Урок 2: Перехід із 32-ох бітного режиму у 64-бітний

Катерина Ковальчук

Грудень 2024р

Анотація

У цьому документі описується процес переходу із 32-ох бітного режиму x86_64 до 64-бітного

1. Вступ

З появою процесорів x86-64 (AMD64, Intel 64 (також відомий як EM64T), VIA Nano) з'явився новий режим, який називається довгим режимом. Довгий режим складається з двох підрежимів: власне 64-розрядного режиму та режиму сумісності (32-розрядний, зазвичай позначається як IA32e у посібниках з AMD64). Нас цікавить саме 64-розрядний режим, оскільки він надає багато нових можливостей, таких як: розширення регістрів до 64-біт (rax, rcx, rdx, rbx, rsp, rbp, rip і т.д.) і введення восьми нових регістрів загального призначення (r8 - r15), а також впровадження восьми нових мультимедійних регістрів (xmm8 - xmm15). 64-розрядний режим - це, по суті, новий світ, оскільки в ньому майже повністю відсутня сегментація, яка використовувалася в 8086-процесорах, а GDT, IDT, пейджинг і т.д. також дещо відрізняються від старого 32-розрядного режиму (так званого захищеного режиму).

2. Компіляція

Процес компіляції повністю поторює процес із першого уроку із потрібними змінами у Makefile.

ПІД ЧАС ПОБУДОВИ ДОКЕР-ЗОБРАЖЕННЯ МОЖЕ ВИНИКНУТИ ТАКА ПОМИЛКА:

```
1 docker build buildenv -t myos-buildenv
2 [+] Building 0.4s (3/3) FINISHED
    docker:default
3 => [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. Зміни у структурі

У нас з'явилася папка kernel та x86_64/boot. Також додався main64.asm та файли .c для виводу тексту на екран. Давайте ж розглянемо, що додалось для успішного переходу в 64-бітний режим.

4. Код

Зміни торкнулись файлів main та main64.asm. Файл header.asm залишився незмінним, тому що нам все ще потрібен multiboot2 та захищений режим для успішного завантаження нашої ОС.

Що ж змінилося у main.asm? Отож, по порядку:

Налаштування стеку:

Стек - це місце у пам'яті комп'ютера, де зберігаються дані викликів функцій, що включає в себе всі локальні змінні функцій та адреси повернень функцій.

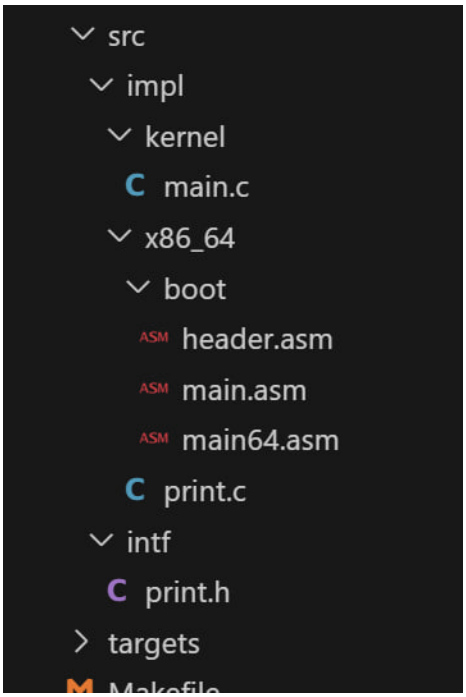


Рис. 1: Зміни у структурі директорії ОС

Пам'ять, яку ми виділили під стек, буде зарезервовано завантажувачем під час завантаження системи.

```
section .bss
stack_bottom:
    resb    4096 * 4
stack_top:
```

Вказівник на адресу поточного місця на стеку зберігає esp регістр, тож ми також її туди запишемо.

```
mov  esp, stack_top
```

Після налаштування стеку ми можемо приступити до переходу в Довгий режим.

Для того, щоб перейти у 64-бітний режим, нам потрібно отримати доступ до 8-го біта регістра EFER.

Extended Feature Enable Register (EFER) - це регістр, який з'явився в процесорах AMD K6. Біти цього регістра призначені для дозволу або заборони інструкцій SYSCALL/SYSRET, керування режимом невиконання, входу/виходу в/з довгого режиму, а також для інших системних налаштувань. Регістр виявився настільки важливим, що був адаптований компанією Intel для своїх процесорів. Його індекс MSR - 0xC0000080.

