МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ
НАЦІОНАЛЬНИЙ ТЕХНІЧНИЙ УНІВЕРСИТЕТ УКРАЇНИ
«КИЇВСЬКИЙ ПОЛІТЕХНІЧНИЙ ІНСТИТУТ
імені ІГОРЯ СІКОРСЬКОГО»

Л. О. Левченко, Ю.А. Тарнавський

ОПЕРАЦІЙНІ СИСТЕМИ

Навчальний посібник

Рекомендовано Методичною радою КПІ ім. Ігоря Сікорського
як навчальний посібник для здобувачів ступеня бакалавра
за освітньо-професійною програмою «Цифрові технології в енергетиці»
спеціальності 122 «Комп’ютерні науки»

Електронне мережне навчальне видання

Київ
КПІ ім. Ігоря Сікорського
2023

УДК 004.451(075.8)
В 19

Автори: *Левченко Лариса Олексіївна*, д-р техн. наук, проф.
Тарнавський Юрій Адамович, канд. техн. наук, доц.

Рецензент: *Гагарін О.О.,* канд. техн. наук, доцент.,
кафедра інженерії програмного забезпечення в енергетиці

Відповідальний редактор: *Сидоренко Ю.В.,* канд. техн. наук, доцент

Гриф надано Методичною радою КПІ ім. Ігоря Сікорського
(протокол № 1 від 07.09.2023 р.)
за поданням Вченої ради Навчально-наукового інституту атомної та теплової енергетики
(протокол № 14 від 29.06.2023 р.)

Левченко Л.О.

В 19 Операційні системи [Електронний ресурс] : навч. посіб. для здобувачів ступеня бакалавра за освіт. програмою «Цифрові технології в енергетиці» спец. 122 «Комп'ютерні науки» / Л. О. Левченко, Ю. А. Тарнавський ; КПІ ім. Ігоря Сікорського. – Електрон. текст. дані (1 файл. – Київ : КПІ ім. Ігоря Сікорського, 2023. – 256 с.

У навчальному посібнику викладено теоретичні основи, методології та принципи побудови сучасних операційних систем, що дозволяє набути практичних навичок основ побудови, функціонування та використання засобів операційних систем для реалізації завдань при розробці та експлуатації інформаційних систем, в тому числі і в галузі енергетики.

Посібник забезпечує студентів необхідними теоретичними, методичними знаннями для опанування відповідної теми силабусу та виконання практичних завдань, запланованих впродовж семестру.

Призначений для здобувачів ступеня бакалавр, які навчаються за спеціальністю 122 «Комп'ютерні науки» освітньо-професійної програми «Цифрові технології в енергетиці». Наведений матеріал є корисним для студентів інших споріднених спеціальностей.

УДК 004.451(075.8)

Реєстр. № НП 23/24-015 Обсяг 12,9 авт. арк.

Національний технічний університет України
«Київський політехнічний інститут імені Ігоря Сікорського»
проспект Берестейський, 37, м. Київ, 03056
https://kpi.ua
Свідоцтво про внесення до Державного реєстру видавців, виготовлювачів
і розповсюджувачів видавничої продукції ДК № 5354 від 25.05.2017 р.

Л. О. Левченко, Ю. А. Тарнавський, 2023
© КПІ ім. Ігоря Сікорського, 2023

ЗМІСТ

ВСТУП…………………………………………………………………. 3

Тема 1. Призначення, функції операційних систем………………………. 4

Лекція 1. Операційні системи: основні визначення та поняття…………… 4

Лекція 2. Функціональні компоненти ОС, архітектура……………………. 22

Тема 2. Підсистема управління оперативною пам'яттю………………… 38

Лекція 3. Підсистема управління оперативною пам'яттю………………… 38

Тема 3. Підсистема організації задач, процеси, потоки………………….. 54

Лекція 4. Керування процесами і потоками………………………………... 54

Лекція 5. Структури процесів і потоків Windows, API функції…………… 66

Тема 4. Регістри процесора……………………………………………… 79

Лекція 6. Регістри процесора………………………………………………... 79

Тема 5. Засоби компіляції та компонування……………………………. 87

Лекція 7. Асемблери (призначення, основні поняття). Базові засоби розробки системних програм…………………………………………….. 87

Тема 6. Типи даних та засоби адресації до них………………………… 108

Лекція 8. Типи даних, засоби адресації до них…………………………… 108

Тема 7. Основні команди мови Асемблер………………………………. 124

Лекція 9. Команди пересилання даних та обміну даними………………… 124

Лекція 10. Арифметичні операції…………………………………………… 132

Лекція 11. Команди передачі управління………………………………….. 143

Лекція 12. Організація циклів……………………………………………… 152

Лекція 13. Команди для роботи з рядковими даними, логічними операціями, зсувами……………………………………………………………. 159

Тема 8. Макрозасоби мови Асемблер……………………………………. 169

Лекція 14. Макрозасоби мови Асемблер………………………………….. 169

Лекція 15. Резидентні програми…………………………………………….. 180

Тема 9. Підсистема управління зовнішніми пристроями………………. 196

Лекція 16. Завдання і організація підсистеми введення-виведення……… 196

Тема 10. Файлові системи……………………………………………………….. 224
Лекція 17. Робота з файлами………………………………………………… 224
Тема 11. Мережні операційні системи…………………………………………... 238
Лекція 18. Мережні операційні системи…………………………………… 238
СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ…………………………………… 255

ВСТУП

Навчальна дисципліна «*Операційні системи*» входить до циклу професійної підготовки навчального плану бакалаврів спеціальності 122 «Комп'ютерні науки» освітньо-професійної програми «Цифрові технології в енергетиці».

Дисципліна складеться з одного кредитного модуля та призначена для викладання студентам першого курсу у другому семестрі.

Навчальний посібник відповідає силабусу навчальної дисципліни «Операційні системи». Відповідно до силабусу передбачено вивчення тем, що дозволяють сформувати у студентів сучасні знання щодо теоретичних основ, принципів побудови, архітектури сучасних операційних систем.

Наведено огляд сучасних операційних систем.

Розглянуто ключові підсистеми операційних систем, а саме, підсистема управління оперативною пам'яттю, підсистема організації задач, керування процесами і потоками, управління файловою системою, управління засобами введення-виведення, мережними засобами, засобами безпеки.

Наведено особливості організації сімейства операційних систем Windows, розглядається специфіка використання системних викликів функцій Win32, необхідні прикладним програмам для доступу до засобів операційних систем.

Розглянуто особливості функціонування операційних систем розподілених комп'ютерних систем, а також особливості операційних систем для хмарних обчислень.

В якості мови програмування, як засобу розробки системних прикладних програм, запропоновано Ассемлер, що дозволяє зрозуміти роботу комп'ютера та створювати програми, які потребують максимальної швидкості виконання.

Метою дисципліни «Операційні системи» є оволодіння студентами теоретичних знань та практичних навичок основ побудови, функціонування, використання засобів операційних систем для реалізації завдань при розробці та експлуатації інформаційних систем.

Предмет дисципліни – вивчення принципів побудови, архітектури, основних функцій, режимів роботи, засобів операційних систем (ОС), які забезпечують функціонування сучасних комп'ютерів, інформаційних та обчислювальних систем різного призначення.

Тема 1. Призначення, функції операційних систем

ЛЕКЦІЯ 1. Операційні системи: основні визначення та поняття

Призначення, основні функції ОС. Історія розвитку операційних систем. Режими роботи. Класифікація ОС. Особливості функціонування ОС розподілених комп'ютерних систем. Особливості ОС для хмарних обчислень. Основні компоненти операційної системи. Основні функції ядра. Типи архітектур ядер операційних систем.

Навіщо потрібні знання операційних систем? Знання операційних систем є основою успішної кар'єри в сфері програмування. Предмет ОС поєднує в собі як математичні методи, так і методи проектування сучасного програмного забезпечення, які використовуються в багатьох інших сучасних областях - при розробці ігор, клієнт-серверних застосунків, бізнес- застосунків, Web-технологій і програмних інструментів.

Знання ОС сприяє становленню зрілого мислення програміста, гарному знанню мережевих технологій і протоколів, віртуальних машин, методів сучасного програмування.

Програмісту потрібні прості високорівневі абстракції, які приховують роботу апаратного забезпечення та надають можливість працювати з відповідними об'єктами за допомогою програмно-апаратного інтерфейсу, наприклад, робота з файловою системою (при роботі з диском типовою абстракцією є файл). Тобто програмісту не треба безпосередньо працювати з основним обладнанням, операційна система надає і програмісту, і користувачу ряд засобів, які використовують програми за допомогою спеціальних команд. Такі можливості і надає операційна система. Існує багато точок зору на те, що таке операційна система. Неможливо дати їй адекватне суворе визначення, тому що вона виконує дуже багато функцій.

Поняття операційної системи, мета її роботи

Операційна система (ОС, в англомовному варіанті – ***operating system***) – базове системне програмне забезпечення, яке керує роботою комп'ютера і є посередником (інтерфейсом) між апаратурою (**hardware**), прикладним програмним забезпеченням (**application software**) і користувачем комп'ютера (**user**) [1].

Призначення операційної системи

Призначення ОС – зручність використання комп'ютерної системи, ефективність та надійність роботи.

Основні цілі роботи операційної системи наступні.

1. Забезпечення зручності, ефективності, надійності, безпеки виконання програм користувача. Для користувача найголовніше - щоб його програма працювала, вела себе передбачувано, видавала необхідні йому правильні результати, не давала збоїв, що не піддавалася зовнішнім атакам. Обчислювальне середовище для такого виконання програм і забезпечує операційна система.

2. Забезпечення зручності, ефективності, надійності, безпеки використання комп'ютера. Операційна система забезпечує максимальну корисність та ефективність використання комп'ютера і його ресурсів, обробляє переривання, захищає комп'ютер від збоїв, відмов і хакерських атак. Ця діяльність ОС може бути не настільки помітною для користувача, але вона здійснюється постійно.

3. Забезпечення зручності, ефективності, надійності, безпеки використання мережевих, дискових та інших зовнішніх пристроїв, підключених до комп'ютера. Особлива функція операційної системи, без якої неможливо використовувати комп'ютер, - це робота з зовнішніми пристроями. Наприклад, ОС обробляє будь-яке звернення до жорсткого диска, забезпечуючи роботу відповідного драйвера (низькорівневої програми для обміну інформацією з диском) і контролера (спеціалізованого процесора, що виконує команди введення-виведення з диском). Будь "флешка", вставлена в USB-слот комп'ютера, розпізнається операційною системою, отримує своє логічне ім'я (в системі Windows - у вигляді літери, наприклад, G) і стає частиною файлової системи комп'ютера на весь час, поки вона не буде витягнута (демонтована).

4. Підкреслимо особливу важливість серед функцій сучасних ОС забезпечення безпеки, надійності і захисту даних. Слід враховувати, що комп'ютер і операційна система працюють в мережевому оточенні, в якому постійно можливі і фактично відбуваються атаки хакерів і їх програм, що ставлять своєю метою порушення роботи комп'ютера, "злом" конфіденційних даних користувача, що зберігаються на ньому, викрадення логінів, паролів, використання комп'ютера як "робота" для розсилки реклам або вірусів та ін. У зв'язку з цим в 2002 р фірма Microsoft оголосила ініціативу щодо надійних і безпечних обчислень (trustworthy computing initiative), метою якої є підвищення надійності та безпеки всього програмного забезпечення, насамперед - операційних систем.

Основні функції ОС:

1). ОС розглядається як інтерфейс (взаємодія) між користувачем та комп'ютером. Таким чином, для користувача ОС виступає у ролі *віртуальної машини*, яка його вивільняє від безпосередньої роботи з устаткуванням. ОС приховує інтерфейс апаратного забезпечення і пропонує програмісту інтерфейс прикладного програмування, яке використовує поняття вищого рівня (абстракції), рис. 1.1.

