Урок 1: Налаштування 32-ох бітного режиму MMU x86_64

Катерина Ковальчук

Грудень 2024р

Анотація

У цьому документі описується процес налаштування 32-бітного режиму MMU x86_64 та виведення 'OK' на екран.

1. Вступ

x86 - це тип центрального процесора, що використовує однойменну архітектуру. З моменту своєї появи в кінці 1970-х років і до сьогодення сімейство x86 значно вплинуло на світ обчислювальної техніки. Цей процесор є основою багатьох комп'ютерів і серверів, без яких сьогодні не може обійтися жодна людина.

Все почалося з базового 8-розрядного мікропроцесора Intel 8080. На його основі був розроблений 16-розрядний мікропроцесор, який заклав основу архітектури x86 в 1978 році і отримав назву Intel 8086. Цей процесор представив новий набір інструкцій, який став відомим як x86. Набір цих інструкцій уможливив виконання складніших завдань і більш ефективні обчислення, проклавши шлях до потужніших комп'ютерів.

Intel не зупинилися на 16-ти бітах, і кожне наступне покоління було кращим за попереднє, приносячи покращення в продуктивності, ефективності та можливостях. У 1985 році дебютував 32-розрядний процесор 80386, який розширив можливості обчислень за рахунок адресації до 4 ГБ пам'яті та підтримки віртуальної пам'яті.

У 1990-х роках розпочалася ера Pentium, яка принесла ще більше підвищення продуктивності та нові функції, такі як покращені обчислення з плаваючою комою. Саме цей бренд закріпив домінування Intel на ринку і став символом продуктивності та інновацій.

Поки Intel святкувала успіх Pentium, їхній конкурент AMD не ловив ґав і на початку 2000-х років представив Athlon та Operon, які були дуже добре сприйняті завдяки своїй ціні та продуктивності, що призвело до конкуренції між двома компаніями.

Ще однією важливою подією в історії архітектури x86 стала поява 64-бітних процесорів, де компанія AMD випередила Intel своєю архітектурою AMD64, після чого Intel перейняла цю технологію і назвала її x86 _64. Це розширення дозволило

процесорам обробляти більші обсяги пам'яті та виконувати складніші обчислення, що зробило їх придатними для сучасних обчислювальних робочих навантажень.

Ця архітектура відіграла важливу роль у розвитку сучасних обчислювальних систем. Пов'язано це з тим, що вона виявилася дуже ефективною і має високу продуктивність і адаптивність, що дозволяє їй залишатися актуальною в цьому швидкозмінному світі. Її сумісність зробила її стандартом для персональних комп'ютерів і серверів. Вона також значно стимулювала розвиток процесорних технологій, таких як конвеєризація, що значно підвищило продуктивність та ефективність.

2. Структура нашої директорії

Структура директорії, де буде знаходитися код нашої операційної системи та всі необхідні специфікації виглядає ось так:

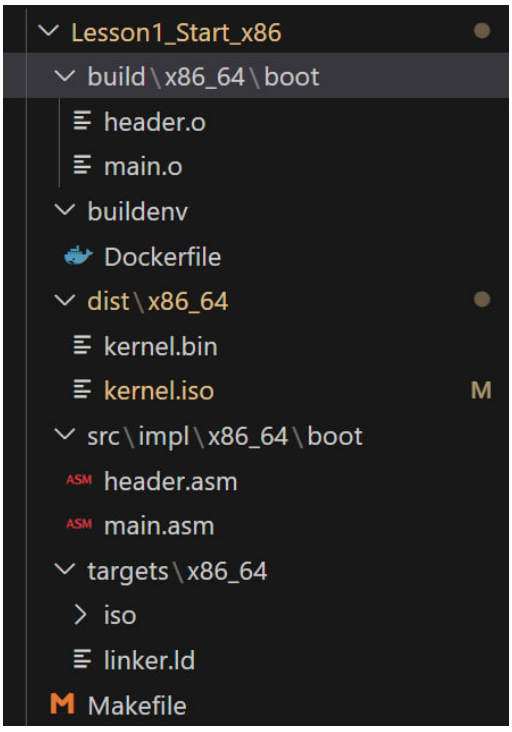

Рис. 1: Структура нашої директорії

Далі іде опис що і де знаходиться.