```
/* x86-64 specific MSRs */
#define MSR_EFER                0xc0000080 /* extended feature register */
```

Рис. 2: MSR EFER

Для того, щоб увімкнути його і перевірити підтримку довгого режиму, нам потрібно скористатися командою CPUID.

CPUID (CPU Identification - ідентифікація процесора) - це команда асемблера, яка використовується для отримання інформації про процесор. З її допомогою програма може визначити тип процесора і його можливості (наприклад, можна визначити, які розширення набору інструкцій підтримуються).

Для того, щоб перевірити підтримку цієї інструкції, нам потрібно перевірити 21 біт в регістрі EFLAGS.

ID (bit 21)	**Identification flag** — The ability of a program to set or clear this flag indicates support for the CPUID instruction.

Рис. 3: 21 біт регістру EFLAGS

Зробити це можна таким способом:

Виштовхнути поточний стан регістра прапорців у стек (pushfd), витягнути значення зі стеку у регістр eax, зберегти початкове значення регістрів у ecx, "увімкнути" 21-й біт (прапорець наявності CPUID) в eax, заштовхнути змінене значення eax у регістр прапорців, знову зберегти поточні прапорці у стек, занети нові значення прапорців у eax, порівняти eax та ecx. Якщо значення 21-го біта відрізняються - значить інструкція CPUID підтримується, якщо ні - не підтримується.

Ось як це виглядатиме в коді:

```
chech_cpuid:
    pushfd
    pop eax
    mov ecx, eax
    xor eax, 1 << 21
    push eax
    popfd
    pushfd
    pop eax
    push ecx
    popfd
    cmp eax, ecx
    je .no_cpuid
    ret

.no_cpuid:
    mov al, "C"
    jmp error

error:
```

```
    ; print "ERR: X" where X is the error code
    mov dword [0xb8000], 0x4f524f45
    mov dword [0xb8004], 0x4f3a4f52
    mov dword [0xb8008], 0x4f204f20
    mov byte  [0xb800a], al
    hlt
```

Далі нам потрібно перевірити, чи доступні розширені можливості CPUID (0x80000000) - Функції з такими номерами надають розширену інформацію про процесор, оскільки ця команда постійно розширюється і туди додається нова інформація. Тоді ми зможемо перевірити 29-й біт регістра edx, який заповнюється викликом функції 0x80000001, і якщо 29-й біт дорівнює 1, то це означає, що ми підтримуємо довгий режим.

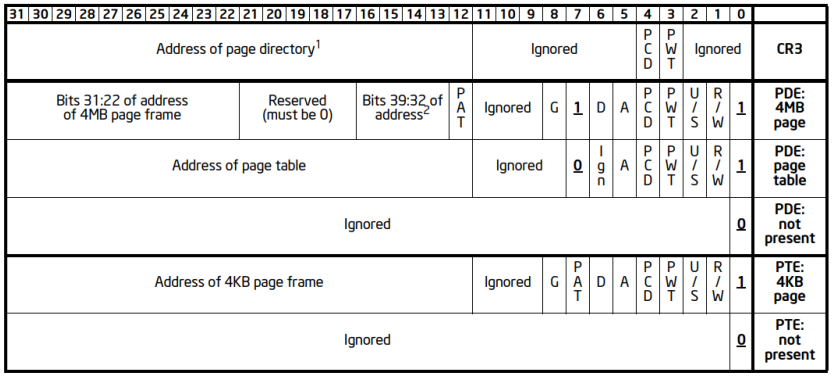
```
check_long_mode:
        mov eax, 0x80000000
        cpuid
        cmp eax, 0x80000001
        jb .no_long_mode

        mov eax, 0x80000001
        cpuid
        test edx, 1 << 29
        jz .no_long_mode

        ret
.no_long_mode:
    mov al, "L"
    jmp error
```

Після того, як ми перевірили, чи підтримується довгий режим, нам потрібно налаштувати сторінки, їхні розміри та їх трансляцію і записати розташування таблиць в пам'ять.

Тут ми можемо побачити, що перші біти у PDE:4MB page це біт наявності (0 біт) і біт дозволу запису (1 біт). Тож перевіряти чи сторінка є у фізичній пам'яті і чи можна в неї записувати інормацію ми будемо за допомогою числа 0b11.

