Реферат

На тему

«Исполнение кода на без операционной системы»

Выполнил:
Сергеев А.Ю.
Группа М3239
2 курс

Преподаватель:
Романовский А.В.

2023 г

Режимы работы процессора

История семейства процессоров x86 насчитывает уже более 45 лет. За это время архитектура успела обрасти множеством полезных и не очень функций. Одной из особенностей современной x86 является наличие режимов работы, история которых ведется с появления первых процессоров данного семейства. Посмотрим, какие режимы работы присутствуют у Intel сейчас.

Real mode

История x86 ведет отсчет с момента создания микропроцессора Intel 8086. Выпущенный в 1978 году, это был первый 16-битный процессор компании. “16-битный” в данном случае означает, что в нем, помимо 8-битных регистров, появились также 16-битные регистры и команды для работы с ними.

В процессоре использовалась 20-битная адресация памяти. Поддержки виртуальной памяти не было: все адреса были физическими. Интересно, что размер адреса превосходил размер регистра, а значит сохранить адрес в одном регистре не получилось бы. Для решения этой проблемы была применена концепция сегментации памяти – было добавлено 4 сегментных регистра: CS, DS, SS, ES, а также 4 адресных: BX, BP, SI, DI. Обращение к памяти происходило в формате AA:BB, где AA – имя сегментного регистра, BB – имя адресного регистра. Запись AA:BB кодировала адрес [AA * 0x10 + BB] (один адрес мог быть закодирован разными комбинациями значений регистров). Это позволяло адресовать 1 MiB физической памяти.

Достаточно скоро у Intel возникла необходимость добавить поддержку виртуальной адресации в свои процессоры. Однако просто изменить модель работы с памятью означает отказаться от совместимости со всеми предыдущими поколениями. Поэтому в будущих процессорах (начиная с 80286) появились новые режимы работы. А режим, в котором сохраняется совместимость с кодом, написанным под 8086, стал именоваться real mode (от real address mode – режим реальной адресации).

В real mode весь код, написанный под 8086, продолжает исполняться так же, как и на 8086. Интересно, что до сих пор все компьютеры на архитектуре x86 изначально запускаются в режиме совместимости с 8086, а затем могут переходить в более “современные” режимы работы. Причем некоторые программы (например, загрузчик GRUB) могут запускаться в real mode, во время работы переходить в более продвинутые режимы, но перед запуском операционной системы возвращаться обратно в real mode.

Замечу также, что современные процессоры поддерживают работу с 32-битными регистрами даже в real mode, т.к. это не нарушает обратную совместимость. Также возможно обращаться к памяти, используя 32-битные регистры.

Protected mode

Protected mode впервые появился в 1982 году в 16-битном процессоре Intel 80286 как режим, поддерживающий виртуальную адресацию. Размер физического адреса был увеличен до 24 бит.

Виртуальная память изначально была реализована с использованием уже имеющегося механизма сегментации. Теперь сегментный регистр не указывал на область памяти, где находился нужный сегмент: вместо этого в сегментном регистре

хранился индекс в таблице GDT (Global Descriptor Table), где и содержалась информация о местонахождении сегмента. Особенность состояла в том, что GDT мог указывать на ячейку в LDT (Local Descriptor Table) – таблице, уникальной для каждого процесса или потока. Это позволяло разделять память, используемую разными процессами и потоками.

Также была добавлена система колец, представлявших разные слои защиты. Программа, исполнявшаяся на конкретном кольце, имела определенный набор прав. В ячейках GDT содержалась информацию о том, из каких колец возможен доступ к ним. Это позволяло ограничивать доступ к сегментам памяти со стороны программ.

С выходом в 1985 году процессора Intel 80386 – приемника Intel 80286, protected mode был расширен. Появились 32-битные регистры и соответствующие им инструкции. Размер адреса памяти также увеличился до 32 бит, что позволило адресовать все 4 GiB поддерживаемой памяти без использования сегментной адресации. На замену сегментной адресации пришла страничная адресация с использованием таблицы страниц и виртуальных адресов, какими мы их знаем.

Long mode

Long mode – режим, в котором длина машинного слова стала равна 64 бита. Размеры регистров были увеличены до 64 бит, а также были добавлены дополнительные регистры R8-R15. Стала возможной поддержка 64-битной адресации памяти (однако на практике до сих пор используется 48-57-битная адресация, т.к. даже 48 бит позволяют адресовать 256 TiB памяти).

Другое

Помимо данных, существует также несколько “побочных” режимов. К примеру, virtual 8086 mode – режим, где процессор, работая в protected mode, способен исполнять программы, написанные для real mode, или unreal mode, являющийся вариацией real mode.

