:          8584 (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:         7
  Section header string table index: 6

Section Headers:
  [Nr] Name              Type            Address          Offset
       Size              EntSize         Flags  Link  Info  Align
  [ 0]                   NULL            0000000000000000  00000000
       0000000000000000  0000000000000000           0     0     0
  [ 1] .text             PROGBITS        0000000000401000  00001000
       000000000000003c  0000000000000000  AX       0     0     16
  [ 2] .data             PROGBITS        0000000000402000  00002000
       000000000000000d  0000000000000000  WA       0     0     4
  [ 3] .bss              NOBITS          0000000000402010  0000200d
       0000000000000010  0000000000000000  WA       0     0     4
  [ 4] .symtab           SYMTAB          0000000000000000  00002010
       0000000000000108  0000000000000018           5     6     8
  [ 5] .strtab           STRTAB          0000000000000000  00002118
       0000000000000044  0000000000000000           0     0     1
  [ 6] .shstrtab         STRTAB          0000000000000000  0000215c
       000000000000002c  0000000000000000           0     0     1
Key to Flags:
  W (write), A (alloc), X (execute), M (merge), S (strings), I (info),
  L (link order), O (extra OS processing required), G (group), T (TLS),
  C (compressed), x (unknown), o (OS specific), E (exclude),
  D (mbind), l (large), p (processor specific)

There are no section groups in this file.

Program Headers:
  Type           Offset             VirtAddr           PhysAddr
                 FileSiz            MemSiz              Flags  Align
  LOAD           0x0000000000000000 0x0000000000400000 0x0000000000400000
                 0x00000000000000e8 0x00000000000000e8  R      0x1000
```

```
  LOAD           0x0000000000001000 0x0000000000401000 0x0000000000401000
                 0x000000000000003c 0x000000000000003c  R E    0x1000
  LOAD           0x0000000000002000 0x0000000000402000 0x0000000000402000
                 0x000000000000000d 0x0000000000000020  RW     0x1000

 Section to Segment mapping:
  Segment Sections...
   00
   01     .text
   02     .data .bss

There is no dynamic section in this file.

There are no relocations in this file.
No processor specific unwind information to decode

Symbol table '.symtab' contains 11 entries:
   Num:    Value          Size Type    Bind   Vis      Ndx Name
     0: 0000000000000000     0 NOTYPE  LOCAL  DEFAULT  UND
     1: 0000000000000000     0 FILE    LOCAL  DEFAULT  ABS main.asm
     2: 0000000000402000     0 NOTYPE  LOCAL  DEFAULT    2 buffer
     3: 0000000000402009     0 NOTYPE  LOCAL  DEFAULT    2 length
     4: 0000000000402010     0 NOTYPE  LOCAL  DEFAULT    3 zeros
     5: 0000000000000000     0 FILE    LOCAL  DEFAULT  ABS exit.asm
     6: 0000000000401000     0 NOTYPE  GLOBAL DEFAULT    1 _start
     7: 000000000040200d     0 NOTYPE  GLOBAL DEFAULT    3 __bss_start
     8: 000000000040200d     0 NOTYPE  GLOBAL DEFAULT    2 _edata
     9: 0000000000402020     0 NOTYPE  GLOBAL DEFAULT    3 _end
    10: 0000000000401030     0 NOTYPE  GLOBAL DEFAULT    1 exit

No version information found in this file.
```

Начнём со списка секций. Здесь есть одно изменение: пропала секция .rela.text. Это логично, так как она уже была использована по назначению — для подстановки адресов меток.

Теперь посмотрите на эту таблицу:

```
 Section to Segment mapping:
  Segment Sections...
   00
   01     .text
   02     .data .bss
```

Загрузчик, который будет создавать процесс, выполняющий нашу программу, будет загружать его по сегментам, а не секциям. Данная таблица говорит о наличии трёх сегментов:

- Первый фиктивный
- Второй с секцией .text
- Третий с секциями .data и .bss

В одном сегменте может быть несколько секций.

Посмотрите на поле Address у секций:

```
Section Headers:
  [Nr] Name              Type            Address          Offset
       Size              EntSize         Flags  Link  Info  Align
  [ 0]                   NULL            0000000000000000  00000000
       0000000000000000  0000000000000000           0     0     0
  [ 1] .text             PROGBITS        0000000000401000  00001000
       000000000000003c  0000000000000000  AX       0     0     16
  [ 2] .data             PROGBITS        0000000000402000  00002000
       000000000000000d  0000000000000000  WA       0     0     4
  [ 3] .bss              NOBITS          0000000000402010  0000200d
       0000000000000010  0000000000000000  WA       0     0     4
...
```

Ранее там были записаны нули. Теперь же там записаны числа, например, 0x401000 для секции .text, 0x402000 для секции .data и 0x402010 для секции .bss. Какое это влияние оказывает на загрузчик? На самом деле, никакое. Эти числа здесь только как подсказка о том, где в процессе исполнения будут находиться в памяти данные секции.

Запустим программу в дебаггере.

```
B+> 0x401000 <_start>      mov    $0x1,%eax
   0x401005 <_start+5>     mov    $0x1,%edi
   0x40100a <_start+10>    movabs $0x402000,%rsi

(gdb) x/8c 0x402000
0x402000:       67 'C'  104 'h' 101 'e' 99 'c'  107 'k' 46 '.'  46 '.'  46 '.'
```

Действительно, выполнение начинается с адреса 0x401000, а по адресу 0x402000 записана наша строка.

Посмотрим на сегменты. (Здесь program header = заголовки сегментов. Отличная терминология.)