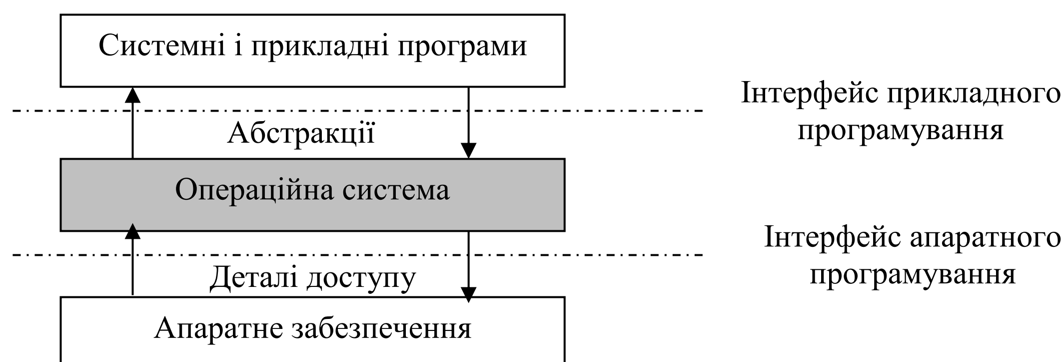


Рис. 1.1. Взаємодія ОС з апаратним забезпеченням і застосунками

Виділення абстракцій дозволяє досягти того, що код ОС і прикладних програм не потребує зміни при переході на нове апаратне забезпечення. Тобто ОС надають *апаратно-незалежне середовище* для виконання прикладних програм.

2). Операційна система виступає у ролі *розподілювача ресурсів*. Під ресурсами розуміють процесорний час, дисковий простір, пам'ять, засоби доступу до зовнішніх пристроїв. ОС виконує роль менеджера цих ресурсів і надає їх прикладним програмам за потребою.

Розрізняють два види розподілу ресурсів:

а) п*росторовий розподіл* – ресурс доступний декільком споживачам одночасно, однак кожний з них може використовувати частину ресурсу. Так поділяється пам'ять;

б) *часовий розподіл* – система ставить споживача у чергу і дає можливість користуватися всім ресурсом обмежений час. Так розподіляється процесор в однопроцесорних системах.

При розподілі ресурсів ОС розв'язує конфліктні ситуації.

3). При одночасній роботі декількох користувачів виникає проблема організації їх безпечної діяльності (збереження інформації на диску, заборона видалення або ушкодження файлів, заборона втручатися в роботу програм інших користувачів). ОС виступає як *захисник користувача та програмного забезпечення.*

4). Операційна система є *постійно функціонуючим ядром*. В багатьох сучасних операційних *системах постійно* працює на комп'ютері лише частина операційної *системи,* яку прийнято називати її ядром.

Таким чином, **операційна система** – це набір програмних і апаратних засобів, призначених для вирішення наступних задач:

1) забезпечення інтерфейсу користувача як розширеної віртуальної машини, а саме: наявності набору команд, меню і графічного інтерфейсу (оболонки, яка організує взаємодію користувача з ОС за допомого. Вікон, значків, меню, миші);
2) раціонального розподілу ресурсів.

Історія розвитку операційних систем (однопрограмні, мультипрограмні, багатокористувацькі, мережні)

1. Режими роботи (пакетні, інтерактивні, системи розподілу часу, системи реального часу)

Перші операційні системи з'явилися в 50-ті роки (1955 – 1965 р.р.). Це були *однопрограмні* ОС і працювали в режимі *пакетної обробки.* Такі системи забезпечували автоматичне послідовне виконання програм (це були завдання - колоди перфокарт декількох користувачів) за допомогою формалізованої мови виконання завдань у пакетному режимі (без можливості взаємодії з користувачем). У певний момент часу *в пам'яті могла перебувати тільки одна програма (системи були однозадачними),* усі програми виконувалися на процесорі від початку до кінця. Системи пакетної обробки стали прообразом сучасних операційних систем, призначених для управління обчислювальним процесом. За такої ситуації ОС

розглядали просто як набір стандартних служб, необхідних прикладним програмам і користувачам.

Другим етапом (1965 – 1970 р.р.) стала *підтримка багатозадачності – мультипрограмні ОС*. Обчислювальний процес був організований на одному центральному процесорові, це так званий *Spooling* – вивантаження та підкачка інформації на зовнішніх носіях та організація черг.

У багатозадачних системах у пам'ять комп'ютера стали завантажувати декілька програм, які виконувалися на процесорі навперемінно. При цьому розвивалися два напрями: *багатозадачна пакетна обробка* і *розподіл часу.* У багатозадачній пакетній обробці завантажені програми, як і раніше, виконувалися в пакетному режимі. *У режимі розподілу часу* із системою могли працювати одночасно кілька користувачів, кожному з яких надавався діалоговий термінал (пристрій, що складається із клавіатури і дисплея).

Для підвищення ефективності використання процесора була використана *ідея мультипрограмування*, яка полягала у наступному: поки одна програма виконує операцію ведення-виведення, процесор не простоює, а виконує іншу програму, по закінченні операції введення-виведення процесор повертається до виконання першої програми. Поява мультипрограмування забезпечила:

- реалізацію захисних механізмів (поява привілейованих і непривілейованих команд для доступу до ресурсів системи, привілейовані – це команди введення-виведення, які виконує виключно ОС; захист пам'яті);
- наявність переривань (зовнішніх – закінчення операції введення-виведення і внутрішніх – ділення на нуль, спроба порушення захисту);
- розвиток паралелізму в архітектурі (прямий доступ пам'яті та організація каналів введення-виведення);
- організацію інтерфейса між прикладною програмою та ОС за допомогою системних викликів;
- планування використання процесора;
- розроблення стратегії управління пам'яттю;
- розроблення файлової системи та доступ до конкретного файлу тільки певної категорії користувачів;
- розроблення засобів комунікації.

Логічним розширенням систем мультипрограмування стали *системи розподілу часу.* В таких системах процесор перемикався між задачами не тільки на час виконання введення-виведення, а і просто через певний час. Ці перемикання здійснюються так часто, що користувачі можуть взаємодіяти зі своїми програмами під час їх виконання, тобто *працювати в інтерактивному режимі*. Тобто з'являється можливість одночасної роботи декількох користувачів на одній комп'ютерній системі. Для зменшення кількості працюючих користувачів була реалізована ідея часткового знаходження виконуваної програми в оперативній пам'яті. Основна частина програми знаходиться на диску, а фрагмент, який потрібно виконати в даний момент, завантажується в оперативну пам'ять, непотрібний фрагмент знову викачується на диск. Це реалізується за *допомогою*

механізму віртуальної пам'яті. В системах розподілу часу користувач отримав можливість в інтерактивному режимі ефективно проводити відладку програми і записувати інформацію на диск, що призвело до необхідності розроблення розвинутих файлових систем.

Третім етапом (1970 – 1980 р.р.) розвитку ОС є *поява сімейств програмно сумісних машин, що працюють під управлінням однієї і тієї ж операційної системи*. Це машини серії IBM / 360, потім PDP. Це були складні та громіздкі системи, написані тисячами програмістів, які містили мільйоні рядків асемблера.

Четвертий етап (1980 – 2008 р.р.) операційних систем пов'язаний з появою персональних комп'ютерів, що призвело до появи *мережевих або розподілених операційних систем.*

В *мережевих операційних системах* користувачі можуть дістати доступ до ресурсів іншого мережевого комп'ютера, тільки вони повинні знати про їх наявність і вміти це зробити. Кожна машина в мережі працює під управлінням своєї локальної операційної *системи,* що відрізняється від операційної *системи* автономного комп'ютера наявністю додаткових засобів (програмною підтримкою для мережевих інтерфейсних пристроїв і доступу до віддалених ресурсів), але ці доповнення не міняють структуру операційної *системи.*

Розподілена система, навпаки, зовні виглядає як звичайна автономна система. Користувач не знає і не повинен знати, де його файли зберігаються – на локальній або віддаленій машині – і де його програми виконуються. Він може взагалі не знать, чи підключений його комп'ютер до мережі. Внутрішня будова *розподіленої операційної системи* має істотні відмінності від автономних систем. Автономні *операційні системи* називаються *класичними операційними системами*.

Класичні *операційні системи* в процесі еволюції виконували **шість основних функцій**:

- планування завдань і використання процесора;
- забезпечення програм засобами комунікації і синхронізації;
- управління пам'яттю;
- управління файловою системою;
- управління введенням-виведенням;
- забезпечення безпеки.

2. Сучасні операційні системи (розквіт у 2000-х рр.)

В даний час ми є свідками небувалого розквіту операційних систем, тому для їх вивчення для студентів відкриваються величезні можливості: випускаються нові ОС для настільних комп'ютерів, кластерів комп'ютерів і паралельних обчислень, мобільних пристроїв, хмарних обчислень.

Безперечним лідером у цій галузі є корпорація Microsoft, яка випустила менш ніж за 10 недавніх років цілу серію *ОС сімейства Windows: Windows XP, Windows 2003, Windows 7, Windows Vista (2007), Windows 2008, Windows 2008 High-*

Performance Computing (HPC), Windows 2010. Найбільш вдалими версіями були *Windows XP* та *Windows 7,* але на даний час припинена їх підтримка [4, 7, 15].

В 2006 р відбулося неймовірна у програмістському світі подія - *Microsoft* відкрила "святу святих", вихідний код ядра ОС Windows сімейства NT (NT/2000/XP/2003/2008/7) і надала в розпорядження університетів та академічних організацій Windows Research Kernel (WRK) - самодокументований вихідний код "дослідницького" ядра Windows. Тепер кожен студент, викладач і дослідник мають можливість вивчати систему Windows "зсередини" і навіть розвивати її, але тільки для цілей навчання і досліджень, а не для комерції.

Розвиваються також діалекти ***ОС Linux (Red Hat, Fedora, Mandrake, Ubuntu, SuSE*** та інші сотні діалектів). Linux - операційна система типу UNIX, ядро якої вільно поширюється з вихідними кодами.

Розвиток Unix-подібних систем відбувається лавиноподібно. Наприклад, компанія HP розробила операційну систему HP-UX. Операційна система Unix поширювалась безкоштовно, оскільки компанія AT & T не мала права займатися комерційною діяльністю. Одна з версій потрапила в Каліфорнійський університет Берклі. Дослідницькою групою комп’ютерних систем (Computer Systems Research Group, скорочено CSRG) Каліфорнійського університету в Берклі 1993 році була створена операційна система FreeBSD (Berkeley Software Distribution), яка базується на ОС BSD версії 4.4BSD-Lite.

FreeBSD — це клон операційної системи UNIX для персональних комп’ютерів, що базуються на архітектурі процесорів Intel (386SX/386DX/486SX/486DX/Pentium/Pentium Pro/…).

FreeBSD працює також на процесорах AMD та VIA, сумісних з Intel, а також на процесорах DEC Alpha. FreeBSD надає широкий набір функцій, які раніше були доступні тільки на більш дорогих комп'ютерах.

Група FreeBSD Project домоглася максимальної продуктивності й надійності системи в ситуаціях реального життя для персональних комп’ютерів. FreeBSD використовувалася для створення багатьох проектів, серед яких: Yahoo!, Hotmail, Apache, Be, Inc., Blue Mountain Arts, Pair Networks, Whistle Communications, *BSDi* і безліч інших.

Фірма *Sun* (у 2010 р увійшла до складу фірми Oracle) розробляє і випускає *ОС Solaris* - одну з найбільш сучасних ОС типу UNIX з розвиненою підтримкою паралельного програмування, новими видами файлових систем, що відрізняється своєю підвищеною надійністю.

Це лише деякі ОС. Існує також багато інших операційних систем. У США і Канаді, як відомо, вельми популярні комп'ютери сімейства Macintosh фірми Apple (Mac) зі своєю операційною системою MacOS, яка є законодавцем мод в області графічних користувацьких інтерфейсів (*GUI - graphical user interface*) та обміну мультимедійною інформацією (наприклад, мовного введення).

Назвемо також ОС фірми IBM для суперкомп'ютерів і комп'ютерів загального призначення (*mainframes*).

Особливо важливо для успішного вивчення операційних систем те, що в даний час багато з них (або їх великі частини, наприклад, ядро) доступні з відкритим вихідним кодом.

За традицією, ще з початку 1990-х рр., ядро ОС Linux також вільно поширюється, з вихідними кодами, що викликало цілу хвилю робіт зі створення нових діалектів Linux, а також з розробки нових ОС для мобільних пристроїв на базі ядра Linux (наприклад, ОС *Google Android*).

Також інтенсивно розвиваються ***ОС для мобільних пристроїв***. Ще кілька років тому найбільш використовуваними ОС в цій області були ОС сімейства *Symbian*. Однак зараз ОС Microsoft Windows Mobile і Google Android активно тіснять Symbian з ринку.