Зная, какие режимы работы современного процессора существуют, попробуем запустить код на “голом” железе, не пользуясь загрузчиками или операционной системой. Нашей целью будет написание программы для real mode, где не используется концепция виртуальной памяти. Для этого разберемся, как происходит запуск кода на системах с BIOS.

Запуск компьютера

После запуска компьютера первым делом необходимо подготовить устройство к исполнению пользовательского кода. Для этого на материнской плате расположен специальный чип ROM – read-only memory. Этот чип содержит в себе BIOS (basic Input/Output system) – систему, выполняющую первичный запуск компьютера. Данные ROM отображаются непосредственно в оперативную память компьютера (в конец первых 4 GiB памяти). В момент запуска компьютера туда передается управление.

Первым делом BIOS определяет имеющееся в компьютере количество оперативной памяти. Это необходимо, ведь в оперативную память в дальнейшем будет загружена пользовательская программа. После определения и проверки всей памяти BIOS загружает и настраивает адресное пространство: создается нижняя и

верхняя память (low & upper memory). В нижней памяти BIOS размещает таблицу прерываний, посредством которой пользователь в будущем сможет взаимодействовать с ним. Там же размещается BDA (BIOS data area), содержащая различные внутренние переменные и поля BIOS.

Затем происходит определение и инициализация внешних устройств. Все найденные ROM внешних устройств отображаются в верхнюю память. Туда же отображается видеопамять.

Наконец, устройство готово к запуску пользовательского кода. BIOS находит загрузочный диск и запускает код с него. В этот момент процессор работает в real mode.

Исполнение кода с диска

Процесс поиска загрузочного устройства и запуск кода с него крайне примитивен. На каждом устройстве (HDD/SSD, floppy-диске, флешке, CD и т.д.) BIOS читает MBR (master boot record) – первые 512 байт. Каждый MBR содержит информацию о текущем устройстве: уникальный номер, таблицу разделов (до 4 разделов на диске) и тп. В первых 440 байтах MBR может быть размещен исполняемый код. Отличительной чертой загрузочного устройства (устройства, содержащего исполняемый код) является magic code – два байта в конце MBR: 0x55 и 0xAA. Любое устройство, содержащее magic code, считается загрузочным.

Как же происходит запуск? В заданном пользователем порядке BIOS проверяет устройства (floppy-диски, CD, USB, HDD, SSD), пока не найдет то, что содержит magic code. Как только такое устройство найдено, BIOS загружает MBR (и только его) в нижнюю память по адресу 0x7C00, после чего передает управление по этому адресу.

Зная это, давайте создадим минимальный исполняемый образ диска. Писать будем на nasm – одном из языков ассемблера для Intel x86. Создадим файл *boot.asm* со следующим содержанием:

```
use16
org 0x7C00
    jmp $
    times (510 - ($ - $$)) db 0x00
    dw 0xAA55
```

Директива `use16` задает способ кодирования команд. Она необходима, т.к. мы собираемся исполнять код в real mode, где исторически размер машинного слова – 16 бит.

Директива `org 0x7C00` показывает, что код ниже будет загружен в память по адресу 0x7C00.

Далее следует команда `jmp $`, соответствующая бесконечному циклу ($ означает “адрес текущей команды”, т.е. jmp выполняет прыжок на самого себя).

Затем идет заполнение файла нулями: `db 0x00` размещает в нашем файле байт 0x00 (db = define byte). Директива `times (510 - ($ - $$))` повторяет команду (510 - ($ - $$)) раз, где $ – адрес текущей команды в файле, $$ – адрес начала секции в файле. Так мы заполняем файл нулями вплоть до 510 байта. Далее с помощью

`dw 0xAA55` мы размещаем в конце MBR необходимый magic code (511 и 512 байты) .

Важно помнить, что в соответствии с архитектурой фон Неймана процессор не разделяет код и остальные данные. Это значит, что если instruction pointer попадает на случайный участок памяти, то эта память будет интерпретирована как набор команд и исполнена. Поэтому важно, чтобы после исполнения вашего кода процессор не переходил к чтению и исполнению случайных данных. Чтобы избежать этого, можно расположить в конце кода бесконечный цикл(`jmp $`) или инструкцию, переводящую процессор в спящий режим (`cli` – отключение прерываний, после чего `hlt` – перевод процессора в режим ожидания прерываний).

