

# 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 сьогодні 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 гав і на початку 2000-х років представив Athlon та Opteron, 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 конвєсризація, 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:

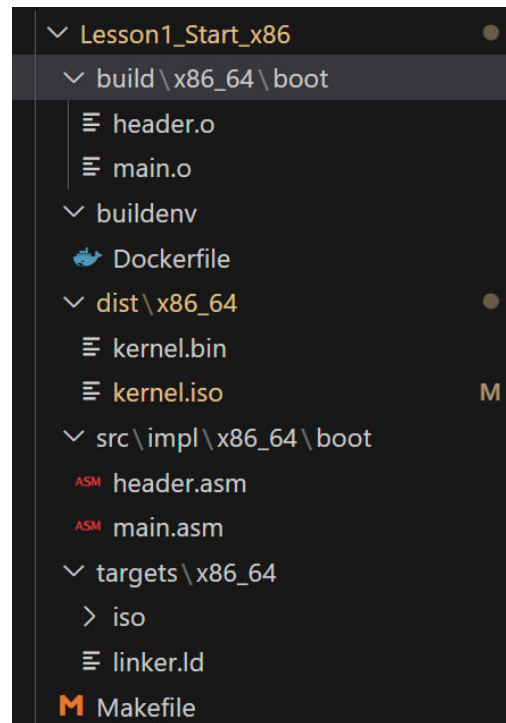


Рис. 1: Структура нашої директорії

Далі іде опис що і де знаходиться.

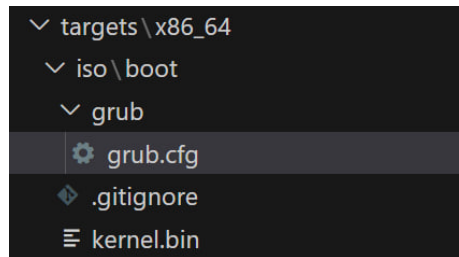


Рис. 2: Папка targets

### 3. Лінування

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 1М, що означає, що всі дані нашої ОС починатимуться з 1М. 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).

```
1 ENTRY( start )
2
3 SECTIONS
4 {
5     . = 1M;
6
7     .boot :
8     {
9         KEEP(*( .multiboot_header ))
10    }
11
12    .text :
13    {
14        *( .text )
15    }
16 }
```

Отже всі дані нашої ОС будуть починатись з 1М, 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`).

```
1 set timeout=0
2 set default=0
3
4 menuentry "my os" {
5     multiboot2 /boot/kernel.bin
6     boot
7 }
```

## 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:

```
1 FROM randomdude/gcc-cross-x86_64-elf
2
3 RUN apt-get update
4 RUN apt-get upgrade -y
```

```

5 RUN apt-get install -y nasm
6 RUN apt-get install -y xorriso
7 RUN apt-get install -y grub-pc-bin
8 RUN apt-get install -y grub-common
9
10 VOLUME /root/env
11 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!**

```

1 docker build buildenv -t myos-buildenv

```

**Це може зайняти кілька хвилин!**

### 3) Вхід в контейнер:

**Linux чи MacOS:**

```

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

```

**Windows (CMD):**

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

#### Windows (PowerShell):

```
1 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) Бїлд:

```
1 make build-x86_64
```

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

```
1 mkdir -p dist/x86_64 && \  
2 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 && \  
3 cp dist/x86_64/kernel.bin targets/x86_64/iso/boot/kernel.bin &&  
   \  
4 grub-mkrescue /usr/lib/grub/i386-pc -o dist/x86_64/kernel.iso  
   targets/x86_64/iso  
5 xorriso 1.5.4 : RockRidge filesystem manipulator, libburnia  
   project.  
  
6  
7 Drive current: -outdev 'stdio:dist/x86_64/kernel.iso '  
8 Media current: stdio file, overwriteable  
9 Media status : is blank  
10 Media summary: 0 sessions, 0 data blocks, 0 data, 29.0g free  
11 Added to ISO image: directory '/'='/tmp/grub.MKKxj9 '  
12 xorriso : UPDATE :      335 files added in 1 seconds  
13 Added to ISO image: directory '/'='/usr/lib/grub/i386-pc '  
14 Added to ISO image: directory '/'='/root/env/targets/x86_64/iso  
   '  
15 xorriso : UPDATE :      643 files added in 1 seconds  
16 xorriso : NOTE : Copying to System Area: 512 bytes from file '/'  
   usr/lib/grub/i386-pc/boot_hybrid.img '  
17 xorriso : UPDATE : Thank you for being patient. Working since 0  
   seconds.
```

```
18 ISO image produced: 6275 sectors
19 Written to medium : 6275 sectors at LBA 0
20 Writing to 'stdio:dist/x86_64/kernel.iso' completed
    successfully.
```

## 5) Вихід із контейнера:

```
1 exit
```

## 6) Емуляція ОС за допомогою QEMU:

```
1 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:

```
1 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.

```
1 section .multiboot_header
2 header_start:
3     ; magic number
4     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.

```
1 ; architecture
2     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.

```
1 ; header length
2     dd header_end - header_start
3     ; checksum
4     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).

```
1 dw 0
2 dw 0
3 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.

```
1 global start
2
3 section .text
4 bits 32
5 start:
6     ; print 'OK'
7     mov dword [0xb8000], 0x2f4b2f4f
8     hlt
```

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

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

Результат виглядатиме ось так:

## Література

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

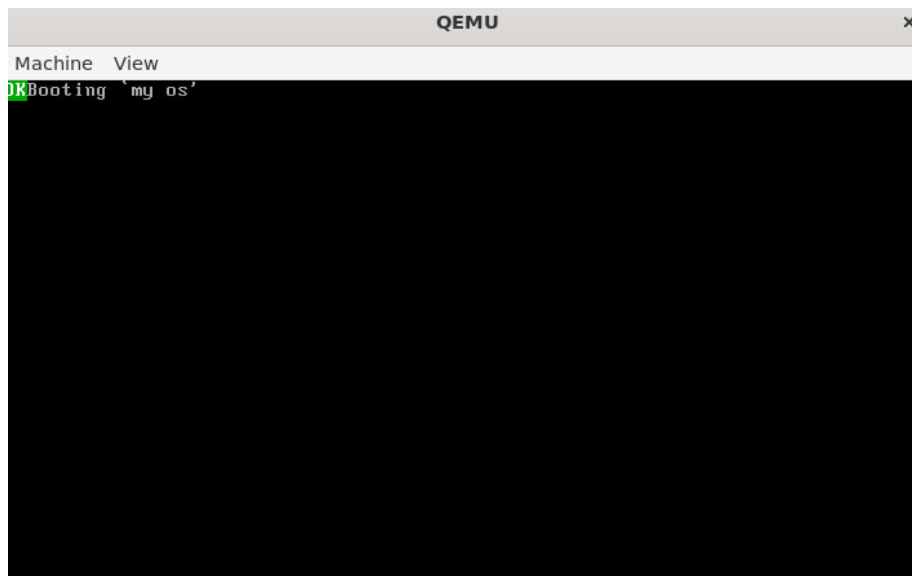


Рис. 3: Вивід першого уроку

- [2] Docker Hub, "randomdude/gcc-cross-x86\_64-elf", [https://hub.docker.com/r/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](https://uk.wikipedia.org/wiki/GNU_GRUB)
- [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](https://en.wikipedia.org/wiki/Legacy_system)