ОС для хмарних обчислень (Web OS) - принципово новий вид ОС, що відображає сучасну тенденцію до організації обчислень як хмарних (*cloud computing*). Хмара - це метафора Інтернету. При хмарних обчисленнях користувач зі свого комп'ютера одержує платний доступ через Інтернет до Web-сервісів, що працюють на комп'ютерах потужних центрів обробки даних (наприклад, на серверах Microsoft). При цьому не тільки використовуване програмне забезпечення (у вигляді набору Web-сервісів), але і самі оброблювані дані користувача зберігаються на серверах "хмарного" центру обробки даних. На своєму комп'ютері користувач має лише простий і зручний і не вимагає великих ресурсів "хмарний" Web-інтерфейс.

Знайомство з хмарними технологіями у користувачів найчастіше пов'язане з відомими файлосховищами - *Dropbox, SkyDrive, Google Drive, Mega, Box*. Але коли необхідно розширити межі присутності в «хмарі» варто вдатися до використання хмарних операційних систем (*Cloud PC*) [9, 10].

Найбільш поширеною ОС для хмарних обчислень є в даний час *Microsoft Windows Azure, Zero PC, CloudMe і CloudTop*.

Великі фірми відкривають вихідні коди своїх операційних систем, залучаючи молодих талановитих фахівців цікавими проектами ОС з відкритим вихідним кодом, так як їм необхідні молоді програмісти і нові цікаві ідеї, які дозволять зробити ОС ще більш потужними, масштабованими, зручними, ефективними, надійними і безпечними [11].

Класифікація операційних систем

В залежності від області застосування ОС сучасні операційні системи поділяються на [1]:

- ***ОС великих ЕОМ (мейнфреймів).*** Такі комп'ютерні системи використовують для надійної обробки значних обсягів даних, при цьому ОС має ефективно підтримувати цю обробку (в пакетному режимі або в режимі розподілу часу). Прикладом ОС такого класу може бути OS/390 фірми IBM; поширених в 1950-х - 1970-х рр. для комп'ютерів: *IBM 360/370*; з вітчизняних - М-220, *БЕСМ-6.* Ці комп'ютери були розроблені в Інституті кібернетики АН УРСР під керівництвом академіка Сергія Олексійовича Лебедєва (1902 - 1974). Це перші комп'ютери, які були створені в континентальній Європі, які зберігають програму у пам’яті комп’ютера. На таких комп'ютерах вирішувалися всі необхідні завдання - від розрахунку зарплати співробітників в організації до розрахунку траєкторій космічних ракет.

• ***ОС суперкомп'ютерів (super-computers)***. Суперкомп'ютери - це потужні багатопроцесорні комп'ютери, найсучасніші з яких мають продуктивність до кількох *petaflops* (1015 речових операцій в секунду; абревіатура *flops* розшифровується як *floating-point operations per second*). Прикладом таких ОС є ОС AIX, UNICOS, SGI IRIX. Суперкомп'ютери використовуються для обчислень, що вимагають великих обчислювальних потужностей, надвисокої продуктивності і великого обсягу пам'яті. У реальній практиці це насамперед задачі моделювання - наприклад, моделювання клімату в регіоні та прогнозування на основі побудованої моделі погоди в даному регіоні на найближчі дні.

Президент США Барак Обама 30 липня 2015 року підписав урядове розпорядження, що вимагає створення до 2025 року найшвидшого суперкомп'ютера у світі. Раніше в США була прийнята науково-дослідницька програма «Стратегічна комп'ютерна ініціатива» (NSCI), завдання якої зводиться до підтримки розвитку нових комп'ютерних технологій та розробки нових ексафлопсних обчислювальних систем, які дозволять країні гідно виступати в високотехнологічної гонці озброєнь.

Комп'ютер буде в 20 разів швидше нинішнього лідера, побудованого в Китаї (Китайським національним університетом оборонних технологій суперкомп'ютер Тяньхе-2 займає перший рядок найпотужніших суперкомп'ютерів, який ось вже два з половиною роки поспіль, що не може не дратувати американських фахівців), і зможе виконувати квінтильйон (мільярд мільярдів - 1 і 18 нулів) операцій в секунду, що відповідає продуктивності в один екзафлопс.

Новий суперкомп'ютер використовуватиметься для наукових розрахунків і проектів у галузі національної безпеки, численних медичних дослідженнях, в першу чергу будуть внесені у комп'ютер усі напрацювання науковців світу щодо вирішення проблеми раку.

Особливістю суперкомп'ютерів є їх паралельна архітектура - як правило, всі вони є багатопроцесорними. Відповідно, ОС для суперкомп'ютерів повинні підтримувати розпаралелювання вирішення завдань і синхронізацію паралельних процесів, які одночасно вирішують підзадачі деякої програми.

• ***ОС кластерних систем.*** Кластери комп'ютерів (computer clusters) - групи комп'ютерів, фізично розташовані поруч і з'єднані один з одним високошвидкісними шинами і лініями зв'язку. Ці комп'ютери з точки зору користувача являють собою єдиний апаратний обчислювальний ресурс.

Для створення кластерів зазвичай використовуються або прості однопроцесорні персональні комп'ютери, або двох- або чотирьох- процесорні SMP-сервери (*Symmetric Multiprocessing*). При цьому не накладається ніяких обмежень на склад і архітектуру вузлів. Кожний з вузлів може функціонувати під керуванням своєї власної операційної системи. Найчастіше використовуються стандартні ОС: Linux, FreeBSD, Solaris, Tru64 Unix, Windows NT. У тих випадках, коли вузли кластера неоднорідні, то говорять про гетерогенні кластери.

Кластери комп'ютерів використовуються для високопродуктивних паралельних обчислень. Найбільш відомі в світі комп'ютерні кластери, розташовані в дослідницькому центрі CERN (Швейцарія) - тому самому, де знаходиться великий адронний коллайдер. Операційна система для кластерів повинна, крім

загальних можливостей, надавати засоби для конфігурації (вибір функціональних блоків системи, їх розташування, установка взаємозв'язків між ними) кластера, управління комп'ютерами (процесорами), що входять до нього, розпаралелювання вирішення завдань між комп'ютерами кластера та моніторингу кластерної комп'ютерної системи. Прикладами таких ОС є ОС фірми Microsoft - Windows 2003 for clusters; Windows 2008 High-Performance Computing (HPC).

• ***Серверні ОС***. Головна характеристика таких ОС — здатність обслуговувати велику кількість запитів користувачів до спільно використовуваних ресурсів. Нині для реалізації серверів частіше застосовують універсальні ОС (UNIX або системи лінії Windows XP);

• ***Персональні ОС.*** Наймасовіша категорія. Особлива увага в персональних ОС приділяється підтримці графічного інтерфейсу користувача і мультимедіа-технологій.

Настільні комп'ютери (desktops) - це найбільш поширені в даний час комп'ютери. Параметри сучасного настільного комп'ютера, найбільш прийнятні для використання сучасних ОС: швидкодія процесора 1 - 3 ГГц, оперативна пам'ять – 1 - 16 Гбт і більше, обсяг жорсткого диска (hard disk drive - HDD) - 250 Гб - 1 Тб і більше (1 терабайт, Тб = 1024 Гб). Все розмаїття сучасних операційних систем (Windows, Linux та ін.) - до послуг користувачів настільних комп'ютерів. При необхідності на настільному комп'ютері можна встановити дві або більше операційних системи, розділивши його дискову пам'ять на кілька розділів (partitions) і встановивши на кожен з них свою операційну систему, так що при включенні комп'ютера користувачеві надається стартове меню, з якого він обирає потрібну операційну систему для завантаження.

Портативні комп'ютери (laptops, notebooks) використовуються ті ж операційні системи, що і для настільних комп'ютерів (наприклад, Windows або MacOS). Характерними рисами портативних комп'ютерів є всілякі вбудовані порти та адаптери для бездротового зв'язку: Wi-Fi (офіційно IEEE 802.11) - вид радіозв'язку, що дозволяє працювати в бездротовій мережі з продуктивністю 10-100 мегабіт в секунду (використовується зазвичай на конференціях, в готелях, на вокзалах , аеропортах - тобто в зоні радіусом у кілька сотень метрів від джерела прийому-передачі); Bluetooth - також радіозв'язок на більш коротких відстанях (10 - 100 м для Bluetooth 3.0), використовувана для взаємодії комп'ютера з мобільним телефоном, навушниками, плейером та ін. *Зовнішні пристрої* (додаткові жорсткі диски HDD, SSD - твердотільні накопичувачі, принтери,) підключаються до ноутбука через порти USB. *Інша характерна риса ноутбуків - це наявність кард-рідерів* - портів для читання всіляких карт пам'яті, використовуваних в мобільних телефонах або цифрових фотокамерах; забезпечується також інтерфейс FireWire (офіційно - IEEE 1394) для підключення цифрової відеокамери; таким чином, ноутбуки добре пристосовані для введення, обробки та відтворення обробки мультимедійної інформації. Популярна різновид ноутбука нині - це **нетбук** - ноутбук, призначений для роботи в мережі, зазвичай менш потужний і тому більш дешевий, а також більш мініатюрний.

• ***ОС розподілених систем*** (distributed system). Розподілені системи – це набір незалежних комп’ютерів, поєднаних між собою каналами зв’язку у мережу, які з

точки зору користувача виглядають як єдине ціле. Обчислення у такій системі розподілені між кількома фізичними процесорами за допомогою методів виявлення, доступу і взаємодії ресурсів . Про їх особливості поговоримо окремо.

• ***ОС мобільних пристроїв*** (mobile intelligent devices - мобільні телефони, комунікатори). Операційні системи для мобільних пристроїв відрізняються більшою компактністю, забезпечують кращий користувальницький інтерфейс ОС. Прикладом таких ОС є Google Android і Microsoft Windows Mobile.

• ***ОС реального часу (real-time systems).*** Системи реального часу - обчислювальні системи, призначені для управління різними технічними, військовими та іншими об'єктами в режимі реального часу. Наприклад, можуть керувати польотом космічного корабля, технологічним процесом або демонстрацією відеороликів. Це системи QNX, VxWorks;

Характеризуються основною вимогою до апаратури та програмного забезпечення, у тому числі до операційної системи: неприпустимість перевищення часу відповіді системи, тобто очікуваного часу виконання типової операції системи. Для ОС вимоги реального часу накладають досить жорстких обмеження - наприклад, в основному циклі роботи системи неприпустимі переривання (бо вони призводять до неприпустимих тимчасових витратах на їх обробку).

• ***Вбудовані ОС.*** До них належать керуючі програми для різноманітних мікропроцесорних систем, які використовують у військовій техніці, системах побутової електроніки, смарт-картах та інших пристроях. До таких систем ставлять особливі вимоги: розміщення в малому обсязі пам'яті, підтримка спеціалізованих засобів введення-виведення, можливість прошивання в постійному запам'ятовуючому пристрої;

• ***Багатопроцесорні ОС***. Багатопроцесорна обробка реалізована в таких *ОС*, як Linux, Solaris, Windows NT та ін.

Особливості функціонування ОС розподілених комп'ютерних систем

У розподіленій системі (*distributed system*) обчислення розподілені між кількома фізичними процесорами (комп'ютерами), об'єднаними між собою в мережу [1, 11].

Слабо пов'язана система (*loosely coupled system*) - розподілена комп'ютерна система, в якій кожен процесор має свою локальну пам'ять, а різні процесори взаємодіють між собою через лінії зв'язку - високошвидкісні шини, телефонні лінії, бездротовий зв'язок (Wi-Fi, EVDO, Wi-Max та ін.).

Переваги розподілених систем:

1. *Поділ (спільне використання) ресурсів*: в розподіленій системі різні ресурси можуть зберігатися на різних комп'ютерах. Немає необхідності дублювати програми або дані, зберігаючи їх копії на декількох комп'ютерах.

2. *Спільне завантаження (load sharing):* кожному комп'ютеру в розподіленій системі може бути доручено певне завдання, яке він виконує паралельно з виконанням іншими комп'ютерами своїх завдань.

3. *Надійність:* при відмові або збої одного з комп'ютерів розподіленої системи його завдання може бути перерозподілено іншому комп'ютеру, щоб збій у мінімальній мірі вплинув або зовсім не вплинув на підсумковий результат.