Давайте создадим из *boot.asm* загрузочное устройство. Для начала скомпилируем код в бинарный файл *boot.bin*:

```
> nasm boot.asm -f bin -o boot.bin
```

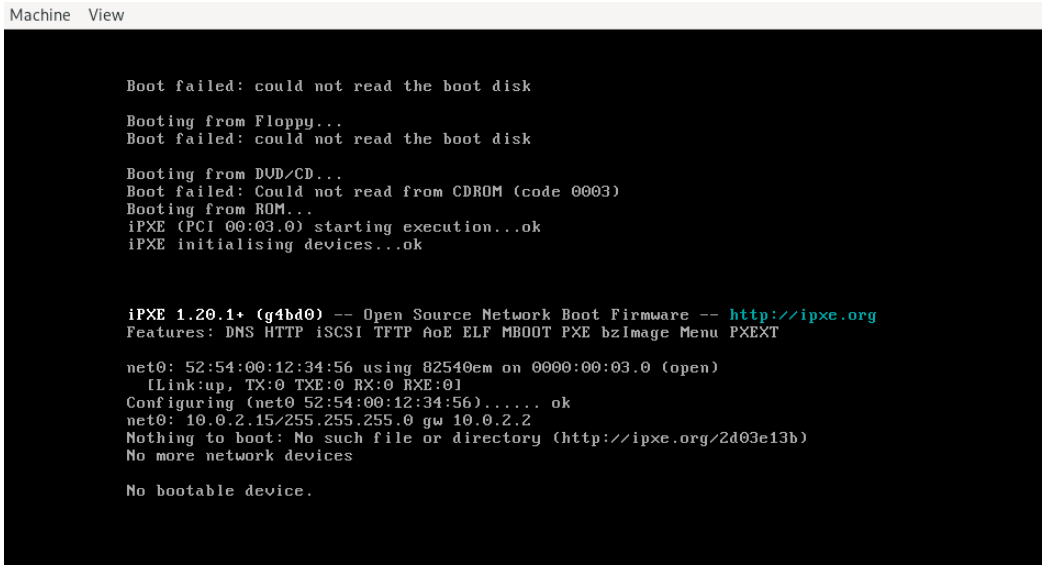
Флаг -f указывает формат результирующего файла, -o – имя.

Мы получили файл с содержанием, соответствующим загрузочному устройству. Запустим наш файл с помощью эмулятора qemu.

Для начала попробуем запустить эмулятор, передав ему пустой файл:

```
> qemu-system-x86_64 -hda /dev/null
```

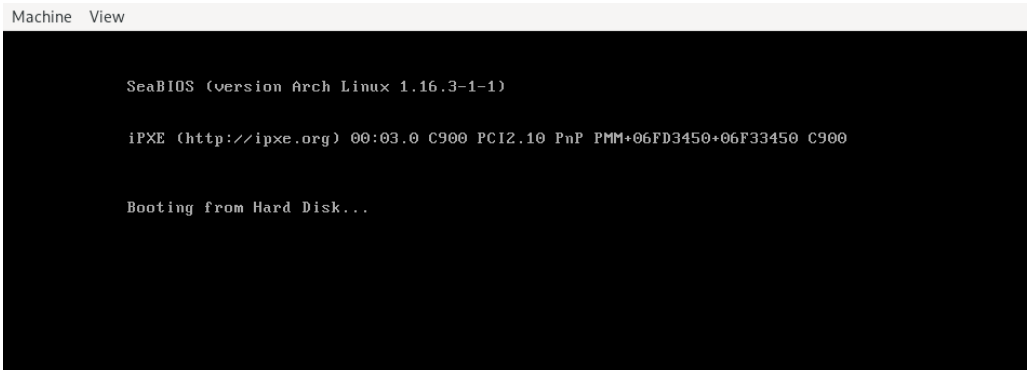
Мы получили вывод, говорящий о том, что эмулятор не смог найти загрузочное устройство.



Теперь давайте передадим эмулятору наш файл:

```
> qemu-system-x86_64 -hda boot.bin
```

В результате мы получаем следующий вывод:



Сообщения на экране могут показаться странными. Однако мы не получили сообщение, как в прошлый раз. Это означает, что эмулятор успешно нашел наше загрузочное устройство. Т.о. мы действительно создали исполняемый файл (хотя и ничего не делающий).

Также мы можем записать наш файл на диск и запустить на реальном железе:

```
> dd if=boot.bin of=/dev/sda
```

Здесь вместо /dev/sda необходимо указать имя вашего диска (необходимо указать сам диск, а не раздел в нем).

ВАЖНО: данная операция сотрет все данные на диске.