```
Program Headers:
  Type           Offset             VirtAddr           PhysAddr
                 FileSiz            MemSiz             Flags  Align
  LOAD           0x0000000000000000 0x0000000000400000 0x0000000000400000
                 0x00000000000000e8 0x00000000000000e8  R      0x1000
  LOAD           0x0000000000001000 0x0000000000401000 0x0000000000401000
                 0x000000000000003c 0x000000000000003c  R E    0x1000
  LOAD           0x0000000000002000 0x0000000000402000 0x0000000000402000
```

```
                0x000000000000000d 0x0000000000000020  RW      0x1000
```

Начнём с поля Offset. Здесь снова речь о позиции в файле. Первый сегмент начинается с позиции 0x0 и по сути делает так, чтобы весь исполняемый файл был покрыт сегментами. Обратите внимание, что в секциях поле Offset тоже изменилось и теперь оно соответствует сегментам. В самом файле сегменты с секциями и сами секции не дублируются.

Теперь о поле VirtAddr. Именно на это поле смотрит загрузчик, когда определяет, где в памяти должен быть данный сегмент. Аналогичное поле в таблице сегментов совпадает с ним. Для сегмента .bss это значение сдвинуто на 0x10. При этом, Offset сдвинут только на 0xd, то есть, формат elf старается не хранить в себе лишние байты ради выравнивания во время исполнения.

Посмотрите на поле Flags, которое говорит о правах (permissions) для данных сегментов. Например, сегмент с секцией .text можно исполнять, но нельзя менять, а сегмент с секциями .data и .bss можно менять, но нельзя исполнять. При нарушении этого мы получим SIGSEGV.

Поле PhysAddr в современных ОС не используется.

5.3.6

Выясните, по каком критерию выясняется, что программа является position independent, и как это влияет на адреса сегментов.

5.3.7

Далее пойдёт обрывочная информация, так как нет полной и чёткой документации о том, как работает компоновщик, и я обнаруживал сюрпризы.

Попробуем изменить использовать нестандартное название секции.

main.asm

```
section .code
global _start
_start:
    mov     rax, 0x3C
    mov     rdi, 0x0
    syscall
```

```
$ nasm -felf64 main.asm -o main.o
$ ld main.o -o main
$ ./main
Segmentation fault (core dumped)
$ readelf -a main
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:               0x401000
  Start of program headers:          64 (bytes into file)
  Start of section headers:          4328 (bytes into file)
  Flags:                             0x0
  Size of this header:               64 (bytes)
  Size of program headers:           56 (bytes)
  Number of program headers:         1
  Size of section headers:           64 (bytes)
  Number of section headers:         5
  Section header string table index: 4

Section Headers:
  [Nr] Name              Type            Address          Offset
       Size              EntSize         Flags  Link  Info  Align
  [ 0]                   NULL            0000000000000000  00000000
       0000000000000000  0000000000000000           0     0     0
  [ 1] .code             PROGBITS        0000000000401000  00001000
       000000000000000c  0000000000000000  A       0     0     1
  [ 2] .symtab           SYMTAB          0000000000000000  00001010
       0000000000000090  0000000000000018           3     2     8
  [ 3] .strtab           STRTAB          0000000000000000  000010a0
       0000000000000022  0000000000000000           0     0     1
  [ 4] .shstrtab         STRTAB          0000000000000000  000010c2
       0000000000000021  0000000000000000           0     0     1
Key to Flags:
  W (write), A (alloc), X (execute), M (merge), S (strings), I (info),
  L (link order), O (extra OS processing required), G (group), T (TLS),
  C (compressed), x (unknown), o (OS specific), E (exclude),
  D (mbind), l (large), p (processor specific)

There are no section groups in this file.

Program Headers:
  Type           Offset             VirtAddr           PhysAddr
                 FileSiz            MemSiz              Flags  Align
  LOAD           0x0000000000000000 0x0000000000400000 0x0000000000400000
                 0x000000000000100c 0x000000000000100c  R      0x1000

 Section to Segment mapping:
```

```
 Segment Sections...
  00     .code

There is no dynamic section in this file.

There are no relocations in this file.
No processor specific unwind information to decode

Symbol table '.symtab' contains 6 entries:
   Num:    Value          Size Type    Bind   Vis      Ndx Name
     0: 0000000000000000     0 NOTYPE  LOCAL  DEFAULT  UND
     1: 0000000000000000     0 FILE    LOCAL  DEFAULT  ABS main.asm
     2: 0000000000401000     0 NOTYPE  GLOBAL DEFAULT    1 _start
     3: 000000000040200c     0 NOTYPE  GLOBAL DEFAULT    1 __bss_start
     4: 000000000040200c     0 NOTYPE  GLOBAL DEFAULT    1 _edata
     5: 0000000000402010     0 NOTYPE  GLOBAL DEFAULT    1 _end

No version information found in this file.
```

Наша программа получила SIGSEGV. Чтобы разобраться в причине, посмотрим на структуру исполняемого файла.

Компоновщик положил нашу секцию в первый сегмент, сдвинув её на 0x1000. Размер MemSiz этого сегмента 0x100c, то есть, его явно хватает для хранения секции .code. Убедимся в этом, воспользовавшись дебаггером.

```
> 0x401000      mov    $0x3c,%eax
  0x401005      mov    $0x0,%edi
  0x40100a      syscall
```

Ответ кроется в правах у сегмента: обратите внимание, что мы не можем его выполнять.

Поведение стандартного скрипта компоновки можно описать как "любые незнакомые секции класть в первый фиктивный сегмент".

(Здесь есть странное место, которое я не понял. Утверждается, что с помощью конструкции MEMORY в linker script-е можно изменить права сегмента. Однако в моём случае, хоть сегмент и получал нужный адрес, был по прежнему не executable.)

Эту проблему можно решить с помощью nasm-a. Дело в том, что в объектном файле секция .code тоже не является исполняемой.

```
$ readelf -a main.o
...
  [ 1] .code             PROGBITS         0000000000000000  00000180
       000000000000000c  0000000000000000  A       0     0     1
...
```

Используем директиву exec в коде в объявлении секции.

```
section .code exec
global _start
_start:
    mov     rax, 0x3C
    mov     rdi, 0x0
    syscall
```

Проверим права теперь.

```
$ readelf -a main.o
...
  [ 1] .code             PROGBITS         0000000000000000  00000180
       000000000000000c  0000000000000000  AX       0     0     1
...
$ readelf -a main
...
Program Headers:
  Type           Offset             VirtAddr           PhysAddr
                 FileSiz            MemSiz             Flags  Align
  LOAD           0x0000000000000000 0x0000000000400000 0x0000000000400000
                 0x00000000000000b0 0x00000000000000b0  R      0x1000
  LOAD           0x0000000000001000 0x0000000000401000 0x0000000000401000
                 0x000000000000000c 0x000000000000000c  R E    0x1000

 Section to Segment mapping:
  Segment Sections...
   00
   01     .code
...
```

Обратите внимание: компоновщик не сделал первый сегмент исполняемым, а поместил секцию .code в отдельный сегмент.