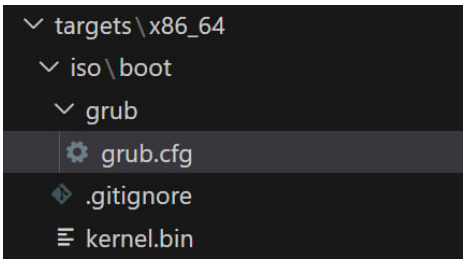

Рис. 2: Папка targets

3. Лінкування

Для того, щоб правильно розмістити секції завантажувача multiboot2 та .text у пам'яті нам потрібно описати як зробити це у файлі Linker.ld (всі визначення пояснюються у інших пунктах.)

Спершу нам потрібно вказати, де знаходиться вхід у нашу ОС (змінна start, яку ми визначимо пізніше у main.asm).

Далі ми визначимо секції нашого коду:

- . — задає поточну адресу в пам'яті на 1M, що означає, що всі дані нашої ОС починатимуться з 1M. Це необхідно для правильного завантаження системи.
- У секцію .boot ми запишемо заголовок завантажувача (з файлу header.asm).
- У секцію .text ми запишемо основний код (main.asm).

```
ENTRY(start)

SECTIONS
{
    . = 1M;

    .boot :
    {
        KEEP(*(.multiboot_header))
    }

    .text :
    {
        *( .text )
    }
}
```

Отже всі дані нашої ОС будуть починатись з 1M, потім ми будемо мати наш завантажувач та інструкції нашого CPU.

3.1. Конфігурація GRUB

GNU GRUB (Великий уніфікований завантажувач) - це завантажувач, який дозволяє користувачам вибирати з декількох встановлених операційних систем одну під час запуску. Він є еталонною реалізацією специфікації Multiboot і може завантажувати будь-яку сумісну операційну систему.

У файлі grub.cfg ми уточнюємо параметри, щоб наш GRUB знав, яку специфікацію завантажувача та з якими параметрами йому використовувати.

Ми задаємо назву нашої ОС (my os), встановлюємо параметри завантаження на одразу після увімкнення (set timeout=0), пункт завантаження (set default=0), як і звідки завантажити ядро ОС (multiboot2 /boot/kernel.bin) та завантажує систему після завантаження ядра (boot).

```
set timeout=0
set default=0

menuentry "my os" {
    multiboot2 /boot/kernel.bin
    boot
}
```

4. Компіляція

Для компіляції я використовувала Makefile та середовище Докер-контейнера, а для емуляції роботи ОС - qemu.

Докер-контейнер - технологія, що дозволяє запакувати усе необхідне програмне забезпечення у ізольоване середовище, що дозволяє запускати програму на будь-якій ОС не переживаючи за сумісність, оскільки усі необхідні залежності вже запаковані у контейнер.

QEMU (Quick Emulator) - Швидкий емулятор -
програма для емуляції та віртуалізації операційних систем та програм на різних архітектурах на хост-системах.
(вкінці буде посилання, за яким його можна буде завантажити.)

Процес компіляції відбувається за таким алгоритмом:

1) Написання Makefile

2) Побудова нашого середовища Докер-контейнера:

Для коректної роботи із докер-контейнером нам потрібен опис, що в цьому контейнері буде знаходитись, для цього нам потрібно створити Dockerfile, в якому буде описано що і куди встановлювати:

```
FROM randomdude/gcc-cross-x86_64-elf

RUN apt-get update
RUN apt-get upgrade -y
```

```
RUN apt-get install -y nasm
RUN apt-get install -y xorriso
RUN apt-get install -y grub-pc-bin
RUN apt-get install -y grub-common

VOLUME /root/env
WORKDIR /root/env
```

Listing 1: Конфігурація Dockerfile

Ми будемо використовувати randomdude/gcc-cross-x86_64-elf зображення Докера, яке містить інструменти компіляції для x86_64. (Завантажити його можна за посиланням: randomdude/gcc-cross-x86_64-elf).

Також ми встановимо останні версії потрібних пакетів:

- RUN apt-get update
- RUN apt-get upgrade -y

І перелічимо, які пакети нам потрібні:

- **nasm** — для компіляції асемблера.
- **xorriso** — для створення ISO-образу (образу нашої ОС. Образ ОС - це початковий двійковий образ, який завантажувач завантажує у пам'ять і передає керування для запуску операційної системи. Образ ОС зазвичай є виконуваним файлом, який містить ядро операційної системи).
- **grub-pc-bin** і **grub-common** — для завантажувача GRUB.