Если у вас компьютер с UEFI вместо BIOS, не забудьте перевести его в режим Legacy Boot для эмуляции BIOS. Также рекомендуется отключить Secure boot.

При тестировании на реальном железе у меня возникли проблемы с запуском кода с USB-носителя: одно устройство из 3 без проблем запустилось с флешки, в то время как два других наотрез отказались. В связи с этим рекомендуется использовать в качестве загрузочного устройства HDD/SSD, CD или floppy (если они вдруг у вас есть).

Дальнейшие действия

Итак, мы научились запускать код, но пока не умеем ничего делать. Давайте это исправим! Научимся выводить что-нибудь на экран. Одним из вариантов это сделать является запись напрямую в видеопамять. Однако мы воспользуемся другим способом: используем BIOS.

Соответствуя своему названию (Basic input/output system), BIOS предоставляет обширный API для работы с многими компонентами компьютера, включая вывод на экран, работу с клавиатурой, дисками и прочим, являясь прослойкой между программистом и реальным железом.

Взаимодействие с BIOS происходит посредством генерации прерываний. При этом от номера прерывания, а также от значений регистров зависит, какая именно функция будет исполнена. Также в регистрах передаются аргументы функции. Более подробную информацию о функциях и их аргументах можно найти на соответствующих ресурсах (я пользовался сайтом http://www.ctyme.com/intr/int.htm, где собраны и описаны многие функции, доступные в BIOS).

К примеру, если вызвать прерывание номер 0x10, поместив в регистр ah значение 0xE, то на экран будет выведен символ из регистра al. Вот как выглядит программа, выводящая символ ‘a’:

```
use16
org 0x7C00
    mov ah, 0xE
    mov al, 'a'
    int 0x10
    jmp $
    ; Magic bytes.
    times ((0x200 - 2) - ($ - $$)) db 0x00
    dw 0xAA55
```

Использование оперативной памяти

Мы научились исполнять произвольный код, а также выводить символы на экран. Но функционал BIOS этим не ограничивается. С его помощью можно получать события ввода с клавиатуры, рисовать на экране, перейдя в графический режим (в противовес текстовому, в котором мы находились до этого) и т.п. Но, несмотря на обилие функций, способ взаимодействия пользователя с BIOS остается прежним – выставление регистров и вызов прерываний. Обо всех функциях, предоставляемых BIOS, можно почитать на соответствующих ресурсах.

Теперь мы можем приступать к написанию достаточно сложных программ.

start	end	size	description	type	
Real mode address space (the first MiB)					
0x00000000	0x000003FF	1 KiB	Real Mode IVT (Interrupt Vector Table)	unusable in real mode	640 KiB RAM ("Low memory")
0x00000400	0x000004FF	256 bytes	BDA (BIOS data area)		
0x00000500	0x00007BFF	29.75 KiB	Conventional memory	usable memory	
0x00007C00	0x00007DFF	512 bytes	Your OS BootSector		
0x00007E00	0x0007FFFF	480.5 KiB	Conventional memory		
0x00080000	0x0009FFFF	128 KiB	EBDA (Extended BIOS Data Area)	partially used by the EBDA	
0x000A0000	0x000BFFFF	128 KiB	Video display memory	hardware mapped	384 KiB System / Reserved ("Upper Memory")
0x000C0000	0x000C7FFF	32 KiB (typically)	Video BIOS	ROM and hardware mapped / Shadow RAM	
0x000C8000	0x000EFFFF	160 KiB (typically)	BIOS Expansions		
0x000F0000	0x000FFFFF	64 KiB	Motherboard BIOS		

Однако большинство программ хранит какие-либо данные, для чего нам вряд ли хватит имеющихся у нас регистров. Посмотрим, где еще мы можем хранить информацию.

В соответствии с таблицей, приведенной выше, в нашем адресном пространстве имеются незанятые регионы. Мы можем использовать под свои нужды адреса 0x0500-0x7BFF и 0x7E00-0x7FFF. Там же мы можем разместить наш стек, записав адрес начала стека в регистр SP.

Ниже приведен пример программы, которая помещает символ ‘a’ в память по адресу 0x7E00, а затем читает его из памяти и выводит на экран:

```
use16
org 0x7C00
    mov byte [data_addr], 'a'

    mov ah, 0xE
    mov al, [data_addr]
    int 0x10
    jmp $
    ; Magic bytes.
    times ((0x200 - 2) - ($ - $$)) db 0x00
    dw 0xAA55

data_addr   equ 0x7E00
```

Загрузка данных с диска

Теперь перед нами встала новая проблема: объем кода, который мы можем поместить в MBR, ограничен 440 байтами (или 446, если мы решим залезть на другие поля MBR). Учитывая, что в среднем размер инструкции варьируется от 1 до 3 байт, получаем, что мы можем написать от 150 до 446 инструкций. Это не так много. Научимся исполнять код, превосходящий по объему 446 байт.