```
$ ./main
```

Программа успешно выполняется.

5.3.8

Изучим linker script-ы. Начнём с изменения entry point-а и VirtAddr-а.

mymain.asm

```
section .code exec
global mymain
extern hismain
mymain:
    call    hismain
```

```
    mov     rax, 0x3C
    mov     rdi, 0x0
    syscall
```

hismain.asm

```
section .code exec
global hismain
hismain:
    mov     rax, 0x0
    ret
```

script.ld

```
ENTRY(mymain)

SECTIONS {
    . = 0x100000;
    .code : {
        *(.code)
    }
}
```

ENTRY(mymain) говорит о том, что entry point-ом будет метка mymain.

Конструкция SECTIONS объединяет секции (пока что в большие секции, а не сегменты) и располагает их в определённом порядке во время исполнения программы, и в том же порядке в самом файле.

Первая строка . = 0x100000 говорит о том, что следующая секция будут располагаться по адресу 0x100000, а следующие за ней — ещё дальше.

Строка .code : { *(.code) } говорит следующее: необходимо взять со всех поданных на вход файлов (*) секцию .code и объединить их в одну секцию под названием .code (справа в конструкции — название выходной секции).

```
$ nasm -felf64 mymain.asm -o mymain.o
$ nasm -felf64 hismain.asm -o hismain.o
$ ld mymain.o hismain.o -o main -T script.ld
$ ./main
$ readelf -a main
...
Section Headers:
  [Nr] Name              Type             Address           Offset
       Size              EntSize          Flags  Link  Info  Align
  [ 0]                   NULL             0000000000000000  00000000
       0000000000000000  0000000000000000           0     0     0
  [ 1] .code             PROGBITS         0000000000100000  00001000
       0000000000000017  0000000000000000  AX       0     0     1
  [ 2] .symtab           SYMTAB           0000000000000000  00001018
```

```
       0000000000000078  0000000000000018           3     3     8
  [ 3] .strtab           STRTAB           0000000000000000  00001090
       0000000000000027  0000000000000000           0     0     1
  [ 4] .shstrtab         STRTAB           0000000000000000  000010b7
       0000000000000021  0000000000000000           0     0     1
Key to Flags:
  W (write), A (alloc), X (execute), M (merge), S (strings), I (info),
  L (link order), O (extra OS processing required), G (group), T (TLS),
  C (compressed), x (unknown), o (OS specific), E (exclude),
  D (mbind), l (large), p (processor specific)

There are no section groups in this file.

Program Headers:
  Type           Offset             VirtAddr           PhysAddr
                 FileSiz            MemSiz              Flags  Align
  LOAD           0x0000000000001000 0x0000000000100000 0x0000000000100000
                 0x0000000000000017 0x0000000000000017  R E    0x1000

 Section to Segment mapping:
  Segment Sections...
   00     .code
...
```

Обратите внимание на то, что у единственного сегмента VirtAddr теперь равен 0x100000, а Offset теперь не 0x0 (то есть теперь таблицы, которые мы сейчас видим, не будут загружены в память при запуске программы (это нужно регулировать отдельно)).

Мы можем включать в выходной файл секции их стандартных файлов (например, файлов стандартной библиотеки).

stdlib.asm

```
section .code exec
global sqrt
sqrt:
    mov     rax, 0x42
    ret
```

script.ld

```
ENTRY(mymain)

SECTIONS {
    . = 0x100000;
    .stdlib : {
        stdlib.o(.code)
    }
```

```
    .code : {
        *(.code)
    }
}
```

Мы образовали ещё одну секцию .stdlib из секции .code в файле stdlib.o. Обратите внимание, что теперь секция .code в файле stdlib.o не попадет в выходную секцию .code. При этом, если мы поменяем порядок объявления выходных секций .stdlib и .code, то секция .code в файле stdlib.o попадёт в выходную секцию .code, а секция .stdlib будет выброшена, так как в неё не попало ничего.

```
$ nasm -felf64 stdlib.asm -o stdlib.o
$ ld mymain.o hismain.o -o main -T script.ld
$ ./main
$ readelf -a main
...
Section Headers:
  [Nr] Name              Type             Address           Offset
       Size              EntSize          Flags  Link  Info  Align
  [ 0]                   NULL             0000000000000000  00000000
       0000000000000000  0000000000000000           0     0     0
  [ 1] .stdlib           PROGBITS         0000000000100000  00001000
       0000000000000006  0000000000000000  AX       0     0     1
  [ 2] .code             PROGBITS         0000000000100006  00001006
       0000000000000017  0000000000000000  AX       0     0     1
  [ 3] .symtab           SYMTAB           0000000000000000  00001020
       00000000000000a8  0000000000000018           4     4     8
  [ 4] .strtab           STRTAB           0000000000000000  000010c8
       0000000000000037  0000000000000000           0     0     1
  [ 5] .shstrtab         STRTAB           0000000000000000  000010ff
       0000000000000029  0000000000000000           0     0     1
Key to Flags:
  W (write), A (alloc), X (execute), M (merge), S (strings), I (info),
  L (link order), O (extra OS processing required), G (group), T (TLS),
  C (compressed), x (unknown), o (OS specific), E (exclude),
  D (mbind), l (large), p (processor specific)

There are no section groups in this file.

Program Headers:
  Type           Offset             VirtAddr           PhysAddr
                 FileSiz            MemSiz              Flags  Align
  LOAD           0x0000000000001000 0x0000000000100000 0x0000000000100000
                 0x000000000000001d 0x000000000000001d  R E    0x1000

 Section to Segment mapping:
```

```
  Segment Sections...
   00     .stdlib .code
...
```

Обратите внимание, что обе секции попали в первый сегмент. Как управлять сегментами, мы увидим чуть позже.

5.3.9

Посмотрим более сложные случаи организации секций и сегментов. Здесь я снова обнаруживал странное и неочевидное поведение компоновщика.

mymain.asm

```
section .code exec
global mymain
extern hismain
mymain:
    call    hismain
    mov     rax, 0x3C
    mov     rdi, 0x0
    syscall

section .foo
foo:
    db      0x1, 0x2
```

script.ld

```
ENTRY(mymain)

SECTIONS {
    . = 0x100000;
    .stdlib : {
        stdlib.o(.code)
    }
    .code : {
        *(.code)
    }
    .foo : {
        *(.foo)
    }
}
```

Представим, что мы хотим дополнительно иметь секцию только с правами на чтение. Оставим всё остальное, как было на прошлом шаге.

