DOI: 10.36910/6775-2524-0560-2020-39-34
УДК: 004.415.25
**Христинець Наталія Анатоліївна**, ст.викладач
https://orcid.org/0000-0002-4836-7632
**Скупейко Дмитро Степанович**, студент
Луцький національний технічний університет

# ОСОБЛИВОСТІ ПРОЕКТУВАННЯ КОМПОНЕНТ МІКРОЯДРА ОПЕРАЦІЙНОЇ СИСТЕМИ ЗАСОБАМИ GCC, GNU BINARY UTILITIES В КОМПОЗИЦІЇ З МОВАМИ АССЕМБЛЕРА ТА С

**Христинець Н.А., Скупейко Д.С. Особливості проектування компонент мікроядра операційної системи засобами GCC, GNU Binary Utilities в композиції з мовами асемблера та С.** Висвітлено питання розробки мікроядра операційної системи: керування пам'яттю, розробка початкового завантажувача та розглянуті питання написання макросів і спеціальних функції у процесі програмування конфігураційного файлу. Сформовано схему збірки ядра від початкового коду до етапу емуляції.
**Ключові слова:** мікроядро операційної системи, Assembler, FASM, TASM, початковий завантажувач, обробник переривань, функція ядра операційної системи, бібліотека, макрос, GCC, GNU Binary Unitilies, компіляція, системні ресурси, збірка ядра.

**Христинець Н.А., Скупейко Д.С. Особенности проектирования компонент микроядра операционной системы средствами GCC, GNU Binary Utilities в композиции с языками ассемблера и С.** Освещены вопросы разработки микроядра операционной системы: управление памятью, разработка начального загрузчика и рассмотрены вопросы написания макросов и специальных функции в процессе программирования конфигурационного файла. Сформирована схема сборки ядра от исходного кода к этапу эмуляции.
**Ключевые слова:** микроядро операционной системы, Assembler, FASM, TASM, начальный загрузчик, обработчик прерываний, функция ядра операционной системы, библиотека, макрос, GCC, GNU Binary Unitilies,компиляция, системные ресурсы, сборник ядра.

**Khrystynets N., Skupeyko D. Features of designing components of the microkernel of the operating system by means of GCC, GNU Binary Utilities in the composition with the assembly language and C.** The issues of developing the microkernel of the operating system are covered: memory management, bootloader development and macro writing and special functions in the process of programming the configuration file. The scheme of assembly of a kernel from a source code to an emulation stage is generated.
**Keywords:** operating system microkernel, Assembly, FASM, TASM, bootloader, interrupt handler, operating system kernel function, library, macro, GCC, GNU Binary Unitilies, compilation, system resources, kernel compilation.

**Постановка проблеми:** Операційна система приховує апаратну складність, керує обчислювальними ресурсами та забезпечує ізоляцію та захист. Основними її компонентами є файлова система, обробник переривань, менеджер пам'яті, планувальник та драйвер пристрою. Ядро операційної системи, завантажене при запуску комп'ютера, керує ресурсами комп'ютера та обробляє запити від системних програм та додатків. Питання створення мікроядра операційної системи передбачає функціональні можливості та стан системи, необхідний для підтримки програм на базі користувача.

**Аналіз досліджень.** Більшість операційних систем, драйверів та утилітних програм написані комерційними організаціями, які поширюють виконувані версії свого програмного забезпечення - версії, які не можуть бути доповнені або змінені [1-3]. Відкритий код вимагає розповсюдження оригінальних матеріалів, які можна вивчити, змінити та доповнити, а результати буде знову вільно поширюватись. Самотужки створити операційну систему непросто. Навіть добре відомі операційні системи, такі як Android та Windows, створені командою з сотень людей. Це говорить про те, що створення операційної системи з нуля це ще й досить фінансово-затратний проект. Проте, якщо володіти переліком речей, які потрібно знати, перш ніж намагатися розпочати розробку операційної системи, то ядро операційної системи з мінімальними вимогами до функціонування написати все ж можна. Для цього необхідний час, бажання, основні знання з комп'ютерних наук, теорії комп'ютерного програмування та знання мов програмування низького та високого рівня.

