Введение в ассемблер

Всем привет, думаю, мне нет смысла представляться, так как все здесь присутствующие либо знакомы со мной, либо уже были на лекции по обратной разработки.

С сегодняшнего дня мы начинаем небольшой интенсив по ассемблеру, к концу которого у вас должно появиться чёткое понимание того, что происходит под капотом программы, а также умение писать на языке ассемблера, что находится в одном шаге от свободного владения машинным кодом — языком, на котором ваш компьютер мыслит.

Зачем вот это вот всё?

Чем это может быть полезно лично вам? Я думаю, многие из присутствующих пришли просто ради интереса или для общего развития, поэтому давайте обсудим практическое применение тех знаний, за которыми вы сюда пришли. Итак, для чего лично вам нужен этот интенсив? Чем вам поможет владение ассемблером?

Резюмируем. *следить за слайдом*

1. Во-первых, ассемблер — это билет в системное программирование. Владея им, вы можете писать различные низкоуровневые вещи. В частности, это работа с подключенным оборудованием, а, стало быть, написание и обновление драйверов. Новое оборудование выходит постоянно, старое нужно поддерживать, высокоуровневые языки не дают достаточно возможностей, а это значит, что нужны ассемблеристы. Специалистов такого уровня днём с огнём не сыскать,

а потребность имеется (даже в Саратове есть вакансии), поэтому зарплаты здесь что надо, но и попасть непросто.

2. Во-вторых, это может оказаться полезным навыком на CTF — соревновании по компьютерной безопасности. Если вы хотите заниматься взломом или, наоборот, расследованием и устранением последствий кибератак, то одно из направлений этой индустрии — обратная разработка, где владение ассемблером пригождается редко, но ценится на вес с золото. *Кирилл не даст соврать, реверс-инженеры — отдельная каста в мире CTF*
3. В-третьих, это может быть полезно в отладке. Такие ситуации возникают нечасто, но бывает, что вы на 100% уверены в валидности своего кода, а поведение всё равно неадекватное. Отлаживая машинный код, вы можете неожиданно найти баг в используемых вами библиотеках (даже если они не опенсорсные) или вообще в самом компиляторе. Такое бывает редко, но метко, и, думаю, хотя бы раз в жизни с этим столкнётся каждый разработчик.
4. В-четвёртых, самое приземлённое. На втором курсе ПИ и ФИИТ читают ассемблер и было бы неплохо быть готовым сдавать лабы на нём. Антонина Гавриловна, бесспорно, замечательно объясняет свой предмет на лекциях, но вот с практическими занятиями не всё так однозначно и многим нужны дополнительные учебные материалы для выполнения заданий. Так что второкурсники сейчас имеют прекрасную возможность догнать и перегнать учебную программу, а

первокурсники — заранее подготовиться к следующему году. Кроме того, первашам-ИВТшникам и, вроде бы, КБшникам тоже читают ассемблер, а это значит, что и им эти пары не помешают, тем более это вообще по их профилю.

5. Есть и другие, более экзотические причины прикасаться к ассемблеру, но пока мы ограничимся этими тремя пунктами.

Структура курса

следить за слайдом

Курс предельно практический и направлен именно на то, чтобы дать вам возможность уже сейчас научиться писать какие-то базовые вещи на языке ассемблера. Это не значит, что теории не будет. Умение писать на ассемблере требует от вас наличие массы специфических знаний и мы разберёмся с основами работы процессора, операционной системы и самим языком. При этом получаемые знания мы тут же будем подхватывать на практике, что закрепит материал в памяти и поможет лучше его понять.

В рамках этого короткого курса, если у нас хватит сил, мы вместе реализуем целых четыре пет-проекта на ассемблере.

1. Первый из них — Hello, world! Поскольку ассемблер — язык далеко не из простых, просто вывод текста на экран, по моему мнению, — уже не такое маленькое достижение.
2. Следующим будет простой калькулятор, выполняющий четыре базовые арифметические операции. Мы научимся работать с

переменными и с числовыми данными, а также познакомимся с консольным вводом.

3. Третий проект будет гораздо интереснее, мы напишем ту самую виселицу из второго семестра инфопроги, у кого она была или будет, но на ассемблере. Программа будет загадывать слово и задача игрока — побуквенно отгадать его. Здесь мы более тесно поработаем со строками, воспользуемся циклами и функциями.
4. И, наконец, вишенка на торте в виде русской рулетки. Мы напишем полноценную графическую программу. В основном окне мы нарисуем кнопку, по клику на которую откроется другое окошко, сообщающее нам результат рулетки. Для этого мы научимся работать с компоновщиком, таблицей импорта, стандартной библиотекой языка Си и библиотекой GTK.