```
$ readelf -a main
...
```

```
Program Headers:
  Type           Offset             VirtAddr           PhysAddr
                 FileSiz            MemSiz             Flags  Align
  LOAD           0x0000000000001000 0x0000000000100000 0x0000000000100000
                 0x000000000000001f 0x000000000000001f  R E    0x1000

 Section to Segment mapping:
  Segment Sections...
   00     .stdlib .code .foo
...
```

Наша секция .foo является executable так как компоновщик положил её в один сегмент с остальными! (Проверьте с помощью дебаггера, что вы действительно можете выполнять код в этой секции. Для этого, напишите в неё инструкции.)

Попробуем сделать секцию .foo writable.

```
section .code exec
global mymain
extern hismain
mymain:
    call    hismain
    mov     rax, 0x3C
    mov     rdi, 0x0
    syscall

section .foo write
foo:
    db      0x1, 0x2
```

```
$ readelf -a main
...
Program Headers:
  Type           Offset             VirtAddr           PhysAddr
                 FileSiz            MemSiz             Flags  Align
  LOAD           0x0000000000001000 0x0000000000100000 0x0000000000100000
                 0x000000000000001f 0x000000000000001f  RWE    0x1000

 Section to Segment mapping:
  Segment Sections...
   00     .stdlib .code .foo
...
```

Теперь секция с кодом тоже стала writable (что неприемлемо). То есть, компоновщик ставит в права сегмента объединение прав всех составляющих его секций.

Я не смог найти информацию о конструкциях, явно указывающих распреде-

ление секций по сегментам. Вместо этого мы делаем это так.

script.ld

```
ENTRY(mymain)

SECTIONS {
    . = 0x100000;
    .stdlib : {
        stdlib.o(.code)
    }
    .code : {
        *(.code)
    }
    . = 0x200000;
    .foo : {
        *(.foo)
    }
}
```

Мы снова явно установили значение VirtAddr на большую величину.

```
$ readelf -a main
...
Program Headers:
  Type           Offset             VirtAddr           PhysAddr
                 FileSiz            MemSiz              Flags  Align
  LOAD           0x0000000000001000 0x0000000000100000 0x0000000000100000
                 0x000000000000001d 0x000000000000001d  R E    0x1000
  LOAD           0x0000000000002000 0x0000000000200000 0x0000000000200000
                 0x0000000000000002 0x0000000000000002  RW     0x1000

 Section to Segment mapping:
  Segment Sections...
   00     .stdlib .code
   01     .foo
...
```

Теперь у нас две сегмента с правильным распределением секций в них, правильными значениями VirtAddr и правильными правами.

Обратите внимание на эти конструкции: . = 0x100000;. Мы можем этим адресам давать имена, а затем использовать эти метки в коде — компоновщик подставит их значения. Кроме того, мы можем давать имена не только адресам, которые мы написали явно, но и тем, которые вывел компоновщик — например, адресу начала секции .code, которая попадает второй секцией к первому сегменту.

script.ld

```
ENTRY(mymain)

SECTIONS {
    . = 0x100000;
    .stdlib : {
        stdlib.o(.code)
    }
    code_begin = .;
    .code : {
        *(.code)
    }
    foo_begin = 0x200000;
    .foo : {
        *(.foo)
    }
}
```

Мы создали две метки: code_begin хранит адрес начала секции .code, а foo_begin хранит адрес начала секции .foo. Мы можем использовать это как обычные метки, определённые в других модулях.

mymain.asm

```
section .code exec
global mymain
extern hismain
extern code_begin
extern foo_begin
mymain:
    mov     rax, code_begin
    mov     rbx, foo_begin
    call    hismain
    mov     rax, 0x3C
    mov     rdi, 0x0
    syscall

section .foo write
foo:
    db      0x1, 0x2
```

```
B+> 0x100006 <mymain>      movabs $0x100006,%rax
  0x100010 <mymain+10>   movabs $0x200000,%rbx
```

5.3.10

Напишите программу и linker script к ней, которые поддерживают другие секции (.data, .bss, .rodata), выставляя им правильные права. При этом эти

секции переименованы (например, .bss -> .uninit).

5.3.11

Проверим тезис о том, что после изучения ассемблера одной архитектуры, ассемблеры других архитектур изучаются легко и быстро (речь, конечно, идёт о прикладном уровне).

Вы можете компилировать код на другие архитектуры локально. Причём легко — zig с этим отлично справляется.

main.c

```
int boo(int x, int y) {
    return x * y;
}

int foo(int x, int y) {
    int a = x + y;
    int b = x - y;
    return boo(a, b);
}