4. *Зв'язок*: в розподіленій системі всі комп'ютери пов'язані один з одним, так що, наприклад, при необхідності можливий віддалений вхід з одного комп'ютера на інший з метою використання ресурсів більш потужного комп'ютера.

У розподіленій системі комп'ютери пов'язані у мережеву інфраструктуру, яка може бути:

- локальною мережею (*local area network* - LAN);
- глобальною або регіональною мережею (*wide area network* - WAN).

По своїй організації розподілені системи можуть бути клієнт-серверними (client-server) або однoранговими (peer-to-peer) системами. В клієнт-серверній системі певні комп'ютери відіграють роль серверів, а інші - роль клієнтів, що користуються їх послугами. Подібна організація розподілених систем найбільш поширена. В тимчасовій розподіленій системі всі комп'ютери рівноправні. Структура клієнт-серверної системи зображена на рис. 1.2.

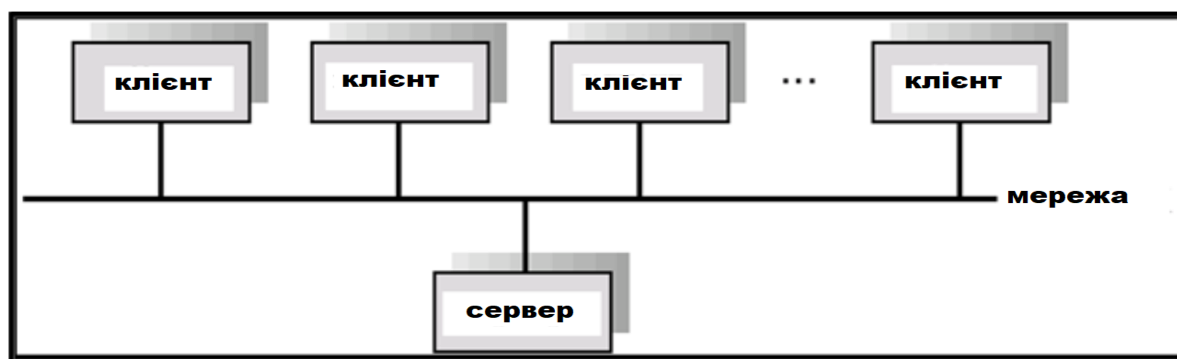


Рис. 1.2. Структура клієнт-серверної системи

Види серверів в клієнт-серверних комп'ютерних системах

Клієнт-серверна архітектура розподілених систем вельми широко поширена і підтримана операційними системами. Тому дуже важливо знати, які види і функції серверів пропонують сучасні розподілені системи.

Файл-сервер (*file server*) - комп'ютер і програмне забезпечення, що надають доступ до підмножини файлових систем, розташованих на дисках комп'ютера-сервера, інших комп'ютерів локальної мережі (LAN). Приклад - серверне програмне забезпечення SAMBA (SMB - скорочення від Server Message Block) для ОС типу UNIX (Linux, FreeBSD, Solaris і т.д.), що забезпечує доступ з Windows-комп'ютерів локальної мережі до файлових систем UNIX-машин. Samba також реалізована для платформи Macintosh / MacOS.

Тобто SAMBA надає доступ до загальних ресурсів в локальній мережі, наприклад, текстових файлів або мережевих принтерів. Для його розгортання на

Windows використовуються штатні засоби ОС, на Linux використовується файловий сервер Samba.

Необхідно налаштувати файловий сервера на серверах з операційними системами Linux і Windows. На сервері буде знаходитися 2 каталоги - публічний і приватний. До файлового сервера підключення буде відбуватися як з операційної системи Windows, так і з Linux, тому що в віртуальної або фізичної локальної мережі можуть знаходитися сервери з різними ОС.

Сервер застосунків (*application server*) - комп'ютер і програмне забезпечення, що надає обчислювальні ресурси (пам'ять і процесор) і необхідне оточення для віддаленого запуску певних класів (як правило, великих) застосунків з інших комп'ютерів локальної мережі. Приклади серверів застосунків - *SysElegance Application Server Basic Edition* - програмний продукт, який додає комп'ютеру з *ОС Windows 10 функціональність сервера терміналів* для централізованого виконання застосунків користувача.

WebSphere Application Server (IBM) надає комплект гнучких і безпечних середовищ виконання EE 7 (платформа Java Enterprise Edition - Java EE 7), доступних локально і в будь-яких приватних, загальнодоступних і гібридних хмарах. Він дозволяє скоротити витрати, отримати додаткову віддачу від вкладень в застосунки і прискорити вихід на ринок. Цей сервер забезпечує гібридні хмарні обчислення, модернізацію існуючих застосунків за допомогою архітектури мікросервісів та управління життєвим циклом API.

WebLogic Server (BEA) – найкращі з відомих серверів застосунків, що працюють в Java Enterprise Edition (JEE). WebLogic - це сімейство продуктів від компанії BEA Systems, яка з 2008 року належить корпорації Oracle. В платформу WebLogic Suite входять сервер застосунків Jakarta EE (Weblogic Server), портал, інтеграційні продукти, засоби для розробки застосунків і JRockit - власна Java Virtual Machine (JVM) компанії.

Сервер баз даних (*database server*) - комп'ютер і програмне забезпечення, що надає доступ іншим комп'ютерам мережі до баз даних, розташованим на комп'ютері-сервері. Приклад: серверне програмне забезпечення для доступу до баз даних *Microsoft SQL Server, Oracle SERVER (Oracle Corporation), IBM DB2, IBM Informix, Sybase.*

Веб-сервер (*Web server*) - комп'ютер і програмне забезпечення, що надає доступ клієнтам через WWW до Web-сторінок, розташованим на комп'ютері-сервері. Приклад: вільно поширюваний Web-сервер *Apache*, *IIS (Internet Information Services), Nginx.*

Проксі-сервер - комп'ютер і програмне забезпечення, що є частиною локальної мережі і підтримують ефективне звернення комп'ютерів локальної мережі до Інтернету, фільтрацію трафіку, захист від зовнішніх атак. *Proxy*-сервер зазвичай вбудований в операційну систему. Спочатку клієнт підключається до проксі-сервера і запитує будь-який ресурс (наприклад e-mail), розташований на іншому сервері. Потім проксі-сервер або підключається до вказаного серверу і отримує ресурс у нього, або повертає ресурс із власного кешу (у випадках, якщо проксі має свій кеш). В деяких випадках запит клієнта або відповідь сервера може бути змінений проксі-сервером в певних цілях. Проксі-сервер дозволяє захищати

комп'ютер клієнта від деяких мережевих атак і допомагає зберігати анонімність клієнта, але також може використовуватися шахраями для приховування адреси сайту, викритого в шахрайстві, зміни вмісту цільового сайту (підміна), а також перехоплення запитів самого користувача.

Сервер електронної пошти (e-mail server) - комп'ютер і програмне забезпечення, що виконують відправку, отримання і "розкладку" електронної пошти для комп'ютерів деякої локальної мережі. Можуть забезпечувати також криптування пошти (*email encryption*) - шифрування електронних листів перед відправкою адресатам з певного мережевого домену (як правило, замовнику) та їх дешифрування після отримання від замовника.

Серверний бек-енд (*Server back-end*) - група (пул), пов'язаних в локальну мережу серверних комп'ютерів, що використовуються замість одного сервера, з метою більшої надійності і надання більшого обсягу ресурсів. Інший термін, близький до цього, - центр обробки даних (data center). Ці поняття особливо актуальні у зв'язку з усе більш широким поширенням хмарних обчислень, що є, з цієї точки зору, найбільш сучасної реалізацією клієнт-серверної схеми взаємодії.

Особливості ОС для хмарних обчислень

Хмарні операційні системи - це революційний напрямок в розвитку ОС. Хмарні обчислення (**Cloud computing**) є одним з найбільш популярних напрямків розвитку IT.

Хмарні обчислення – це модель обчислень, заснована на динамічно масштабованих (**scalable**) і віртуалізованних ресурсах (даних, застосунках, ОС та ін.), які доступні і використовуються як сервіси через Інтернет і реалізуються за допомогою високопродуктивних центрів обробки даних (***data centers***) [9, 10].

Всі провідні компанії світу, які роблять свій бізнес на комп'ютерах і в інтернеті, активно розробляють свої хмарні операційні системи. Наприклад, компанія Google запустила в продаж цілий модельний ряд ноутбуків і нетбуків під управлінням операційної системи Google Chrome OS. Компанія Apple вже цієї осені запускає нову операційну систему Apple iOS 5, яка повністю буде «заточена» під «хмари».

Хмара – це дисковий простір, на якому ви можете зберігати, переглядати та оперувати зі своїми файлами, а фізично - це сервери, які розташовуються в різних країнах та континентах. На одному з таких серверів і знаходяться ваші файли.

В даний час всі великі компанії (*Microsoft, IBM, HP, Dell, Oracle, Sun* та ін.) розробляють свої системи хмарних обчислень; є тенденція до інтеграції цих корпоративних систем в єдине доступне користувачеві "хмара".

Відповідно до історії основних компаній-розробників хмарних технологій – Amazon, Google, Microsoft.

Переваги хмарних операційних систем перед стаціонарними (встановленими на комп'ютері) полягають в наступному:

1. Не потрібно встановлювати ніякого програмного забезпечення. Всю обробку можна проводити на сервісах в інтернеті. Тобто з будь-якого

комп’ютера/планшета/телефону необхідно тільки підключитися до Інтернету для отримання доступу до вашої особистої інформації. Це дозволяє прискорити процес, наприклад редагування відео, адже на сервісі обробка проводиться на такому потужному комп'ютері, який пересічний користувач навряд чи зможе собі дозволити. Після редагування завантажуєте результат до себе на комп'ютер або (якщо це можливо, а це в більшості випадків можливе) зберігайте його там же на сервері.

2. При використанні хмарних операційних систем відпадає необхідність зберігати у себе на комп'ютері будь-який або контент (фільми, музику і т.д.). Все це зберігається на сервісі.

3. Немає необхідності купувати дорогий потужний комп'ютер (ноутбук). Досить мати той, який дозволяє комфортно користуватися інтернетом, при чому робота з хмарними сервісами не залежить від операційної системи, встановленої на вашому комп’ютері.

4. Багато платних програм є безкоштовними (або дешевими) веб-застосунками.

Тобто хмарні технології надають користувачу ресурси та програмні засоби (Інтернет-сервіси). Хмарні технології, в першу чергу, - це відомі файлосховища - *Google Drive, Mega, Dropbox, SkyDrive, Box, OneDrive*. Користувачу необхідно мати доступ до глобальної мережі Інтернет по стабільному високошвидкісному каналу з широкою пропускною здатністю. І саме швидкість обробки інформації залежить виключно від високошвидкісному каналу.

Недолік хмарних обчислень полягає в тому, що користувач виявляється повністю залежним від використовуваної ним "хмари" (в якому доступні використовувані ним дані і програми) і не може управляти не тільки роботою "хмарних" комп'ютерів, але навіть резервним копіюванням своїх даних. У зв'язку з цим виникає цілий ряд важливих питань про безпеку хмарних обчислень, збереження конфіденційності даних користувача і т.д .; далеко не всі з них на даний момент вирішені.

Існує декілька віртуальних ОС. Найстаріша перша хмарна система – це **eyeOS.** Це open-source віртуальна надбудова над ОС, яка нараховує більше 70 застосунків. Наступна хмарна ОС – це **iCloud,** яка являє собою робочий простір, має набір всіляких застосунків на будь-які випадки у житті, а також забезпечує доступ до хмарного сховища та синхронізацію мобільних пристроїв з диском *iCloud*. Але цю систему викупила компанія Apple і тому вона доступна тільки власникам продукції компанії Apple.

Швейцарський *startup* під назвою **Cloudo** схожий на розроблені хмарні системи, але має дуже гарний графічний інтерфейс. Для повноцінної роботи потрібний один з браузерів Chrome, Firefox, Safari.

Компанія *Amazon* розробила відому віртуальну ОС **Global Hosted Operation System**, але вона не поширюється на приватних користувачів.