Є багато принципів проектування операційної системи: принцип віртуалізації, генерованості, сумісності і т.д. Часто в літературі [4-5] вони розподілені по двом групам. Перша полягає в розмежуванні механізму та політики шляхом впровадження гнучких механізмів підтримки, друга – в оптимізації для загального випадку з огляду на те, де буде використовуватися ОС, що користувач захоче виконати на цій машині і які вимоги до навантаження. Існує три типи операційних систем, що часто використовуються. Перша - Monolithic OS, де вся ОС працює в просторі ядра і знаходиться в режимі супервізора. Друга - це модульна ОС, в якій деяка частина системного ядра буде розташована в незалежних файлах, званих модулями, які можна додати до системи під час виконання. І третя -

Micro OS, де ядро розбивається на окремі процеси, відомі як сервери. Деякі сервери працюють у просторі ядра, а деякі - у просторі користувача. Для створення операційної системи потрібно орієнтовно тисячі рядків коду. Їх розробка використовує мови програмування, такі як C, C++ та програмування на Assembler.

**Виклад основного матеріалу й обґрунтування отриманих результатів.** Фактично створення мікроядра має на меті створення програми для виконання процесором, або для керування його регістрами. Використовували мову Assembler, зокрема ассемблер FASM. На цій мові повністю написаний код початкового завантажувача (рис. 1), у якому відбувалось, наприклад:

- завантаження глобальної таблиці дескрипторів сегментів gdt32 у спеціальний регістр gdtr;
- заборона переривань (cli);
- перехідно до захищеного режиму роботи процесора, зміна біта PE на 1 та близький стрибок на мітку start32 (Intel рекомендує так робити);
- зміщення на адресу кратну 16 (align 16), для швидшого звернення до даних;
- опис дескрипторів глобальної таблиці дескрипторів (мітка gdt32);
- опис розміру структури таблиці дескрипторів та її адреса;
- мітка start32: підготовка регістрів до стрибка на ядро, заповнення додаткового регістрів даними про ідентифікатор диску, список завантажених модулів ядра та карту оперативної пам'яті;
- стрибок на ядро (jmp 0FFC0000h).

```
610
611                 ; Load values into GDTR (Global Description Table Register)
612                 lgdt    [gdtr32]
613                 ; Unable interrupts
614                 cli
615                 ; Switch to protected mode
616                 mov     EAX, CR0
617                 or      EAX, 80000001h
618                 mov     CR0, EAX
619                 ; Switch to 32-bin code
620                 jmp     8:start32
621 ; Description Segment Table for 32-bin kernel
622 align 16
623 gdt32:
624                 dq      0                       ; NULL - 0
625                 dq      00CF9A000000FFFFh       ; CODE32 - 8
626                 dq      00CF92000000FFFFh       ; DATA32 - 16
627                 dq      00009A000000FFFFh       ; CODE16
628                 dq      000092000000FFFFh       ; DATA16
629 gdtr32:
630                 dw      $ - gdt32 - 1
631                 dd      gdt32
632 ; 32-bit code
633 use32
634 start32:
635                 ; Set segment register and stack
636                 mov     EAX, 16
637                 mov     DS, AX
638                 mov     ES, AX
639                 mov     FS, AX
640                 mov     GS, AX
641                 mov     SS, AX
642                 mov     ESP, 0FFFFDFFCh
643
644                 ; Put in DL the boot disk number
645                 mov     DL, [disk_id]
646                 ; Put in EBX the address of the list of downloaded files
647                 mov     EBX, module_list
648                 ; Put in ESI memory map address
649                 mov     ESI, memory_map
650
651             ; Pass to the kernel
652                 jmp     0FFC00000h
```