int main() {
    int x = 2;
    int y = 3;
    int z = foo(x, y);
}
```

```
$ zig build-obj -target arm-linux main.c
$ arm-linux-gnueabihf-objdump -d main
```

Так вы можете выполнить компиляцию на arm, а затем дизассемблировать и посмотреть инструкции. К сожалению, запуск хоть и возможен (например, с помощью эмулятора qemu), весьма непрост.

Изучите регистры, формат инструкций и протокол вызова функций в архитектурах arm и risc-v.

Альтернативно вы можете воспользоваться https://godbolt.org/

5.4. Введение в Embedded Programming

5.4.1

Здесь мы научимся писать программы, выполняющиеся непосредственно на процессоре, и тривиальные операционные системы. Главным источником информации будет данный сайт: https://wiki.osdev.org . Изученной нами на протяжении данного курса информации должно быть достаточно, чтобы

понимать tutorial-ы с уровнем "beginner". Мы будем использовать эмулятор qemu для запуска и отладки нашей ОС.

Представим, что мы хотим запустить написанную нами на языке ассемблера и C программу непосредственно на процессоре, без поддержки существующей операционной системы. Для начала нужно понять, а что делает компьютер в самом начале, ведь, очевидно, он не съинтерпретирует диск как файловую систему, не найдет там наш elf-файл (а почему именно elf, а не PE?), не поймет, что это elf-файл, не выполнит его загрузку в процесс, не прикомпонует его к shared objects. В общем, проблем намного больше, чем может показаться на первый взгляд.

Итак, компьютер (не процессор) делает следующее: берёт первый сектор (512 байт) данных с диска (при этом, последние два байта должны быть 0x55 и 0xAA), загружает их в адрес 0x7C00 и прыгает в этот адрес. Это означает, что в первом секторе диска у нас должен быть некий код, который считает файловую систему, найдёт исполняемый файл ядра и загрузит его. Уложить это в 512 байт не получится, поэтому обычно эта логика пишется в нескольких первых секторах, а первый сектор загружает их втупую (не интерпретируя, как файловую систему) и прыгает в них.

Есть хорошая новость: драйвер для файловой системы мы писать здесь не будем. Мы воспользуемся загрузчиком grub (который вы скорее всего используете для запуска своей ОС). Есть и альтернативные загрузчики, например, limine и lilo.

Использование grub-а довольно простое.

1. Мы скомпилируем ядро в обычный 32-bit elf-файл.
2. Мы создадим директорию и положим в неё файлы так, как мы хотим, чтобы они располагались на загрузочном диске.
3. Мы создадим конфигурационный файл для grub-а в этой директории в /boot/grub и запишем в нём путь до нашего elf-файла.
4. Мы воспользуемся утилитой grub-mkrescue, который создаст загрузочный диск по такому же принципу, по какому мы это делали с помощью монтирования и loop devices.

5.4.2

Я сразу покажу все файлы, а затем мы выполним анализ этого.

src/boot.asm

```
%define FLAGS 0x3
%define MAGIC 0x1BADB002
%define CHECKSUM -(MAGIC + FLAGS)

section .multiboot
    align   0x4
```

```
    dd      MAGIC
    dd      FLAGS
    dd      CHECKSUM

section .text
global _start
_start:
    lea     edi, 0xB8000
    mov     byte [edi + 0], 'H'
    mov     byte [edi + 2], 'e'
    mov     byte [edi + 4], 'l'
    mov     byte [edi + 6], 'l'
    mov     byte [edi + 8], 'o'

    cli
.1: hlt
    jmp .1
```

script.ld

```
ENTRY(_start)

SECTIONS {
    . = 0x200000;
    .text : ALIGN(0x1000) {
        *(.multiboot)
        *(.text)
    }
}
```

grub.cfg

```
menuentry "Rach" {
    multiboot /boot/kernel
}
```

Makefile

```
BUILD_DIR=$(abspath build)

SRCS_ASM=$(wildcard src/*.asm)
OBJS_ASM=$(patsubst src/%.asm, $(BUILD_DIR)/%.o, $(SRCS_ASM))

ASFLAGS=-felf32
LDFLAGS=-T script.ld -m elf_i386

$(BUILD_DIR)/%.o: src/%.asm prepare
    nasm $(ASFLAGS) $< -o $@
```

```
kernel: $(OBJS_ASM) $(OBJS_C)
    ld $(LDFLAGS) $(OBJS_ASM) $(OBJS_C) -o $(BUILD_DIR)/kernel

image: kernel
    mkdir -p isodir/boot/grub
    cp $(BUILD_DIR)/kernel isodir/boot/kernel
    cp grub.cfg isodir/boot/grub/grub.cfg
    grub-mkrescue -o $(BUILD_DIR)/kernel.iso isodir
    rm -rf isodir

qemu: image
    qemu-system-i386 -cdrom $(BUILD_DIR)/kernel.iso

prepare:
    mkdir -p $(BUILD_DIR)

clean:
    rm -rf $(BUILD_DIR)
```

Для запуска этого вам необходимо поставить в систему:

- grub
- xorriso
- mtools
- qemu
- zig

Начнём с файла src/boot.asm

Данный файл начинается с секции, которая хранит только последовательность чисел, по которой grub увидит этот файл. (Зачем это нужно, ведь полученный диск является обычным диском, partition-ы которого имеют обычные файловые системы (можете потом это проверить), и grub может просто найти наш elf-файл по заданному в конфигурационном файле пути? Полагаю, чтобы grub мог загружать ядро и из неизвестной ему файловой системы.)

В самой программе я записываю простым способом строку в адрес 0xB8000 с шагом в 2 байта. Это выглядит как что-то бессмысленное, но это не так. По умолчанию BIOS устанавливает монитор в специальный режим терминала (ищите по запросу VESA), при котором всё, что вы будете писать в адрес 0xB8000 будет дублироваться на мониторе. При этом, каждый чётный байт обозначает символ, а каждый нечётный байт — цвет этого символа. Я это сделал, чтобы наша ОС показала хоть какие-то признаки жизни.

Инструкция cli, грубо говоря, ставит на паузу анализ прерываний (interrupts), приходящих от оборудования. Когда какое-то оборудование хочет нам что-то сказать, оно "пингует" нас прерыванием. При этом, само прерывание не несёт в себе информации — получив его, мы просто вступаем в "диалог" с

оборудованием. BIOS автоматически настраивает некие прерывания (я не знаю, перезаписывает ли их grub).

Инструкция hlt останавливает процессор. Однако, если придет прерывание от оборудования, то после его обработки, процессор продолжит своё выполнение. Поэтому мы и используем инструкцию cli. (Про jmp на hlt мне мало известно. Говорят, что процессор всё равно может возобновить работу, поэтому мы и добавляем этот прыжок.)

Linker script script.ld, полагаю, должен быть почти очевиден. Мы добавили сдвиг на 0x200000, так как на меньших адресах grub хранит свою информацию. Аттрибут ALIGN выравнивает секцию на соответствующую величину.

Конфигурационный файл для grub-a grub.cfg содержит лишь путь до нашего исполняемого файла.

Посмотрим на файл Makefile. Мы добавляем компоновщику флаг -m elf_i386, чтобы выполнить компоновку в elf 32-bit. (По какой-то причине компоновщик у меня игнорировал OUTPUT_FORMAT в linker script-е.) После компиляции мы кладём исполняемый файл и конфигурационный файл grub-a в отдельную директорию и вызываем grub-mkrescue, который создаёт загрузочный диск. Наконец, мы запускаем виртуальную машину qemu, передав ей загрузочный диск в cd-rom.

Выполните make qemu. qemu запустится в обычном окне. Вы на долю секунды увидете текст от BIOS-а в qemu, затем появится grub, который предложит вам выбрать ОС для загрузки из списка, в котором будет только одна ОС. После загрузки ОС вы увидете текст Hello и мигающий курсор на первом символе (как поменять его позицию, можете почитать самостоятельно).

5.4.3

Установить на один установочный диск несколько ОС так, чтобы с помощью grub-а можно было выбирать, какую запустить.

5.4.4

Мы можем выполнять отладку нашей ОС с помощью gdb. Добавим дополнительный рецепт в Makefile.