Далі встановимо:

- **директорію для обміну файлами** між контейнером та ОС, на якій ми запускаємо його: VOLUME /root/env.
- **робочу директорію всередині контейнера**: WORKDIR /root/env.

ПЕРЕД ТИМ ЯК ВИКОНУВАТИ ЦІ КОМАНДИ ТРЕБА ПЕРЕСВІДЧИТИСЬ, ЩО У ВАС ЗАПУЩЕНИЙ ДОКЕР!

```
docker build buildenv -t myos-buildenv
```

Це може зайняти кілька хвилин!
ПІД ЧАС ПОБУДОВИ ДОКЕР-ЗОБРАЖЕННЯ МОЖЕ ВИНИКНУТИ ТАКА ПОМИЛКА:

```
docker build buildenv -t myos-buildenv
[+] Building 0.4s (3/3) FINISHED

    docker:default
```

```
 => [internal] load build definition from Dockerfile
                                                    0.0s
 => => transferring dockerfile: 294B
                                                    0.0s
 => [internal] load .dockerignore
                                                    0.0s
 => => transferring context: 2B
                                                    0.0s
 => ERROR [internal] load metadata for docker.io/randomdude/gcc
    -cross-x86_64-elf:latest
                                                    0.4s
------
 > [internal] load metadata for docker.io/randomdude/gcc-cross-
    x86_64-elf:latest:
------
Dockerfile:1
--------------------
   1 | >>> FROM randomdude/gcc-cross-x86_64-elf
   2 |
   3 |     RUN apt-get update
--------------------
ERROR: failed to solve: randomdude/gcc-cross-x86_64-elf: error
    getting credentials - err: exit status 1, out: ``
```

Це може бути пов'язане із правами доступу до цього образу.
Це можна виправити такою командою:

docker pull randomdude/gcc-cross-x86_64-elf

Та повторити спробу побудови Докер-Зображення.

3) Вхід в контейнер:

Linux чи MacOS:

```
docker run --rm -it -v "$(pwd)":/root/env myos-buildenv
```

Windows (CMD):

```
docker run --rm -it -v "%cd%":/root/env myos-buildenv
```

Windows (PowerShell):

```
docker run --rm -it -v "${pwd}:/root/env" myos-buildenv
```

Деталі (або поширені проблеми):

- Використовуйте команди linux, якщо ви використовуєте WSL, msys2 або git bash.
- Якщо у вас виникли проблеми з незагальним диском, переконайтеся, що ваш демон docker має доступ до диска, на якому знаходиться ваше середовище розробки. Для Docker Desktop це можна зробити у «Налаштуваннях > Спільні диски» або «Налаштуваннях > Ресурси > Спільний доступ до файлів».

4) Білд:

```
make build-x86_64
```

Якщо у вас все правильно збудувалось, то ви маєте побачити приблизно ось такий вивід на екрані:

```
mkdir -p dist/x86_64 && \
x86_64-elf-ld -n -o dist/x86_64/kernel.bin -T targets/x86_64/linker.ld build/x86_64/boot/header.o build/x86_64/boot/main.o && \
cp dist/x86_64/kernel.bin targets/x86_64/iso/boot/kernel.bin && \
grub-mkrescue /usr/lib/grub/i386-pc -o dist/x86_64/kernel.iso targets/x86_64/iso
xorriso 1.5.4 : RockRidge filesystem manipulator, libburnia project.

Drive current: -outdev 'stdio:dist/x86_64/kernel.iso'
Media current: stdio file, overwriteable
Media status : is blank
Media summary: 0 sessions, 0 data blocks, 0 data, 29.0g free
Added to ISO image: directory '/'='/tmp/grub.MKKxj9'
xorriso : UPDATE :     335 files added in 1 seconds
Added to ISO image: directory '/'='/usr/lib/grub/i386-pc'
Added to ISO image: directory '/'='/root/env/targets/x86_64/iso'
xorriso : UPDATE :     643 files added in 1 seconds
xorriso : NOTE : Copying to System Area: 512 bytes from file '/usr/lib/grub/i386-pc/boot_hybrid.img'
xorriso : UPDATE : Thank you for being patient. Working since 0 seconds.
ISO image produced: 6275 sectors
Written to medium : 6275 sectors at LBA 0
Writing to 'stdio:dist/x86_64/kernel.iso' completed successfully.
```