```
section .bss
align 4096
page_table_L4:
    resb 4096
page_table_L3:
    resb 4096
page_table_L2:
    resb 4096
```

```
setup_page_tables:
    mov eax, page_table_L3
    or eax, 0b11 ; present, writable
```



31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
Address of page directory[1]																				Ignored							PCD	PWT	Ignored			CR3
Bits 31:22 of address of 4MB page frame										Reserved (must be 0)						Bits 39:32 of address[2]			PAT	Ignored			G	1	D	A	PCD	PWT	U/S	R/W	1	PDE: 4MB page
Address of page table																				Ignored				0	Ign	A	PCD	PWT	U/S	R/W	1	PDE: page table
Ignored																															0	PDE: not present
Address of 4KB page frame																				Ignored			G	PAT	D	A	PCD	PWT	U/S	R/W	1	PTE: 4KB page
Ignored																															0	PTE: not present

Figure 5-4. Formats of CR3 and Paging-Structure Entries with 32-Bit Paging

Рис. 4: Біти сторінок

```
    mov [page_table_L4], eax

    mov eax, page_table_L2
    or eax, 0b11 ; present, writable
    mov [page_table_L3], eax

    mov ecx, 0 ; counter

.loop:
        mov eax, 0x200000 ; 2MiB
        mul ecx
        or eax, 0b10000011 ; present, writable, huge page
        mov [page_table_L2 + ecx * 8], eax

        inc ecx ; increment counter
        cmp ecx, 512 ; checks if the whole table is mapped
        jne .loop ; if not, continue

        ret
```

Тут ми також налаштовуємо, що у таблиці сторінок рівня 2 всі сторінки будуть великого розміру (2Мб), за допомогою 7-го біта формату сторінок:

7 (PS)	Page size; must be 1 (otherwise, this entry references a page table; see Table 5-5)

Рис. 5: 7-й біт характеристики сторінки

Нам також знадобиться PAE, щоб правильно працювати в довгому режимі.

Розширення фізичної адреси (Physicall Address Extention) Дозволяє працювати з фізичними адресами, більшими за 4 ГБ, навіть на 32-бітних системах. Коли біт

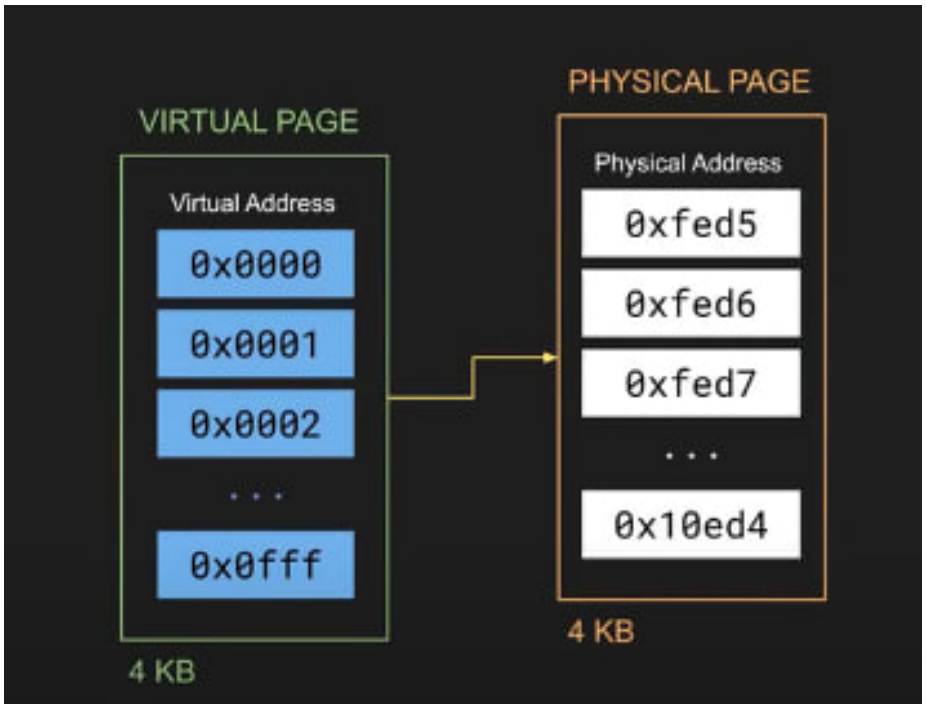


Рис. 6: Трансляція віртуальної адреси у фізичну

PAE вимкнено, кожен запис у таблиці сторінок визначає 32-бітну базову адресу у фізичній пам'яті. Коли біт увімкнено, запис розширюється до 64 біт (хоча більшість процесорів підтримують менше цього числа). Ми можемо увімкнути PAE, увімкнувши 5-й біт регістра cr4.

```
mov eax, cr4
or eax, 1 << 5
mov cr4, eax
```

Після увімкнення PAE ми можемо увімкнути «Довгий режим» за допомогою регістра EFER. Ми можемо зробити це, увімкнувши 8 біт цього регістру.