Подписываемся на каналы

Секунда нативочки. Я бы прорекламировал здесь свой тг-канал, но делать этого я, конечно же, не стану, вместо этого предлагаю вам достать свои телефоны, отсканировать коды и подписаться на канал Студенческих клубов разработки, где будут делаться объявления в том числе от других клубов, а также публиковаться записи лекций.

А пока кто-то ещё не успел достать свой телефон, я задам вопрос в зал: сколько гендеров вы знаете и можете назвать?

Ассемблеры

На самом деле, множество гендеров является бесконечным и включает в себя целый список ассемблеров:

TASM ассемблер, который давным-давно был разработан компанией Borland, и имеет 16- и 32-разрядную реализации. Именно он, согласно учебной программе, изучается в начале курса машинно-зависимых языков программирования у Антонины Гавриловны. Его разработка была прекращена много лет назад, да и сама Borland уже сто лет в обед как приказала долго жить. Именно на TASM вы будете писать свои программы под DOS

FASM ассемблер, который отличается особенно простым синтаксисом. В каком-то смысле он чуть более высокоуровневый, чем остальные собраья. Игрушечным его не назовёшь, почти полностью на нём написаны операционные системы Menuet и Kolibri.

GAS мощный ассемблер от GNU, который известен своими возможностями по работе с макросами и директивами. Именно он используется в компиляторе gcc. GAS особенно популярен в Linux-сообществе, но имеет очень специфичный синтаксис.

MASM ассемблер от Microsoft, изначально предназначенный для написания программ под MS-DOS, а затем и Windows. Он поддерживает и другие операционные системы, но ориентирован всё же на майкрософтовскую продукцию. MASM развивается весьма медленно, последняя версия датируется 2017 годом. На этом ассемблере вы будете писать в рамках машинно-зависимых языков программирования ближе ко второй половине семестра.

NASM (Netwide Assembler) это свободный ассемблер для архитектуры Intel x86, известный своей простотой и мощью. Он поддерживает 16-, 32- и 64-разрядные программы и предлагает богатый набор макросов. NASM часто используется для написания высокопроизводительных участков кода, драйверов устройств и операционных систем.

Есть и множество других ассемблеров, как универсальных, так и очень специфичных. Некоторые из них суть клоны уже перечисленных. Так, YASM — это тот же NASM, просто под другой лицензией.

Работать мы будем как раз на NASM под линуксом, потому что я так решил вуахаххаа. У меня уже была лекция, где я рассказал о самых лайтовых приколах винды, и я не хотел бы к этому возвращаться, поэтому мой выбор в пользу линукса кажется очевидным. Можно, конечно, было бы работать и на MASM, и на GAS, но мне не нравится GASовский синтаксис, а, работая под линуксом, было бы непоследовательно брать майкросовтовский ассемблер. Поэтому берём в руки то, что считается здесь лучшей практикой и достойной альтернативой GASy, — NASM.

Запускаем компьютеры!

Теперь, разобравшись с организационными вопросами, мы можем приступить непосредственно к занятию и наш план-минимум на сегодня — реализация Hello, world!

Исполняемый файл

Исполняемый файл — это файл, в котором содержатся машинные инструкции, следуя которым, процессор выполняет заданный

алгоритм. Кроме них там же лежат самого разного рода данные, необходимые для запуска и поддержания работы процесса.

Структура исполняемых файлов строго стандартизирована и в Linux она соответствует исполняемому и связываемому формату, сокращённо ELF.

Когда мы запускаем исполняемый файл, он попадает в распоряжение загрузчика, который, опираясь на метаданные, подготовит программу к запуску.

Структура ELF

Исполняемый файл начинается с заголовка ELF, в котором описывается и архитектруа процессора, и метод кодирования данных, и многие другие технические шоколадки.

Далее следует заголовок программы, в котором описываются её сегменты — комбинации секций — непересекающиеся блоки с данными, из которых состоит исполняемый файл.

Ну а завершается всё непосредственно секциями и таблицей секций. Как видите, схема очень простая, гораздо проще виндового PE, где скрестили собаку с носорогом (MZ и COFF).