```
...
debug: image
    qemu-system-i386 -cdrom $(BUILD_DIR)/kernel.iso -gdb tcp::1234 -S
...
```

Теперь мы можем подключиться к Qemu по порту 1234. Для этого добавим дополнительные команды в .gdbinit.

```
layout asm
layout regs
```

```
set disassembly-flavor intel
target remote localhost:1234
symbol-file build/kernel
b _start
c
```

Команда set disassembly-flavor intel меняет синтакис кода в gdb на intel.

Команда target remote localhost:1234 подключает gdb к Qemu на порту 1234.

Команда symbol-file build/kernel говорит gdb использовать символы из данного исполняемого файла. В противном случае gdb, например, не будет знать, что 0x200000 это именно секция .multiboot, и где находится метка _start.

Выполните make debug и в отдельном терминале gdb. Пройдите этап с grub-ом, и Qemu остановится на метке _start.

```
Register group: general
eax            0x2badb002       732803074
ecx            0x0              0
edx            0x0              0
ebx            0x10000          65536
esp            0x7ff00          0x7ff00
ebp            0x0              0x0
esi            0x0              0
edi            0x0              0
eip            0x200010         0x200010 <_start>
eflags         0x46             [ IOPL=0 ZF PF ]
cs             0x10             16
ss             0x18             24
ds             0x18             24
es             0x18             24
fs             0x18             24

B+> 0x200010 <_start>      lea    edi,ds:0xb8000
   0x200016 <_start+6>     mov    BYTE PTR [edi],0x48
   0x200019 <_start+9>     mov    BYTE PTR [edi+0x2],0x65
   0x20001d <_start+13>    mov    BYTE PTR [edi+0x4],0x6c
   0x200021 <_start+17>    mov    BYTE PTR [edi+0x6],0x6c
   0x200025 <_start+21>    mov    BYTE PTR [edi+0x8],0x6f
    0x200029 <_start+25>    cli
    0x20002a <_start.1>     hlt
   0x20002b <_start.1+1>   jmp    0x20002a <_start.1>
    0x20002d                add    BYTE PTR [eax],al
    0x20002f                add    al,dl
   0x200031                 add    BYTE PTR [eax],dl
   0x200033                 add    BYTE PTR [esp+ebx*1],dl
```

```
0x200036          jl     0x200038
0x200038          or     BYTE PTR [eax],ch
0x20003a          cmp    al,0x2d
```

Секция .text действительно находится в 0x200010. Выполните инструкцию за инструкцией и посмотрите, как буквы по одной появляются на виртуальной машине.

Пора раскрыть информацию о сегментных регистрах. Обратите внимание на последние шесть регистров в списке (в моём случае, чтобы увидеть шестой, нужно прокрутить вниз): cs, ss, ds, es, fs, gs. Это сегментные регистры и в зависимости от состояния процессора они выполняют одну из двух ролей.

Сразу при запуске процессор находится real mode (реальном режиме). В таком режиме может исполняться только 16-bit код (то есть, мы явно не в этом режиме), а сегментные регистры используются для следующей цели. Так как мы работаем в 16-bit, у нас есть только 16-bit регистры. Это означает, что числом в регистре мы можем кодировать только число до $1 << 16$.

Чтобы немного повысить диапазон чисел, которые можно кодировать, для каждой инструкции, использующей обычный регистр, выбрали один из шести сегментных регистров (причём, не всегда очевидным образом). Пусть в инструкции MOV выбран сегментный регистр B. Тогда MOV [A], 0x1 выполнит перемещение в адрес $A + (B << 4)$. Такой адрес записывают как B:A. (Совсем так же, как записывают, что результат инструкции mul помещается в rdx:rax, хотя это абсолютно разные вещи.)

Идея такого следующая. Обычно, когда мы выполняем какую-то подпрограмму, мы работает с небольшим отрезком памяти. Это означает, что мы можем сразу установить необходимые регистры и при выполнения этой подпрограммы их не трогать. После завершения мы сменим сегментные регистры, чтобы работать с другим отрезком памяти.

На практике же, в добавок к тому, что написание кода на ассемблере сложное, манипуляции с сегментами делают этот процесс ещё больнее. Обычно в обучениях написанию ОС вы будете видеть, что в них просто сразу записывают нули и больше их никогда не трогают.

Существует также protected mode (защищённый режим) работы процессора. Все ОС стараются немедленно в него перейти. (Правда, это ломает вспомогательные функции BIOS-a. Чтобы ими пользоваться, можно временно переключаться на real mode (для чего мы будем иметь чередующиеся куски кода на 16-bit и 32-bit, что также неприятно контролировать).) GRUB уже перешёл для нас в protected mode. О нём мы узнаем подробнее чуть позже.

5.4.5

Писать программы на ассемблере очень сложно. Как и в случае с языком C, мы хотим как можно скорее перейти на более высокоуровневый язык программирования. Перейти к C на самом деле довольно просто. Однако обычным gcc пользоваться очень сложно, так как он то и дело использует оптимизации, которые непросто поддерживать: shared objects и секцию .rela.dyn, position independent code и секцию .got, stdlib, даже если есть флаг на неиспользование её, и т. д.

Чтобы эффективно отключить всё это, следует скомпилировать себе gcc со специальными freestanding настройками. Здесь указано, как это сделать: https://wiki.osdev.org/GCC_Cross-Compiler . Однако, я решил пойти немного другим путём, так как компиляция компилятора — довольно затратное действие (на слабой машине процесс компиляции будет длиться несколько часов). Мы воспользуемся компилятором zig, так как он отлично поддерживает freestanding-компиляцию. Позволю вам прикоснуться к прекрасному.

Makefile