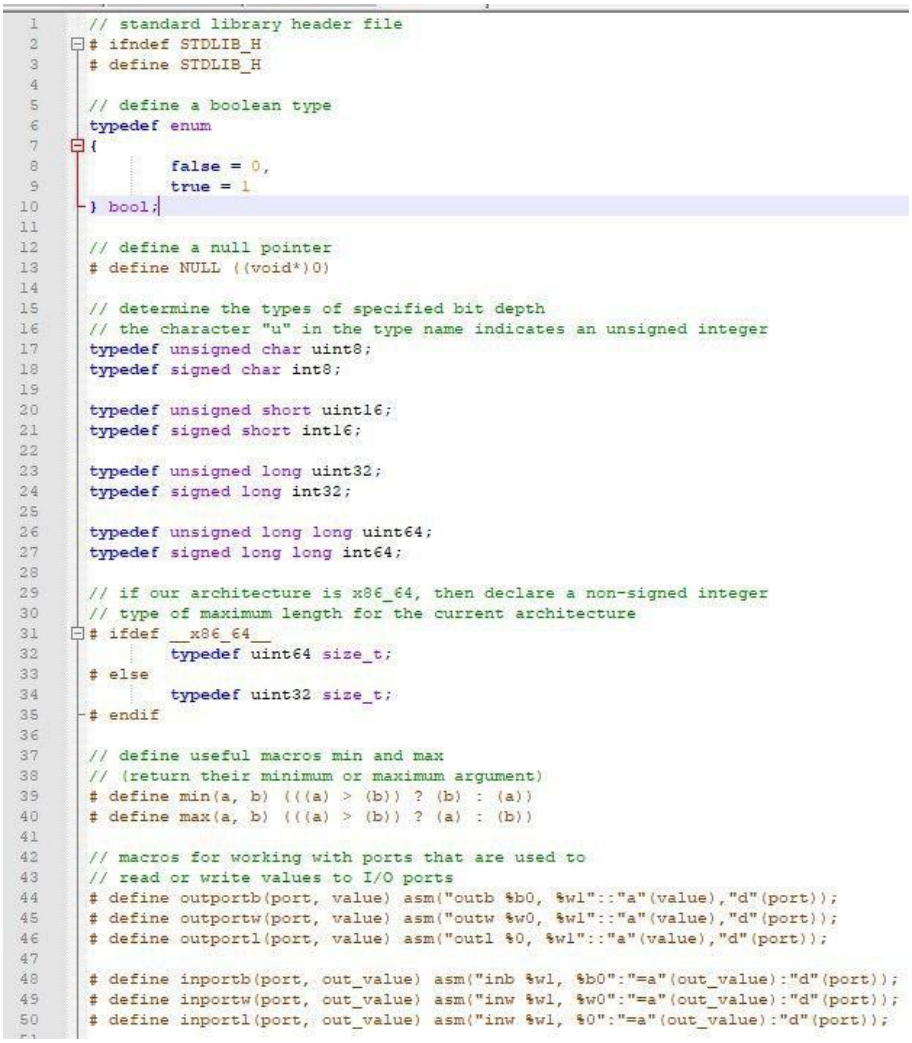
Рисунок 1 – Фрагмент кінцівки коду початкового завантажувача

Команда *cli* забороняє переривання від контролера переривань, щоб не можна було перервати даний набір команд (на рисунку 1), щоб забезпечити атомарність операції та весь перехід до захищеного режиму роботи процесора.

Після написання на FASM початкового завантажувача, здійснювалась ще певна підготовка на ассемблері для передачі параметрів у головну функцію ядра (*kernel_main*): надання відповідних параметрів стеку та виклик функції ядра. Коли з'явилась можливість переходу на головну функцію, було написано власно розроблену частину функціоналу стандартної бібліотеки C: у *stdlib* оголошено базові типи даних, різні макроси для роботи з портами вводу-виводу, створено функції для роботи з ділянками пам'яті та рядками. На основі цієї стандартної бібліотеки (рис. 2) базується робота модулів ядра та робота драйверів:

- оголошено тип даних bool;