Операційна система **Zero PC -** одна з найбільш функціональних хмарних операційних систем в мережі Інтернет. Сервіс працює як в браузері, так має готові

програми для *iPhone, iPad, Android і Amazon Appstore*. Крім того, сервіс підтримує авторизацію через соціальні мережі *Facebook, Google +, Twitter*.

iSpaces – проста у функціональному сенсі віртуальна операційна система, яка надає основні інструменти для роботи загалом офісного характеру. Легко інтегрується х файловими сховищами DropBox та Box.

ZimDesk – має багато функцій, як і попередня ОС, але надає більш широкі можливості щодо налаштування графічного інтерфейсу. Також має великий пакет вбудованих застосунків.

Joli OS є однією з найбільш вдалих хмарних операційних систем, в основу якої був використаний дистрибутив Ubuntu. Цей програмний продукт також можна використовувати і як інтернет- застосунок, застосунок для Chrome OS, а у перспективі і з'являться версії для Apple AppStore та Google Play.

Google Chrome OS – це сучасний представник сімейства хмарних операційних систем. Операційна система базується на ядрі Linux, яке швидко дозволяє завантажити систему. Має велику кількість сервісів Google, які дозволяють вирішувати різноманітні завдання.

З точки зору користувачів, існує сукупність "хмар" (загальнодоступних, корпоративних, приватних та ін.), що надаються різними компаніями для використання потужних обчислювальних ресурсів, яких немає у індивідуального користувача. Як правило, "хмарні" сервіси платні. З безкоштовних назвемо Windows Live (*http://www.live.com*).

Серйозною проблемою організації хмарних обчислень з точки зору апаратури центрів обробки даних є економія електроенергії та проблема розподілу завантаження, так як хмарні обчислення в кожному центрі обробки даних мають (або в найближчому майбутньому матимуть) мільйони віддалених користувачів.

Найбільш популярна хмарна платформа - ***Microsoft Windows Azure*** *(хмарна ОС)* і *Microsoft Azure Services Platform* (реалізована на основі Microsoft.NET). Windows Azure можна розглядати як "ОС в хмарі". Користувачеві немає необхідності турбуватися про її інсталяцію на його комп'ютері, який може не мати для цього необхідних ресурсів. Все, що потрібно, це мати Web-браузер і мінімальний пакет надбудов (*plug-ins*) для запуску і використання через браузер хмарних сервісів.

Платформа *Windows Azure* містить:

- Windows Azure - операційну систему в "хмарі", яка надає обчислювальні ресурси, засоби зберігання даних та інструменти управління сервісами.
- SQL Azure - реляційну базу даних, яка надає основні можливості MS SQL Server зі зберігання даних, надається як сервіс.
- Windows Azure AppFabric - програмні модулі, що забезпечують комунікації (Service Bus) і контроль доступу (Access Control). Використовуються для забезпечення взаємодій між застосунками споживача і застосунками хмари.

Microsoft Azure Service Platform - це група хмарних технологій, кожна з яких надає певний набір сервісів для розробників. Компоненти платформи можуть використовуватися локальними застосунками, що працюють на різних системах,

включаючи різні версії Windows, мобільні пристрої та ін. Ці компоненти включають:

• *ОС Windows Azure*: надає середовище для виконання застосунків, заснованих на ОС Windows, і зберігання даних на серверах в дата центрах Microsoft, тобто керує дисковим простором, застосунками і мережами;

• *Windows .NET Services*: забезпечують розподілену інфраструктуру для хмарних і локальних застосунків;

• *Microsoft SQL Services*: надають сервіси для роботи з даними, які засновані на SQL Server;

• *Live Services*: через Live Framework надається доступ до даних з застосунків на Microsoft Live. Live Framework також дозволяє синхронізувати ці дані між десктопами і пристроями, шукати і завантажувати програми та інше.

Кожний компонент платформи грає свою власну роль.

Основні компоненти операційної системи

Розглянемо основні частини ОС [13, 14].

- *Ядро* (*kernel*) - низькорівнева основа будь-якої операційної системи, яка виконується апаратурою виключно у привілейованому режимі. Ядро завантажується в пам'ять один раз і знаходиться в пам'яті резидентно, тобто постійно, за одними і тими же адресами овновної пам’яті. Ядро управляє всією операційною системою, яка містить: драйвери пристроїв, підпрограми управління пам'яттю, планувальник завдань, який реалізує системні виклики і т.п.

- *Підсистема управління ресурсами (resource allocator)* - частина операційної системи, що управляє обчислювальними ресурсами комп'ютера - оперативною і зовнішньою пам'яттю, процесором і ін.

- *Керуюча програма (control program, supervisor)* - підсистема ОС, яка керує виконанням інших програм і функціонуванням пристроїв введення-виведення.

Основні функції ядра

Ядро операційної системи, як правило, містить програми для реалізації наступних функцій:

- обробка переривань;
- створення і знищення процесів;
- перемикання процесів зі стану у стан;
- диспетчеризація;
- призупинення та активізація процесів;
- синхронізація процесів;
- організація взаємодії між процесами;
- маніпулювання блоками управління процесами;
- підтримка операцій введення-виведення;
- підтримка розподілу та перерозподілу пам'яті;
- підтримка роботи файлової системи;

- підтримка механізму виклику-повернення при зверненні до процедур;
- підтримка певних функцій з ведення обліку роботи машини.

Існує декілька ***типів архітектур ядер операційних систем***:

1. Монолітне ядро. Це така схема операційної системи, при якій всі компоненти її ядра є складовими частинами однієї програми, використовують загальні структури даних і взаємодіють один з одним шляхом безпосереднього виклику процедур. Всі частини монолітного ядра працюють в одному адресному просторі. Прикладом систем з монолітним ядром є більшість UNIX-систем такі як BSD, Linux; ядро MS-DOS, ядро KolibriOS. Недоліком такої архітектури є збій в одному з компонентів може порушити працездатність всієї системи.

2. Модульне ядро. Модульні ядра, як правило, не вимагають повної перекомпіляції ядра при зміні складу апаратного забезпечення комп'ютера, вони надають механізм підзавантаження модулів ядра, який підтримує те чи інше апаратне забезпечення (наприклад, драйвери). При цьому підзавантаження модулів може бути як динамічним (без перезавантаження ОС), так і статичним (при перезавантаженні ОС після переконфігурування системи). Прикладом модульного ядра слугує VFS - «віртуальна файлова система», спільно використовувана багатьма модулями файлових систем в ядрі Linux.

3. Мікроядро. Надає тільки елементарні функції управління процесами і мінімальний набір абстракцій для роботи з обладнанням. Велика частина роботи здійснюється за допомогою спеціальних процесів користувачів – сервісів, тобто розміщення всіх або майже всіх драйверів і модулів у сервісних процесах. У мікроядерній операційній системі можна, не перериваючи її роботи, завантажувати і вивантажувати нові драйвери, файлові системи, істотно спрощується процес налагодження компонентів ядра, так як нова версія драйвера може завантажуватися без перезапуску всієї операційної системи. Мікроядерна архітектура підвищує надійність системи. Прикладами такої архітектури є: Symbian OS; Windows CE; OpenVMS; Mach, що використовується в GNU/Hurd і Mac OS X; QNX; AIX; Minix; ChorusOS; AmigaOS; MorphOS.

4. Екзоядро. Це ядро операційної системи, що надає лише функції для взаємодії між процесами і безпечного виділення та звільнення ресурсів. Передбачається, що API для прикладних програм будуть надаватися зовнішніми по відношенню до ядра бібліотеками (звідки і назва архітектури). Можливість доступу до пристроїв на рівні контролерів дозволить ефективніше вирішувати деякі завдання, які погано вписуються в рамки універсальної ОС, наприклад, реалізація СУБД матиме доступ до диска на рівні секторів диска, а не файлів і кластерів, що позитивно позначиться на швидкодії. Прикладами такої архітектури є: XOK, ExOS.

5. Наноядро - архітектура ядра операційної системи, в рамках якої ядро виконує лише одну задачу - обробку апаратних переривань, що генеруються пристроями комп'ютера. Після обробки переривань від апаратури наноядро, в свою чергу, посилає інформацію про результати обробки (наприклад, отримані з клавіатури символи) вище розташованого програмного забезпечення за допомогою того ж механізму переривань. Прикладом є KeyKOS - найперша ОС на наноядреі,

Symbian OS v8.1- S60 2rd Edition (Nokia N70, Nokia N90), Symbian OS v9.1- S60 3rd Edition (Nokia N96).

6. Гібридні ядра - це модифіковані мікроядра, що дозволяють для прискорення роботи запускати «несуттєві» частини в просторі ядра. Мають свої переваги і недоліки.

Контрольні питання:

1. Що таке операційна система?
2. Яке призначення операційної системи?
3. Які основні функції виконує операційна система?
4. Які види ресурсів розподіляє операційна система?
5. У чому полягає ідея мультипрограмування?
6. Яка вам відома класифікація операційних систем?
7. У чому полягають особливості функціонування ОС розподілених комп'ютерних систем?
8. Які види серверів в клієнт-серверних комп'ютерних системах?
9. Які особливості хмарних операційних систем?
10. Які основні компоненти операційної системи?
11. Які функції ядра операційної системи?
12. Які типи архітектур ядер операційних систем вам відомі?

Лекція 2. Функціональні компоненти ОС, архітектура

Функціональні компоненти ОС. Базові поняття архітектури ОС. Системні виклики. Класифікація архітектур системи (набору) команд комп'ютера. Засоби апаратної підтримки. Основні характеристики ОС FreeBSD. Особливості ОС Windows

Щоб краще зрозуміти місце і роль операційної системи в процесі обчислень, розглянемо комп'ютерну систему в цілому (рис. 2.1) [5].

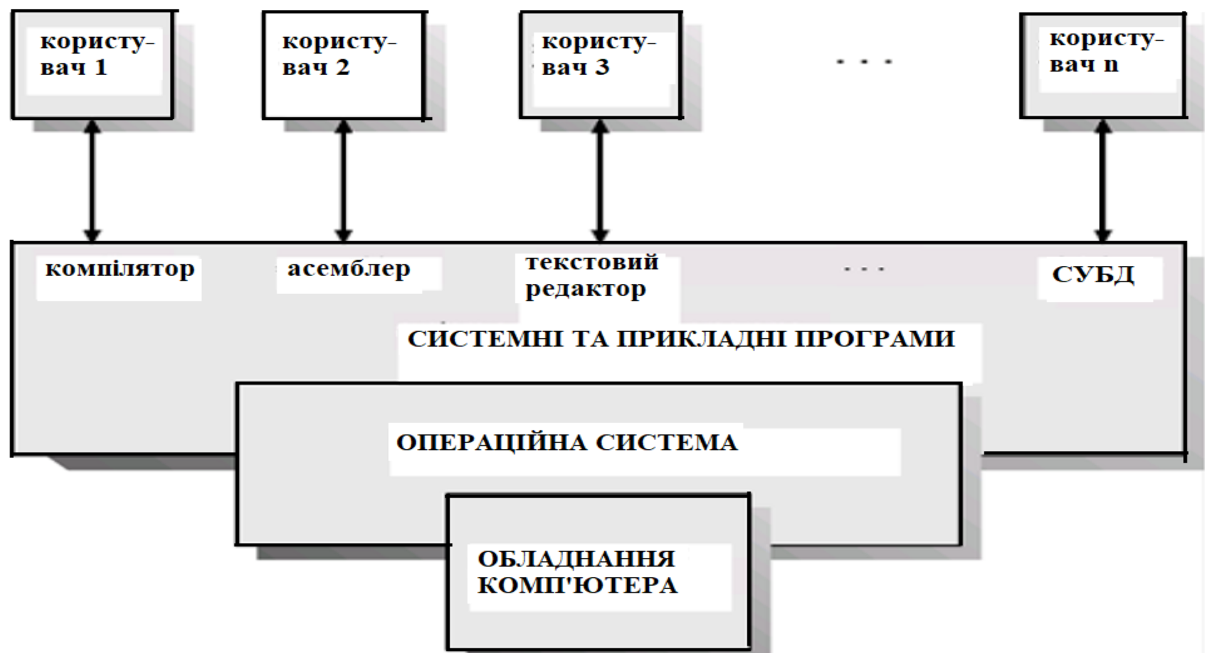


Рис. 2.1. Функціонування комп’ютерної системи

Вона складається з наступних компонентів:

1. Апаратура (hardware) комп'ютера, основні частини якої - *центральний процесор* (*Central Processor Unit - CPU*), який виконує команди (інструкції) комп'ютера; *пам'ять* (*memory*), що зберігає дані і програми, і *пристрої введення-виведення*, або зовнішні пристрої (*input-output devices, I/O devices*), що забезпечують введення інформації в комп'ютер і виведення результатів роботи програм у формі, яка сприймається користувачем-людиною або іншими програмами. Часто на програмістському сленгу апаратуру називають "залізом".