```
mov ecx, 0xC0000080
rdmsr
or eax, 1 << 8
wrmsr
```

І, нарешті можна увімкнути трансляцію, встановивши 31 біт cr0 у 1.

```
mov eax, cr0
or eax, 1 << 31
mov cr0, eax
```

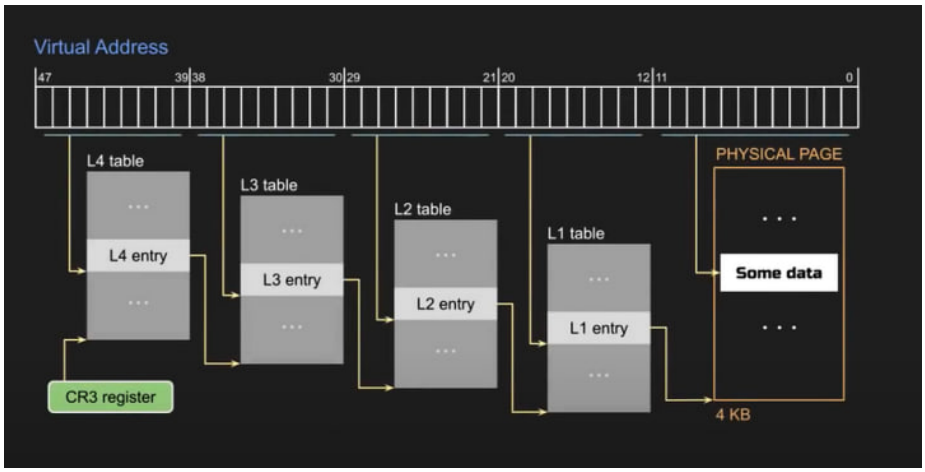


Рис. 7: Структура таблиці сторінок

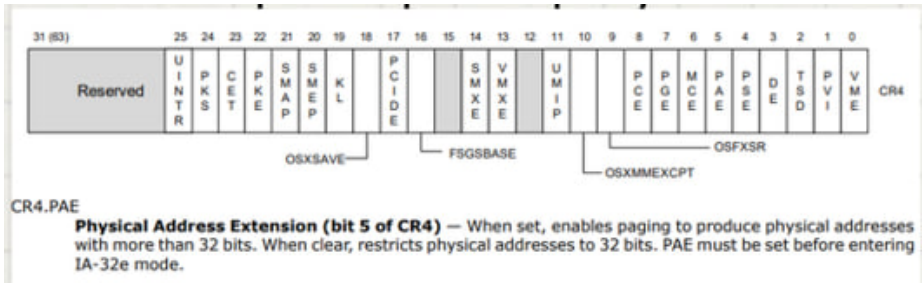


Рис. 8: Структура регістра CR4

Успішно налаштувавки 64-бітовий режим і увімкнувши його ми все ще знаходимось у 32-ох бітному підрежимі. Для того, щоб усіпшно перейти у Довгий режим нам потрібно налаштувати GDT.
GDT (Global Descriptor Table) - глобальна таблиця дескрипторів, службова структура даних в архітектурі x86, що визначає глобальні (загальні для всіх завдань) сегменти .

```
section .read_only_data
gdt64:
        dq 0 ; zero entry
.code_segment: equ $ - gdt64
        dq (1 << 43) | (1 << 44) | (1 << 47) | (1 << 53) ; code
            segment
.pointer:
        dw $ - gdt64 - 1 ; length
        dq gdt64 ; address
```

5	PAE	Physical Address Extension	If set, changes page table layout to translate 32-bit virtual addresses into extended 36-bit physical addresses.

Рис. 9: 5-й біт регістра CR4