```
...
SRCS_C=$(wildcard src/*.c)
OBJS_C=$(patsubst src/%.c, $(BUILD_DIR)/%.o, $(SRCS_C))
...
ZIGFLAGS=-target x86-freestanding
...
$(BUILD_DIR)/%.o: src/%.c prepare
    zig build-obj $(ZIGFLAGS) $< -femit-bin=$@
...
```

Мы передаём zig-у флаг -target x86-freestanding. Я не буду здесь это показывать, можете проверить самостоятельно, что структура исполняемого файла корректная.

kernel.c

```
void kernel_main() {
    char *ptr = (char*)0xB8000;
    char str[6] = "Hello";
    int i;
    for (i = 0; i < 5; i++) {
        ptr[i * 2] = str[i];
    }
}
```

Сделаем то же самое, что и в коде на языке ассемблера в прошлый раз. Здесь нам нужно подумать внимательно: а не пользуемся ли мы неправильными секциями? Мы создали локальный массив. Где он хранится? На стеке, как и адрес возврата для фукнции kernel_main. А где стек? Наверное, grub нам

его создал, но чёткого ответа на то, где он, нет. Тогда создадим стек сами.

Мы поместим стек в секцию .bss. (Кстати, а если объявить строку как const char*, то в какой секции она будет?)

boot.asm

```
...
section .bss
    align   0x10
stack_bottom:
    resb    16384
stack_top:

section .text
extern kernel_main
global _start
_start:
    lea     esp, stack_top
    call    kernel_main

    cli
.1: hlt
    jmp .1
```

Мы создали секцию .bss, в которой объявили последовательность, размера 16384. (Число взято с неба. Но имейте ввиду: если вы переполните стек, понять это при отладке может быть очень непросто.) Напомню, что стек растет в сторону уменьшения адреса.

Далее мы устанавливаем значение регистра esp на вершину нашего стека и вызываем функцию, определённую на языке C.

script.ld

```
ENTRY(_start)

SECTIONS {
    . = 0x200000;
    .text : ALIGN(0x1000) {
        *(.multiboot)
        *(.text)
    }
    .bss : ALIGN(0x1000) {
        *(.bss)
    }
}
```

Нам нужно добавить секцию .bss. Здесь есть очередная странность в логике работы компоновщика: секция .bss попадает в отдельный сегмент только

из-за того, что мы добавили аттрибут ALING.

При запуске этой ОС вы снова увидете строку Hello.

5.4.6

Реализуйте удобные функции для вывода текста на экран, в том числе, printf (естественно, не имеющий отношения к stream-ам).

5.4.7

Последнее, что мы здесь сделаем, это настроим прерывания (interrupts) и таблицу дескрипторов прерываний (interrupt descriptor table/IDT).

Прерывания приходят из трёх источников:

- Ошибочное состояние процессора (обычно, это называют exceptions). Например, eip указывает на некорректную инструкцию, или мы выполнили деление на ноль, или мы пытаемся загрузить в сегментный регистр другой сегмент, который не соответствует правам.
- Прерывание от другого устройства. В этом случае устройство лишь говорит нам о желании "поговорить", а сам разговор выполняется инструкциями in и out. Писать драйверы для устройств сложно (хотя, для PS/2 клавиатуры не слишком сложно), поэтому мы это здесь делать не будем.
- Выполнение инструкции int (обычно, это называют software interrupts). Мы уже выполняли эту инструкцию, когда писали программы на Linux-е. Мы использовали эту инструкцию, как вызов функции, которая сделает то, что мы не имеем права сделать. При использовании этой инструкции на время текущие права меняются на полные, чтобы ОС могла выполнить запрошенную операцию. Это похоже на setuid флаг у программ, который может включить только владелец файла. При запуске таких программ на время их выполнения текущим пользователем становится их владелец.

Прерывание характеризуется номером, значение которого от 0 до 255. Также некоторые exceptions имеют код ошибки.

Чтобы настроить прерывания, нам необходимо иметь следующие структуры:

- Массив размера 256, состоящий из entries — указателей на функции и некоторых флагов. При возникновении прерывания с номером x будет "вызвана" функция по указателю в ячейке x.
- Записанные подряд размер массива в байтах минус один в 16-битном типе и адрес начала массива. Адрес этой структуры мы запишем в регистр таблицы прерывание.

По-хорошему, нужно вдумчиво подходить к месту размещения этих двух структур. Но мы просто объявим их глобально в коде, из-за чего компилятор

их поместит в секцию .bss. Обычно же таблица прерываний находится в очень маленьких адресах.

idt.h

```
struct _idt_entry_t {
    uint16_t base_low;
    uint16_t selector;
    uint8_t  always0;
    uint8_t  flags;
    uint16_t base_high;
} __attribute__((packed));
typedef struct _idt_entry_t idt_entry_t;

struct _idt_table_t {
    uint16_t limit;
    uint32_t base;
} __attribute__((packed));
typedef struct _idt_table_t idt_table_t;

#define N_IDT_ENTRY 256
idt_entry_t idt_entries[N_IDT_ENTRY];
idt_table_t idt_table;

void idt_flush(idt_table_t*);
void handler0();
void handler13();

void idt_set_entry(uint8_t id, uint32_t base, uint16_t selector, uint8_t flags) {
    idt_entries[id].base_low    = base & 0xFFFF;
    idt_entries[id].base_high   = (base >> 16) & 0xFFFF;

    idt_entries[id].selector    = selector;
    idt_entries[id].always0     = 0;
    idt_entries[id].flags       = flags;
}

void idt_init() {
    idt_table.limit = sizeof(idt_entry_t) * N_IDT_ENTRY - 1;
    idt_table.base = (uint32_t)&idt_entries;