2. Операційна система (operating system) - системне програмне забезпечення, яке керує використанням апаратури комп'ютера різними програмами та користувачами.

3. Прикладне програмне забезпечення (applications software) - програми, призначені для вирішення різних класів задач. До них належать, зокрема, компілятори, що забезпечують трансляцію з програм з мов програмування, наприклад, C++, в машинний код (команди); системи управління базами даних (СКБД); графічні бібліотеки, ігрові програми, офісні програми. Прикладне програмне забезпечення утворює наступний, більш високий рівень, порівняно з

операційною системою, і дозволяє вирішувати на комп'ютері різні прикладні та повсякденні завдання.

4. Користувачі (users) - люди та інші комп'ютери (рис. 2.1). Віднесення користувача-людини до компонентів комп'ютерної системи - зовсім не жарт, а реальність: будь-який користувач фактично стає частиною обчислювальної системи в процесі своєї роботи на комп'ютері, так як повинен підкорятися певним суворим правилам, порушення яких призведе до помилок або неможливості використання комп'ютера, і виконувати великий обсяг типових рутинних дій - майже як сам комп'ютер. Одна з важливих функцій ОС якраз і полягає в тому, щоб позбавити користувача від більшої частини такої рутинної роботи (наприклад, резервного копіювання файлів) і дозволити йому зосередитися на творчій роботі. Інші комп'ютери в мережі також можуть виступати в ролі користувачів (клієнтів) по відношенню до даного комп'ютера, який виступає в ролі сервера, який використовується, наприклад, для зберігання файлів або виконання великих програм.

Користувачам комп'ютера доступні верхні рівні програмного забезпечення - системні та прикладні програми (наприклад, компілятори, текстові редактори, системи управління базами даних). Ці програми взаємодіють з операційною системою, яка, в свою чергу, управляє роботою комп'ютера.

Функціональні компоненти операційних систем

Операційну систему можна розглядати як сукупність функціональних компонентів, кожна з яких відповідає за реалізацію певної функції системи. Спосіб побудови системи зі складових частин та їхній взаємозв'язок визначає *архітектуру операційної системи* [3, 6].

1. Керування процесами і потоками. Це одна з найважливіших функцій ОС є виконання прикладних програм. Код і дані прикладних програм зберігаються в комп'ютерній системі на диску в спеціальних виконуваних файлах. При запуску користувачем або операційною системою виконуваного файлу в системі буде створено **базову одиницю обчислювальної роботи, що називається *процесом* (process).**

Процес - це програма під час її виконання. Операційна система розподіляє ресурси між процесами. До таких ресурсів належать процесорний час, пам'ять, пристрої введення-виведення, дисковий простір у вигляді файлів. При розподілі пам'яті з кожним процесом пов'язується його *адресний простір* — набір адрес пам'яті, до яких йому дозволено доступ. В адресному просторі зберігаються код і дані процесу. При розподілі дискового простору для кожного процесу формується список відкритих файлів, аналогічним чином розподіляють пристрої введення-виведення.

Процеси забезпечують захист ресурсів, якими вони володіють. Наприклад, до адресного простору процесу неможливо безпосередньо звернутися з інших процесів (він є захищеним), а при роботі з файлами може бути задано режим, що забороняє доступ до файлу всім процесам, крім поточного.

Розподіл процесорного часу між процесами необхідний через те, що процесор виконує інструкції одну за одною (тобто в конкретний момент часу на ньому може фізично виконуватися тільки один процес), а для користувача процеси мають виглядати як послідовності інструкцій, виконувані паралельно. Щоб домогтися такого ефекту, ОС надає процесор кожному процесу на деякий короткий час, після чого перемикає процесор на інший процес; при цьому виконання процесів відновлюється з того місця, де їх було перервано. У *багатопроцесорній системі* процеси можуть виконуватися паралельно на різних процесорах.

Сучасні ОС крім багатозадачності можуть підтримувати *багатопотоковість* (*multithreading*), яка передбачає в рамках процесу наявність кількох послідовностей інструкцій *(потоків, threads*), які для користувача виконуються паралельно, подібно до самих процесів в ОС. На відміну від процесів потоки не забезпечують захисту ресурсів (наприклад, вони спільно використовують адресний простір свого процесу).

2. Керування пам'яттю. Під час виконання програмного коду процесор бере інструкції і дані з оперативної (основної) пам'яті комп'ютера. При цьому така пам'ять відображається у вигляді масиву байтів, кожний з яких має адресу. Основна пам'ять є одним із видів ресурсів, які розподіляються між процесами. ОС відповідає за виділення пам'яті під захищений адресний простір процесу і за вивільнення пам'яті після того, як виконання процесу буде завершено. Обсяг пам'яті, доступний процесу, може змінюватися в ході виконання, у цьому разі говорять про динамічний розподіл пам'яті.

ОС повинна забезпечувати можливість виконання програм, які окремо або в сукупності перевищують за обсягом доступну основну пам'ять. Для цього в ній має бути реалізована технологія *віртуальної пам'яті.* Така технологія дає можливість розміщувати в основній пам'яті тільки ті інструкції і дані процесу, які потрібні в поточний момент часу, при цьому вміст іншої частини адресного простору зберігається на диску.

3. Керування введенням-виведенням. Операційна система відповідає за керування пристроями введення-виведення, підключеними до комп'ютера. Підтримка таких пристроїв в ОС зазвичай здійснюється на двох рівнях. До першого, нижчого, рівня належать *драйвери пристроїв* — програмні модулі, які керують пристроями конкретного типу з урахуванням усіх їхніх особливостей. До другого рівня належить *універсальний інтерфейс введення-виведення,* зручний для використання у прикладних програмах.

ОС має реалізовувати загальний інтерфейс драйверів введення-виведення, через який вони взаємодіють з іншими компонентами системи. Такий інтерфейс дає змогу спростити додавання в систему драйверів для нових пристроїв.

Сучасні ОС надають великий вибір готових драйверів для конкретних периферійних пристроїв. Чим більше пристроїв підтримує ОС, тим частіше вона практично використовується.

4. Керування файлами та файлові системи. Для користувачів ОС і прикладних програмістів дисковий простір надається у вигляді сукупності *файлів,* організованих у *файлову систему.*

Файл — це набір даних у файловій системі, доступ до якого здійснюється за іменем. Термін „файлова система” може вживатися для двох понять: принципу організації даних у вигляді файлів і конкретного набору даних (зазвичай відповідної частини диска), організованих відповідно до такого принципу. У рамках ОС може бути реалізована одночасна підтримка декількох файлових систем.

Файлові системи розглядають на логічному і фізичному рівнях. *Логічний* рівень визначає зовнішнє подання системи як сукупності файлів (які зазвичай розташовуються у каталогах), а також виконання операцій над файлами і каталогами (створення, вилучення тощо). *Фізичний* рівень визначає принципи розміщення структур даних файлової системи на диску або іншому пристрої.

5. Мережна підтримка. Засоби мережної підтримки забезпечують *клієнт-серверну технологію*, а саме, ОС має можливість: а) надавати локальні ресурси (дисковий простір, принтери тощо) у загальне користування через мережу, тобто функціонувати як сервер; б) звертатися до ресурсів інших комп'ютерів через мережу, тобто функціонувати як клієнт. Реалізація функціональності сервера і клієнта базується на *транспортних засобах,* відповідальних за передачу даних між комп'ютерами відповідно до правил, обумовлених мережними протоколами (набір правил, який визначає формат даних для пересилання по мережі).

6. Безпека даних. Під безпекою даних в ОС розуміють забезпечення надійності системи (захисту даних від втрати у разі збоїв) і захист даних від несанкціонованого доступу (випадкового чи навмисного).

Для захисту від несанкціонованого доступу ОС має забезпечувати наявність засобів *аутентифікації* користувачів (такі засоби дають змогу з'ясувати, чи є користувач тим, за кого себе видає; зазвичай для цього використовують систему паролів) та їхньої *авторизації* (дозволяють перевірити права користувача, що пройшов аутентифікацію, на виконання певної операції).

7. Інтерфейс користувача. Розрізняють два типи засобів взаємодії користувача з ОС: *командний інтерпретатор* (*shell*) і *графічний інтерфейс користувача* (*GUI*), який включає вікна, значки, меню та мишу.

Командний інтерпретатор дає змогу користувачам взаємодіяти з ОС, використовуючи спеціальну командну мову (інтерактивне або через запуск на виконання командних файлів). Команди такої мови змушують ОС виконувати певні дії (наприклад, запускати програми, працювати із файлами).

Графічний інтерфейс користувача надає йому можливість взаємодіяти з ОС, відкриваючи вікна і виконуючи команди за допомогою меню або кнопок. Підходи до реалізації графічного інтерфейсу доволі різноманітні: наприклад, у Windows-системах засоби його підтримки вбудовані в систему, а в UNIX вони є зовнішніми для системи і спираються на стандартні засоби керування введенням-виведенням.

Базові поняття архітектури ОС

Набір компонентів ОС, які відповідають за певні функції, та порядок їх взаємодії між собою та із зовнішнім середовищем, називається **архітектурою ОС.** *Набір функціональних можливостей компонентів системи становлять* **механізм**.

Використання зазначених можливостей утворюють **політику**. Наприклад, механізм доступу до периферійних пристроїв реалізують драйвери, а політику використання цих механізмів задає програмне забезпечення по введенню-виведенню [1, 5].

Базові компоненти ОС, які відповідають за найважливіші функції, зазвичай перебувають у пам'яті постійно і виконуються у привілейованому режимі, називаються *ядром операційної системи* (*operating system kernel*).

До найважливіших функцій ОС, які виконує ядро належить обробка переривань, керування пам'яттю, керування введенням – виведенням. До надійності та продуктивності ядра висувають підвищені вимоги.

Для забезпечення ефективного керування ресурсами комп'ютера ОС повинна мати певні привілеї щодо прикладних програм. Тобто ОС має можливість втручатися в роботу прикладних програм, а прикладні програми не можуть втручатися в роботу ОС. Для реалізації таких привілеїв розроблена апаратна підтримка: процесор підтримує два режими роботи: *привілейований* (*захищений режим, режим ядра, kernel mode*) і *режим користувача* (*user mode*). У режимі користувача недопустимі команди, які є критичними для роботи системи (перемикання задач, звернення до пам'яті за заданими межами, доступ до пристроїв введення-виведення тощо).

Після завантаження системи ядро перемикає процесор у привілейований режим і отримує повний контроль над комп'ютером. Кожний застосунок запускається та виконується в режимі користувача. Коли потрібно виконати дію, реалізовану у ядрі, застосунок робить *системний виклик* (*system call*). Ядро перехоплює його, перемикає процесор у привілейований режим, виконує дію, перемикає процесор назад у режим користувача і повертає результат застосунка.

Системний виклик – це засіб доступу до певної функції ядра ОС з прикладних програм. Набір системних викликів визначає дії, які ядро може виконати за запитом процесів користувача. Системні виклики задають інтерфейс між прикладною програмою і ядром ОС. Для позначення цього набору функцій використовується термін *інтерфейс програмування застосунків* (Application Programming Interface - API), який реалізований як набір бібліотечних функцій.

Окрім ядра, важливими складовими роботи ОС є системні функції, які виконують режим користувача. До такого *системного програмного забезпечення належать:*

- Системні програми (утиліти), наприклад, командний інтерпретатор, програми резервного копіювання та відновлення даних, засоби діагностики та адміністрування;
- Системні бібліотеки, в яких реалізовано функції для роботи застосунків користувачів.

Класифікація архітектур системи (набору) команд комп'ютера

Комп'ютерні системи відрізняються між собою не тільки за своїми параметрами і своєму призначенню, але і за своїми внутрішніми архітектурними

принципами системи команд. Найбільш відомі такі підходи до архітектури комп'ютерних систем [1, 5].