- тип NULL;
- цілочисельні типи з різною розрядністю у версіях зі знаком та без;
- тип size_t в залежності від розрядності машини;
- макроси для визначення максимального та мінімального значення;
- макроси для роботи з портами I/O.



```
// standard library header file
# ifndef STDLIB_H
# define STDLIB_H

// define a boolean type
typedef enum
{
        false = 0,
        true = 1
} bool;

// define a null pointer
# define NULL ((void*)0)

// determine the types of specified bit depth
// the character "u" in the type name indicates an unsigned integer
typedef unsigned char uint8;
typedef signed char int8;

typedef unsigned short uint16;
typedef signed short int16;

typedef unsigned long uint32;
typedef signed long int32;

typedef unsigned long long uint64;
typedef signed long long int64;

// if our architecture is x86_64, then declare a non-signed integer
// type of maximum length for the current architecture
# ifdef __x86_64__
        typedef uint64 size_t;
# else
        typedef uint32 size_t;
# endif

// define useful macros min and max
// (return their minimum or maximum argument)
# define min(a, b) (((a) > (b)) ? (b) : (a))
# define max(a, b) (((a) > (b)) ? (a) : (b))

// macros for working with ports that are used to
// read or write values to I/O ports
# define outportb(port, value) asm("outb %b0, %w1"::"a"(value),"d"(port));
# define outportw(port, value) asm("outw %w0, %w1"::"a"(value),"d"(port));
# define outportl(port, value) asm("outl %0, %w1"::"a"(value),"d"(port));

# define inportb(port, out_value) asm("inb %w1, %b0":"=a"(out_value):"d"(port));
# define inportw(port, out_value) asm("inw %w1, %w0":"=a"(out_value):"d"(port));
# define inportl(port, out_value) asm("inw %w1, %0":"=a"(out_value):"d"(port));
```

Рисунок 2 – Фрагмент програмного коду хедера бібліотеки з модулями ядра і драйверами

Наприклад, перший макрос outportb(port, value) – робить зручним вивід у заданий порт (параметр port) значення розміром 1 байт (параметр value). А функція для роботи з ділянками пам'яті void memset (void *mem, char value, size_t count) – спрощує розміщення символу value у перших count позиціях ділянки mem та повертає вказівник на цю ділянку пам'яті mem (рис. 3):

```
// macros for working with ports that are used to
// read or write values to I/O ports
# define outportb(port, value) asm("outb %b0, %w1"::"a"(value),"d"(port));
# define outportw(port, value) asm("outw %w0, %w1"::"a"(value),"d"(port));
# define outportl(port, value) asm("outl %0, %w1"::"a"(value),"d"(port));

# define inportb(port, out_value) asm("inb %w1, %b0":"=a"(out_value):"d"(port));
# define inportw(port, out_value) asm("inw %w1, %w0":"=a"(out_value):"d"(port));
# define inportl(port, out_value) asm("inw %w1, %0":"=a"(out_value):"d"(port));

// declare prototypes of the functions of the standard library;
// first functions for working with sections of memory,
// then functions for working with strings;
// what each function does is described in the file stdlib.c
void memset(void *mem, char value, size_t count);
void memset_word(void* mem, uint16 value, size_t count);
void memcpy(void *dest, void *src, size_t count);
int memcmp(void *mem1, void *mem2, size_t count);
void *memchr(void *mem, char value, size_t count);

size_t strlen(char *str);
void strcpy(char *dest, char *src);
void strncpy(char *dest, char *src, size_t max_count);
int strcmp(char *str1, char *str2);
char *strchr(char *str, char value);

# endif
```

Рисунок 3 – Фрагмент коду з прототипами функцій

Робочий процес програмування конфігураційного файлу тривав досить довгий час, оскільки природно під час написання такого об'єму коду виникали питання вірного компілювання через не читання файлів з ядром (його не виявляла система), деякі питання збору образу системи, правильного читання карти пам'яті оперативної пам'яті і ще різного роду помилки з неправильною ініціалізацією стеку ядра та таблиці сторінок ядра. Проте, результат проектування з додатковим вивченням тем з відповідної літератури та інтернет-джерел [6-11] показав позитивні вирішення цих та інших питань.

