

# 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.

## 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. Даваймо ж розглянемо, що додалось для успішного переходу в 64-бітний режим.

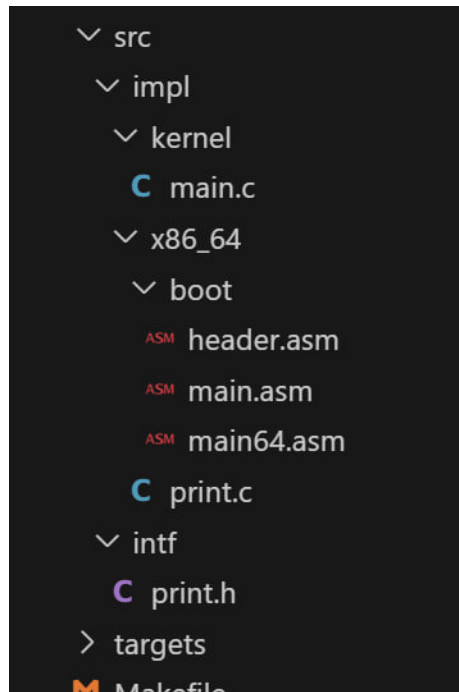


Рис. 1: Зміни у структурі директорії ОС

## 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.

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

```
1 section .bss
2 stack_bottom:
3     resb    4096 * 4
4 stack_top:
```

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

```
1 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.

<b>ID (bit 21)</b>	<b>Identification flag</b> — 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, занести 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:

```
1  chech_cpuid:  
2      pushfd  
3      pop  eax  
4      mov  ecx,  eax  
5      xor  eax, 1 << 21  
6      push eax  
7      popfd  
8      pushfd
```

```

9      pop eax
10     push ecx
11     popfd
12     cmp eax, ecx
13     je .no_cpuid
14     ret
15
16 .no_cpuid:
17     mov al, "C"
18     jmp error
19
20 error:
21     ; print "ERR: X" where X is the error code
22     mov dword [0xb8000], 0x4f524f45
23     mov dword [0xb8004], 0x4f3a4f52
24     mov dword [0xb8008], 0x4f204f20
25     mov byte [0xb800a], al
26     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.

```

1  check_long_mode:
2      mov eax, 0x80000000
3      cpuid
4      cmp eax, 0x80000001
5      jb .no_long_mode
6
7      mov eax, 0x80000001
8      cpuid
9      test edx, 1 << 29
10     jz .no_long_mode
11
12     ret
13 .no_long_mode:
14     mov al, "L"
15     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 інформацію ми будемо за допомогою числа 0b11.

```

1  section .bss

```

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 <sup>1</sup>																				Ignored				P C D	P W T	Ignored					CR3		
Bits 31:22 of address of 4MB page frame								Reserved (must be 0)				Bits 39:32 of address <sup>2</sup>				P A T	Ignored				G	1	D	A	P C D	P W T	U / S	R / W	1	PDE: 4MB page			
Address of page table																				Ignored				0	I g n	A	P C D	P W T	U / S	R / W	1	PDE: page table	
Ignored																										0	PDE: not present						
Address of 4KB page frame																				Ignored				G	P A T	D	A	P C D	P W T	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

```

2 align 4096
3 page_table_L4:
4     resb 4096
5 page_table_L3:
6     resb 4096
7 page_table_L2:
8     resb 4096