    memset(&idt_entries, 0, sizeof(idt_entry_t) * N_IDT_ENTRY);
    idt_set_entry(0, (uint32_t)handler0, 0x10, 0x8E);
    idt_set_entry(13, (uint32_t)handler13, 0x10, 0x8E);
    idt_flush(&idt_table);
}
```

Итак, первыми объявляются две вышеупомянутые структуры idt_entry_t и idt_table_t и их объекты на стеке: массив idt_entries и одиночный объект idt_table. Обратите внимание, что указатель на функцию в структуре idt_entry_t разделён на два отрезка, которые названы base_low и base_high. О том, что это за флаги записаны в entries, можете не задумываться.

А вот о поле selector можно задуматься. Это номер сегмента (или селектора), который будет автоматически выбран при возникновении прерывания. Это необходимо для временного повышения прав.

Ещё до установки прерываний нам необходимо настроить глобальную таблицу дескрипторов (global descriptor table/GDT). Однако, grub уже сделал это за нас. Посмотрите ещё раз на сегментные регистры в дебаггере: grub использует в качестве сегмента кода 0x10, в качестве сегмента данных 0x18. Мы не будем настраивать свою GDT, а воспользуемся трудами grub-a. Нам необходимо в качестве селектора передать сегмент кода 0x10.

Сегменты задают уровни прав (на самом деле, когда-то они задавали и сегменты памяти, к которым мы можем обращаться, но так теперь их использовать сложно). Когда мы выполняем код пользовательской программы, должен быть выбран сегмент с низкими правами (которые, в том числе, не позволяют выбрать сегмент с полными правами). При обработке прерывания автоматически будет выбираться тот сегмент, который записан в поле selector в нашей структуре.

На самом деле, речь о user mode может быть достаточно длинной, так как там, помимо сегментов с правами, есть ещё страницы и процессы (tasks). Мы не будем здесь это затрагивать и ограничимся прерываниями.

Функции handlerX — это функции-обработчики, которые будут вызываться при возникновении прерываний. Мы вынуждены реализовать их на языке ассемблера, так как возвращаться из них нужно не так, как из обычных функций. Функция idt_flush также требует использования инструкций, которые нельзя получить на языке C. Я решил для демонстрации реализовать обработчики только для прерываний с номерами 0 и 13.

Здесь используется функция memset. Напишите её самостоятельно.

5.4.8

idt.asm

```
global idt_flush
idt_flush:
   mov     eax, [esp + 4]
   lidt    [eax]
   ret
```

Функция для установки регистра для IDT проста: она вызывает инструкцию lidt, передав ей адрес структуры.

Нам осталось определить обработчики. При возникновении прерывания процессор кладет на стек значения регистров eip, cs, eflags, esp, ss, которые были в момент возникновения прерывания, и адрес возврата. Затем, если это был exception, но на стек код ошибки, если у этого номера он есть (например, у номера 13). Интересно отметить, что если прерывание с тем же номером вызвать инструкцией int, то код ошибки положен не будет. Понять в обработчике, есть ли код ошибки, придётся самостоятельно (программирование на x86 — это боль).

При возвращении из обработчика необходимо убрать всё, что было положено на стек. Для этого используют инструкцию iret.

idt.asm

```
...
extern handler_c

global handler0
handler0:
    cli
    push    dword 0
    push    dword 0
    call    handler_c
    sti
    add     esp, 0x8
    iret

global handler13
handler13:
    cli
    push    dword 13
    call    handler_c
    sti
    add     esp, 0x8
    iret
```

Мы хотим иметь общий обработчик на C, который будет знать номер прерывания и код ошибки. Для этого в handler0 мы кладём бессмысленный код ошибки. Из-за того, как передаются аргументы в функции, мы сможем использовать эти числа в функции на C. Обработку прерывания по-хорошему нужно проводить с выключенными прерываниями, так как мы можем временно переводить процессор в "промежуточные" состояния. Но здесь мы такого не делаем.

Напомню, что код ошибки кладётся только при возникновении exception-а. Я позже его намеренно вызову. Если же вы выполните инструкцию int 10, то ваш стек съедет.

idt.h

```
...
void handler_c(int id, int error_code) {
    printf("Interrupt: %d %d\n", id, error_code);
}
```

А вот и обработчик на языке C, который просто выводит информацию. Вы ведь написали написали функции для вывода? (Подставьте здесь свои.)

kernel.c

```
...
#include "idt.h"

void kernel_main() {
    idt_init();
}
```

Здесь мы просто вызвали функцию, настраивающую ITD.

boot.asm

```
...
_start:
    lea     esp, stack_top
    call    kernel_main
    int     0x0
    mov     ax, 0x8
    mov     ss, ax

    cli
.1: hlt
    jmp .1
```

Теперь демонстрация прерываний. Выполните make qemu.

После выполнения инструкции int 0x0, вы увидете строку Interrupt: 0, 0. Затем выполнения этой функции возобновиться. Далее я пытаюсь сменить сегмент на 0x8, но я не имею на это права. При выполнении инструкции mov ss, ax будет выведено Interrupt 13, 8, так как именно это прерывание General Protection Fault вызывается в этом случае. Здесь 8 — это код ошибки, который, похоже, говорит о том, что мы попытались выбрать сегмент 0x8.

Далее происходит интересная вещь — после завершения выполнения обработчика будет предпринята ещё одна попытка выполнения инструкции mov ss, ax. При этом, регистра ax станет равным нулю, но на этот сегмент мы тоже не можем переключаться. Поэтому далее мы будем бесконечно получать Interrupt 13, 0.

Почему после обработки exception-а выполнения продолжается с той же инструкции? На самом деле, при выполнении пользовательских программ

много что може потребовать вмешательства прерываний. Иногда при обращении к памяти происходит Page Fault (посмотрите номер этого прерывания). Это означает, что куска памяти, к которому вы обратились, сейчас нет в оперативной памяти. В этом случае обработчик прерывания находит место в оперативной памяти, копирует туда память программы, после чего снова выполняется инструкция обращения к памяти, которая на этот раз не вызывает Page Fault. При этом пользовательская программа никак не может узнать об этом Page Fault-е.

5.4.9

Добавьте обработчики для всех прерываний. Обычно для этого используют макросы.

5.4.10

Выполните "Bare Bones" для архитектур arm и risc-v на https://wiki.osdev.org , включая запуск в qemu.

Попробуйте отдавать предпочтение компилятору zig, вместо gcc.