Керування пам'яттю при розробці мікроядра зводиться до:

- ініціалізації самої пам'яті на основі даних з карти пам'яті RAM;
- виділення фізичних сторінок;
- звільнення фізичних сторінок;
- отримання об'єму вільної пам'яті;
- проектування фізичної сторінки на віртуальну адресу (необхідно для спрощення реалізації роботи функцій по виділенню та звільненню фізичних та віртуальних сторінок);
- отримання даних про конкретну віртуальну адресу.

Проектування фізичної сторінки (чи кількох сторінок) на задану віртуальну адресу здійснювалось за допомогою спеціально заданого блоку фізичної пам'яті, який має спеціально задану віртуальну адресу.

Функція набуває наступних параметрів:

1. Фізична адреса каталогу сторінок, тобто початкова адреса таблиці сторінок рівня 1.
2. Віртуальна адреса, яка буде використовуватись для спілкування зі спроектованою фізичною сторінкою.
3. Фізична адреса початкової фізичної сторінки, з цієї адреси бере початок виділення пам'яті .
4. Кількість фізичних сторінок, під які буде виділятись пам'ять.
5. Прапорець властивостей сторінок.

Функція повертає *bool* значення: *true* – якщо робота функції була успішною та *false* – у протилежному випадку. Сама функція – це цикл *for*, умовою якого, є «поки не виділиться необхідна кількість фізичних сторінок». Всередині циклу відбувається розбивка віртуальної 32-бітної адреси на індекси 1-го рівня, 0-го рівня та зміщення – це необхідно для того, щоб знайти необхідну фізичну сторінку у таблиці сторінок. Далі, знаходиться необхідний блок фізичної сторінки (чи кілька блоків), якщо він доступний (тобто не був виділений раніше), то він позначається як виділений, увесь записується нулями і встановлюються його властивості (присутність в оперативній пам'яті, доступність до запису, і чи сторінка доступна для користувача). Якщо ж фізична сторінка була успішно виділена, то збільшується значення адреси фізичного блоку – переміщення до наступного блоку, щоб його виділити і т.д. Усі блоки фізичної пам'яті мають розмір 4КБ. Дана процедура відбувається почергово, поки не виділиться необхідна кількість фізичних сторінок. В кінці функція повертає значення *true*.

Процес збірки образу системи від source-коду до вже готового образу системи здійснювався почергово. У Makefile були написані (рис. 4) усі команди для роботи з source-кодом від компіляції до збору в єдиний бінарний файл ядра.

Порядок написання наступний:

- вказується цільова архітектура (параметр для лінкера);
- параметри для компілятора GCC, тут сказано, що об'єктні файли компілюються як 32-бітні, без стандартної бібліотеки мови С (адже використовується самописний функціонал частини стандартної бібліотеки С), з рівнем оптимізації коду -O2;
- перераховуються усі об'єктні файли (all), які необхідні для успішного лінкування;
- відбувається безпосереднє лінкування (команда ld).

Після цього перераховані команди ассемблерування та компілювання source-коду перетворюються в об'єктні файли.

Загалом, структуру збірки можна подати такою схемою (рис. 4):

Кожен source-файл з кодом на С та ассемблері компілюється/ассемблерується в об'єктні файли (у випадку з кодом початкового завантажувача — створюється бінарний файл). Об'єктні файли (модулі ядра, драйвери) у свою чергу за допомогою лінкера LD із набору GNU Binary Utilities збираються у файл ядра, параметри для лінкера описані в окремому скрипті. Далі відбувається безпосереднє створення образу системи, який містить у собі початковий завантажувач (розташовується на початку образу) та саме ядро ОС. Код програми створення образу написано на С, сама програма приймає параметри з окремого linux-скрипту, де вказано що первинний завантажувач

має міститись у перших 512 байтах, за ним вторинний і далі вже ядро. Образ системи має формат .img і може вже бути запущений на якійсь віртуальній машині або на реальному ПК.


Рисунок 4 – Етапи збірки мікроядра

Таким чином змонтована найсуттєвіша частина операційної системи, що керує системними ресурсами – ядро, яке працює у привілейованому режимі.