1 setup_page_tables:
2     mov eax, page_table_L3
3     or  eax, 0b11 ; present, writable
4     mov [page_table_L4], eax
5
6     mov eax, page_table_L2
7     or  eax, 0b11 ; present, writable
8     mov [page_table_L3], eax
9
10    mov ecx, 0 ; counter
11
12 .loop:
13    mov eax, 0x200000 ; 2MiB
14    mul ecx
15    or  eax, 0b10000011 ; present, writable, huge page
16    mov [page_table_L2 + ecx * 8], eax
17
18    inc ecx ; increment counter
19    cmp ecx, 512 ; checks if the whole table is mapped
20    jne .loop ; if not, continue
21
22    ret

```

Тут ми також налаштуємо, 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

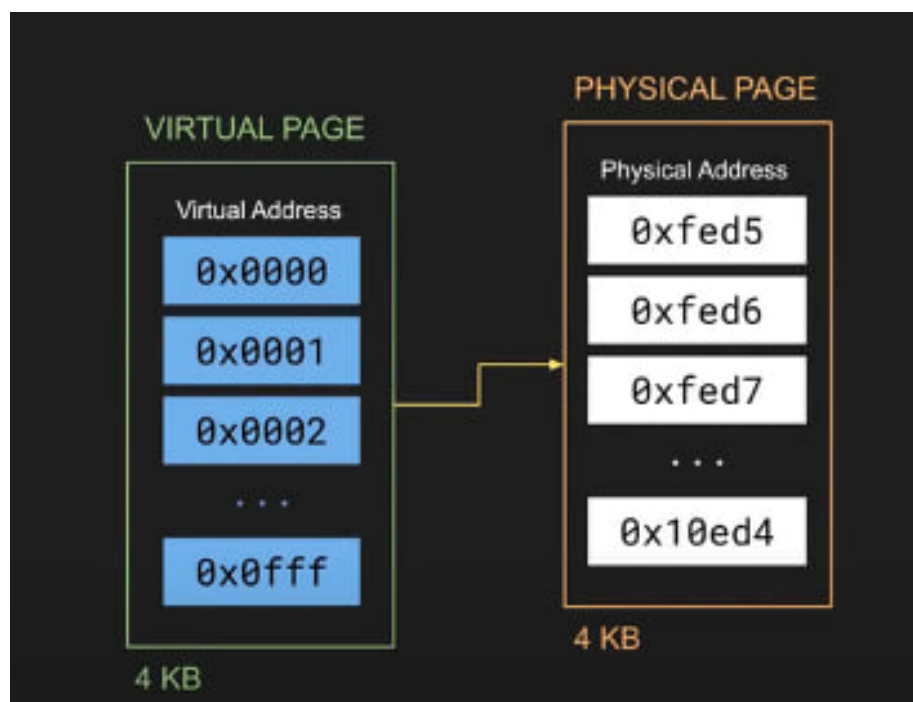


Рис. 6: Трансляція віртуальної адреси у фізичну

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

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

```

1 mov eax, cr4
2 or  eax, 1 << 5
3 mov cr4,  eax

```

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

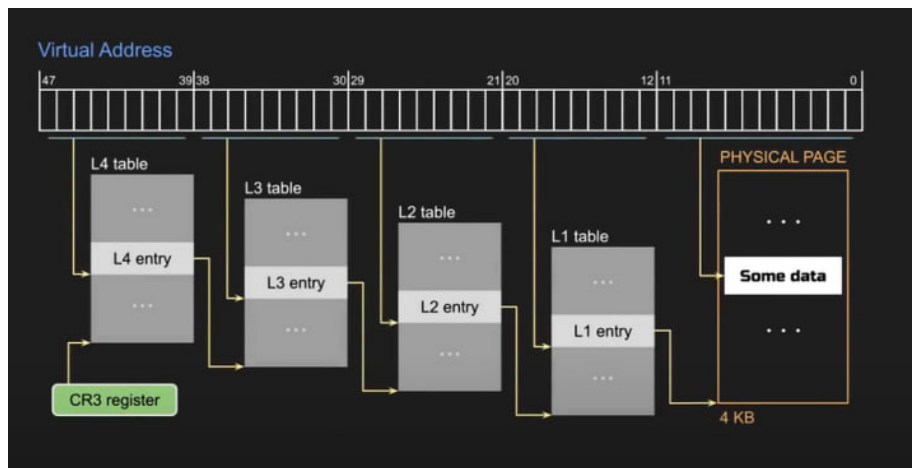


Рис. 7: Структура таблиці сторінок

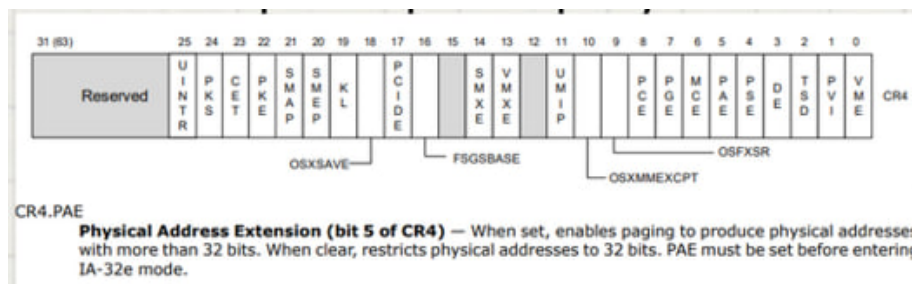


Рис. 8: Структура регістра CR4