5) Вихід із контейнера:

```
exit
```

6) Емуляція ОС за допомогою QEMU:

```
qemu-system-x86_64 -cdrom dist/x86_64/kernel.iso
```

Деталі (або поширені проблеми):

- Закрийте емулятор після завершення роботи, щоб не блокувати запис до kernel.iso для майбутніх збірок.
- Якщо наведена вище команда не спрацювала, спробуйте один з наступних способів:
  - Windows: qemu-system-x86_64 -cdrom dist/x86_64/kernel.iso -L "C:\Program Files\qemu"
  - Linux: qemu-system-x86_64 -cdrom dist/x86_64/kernel.iso -L /usr/share/qemu/.

7) Очищення зображення Докер-контейнера після завершення роботи:

```
docker rmi myos-buildenv -f
```

5. Код та що нам знадобиться для завантаження ОС

Цей код буде знаходитись у файлі header.asm.

Перед тим, як перейти до запуску нашої ОС, потрібно її завантажити. Для цього ми будемо використовувати завантажувач multiboot2.

5.1. Multiboot2

Кожна операційна система повинна мати власний завантажувач. Специфікація Multiboot2 має на меті організувати усталений інтерфейс між завантажувачем та операційною системою, щоб будь-який відповідний завантажувач міг завантажити будь-яку відповідну операційну систему. Ця специфікація не описує і не визначає, як працює завантажувач, а лише те, як він повинен взаємодіяти з операційною системою, яку він завантажує. Специфікація multiboot2 орієнтована на 32-розрядні системи на персональних комп'ютерах. Щоб завантажити образ операційної системи, потрібно виконати декілька підготовчих дій, а саме - створити образ нашої ОС.

Ідеальним варіантом образу є звичайний 32-розрядний
виконуваний файл будь-якого формату, який зазвичай використовується операційною системою. Для спрощення цього процесу використовується специфікація Multiboot2. Для визначення сумісності з цією специфікацією використовується чарівний заголовок Multiboot2, який дозволяє завантажувачу не розбиратися у всіх варіаціях a.out (вихідні файли, створені компіляторами), а просто завантажити образ.

Цей образ може бути приєднано до адреси завантаження не за замовчуванням, щоб уникнути завантаження через область вводу/виводу або інші зарезервовані області.

Цей заголовок має міститися у перших 32768 байтах образу ОС і бути вирівняним по 64 бітам!

Для ідентифікації специфікаційного заголовка Multiboot2 використовується магічне число 0xE85250D6 для визначення версії та сумісності.

```
section .multiboot_header
header_start:
        ; magic number
        dd 0xe85250d6 ; multiboot2
```

5.2. Захищений режим

Також, для успішного завантаження ОС, потрібно уточнити архітектуру, де 0 - це 32-розрядний захищений режим i386, 4 - 32-розрядний MIPS.

Захищений режим - основний режим роботи сучасних процесорів Intel. Дозволяє працювати з декількома віртуальними адресними просторами, кожен з яких має 4 ГБ адресованої пам'яті і дозволяє забезпечити суворий захист пам'яті та апаратного вводу/виводу.

```
; architecture
        dd 0 ; protected mode i386
```

MIPS (Microprocessor without Interlocked Pipeline Stages [1]) - мікропроцесорна архітектура, розроблена компанією MIPS Computer Systems (нині MIPS Technologies) відповідно до концепції проектування RISC-процесорів, тобто концепції зі скороченим набором інструкцій.

5.3. Налаштування перевірки правильності завантаження

Для перевірки правильності завантаження нам потрібно встановити контрольну суму. Контрольна сума - це 32-бітове беззнакове значення, яке при додаванні до magic, архітектури та довжини заголовка має значення 0.

```
; header length
        dd header_end - header_start
        ; checksum
        dd 0x100000000 - (0xe85250d6 + 0 + (header_end -
            header_start))
```

Мітки «header_start» та «header_end» позначають межі заголовка мультизавантаження в пам'яті.

Ми закінчуємо заголовок тегом зі структурою (тип, прапори, розмір).

```
dw 0
dw 0
dd 8
```

Після того, як ми закінчили налаштовувати завантаження нашої ОС можна приступати до написання коду для її вхідної точки.

Вхідна точка у нашу операційну систему буде знаходитися у файлі main.asm.