**Висновки та перспективи подальшого дослідження.** З використанням мов програмування ассемблера та С, засобів GCC та GNU Binary Utilities було змонтовано код, який містить необхідну мінімальну кількість функцій, даних і є найважливішою частиною для правильної реалізації операційної системи. Архітектура розробленого мікроядра невелика і ізольована, подальші ж дослідження можуть бути спрямовані на розробку великого програмного комплексу, що складатиметься з десятків модулів.

**Список бібліографічного опису**

1. Таненбаум Э. Современные операционные системы / Э. Таненбаум, П. Босх. – Питер, 2017. – 1120 с. – (4-е издание).
2. Лав Р. Ядро Linux: описание процесса разработки / Роберт Лав., 2019. – 496 с. – (3-е издание).
3. Назаров С. Операционные среды, системы и оболочки Основы структурной и функциональной организации / Сергей Назаров., 2007. – 504 с.
4. Дейтел Х. М. Операционные системы. Основы и принципы / Х. М. Дейтел, П. Д. Дейтел, Д. Р. Чофнес., 2016. – 382 с. – (3-е изд).
5. Рузайкин Г. Принципы проектирования и организации ОС [Електронний ресурс] / Геля Рузайкин // Открытые системы. СУБД, №6. – 1998. – Режим доступу до ресурсу: https://www.osp.ru/os/1998/06/179610/
6. Мартин Р. Идеальный программист. Как стать профессионалом разработки ПО / Роберт Мартин., 2019. – 224 с. – (Программирование).
7. Аблязов Р. З. Программирование на ассемблере на платформе x86-64 / Р. З. Аблязов. – Москва: ДМК Пресс, 2016. – 302 с. – (Профессиональное программирование).
8. Пильщиков В. Н. Программирование на языке ассемблера IBM PC / В. Н. Пильщиков., 2014. – 288 с. – (Диалог-МИФИ).
9. Эриксон Д. Хакинг. Искусство эксплойта / Джон Эриксон., 2014. – 496 с.
10. Максимов А. В. Оптимальное проектирование ассемблерных программ математических алгоритмов. Лабораторный практикум / Александр Викторович Максимов. – Санкт-Петербург: Лань, 2017. – 128 с. – (Учебники для вузов. Специальная литература).
11. Как писать на ассемблере в 2018 году [Електронний ресурс] – Режим доступу до ресурсу: https://habr.com/ru/post/345748/

**References**

1. Tanenbaum E. Modern operating systems / E. Tanenbaum, P. Bosch. - Peter, 2017 .-- 1120 s. - (4th edition).
2. Love R. The Linux kernel: a description of the development process / Robert Love., 2019. - 496 p. - (3rd edition).
3. Nazarov S. Operating environments, systems and shells Fundamentals of the structural and functional organization / Sergey Nazarov., 2007. - 504 p.
4. Deitel H. M. Operating Systems. Fundamentals and principles / Kh. M. Daytel, P.D. Daytel, D.R. Chofnes., 2016 .-- 382 p. - (3rd ed).
5. Ruzaykin G. Principles of design and organization of OS [Electronic resource] / Gelya Ruzaykin // Open Systems. DBMS, No. 6. - 1998. - Access mode to the resource: https://www.osp.ru/os/1998/06/179610/
6. Martin R. The ideal programmer. How to become a professional in software development / Robert Martin., 2019 .-- 224 p. - (Programming).
7. Ablyazov R.Z. Programming in assembler on the x86-64 platform / R.Z. Ablyazov. - Moscow: DMK Press, 2016 .-- 302 p. - (Professional programming).
8. Pilshchikov V. N. Programming in assembly language IBM PC / V. N. Pilshchikov., 2014. - 288 p. - (Dialogue-MIFI).
9. Erickson D. Hacking. The Art of Exploit / John Erickson., 2014 .-- 496 p.
10. Maksimov A. V. Optimal design of assembler programs of mathematical algorithms. Laboratory workshop / Alexander Viktorovich Maksimov. - St. Petersburg: Doe, 2017 .-- 128 p. - (Textbooks for universities. Special literature).
11. How to write in assembler in 2018 [Electronic resource] - Access to resource mode: https://habr.com/en/post/345748/