```

1 mov ecx, 0xC0000080
2 rdmsr
3 or eax, 1 << 8
4 wrmsr

```

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

```

1 mov eax, cr0
2 or eax, 1 << 31
3 mov cr0, eax

```

Успішно налаштувавши 64-бітовий режим і увімкнувши його ми все ще знаходимось у 32-ох бітному підрежимі. Для того, щоб успішно 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.

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

```

1 section .read_only_data
2 gdt64:
3     dq 0 ; zero entry
4 .code_segment: equ $ - gdt64
5     dq (1 << 43) | (1 << 44) | (1 << 47) | (1 << 53) ; code
        segment
6 .pointer:
7     dw $ - gdt64 - 1 ; length
8     dq gdt64 ; address

```

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

- **Прапорець Р (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.



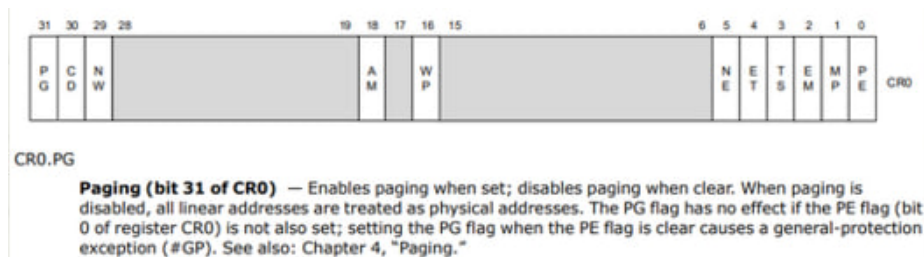


Рис. 11: Структура CR0

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.

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 gdt - 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.

```

1 global long_mode_start
2 extern kernel_main
3
4 section .text
5 bits 64
6 long_mode_start:
7     ; load null into all data segment registers
8     mov ax, 0
9     mov ss, ax
10    mov ds, ax
11    mov es, ax
12    mov fs, ax
13    mov gs, ax
14
15    call kernel_main
16    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:

```
1 #include "print.h"
2
3 void kernel_main() {
4     print_clear();
5     print_set_color(PRINT_COLOR_YELLOW, PRINT_COLOR_BLACK);
6     print_str("Welcome to our 64-bit kernel!");
7 }
```

Ми отримусмо такий результат:

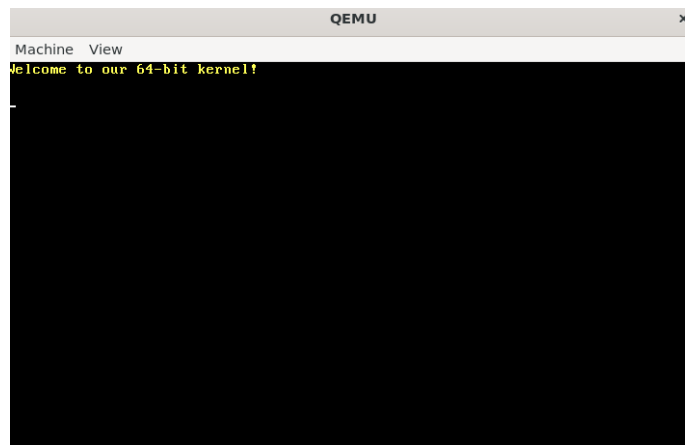


Рис. 12: Результат другого уроку

## Література

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