CISC (Complicated Instruction Set Computers - комп'ютери з ускладненою системою команд) - історично перший підхід до комп'ютерної архітектури, суть якого полягала у тому, що в систему команд комп'ютера включаються складні за семантикою операції, які реалізують типові дії, що часто використовуються при програмуванні і при реалізації мов - наприклад, виклик рекурсивних процедур і автоматичне оновлення дисплей-регістрів, групові операції пересилання рядків і масивів та ін. Типовими представниками CISC-комп'ютерів були: із зарубіжних комп'ютерних систем - машини серії IBM 360/370, з російських - багатопроцесорні обчислювальні комплекси (МВК) "Ельбрус". Основним недоліком CISC-архітектура полягає в тому, що подібні групові операції на час їх виконання фактично зупиняли роботу конвеєра (pipeline, послідовність команд, які виконують стандартне виведення) - реалізованої в будь комп'ютерній архітектурі апаратної оптимізації, паралельного виконання кількох сусідніх команд за умови їх незалежності один від одного за даними.

RISC (Reduced Instruction Set Computers - комп'ютери зі спрощеною системою команд) - спрощений підхід до архітектури комп'ютерів, запропонований на початку 1980-х рр. професором Девідом Паттерсоном (університет Берклі, США) і його студентом Девідом Дітцель (згодом - великим вченим, керівником компанії Transmeta). Приклади сімейств RISC-комп'ютерів: SPARC, MIPS, PA-RISC, PowerPC. Принципи даного підходу: спрощення семантики команд, відсутність складних групових операцій (які можуть бути реалізовані послідовностями команд, що містять цикли); однакова довжина команд (32 біта - архітектура була розроблена в розрахунку на 32-бітові процесори); виконання арифметичних операцій тільки в регістрах і використання спеціальних команд зчитування з пам'яті в регістр і записи з регістра в пам'ять; відсутність спеціалізованих регістрів (наприклад, дисплей-регістрів для адресації доступних областей локальних даних в стеку); використання великого набору регістрів (реєстрового файлу) загального призначення-512, 1024, 2048 регістрів і т.д., в залежності від конкретної моделі процесора; передача при виклику процедур параметрів через регістри. Подібна архітектура дає широкий простір для оптимізацій, виконуваних компіляторами, що і демонструють компілятори *Sun Studio* розробки фірми Sun/Oracle для ОС Solaris і Linux. RISC-архітектура досі використовується при розробці нових комп'ютерів.

VLIW (Very Long Instruction Word) - комп'ютери з широким командним словом) - підхід до архітектури комп'ютерів, що склався в 1980-х - 1990-х рр. Основна ідея даного підходу - статичне планування паралельних обчислень компілятором на рівні окремих послідовностей команд і підкоманд. При даній архітектурі кожна команда є "широкою" (*long*) і містить кілька підкоманд, виконуваних паралельно за один машинний такт на декількох однотипних пристроях процесора - наприклад, в такому комп'ютері може бути два пристрої додавання, два логічних пристрої, два пристрої для виконання переходів і т.д. Завданням компілятора є оптимальне планування завантаження всіх цих пристроїв в кожному машинному такті і генерація таких (широких) команд, які дозволили б

оптимально завантажити на кожному такті кожне з пристроїв. Перевагою такої архітектури є можливість розпаралелювання обчислень, недоліком - складність (в порівнянні з RISC-архітектурою). Приклади комп'ютерів таких архітектур: із зарубіжних - комп'ютери Cray X/MP та ін., Розроблені комп'ютерним генієм Сей муром Креем (Cray) і його фірмою Cray Research; з вітчизняних - багатопроцесорний обчислювальний комплекс "Ельбрус-3".

EPIC (Explicit Parallelism Instruction Computers - комп'ютери з явним розпаралелюванням) - по архітектурі аналогічні VLIW, але з додаванням ряду важливих удосконалень: наприклад, спекулятивних обчислень - паралельного виконання обох гілок умовної конструкції з обчисленням умови. Підхід склався і використовується з 1990-х рр. Приклади процесорів даної архітектури - Intel IA-64, AMD-64.

Multi-core computers (багатоядерні комп'ютери) - отримала найбільш широку популярність в даний час архітектура комп'ютерів, при якій кожен процесор має кілька ядер (cores), об'єднаних в одному кристалі і паралельно працюють на одній і тій же загальній пам'яті, що дає широкі можливості для паралельних обчислень. В даний час *відомі багатоядерні процесори* фірми Intel (Cure 2 Duo, Dual Core та ін.), А також потужні багатоядерні процесори фірми Sun / Oracle: Ultra SPARC-T1 ("Niagara") - 16-ядерний процесор; Ultra SPARC-T2 ("Niagara2") - 32-ядерний процесор. Всі провідні фірми світу зайняті розробкою і випуском все більш потужних багатоядерних процесорів. Відповідно, творці операційних систем для таких комп'ютерів розробляють базові бібліотеки програм, що дозволяють повною мірою використовувати можливості паралельного виконання на багатоядерних процесорах.

Hybrid processor computers (комп'ютери з гібридними процесорами) - новий, все ширше поширюється підхід до архітектури комп'ютерів, при якому процесор має гібридну структуру - складається з (багатоядерного) центрального процесора (CPU) і (також багатоядерного) графічного процесора (GPU - Graphical Processor Unit). Така архітектура була розроблена, у зв'язку з необхідністю паралельної обробки графічної і мультимедійної інформації, що особливо актуально для комп'ютерних ігор, перегляді на комп'ютері високоякісного цифрового відео та ін. Гібридна архітектура є новим "інтелектуальним викликом" для розробників компіляторів, яким необхідно розробити і реалізувати адекватний набір оптимізацій, як для центральних, так і для графічних процесорів. Прикладами таких архітектур є нові процесори фірми AMD (*Advanced Micro Devices Inc)* покоління Ryzen серії 5000, які вважаються найшвидшими ігровими процесорами в світі, а також графічні процесори серії Tesla фірми NVidia (NVIDIA Ampere, NVIDIA Turing).

Засоби апаратної підтримки операційних систем

Сучасні апаратні архітектури комп’ютерів реалізують засоби підтримки операційних систем, а саме: система переривань, привілейований режим процесора, засоби перемикання задач, підтримка керування пам’яттю (механізми

трансляції адрес, захист пам’яті), системний таймер, захист пристроїв введення – виведення, базова система введення – виведення (BIOS) [2].

Механізм переривань є основним засобом керування роботою ОС. За допомогою системи переривань процесор отримує інформацію про події, не пов’язані з основною його роботою. Переривання бувають двох типів; *апаратні і програмні.*

Апаратне переривання – це спеціальний сигнал (запит переривання), який передається процесору від апаратного пристрою. До апаратних переривань належать:

- переривання введення-виведення, які надходять від контролера периферійного пристрою (наприклад, таке переривання генерує контролер клавіатури при натисканні клавіші);
- переривання, пов’язані з апаратними або програмними помилками (такі переривання виникають у разі збою контролера диска, доступу до забороненої області пам’яті, ділення на нуль).

Програмні переривання генерує прикладна програма, яка виконує *інструкцію переривання,* управління передається оброблювачу переривання. Обробка програмних переривань не відрізняється від обробки апаратних переривань.

Для реалізації *привілейованого режиму процесора* в одному з його регістрів передбачено спеціальний біт ***IF*** (***interrupt flag,*** біт режиму), який показує у якому режимі знаходиться процесор. У разі програмного або апаратного переривання процесор автоматично перемикається у привілейований режим, ядро завжди отримує керування саме у цьому режимі.

Засоби перемикання задач дають змогу зберігати вміст регістрів процесора (контекст задачі) у разі припинення задачі та відновлювати дані перед її подальшим виконанням.

Механізм трансляції адрес забезпечує перетворення адрес пам’яті, з якими працює програма, в адреси фізичної пам’яті комп’ютера. Апаратне забезпечення генерує фізичну адресу, використовуючи спеціальні таблиці трансляції.

Захист пам’яті забезпечує перевірку прав доступу до пам’яті під час кожної спроби його отримати. Засоби захисту пам’яті містять інформацію про права доступу та про ліміт (розміри ділянки пам’яті, до якої можна отримати доступ).

Системний таймер є апаратним пристроєм, який генерує переривання таймера через певні проміжки часу. Такі переривання обробляє ОС і використовуються для визначення часу перемикання задач.

Захист пристроїв введення – виведення підтримується тим, що усі інструкції введення – виведення виконуються як привілейовані.

Базова система ведення – виведення (BIOS) - службовий програмний код, який зберігається у постійному запам’ятовуючому пристрої і призначений для ізоляції ОС від конкретного апаратного забезпечення.

Більша частина коду ОС написана мовою високого рівня C та C++, використання мови асемблера використовується тоді, коли продуктивність компонента є критичною для системи.

Для забезпечення *програмної сумісності* (це означає виконання в середовищі однієї операційної системи програми, розроблені для іншої ОС) проводяться

роботи зі стандартизації інтерфейсу операційних систем в рамках проекту POSIX (Portable Operating System Interface), а саме: наявність стандарту на мови програмування (C та C++) і стандартних компіляторів та наявність стандарту на інтерфейс операційної системи (API).

Операційна система MS DOS

Операційна система MS DOS з'явилась в 1981 р. одночасно з комп'ютерами типу IBM PC і стала для них домінуючою. Популярність цієї операційної системи була такою великою, що основні її компоненти, такі як базова система введення-виведення та файлова система, використовуються і досі в сімействі операційних систем Windows [1, 5].

Серед позитивних якостей MS DOS слід відзначити:

- розвинену командну мову;
- можливість організації багаторівневих каталогів;
- роботу з усіма послідовними пристроями як із файлами;
- можливість під'єднання користувачем додаткових драйверів зовнішніх пристроїв;
- можливість запуску фонових задач одночасно з діалоговою роботою користувача.
- Найважливішою характерною особливістю MS DOS є її модульність. Основними модулями системи є:

- базова система введення-виведення BIOS (Basic Input Output System);
- блок початкового завантаження Boot Record;
- модуль розширення базової системи введення-виведення IO.SYS;
- модуль обробки переривань MSDOS.SYS;
- командний процесор COMMAND.COM.

Кожний із цих модулів виконує певну частину функцій, покладених на MS DOS. Так, BIOS міститься в постійній пам'яті. Блок початкового завантаження або завантажувач завжди записаний у першому секторі системного диска. Модулі IO.SYS та MSDOS.SYS зберігаються на системному диску, місцеположення їх відоме завантажувачеві. Командний процесор - це звичайний файл, який може займати довільне місце на системному диску.

Коротко охарактеризуємо основні функції модулів MS DOS. BIOS призначена для автоматичного тестування основних апаратних компонент у разі вмикання комп'ютера, а із закінченням тестування викликає завантажувач і передає йому керування. Третьою важливою функцією BIOS є обслуговування системних переривань нижнього рівня, тобто тих які вимагають безпосереднього керування апаратними компонентами (дисплеєм, клавіатурою, магнітними дисками, принтерами, комунікаційними каналами). Таким чином, BIOS є програмною оболонкою навколо апаратних засобів комп'ютера, яка надає можливість іншим програмам, у тому числі й самій операційній системі, звертатися до апаратних компонент через механізм переривань.

Завантажувач - не невелика програма, єдина функція якої полягає у зчитуванні в оперативну пам'ять двох інших частин MS DOS - IO.SYS і MSDOS.SYS.

BIOS, яка розміщується в ПЗП і є інваріантною відносно операційної системи, що використовується на даному комп'ютері. Зміна BIOS - нетривіальне завдання, оскільки воно дуже тісно пов'язане з особливостями апаратури конкретної моделі персонального комп'ютера.

Розширення BIOS за допомогою додаткового модуля MS DOS надає гнучкості операційній системі, дає змогу "переривати" за допомогою механізму переривань функції BIOS і вмикати програми, що обслуговують нові зовнішні пристрої (драйвери). Драйвери розробляються не тільки для нових зовнішніх пристроїв, а й для тих, які стандартно входять до складу апаратури в тих випадках, коли обмін інформацією з ними має відбуватися інакше, ніж у стандартній версії MS DOS.