Увесь код знаходиться у секції .text, тож цей код не буде винятком (section .text). Також нам потрібно вказати, що наші інструкції знаходяться у 32-бітному режимі (bits 32). До того ж, нам знадобиться глобальна змінна start для позначення точки входження до виконання нашої програми.

```
global start

section .text
bits 32
start:
    ; print 'OK'
    mov dword [0xb8000], 0x2f4b2f4f
    hlt
```

Виводити 'OK' на екран ми будемо шляхом прямого запису цього тексту у текстовий режим відеопам'яті (0xb8000).

$0x2f4b2f4f = 0x2f$ (green-yellow) $+ 0x4b$ ('K') $\quad +0x2f$ (green-yellow) $+ 0x4f$ ('O')

Результат виглядатиме ось так:

Література

[1] SpaceLab, "Docker: Інструкції для початківців", https://spacelab.ua/articles/Docker/

[2] Docker Hub, "randomdude/gcc-cross-x86_64-elf", https://hub.docker.com/r/randomdude/gcc-cross-x86_64-elf

[3] PDOS, "Tools for Operating System Development", https://pdos.csail.mit.edu/6.828/2018/tools.html

[4] GNU, "Multiboot2 Manual", https://www.gnu.org/software/grub/manual/multiboot2/multiboot.html

[5] Wikipedia, "GNU GRUB", https://uk.wikipedia.org/wiki/GNU_GRUB

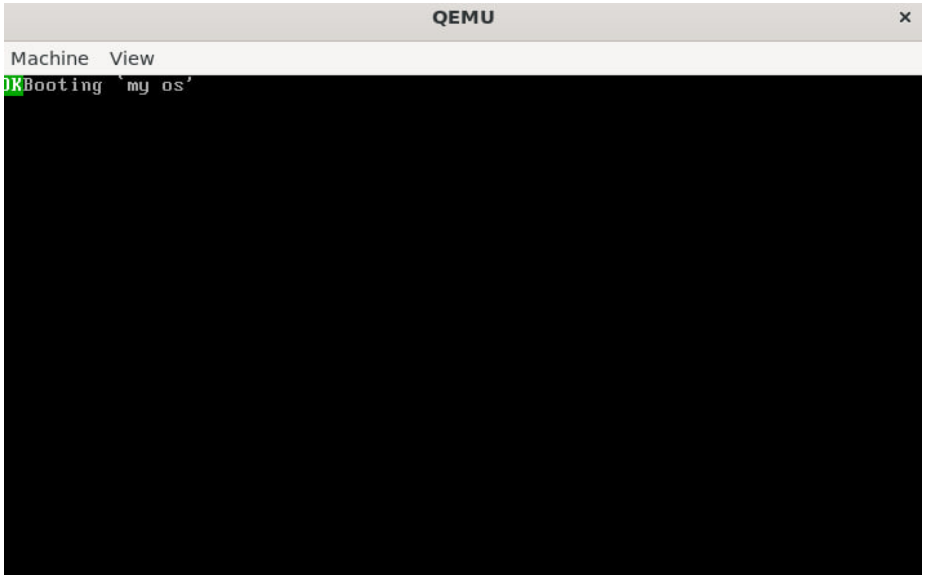

Рис. 3: Вивід першого уроку

[6] Intel, "Intel 64 and IA-32 Architectures Software Developer's Manual", https://www.intel.com/content/www/us/en/content-details/782158/intel-64-and-ia-32-architectures-software-\developer-s-manual-combined-volumes-1-2a-2b-2c-2d-3a-3b-3c-3d-and-4.html?wapkw=intel%2064%20and%20ia-\32%20architectures%20software%20developer%27s%20manual&docid=782161

[7] PSTNET, "E-BASIC CColor Color Object", https://support.pstnet.com/hc/en-us/articles/360021200914-E-BASIC-CColor-\Color-Object-RteColor-provides-expanded-colors-and-HTML-color-listing-17814

[8] YouTube, "Video 1", https://www.youtube.com/watch?v=FkrpUaGThTQ

[9] CrateCode, "X86 Processors", https://cratecode.com/info/x86-processors

[10] David Callanan, "OS Series Ep1", https://github.com/davidcallanan/os-series/tree/ep1

[11] Wikipedia, "x86", https://uk.wikipedia.org/wiki/X86

[12] Wikipedia, "Legacy system", https://en.wikipedia.org/wiki/Legacy_system