Какой бы код мы не написали в нашем файле, во время старта компьютера лишь первые 512 байт будут загружены в оперативную память. Это значит, что нам необходимо самостоятельно загрузить оставшийся код. Мы можем загрузить его в адреса из диапазонов 0x0500-0x7BFF или 0x7E00-0x7FFF. Для загрузки также воспользуемся BIOS.

Чтобы вызвать функцию чтения с диска, необходимо вызвать прерывание с номером 0x13, поместив в регистр AH значение 0x02. В регистрах мы передаем следующие параметры: AL – количество секторов, которые необходимо прочитать, CH – младшая часть номера цилиндра, CL – номер сектора (биты 0-5, нумерация с 1) и старшая часть номера цилиндра (биты 6-7), DH – номер головки диска, DL – номер диска (для HDD старший бит должен быть равен 1), ES:BS – адрес, куда будут записаны прочитанные данные. Будем считать, что мы работаем с диском 0, а размер сектора будем считать равным 512 байт (в реальности размер сектора у носителя может быть другим).

Приведенный ниже код читает один сектор – сектор 2 цилиндра 0 головки 0 диска 0 – и записывает прочитанные данные по адресу 0x7E00, после чего совершает туда прыжок (в данном случае мы полагаемся на то, что наш диск имеет номер 0, а размер сектора равен 512 байтам, т.е. start расположен в начале второго сектора).

```
use16
org 0x7C00
    mov ah, 0x02
    mov al, 1
    mov dl, 0x80
    mov ch, 0
    mov dh, 0
    mov cl, 1
    xor bx, bx
    mov es, bx
    mov bx, code_addr
    int 0x13

    jmp code_addr

    ; Magic bytes.
    times ((0x200 - 2) - ($ - $$)) db 0x00
    dw 0xAA55

code_addr   equ 0x7E00
stack_addr  equ 0x7C00

start:
    mov sp, stack_addr

    mov ah, 0xE
    mov al, 'a'
    int 0x10
    jmp $
```

После того, как мы совершили прыжок, хранить MBR в памяти нам больше не нужно. Адреса 0x7C00-0x7DFF теперь можно использовать в своих нуждах. В данном случае мы разместили там стек, записав в stack pointer адрес 0x7C00.

Итог

Первоначальной целью моего исследования было создание работоспособного кода, который бы исполнялся на “сыром” железе (т.е. железе без операционной системы). Чтобы лучше разобраться в процессе, а также чтобы немного попрактиковаться в использовании языка ассемблера, я написал микро-проект, реализующий минимальный функционал игры “Змейка”.

Данная программа использует графический режим вывода (где пользователь способен изменять цвет каждого отдельного пикселя на экране) с разрешением 320x200. В памяти хранятся позиции всех клеток, занимаемых змейкой, в виде циклического буфера. На каждой итерации в буфер добавляется новая клетка, а последняя клетка удаляется. Новая клетка рисуется на экране, а удаленная стирается.

Обработка происходит в бесконечном цикле. В конце каждой итерации вызывается функция BIOS, выполняющая ожидание в течение нескольких миллисекунд (иначе игра будет работать слишком быстро, а ее скорость будет очень зависеть от частоты процессора).

Для управления направлением движения змейки каждую итерацию происходит проверка наличия событий ввода с клавиатуры (события ввода с клавиатуры сохраняются BIOS'ом в специальном буфере, откуда их можно прочитать с помощью соответствующей функции). В случае, если была нажата одна из кнопок “Влево”, “Вправо”, “Вперед”, “Назад”, направление змейки меняется соответствующим образом.

Чтобы змейка могла расти, с определенным периодом на поле добавляется еда. В качестве случайного значения для координат еды выступает значение в памяти по адресу 0x046C. По этому адресу располагается счетчик количества тактов, прошедшего с запуска компьютера.

Код игры можно найти по ссылке: https://github.com/as3rg/bare-metal-snake

Источники

1. https://www.wikipedia.org
2. https://wiki.osdev.org
3. http://www.ctyme.com
4. https://github.com/siutin/BareMetalOS
5. https://github.com/programble/bare-metal-tetris
6. https://github.com/cirosantilli/x86-bare-metal-examples