```
/* EFER bits: */
#define _EFER_SCE               0  /* SYSCALL/SYSRET */
#define _EFER_LME               8  /* Long mode enable */
#define _EFER_LMA              10 /* Long mode active (read-only) */
#define _EFER_NX               11 /* No execute enable */
#define _EFER_SVME              12 /* Enable virtualization */
#define _EFER_FFXSR             14 /* Enable Fast FXSAVE/FXRSTOR */
```

Рис. 10: Біти регістра EFER

Вона має починатись з 0 входу (dq 0), після ми повинні увімкнути потрібні сегменти таблиці дескрипторів:

- **Прапорець P (сегмент присутній) (47 біт):** Показує, чи присутній сегмент у пам'яті (встановлений) або відсутній (скинутий). Якщо цей прапорець чистий, процесор генерує виключення відсутності сегмента (#NP), коли селектор сегмента, що вказує на дескриптор сегмента, завантажується у сегментний регістр.
- **Прапорець S (тип дескриптора) (44 біт):** Визначає, чи є дескриптор сегмента системним сегментом (S очищено), чи сегментом коду або даних (S встановлено).
- **Сегмент виконуваного коду (43 біт):** Прапорець D вказує на довжину за замовчуванням для ефективних адрес та операндів:
  - Якщо прапорець встановлено - використовуються 32-розрядні адреси та 32-розрядні або 8-розрядні операнди.
  - Якщо прапорець не встановлено - використовуються 16-розрядні адреси та 16-розрядні або 8-розрядні операнди.

  Префікс 66H вибирає розмір операнда, відмінний від значення за замовчуванням, а префікс 67H — розмір адреси.
- **Прапорець L (64-бітовий сегмент коду) (53 біт):** У режимі IA-32e біт 21 другого подвійного слова дескриптора визначає, чи сегмент коду містить 64-бітовий код:
  - Значення 1 — інструкції виконуються у 64-бітному режимі.
  - Значення 0 — інструкції виконуються у режимі сумісності.

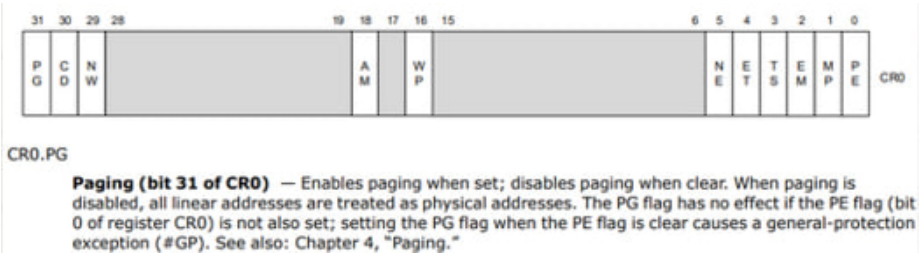


Рис. 11: Структура CR0

Якщо встановлено L-біт, D-біт має бути очищений. Біт 21 не використовується поза режимом IA-32e. Уникати завантаження CS із дескриптора з встановленим L-бітом поза режимом IA-32e.

Нарешті треба налаштувати вказівник на цю структуру та завантажити її у gdtr - регістр глобальної таблиці дескрипторів.

І після того, як ми все увімкнули, ми можемо записати 0 у всі регістри сегмента даних у файлі main64.asm, щоб коректно завантажити цей режим.

```
global long_mode_start
extern kernel_main

section .text
bits 64
long_mode_start:
    ; load null into all data segment registers
    mov ax, 0
    mov ss, ax
    mov ds, ax
    mov es, ax
    mov fs, ax
    mov gs, ax

    call kernel_main
    hlt
```

Код із print.c досить простий - там знаходяться функції для виведення тексту на екран, а в print.h - оголошення цих функцій:

clear_row(size_t row): Очищає конкретний рядок на екрані, встановлюючи кожен символ на пробіл з поточним кольором.

print_clear(): Очищає весь екран, викликаючи clear_row() для всіх рядків.

print_newline(): Переміщує курсор на наступний рядок, прокручуючи екран вгору, якщо потрібно.

print_char(char character): Виводить один символ на екран в поточну позицію курсора. Якщо символ — це новий рядок, викликається print_newline().

print_str(char* str): Виводить нуль-термінований рядок на екран, викликаючи print_char() для кожного символу в рядку.

print_set_color(uint8_t foreground, uint8_t background): Встановлює кольори тексту, комбінуючи передній та задній фон.

За допомогою цих функцій, які ми викликаємо в main.c:

```
#include "print.h"

void kernel_main() {
    print_clear();
    print_set_color(PRINT_COLOR_YELLOW, PRINT_COLOR_BLACK);
    print_str("Welcome to our 64-bit kernel!");
}
```

Ми отримуємо такий результат:

Рис. 12: Результат другого уроку

Література

[1] OSDev, "GDT Tutorial", https://wiki.osdev.org/GDT_Tutorial

[2] Wikipedia, "Control registers in Intel x86 series", https://en.wikipedia.org/wiki/Control_register#Control_registers_in_Intel_x86_series

[3] Felix Cloutier, "CPUID instruction", https://www.felixcloutier.com/x86/cpuid

[4] Wikipedia, "CPUID", https://uk.wikipedia.org/wiki/CPUID

[5] 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

[6] YouTube, "Video 2", https://www.youtube.com/watch?v=wz9CZBeXR6U

[7] OSDev, "GDT Tutorial", https://wiki.osdev.org/GDT_Tutorial