Секция

Но вернёмсся к секциям. Каждая секция в себе содержит вполне осмысленный набор данных. Это может быть:

- Либо служебная информация для загрузчика
- Либо код программы (то есть машинные инструкции)
- Либо данные программы (переменные, массивы и другие структуры)

слайд

Поведение любой секции определяется спецификаторами:

Спецификатор	Значение
alloc/noalloc	выгружать ли секцию в память?
exec/noexec	разрешено ли выполнение?
write/nowrite	разрешена ли запись?
progbits/nobits	хранится ли секция в файле?
align	параметр выравнивания секции

слайд

Кроме спецификаторов у секции есть имя. Есть также несколько стандартных имён, для которых NASM автоматически устанавливает спецификаторы:

```
section .text progbits alloc   exec nowrite align=16; код
section   .data   progbits   alloc   noexec        write   align=4;
инициализированные данные
section   .bss         nobits   alloc   noexec       write   align=4;
неинициализированные данные
section other progbits alloc noexec nowrite align=1; любая другая
секция
```

слайд

Спецификаторы можно не прописывать для секций со стандартными именами:

```
section .text ; код
section .data ; инициализированные данные
section .bss  ; неинициализированные данные
section other progbits alloc noexec nowrite align=1; любая другая
секция
```

Пишем в память

Инструкция `db` позволяет писать произвольные данные. Воспользуемся этим, чтобы положить строку в секцию данных:

```
section .data   ; секция данных
db "Hello, world!",1
```

Добавляем код

Некоторые вещи (от функций до переменных) мы можем экспортировать, то есть сделать доступными для других программ, с помощью директивы `global`

В частности, с её помощью мы указываем точку входа:

```
global _start           ; делаем метку метку _start видимой извне

section .data   ; секция данных
db "Hello, world!",1

section .text           ; объявление секции кода
_start:                 ; метка _start - адрес точки входа
```

Метка `_start`?

Метка — абстрактное понятие языка ассемблера. Метка олицетворяет адрес следующей за ней инструкции.

```
global _start           ; делаем метку метку _start видимой извне

section .data           ; секция данных
message:                ; указывает на букву H из следующей строки:
db "Hello, world!",1

section .text           ; объявление секции кода
_start:                 ; метка _start - адрес точки входа
```

Как оперировать данными?

Процессор не стучится по каждому чиху в ОЗУ. Данные, с которыми он работает прямо сейчас, хранятся в его собственной памяти - регистрах.

Регистров много, они имеют разное имя, разное назначение и разный размер (не больше 64 бит)

Регистры общего назначения

таблица на слайде

Пишем в регистр

Чтобы положить какие-то данные в регистр, существует инструкция `mov`:

```
global _start           ; делаем метку метку _start видимой извне


section .data           ; секция данных
message:                ; указывает на букву H из следующей строки:
db "Hello, world!",1


section .text           ; объявление секции кода
_start:                 ; метка _start - адрес точки входа
mov rax, 60
mov rdi, 0
```

Системные вызовы

Системный вызов приостанавливает работу нашей программы и передаёт управление ядру ОС, чтобы она сама всё разрулила. Естественно, принимая решения, она будет опираться на те данные, которые мы положили в регистр перед этим вызовом.

Системный вызов делается с помощью инструкции `syscall`

слайд

Как правило, в регистре `rax` от нас ожидается номер системного вызова — что-то вроде имени функции, а в других регистрах — аргументы.

Так, чтобы завершить работу программы, нужно выполнить вызов 60, а в `rdi` положить код возврата.

слайд

Так, чтобы завершить работу программы, нужно выполнить вызов 60, а в `rdi` положить код возврата.

```
global _start           ; делаем метку метку _start видимой извне

section .data           ; секция данных
message:                ; указывает на букву H из следующей строки:
db "Hello, world!",1

section .text           ; объявление секции кода
_start:                 ; метка _start - адрес точки входа
mov rax, 60             ; системный вызов exit
mov rdi, 0              ; код возврата 0
syscall                 ; делаем системный вызов
```

слайд

Все системные вызовы Linux можно найти на этой странице:

![https://filippo.io/linux-syscall-table/](presentations/01/syscalls.svg)

Консольный вывод

Консольный вывод в Linux — это поток stdout, он имеет дескриптор 1.

Чтобы оправить данные в поток, существует системный вызов 1, который ожидает:

- В `rdi` дескриптор

- В `rsi` указатель на начало данных
- В `rdx` количество байт

слайд

```
global _start           ; делаем метку метку _start видимой извне

section .data           ; секция данных
message:                ; указывает на букву H из следующей строки:
db "Hello, world!",1

section .text           ; объявление секции кода
_start:                 ; метка _start - адрес точки входа
mov rax, 1              ; системный вызов write
mov rdi, 1              ; дескриптор stdout (1)
mov rsi, message        ; адрес начала строки
mov rdx, 13             ; количество байт в ней
syscall                 ; делаем системный вызов
mov rax, 60             ; системный вызов exit
mov rdi, 0              ; код возврата 0
syscall                 ; делаем системный вызов
```

Hello, world!