Крім цих функцій **модуль розширення базової системи введення-виведення IO.SYS** завершує завантаження MS DOS в пам'ять. Для цього він передає керування на завантажений в оперативну пам'ять модуль обробки переривань MSDOS.SYS, в якому встановлюються внутрішні робочі таблиці, ініціюються вектори переривань з номерами 32-39 і виконується підготовка до завантаження командного процесора. Після цього керування повертається в модуль розширення BIOS, який завантажує командний процесор із диска в оперативну пам'ять і передає йому керування.

Модуль обробки переривань MSDOS.SYS, крім вищезазначених функцій, утворює верхній рівень системи, з яким взаємодіє більшість прикладних програм. Компонентами даного модуля є підпрограми, які забезпечують роботу файлової системи, пристроїв введення-виведення, обслуговування деяких спеціальних ситуацій, пов'язаних із завершенням програм і обробкою помилок.

На відміну від двох попередніх модулів **командний процесор COMMAND.COM** трактується як звичайна програма. Його основні функції такі:

- прийом і розпізнавання команд, одержаних із клавіатури або з командного файлу;
- завантаження й виконання зовнішніх команд MS DOS і прикладних програм (файли типу COM і EXE);
- виконання файлу автозапуску (AUTOEXEC.BAT).

Завантаження операційної системи MS DOS відбувається коли комп'ютер вмикається або в разі перезавантаження (шляхом одночасного натиснення на три клавіші Ctrl+Alt+Del або на спеціальну кнопку RESET, що знаходиться на системному блоці).

Зручна обстановка для користувач може бути створена в результаті конфігурування і початкового настроювання системи, тобто в разі завантаження операційної системи автоматично можуть задаватися певні початкові умови, які впливають на подальшу роботу користувача. Здійснюється це за допомогою двох файлів: **файлу конфігурації CONFIG.SYS** та **файлу автозапуску AUTOEXEC.BAT**.

Файлом конфігурації можна завантажувати додаткові драйвери (команда DEVICE), розширювати об'єм доступної оперативної пам'яті (команда

DOS=HIGH), змінювати за допомогою драйверів деякі параметри, які впливають на роботу зовнішніх пристроїв. Крім цього, у файлі CONFIG.SYS можна вказати, яка кількість файлів у системі може бути відкрита одночасно (команда FILES), кількість буферів для обміну із зовнішніми накопичувачами (команда BUFFERS), встановити формат виведення дати, часу та іншої інформації відповідно до узгоджень, прийнятих у тій чи іншій країні (команда COUNTRY) і т.д.

Наведемо приклад типового файлу CONFIG.SYS:
FILES=50 BUFFERS=32 DEVICE=HIMEM.SYS DOS=HIGH DEVICE=RAMDRIVE.SYS /E COUNTRY=380, 866

Означає, що максимальне число відкритих файлів дорівнює 50 (FILES=50), необхідне число буферів (BUFFERS=32) для операцій введення-виведення з диском дорівнює 32, треба підключити драйвер управління розширеною пам'яттю (DEVICE= HIMEM.SYS); у верхню область оперативної пам’яті завантажити резидентні програми і драйвери (DOS=HIGH); драйвер віртуальних дисків в розширеній пам’яті (DEVICE=RAMDRIVE.SYS /E) , налаштування DOS на конкретну країну (COUNTRY=380) 380 – Україна, 033 – Франція. 866 – DOS-розкладка кирилиці.

Файл автозапуску AUTOEXEC.BAT відноситься до так званих пакетних файлів (файлів з розширенням .BAT). Створити будь-який пакетний файл можна за допомогою будь-якого текстового редактора. Він складається з команд операційної системи та виконуваних програм (файлів із розширенням .EXE або .COM), які виконуються після запуску пакетного файлу. Пакетний файл AUTOEXEC.BAT містить команди, які повинні виконуватися щоразу, коли завантажується операційна система. В разі створення файлу автозапуску AUTOEXEC.BAT його потрібно помістити в кореневий каталог системного диску. При виконанні цього файлу завершується завантаження операційної системи.

Мова команд операційної системи MS DOS є основним засобом спілкування користувача із системою. Команда MS DOS має такий вигляд:

mmm [a1 a2 … an] [/f1 /f2 … /fk].

Тут *mmm* - назва команди (програми). Це є обов'язковий елемент. Аргументи *a1 a2 ... an*, які не є обов'язковими (факт необов'язковості позначається квадратними дужками) і вимагаються не в кожній команді, як правило, вказують на ті об'єкти, з якими має справу дана команда (імена накопичувачів, каталогів, файлів і т.д.). Параметри */f1 /f2 ... /fk* слугують для завдання різних модифікацій і режимів в разі виконання даної команди.

Існує два типи команд операційної системи MS DOS: внутрішні та зовнішні.

Внутрішні команди - це найпростіші та найчастіше використовувані команди системи. Вони є частиною командного процесора COMMAND.COM і завантажуються в пам'ять під час завантаження операційної системи.

Зовнішні команди реалізовані у вигляді окремих виконуваних програм, тобто у вигляді файлів з розширенням .EXE або .COM і знаходяться вони в системному

каталозі. Наведемо деякі, найуживаніші команди операційної системи MS DOS із їх коротким описом.

Командний процесор має команди для роботи з каталогами і команди для роботи з файлами. Перелік цих команд є темою окремої лекції. Кому цікаво можу надіслати.

Основні характеристики ОС FreeBSD

FreeBSD - це клон операційної системи UNIX для персональних комп'ютерів, що базуються на архітектурі процесорів *Intel (386SX/386DX/486SX/486DX/Pentium/Pentium Pro/...).* FreeBSD працює також на процесорах AMD та VIA, сумісних з Intel, а також на процесорах DEC Alpha. FreeBSD надає широкий набір функцій, що раніше були доступні тільки на більш дорогих комп'ютерах [1, 5]. Вони містять:

- *витісняючу багатозадачність (Preemptive multitasking)* з динамічним налаштуванням пріоритетів, що забезпечує гнучкий поділ ресурсів комп'ютера поміж застосунками й користувачами; витісняюча багатозадачність означає, що рішення про перемикання процесора з виконання одного процесу на виконання іншого процесу ухвалюється планувальником операційної системи, а не найактивнішим завданням. При витісняючій багатозадачності механізм планування завдань цілком зосереджений в операційній системі, і програміст пише своє застосування, не піклуючись про те, що воно виконуватиметься паралельно з іншими завданнями. При цьому операційна система виконує наступні функції: визначає момент зняття з виконання активного завдання, запам'ятовує її контекст, вибирає з черги готових завдань наступну і запускає її на виконання, завантажуючи її контекст.

- *багатокористувацький доступ*, який означає, що водночас в системі можуть працювати кілька користувачів, котрі використовують різні застосунки. Такі периферійні ресурси, як принтер і магнітна стрічка, також поділяються поміж усіма користувачами системи;

- *багатопоточність* - можливість паралельно виконувати кілька видів операцій в одній прикладній програмі. Паралельні обчислення дозволяють більш ефективно використовувати ресурси центрального процесора й потребують меншого сумарного часу на виконання задач;

- *повну мережну підтримку стеку протоколів TCP/IP (Transmission Control Protocol / Internet Protocol),* включаючи протоколи SLIP (Serial Lines Internet Protocol), PPP (Point-to-Point Protocol), а також мережевої файлової системи NFS (Network File System) та NIS (Network Information System). Це означає, що FreeBSD система може легко взаємодіяти з іншими операційними системами, а також працювати як сервер, що надає такі важливі функції, як NFS та електронна пошта. У FreeBSD можна зорганізувати WWW- чи ftp-сервер, за допомогою якого можна подавати свою організацію в Internet, установлювати роутінг (маршрутизацію) і систему безпеки (firewall), що захищає корпоративну мережу від небажаного впливу зовнішнього інформаційного середовища;

- *захист пам'яті,* який гарантує, що застосунки (чи користувачі) не можуть зашкоджати один одному. У будь-якому разі знищення одного застосунка жодним чином не зачіпає роботу інших;
- FreeBSD - *32-бітна операційна система,* і була такою із самого початку;
- *промисловий стандарт X Window System (X11R6),* котрий надає графічний користувацький інтерфейс (GUI), підтримує більшість VGA-карт, моніторів і надходить із усіма вихідними кодами;
- *двійкову сумісність* з багатьма програмами, створеними для систем SCO, BSDI, NetBSD, Linux та 386BSD;
- *велику кількість готових до роботи* застосунків, які містяться в колекції пакетів і переносяться (Port Packages Collection);
- *вихідні коди FreeBSD*, сумісні з багатьма комерційними системами UNIX та більшістю застосунків; вони якщо й потребують, то зовсім небагатьох змін для своєї компіляції;
- сторінкову організацію *віртуальної пам'яті (VM)* з підкачуванням сторінок за вимогою процесора і загальний кеш для VM та буфера I/O, котрі дозволяють задовольняти надмірні потреби застосунків, у той же час не заподіюючи незручностей іншим користувачам;
- *поділювані бібліотеки* (Unix-овий еквівалент MS-Windows DLL), котрі забезпечують ефективне використання дискового простору та пам'яті;
- *повний набір засобів розробляння для мов C, C++ і Fortran.* У колекції пакетів можна віднайти багато інших мов для найсучасніших досліджень та розроблянь;
- *вихідні коди всієї системи*. Маючи їх, можна одержувати найвищий рівень контролю над середовищем FreeBSD;
- *велика кількість on-line документації.*

FreeBSD базується на ОС BSD версії 4.4BSD-Lite, розробленої дослідницькою групою комп’ютерних систем (Computer Systems Research Group, скорочено CSRG) Каліфорнійського університету в Берклі 1993 року, і несе в собі традиції розробляння систем BSD. Група FreeBSD Project домоглася максимальної продуктивності й надійності системи в ситуаціях реального життя, витративши на це досить багато часу, тоді як багато комерційних гігантів ще б'ються над розв’язанням цих задач для операційних систем для PC.

FreeBSD використовується для створення багатьох проектів, з-посеред яких: Yahoo!, Hotmail, Apache, Be, Inc., Blue Mountain Arts, Pair Networks, Whistle Communications, *BSDi* і безліч інших.

Особливості ОС Windows

Windows являє собою операційну систему з гібридним ядром, в якій основні системні функції по *управлінню процесами, пам'яттю, пристроями, файловою системою і безпекою* реалізовані в компонентах, що працюють в *режимі ядра*; але існує ряд важливих системних компонентів *користувацького режиму*, наприклад

*системні процеси входу в систему, локальної аутентифікації, диспетчера сеанс*ів, а також підсистеми оточення [2, 3].

Компоненти режиму користувача

У режимі користувача працюють наступні види процесів:

- *системні процеси (system processes)* - компоненти Windows, що відповідають за вирішення критично важливих системних завдань (аварійне завершення одного з цих процесів викликає крах чи нестабільну роботу всієї системи), але виконуються в режимі користувача. Основні системні процеси:
  - ✓ **Winlogon.exe** - процес входу в систему і виходу з неї;
  - ✓ **Smss.exe** (*Session Manager - диспетчер сеансів*) - процес виконує важливі операції при ініціалізації системи (завантаження необхідних DLL, запуск процесів Winlogon і Csrss та ін.), А потім контролює роботу Winlogon і Csrss;
  - ✓ **Lsass.exe** (*Local Security Authentication Subsystem Server* - сервер підсистеми локальної аутентифікації) - процес перевіряє правильність введених імені користувача і пароля;
  - ✓ **Wininit.exe** - процес ініціалізації системи (наприклад, запускає процеси Lsass і Services);
  - ✓ **Userinit.exe** - процес ініціалізації середовища користувача (наприклад, запускає системну оболонку - за замовчуванням, Explorer.exe);
  - ✓ **Services.exe** (SCM, *Service Control Manager* - диспетчер управління службами) - процес, який відповідає за виконання служб - див. Нижче;
- *служби (сервіси, services)* - програми, що працюють у фоновому режимі і не потребують взаємодії з користувачем. Служби можуть бути як частиною операційної системи (наприклад, Windows Audio - служба для роботи зі звуком, або Print Spooler - диспетчер друку), так і частиною користувальницького застосунку (наприклад, служба СУБД SQL Server). За служби відповідає системний процес Services.exe;
- *застосунки користувачів (user applications)* - прикладні програми, що запускаються користувачем;
- *підсистеми оточення (environment subsystems)* - компоненти, що надають доступ застосунків до деякій підмножині системних функцій. Windows підтримує дві підсистеми оточення:
  - ✓ власне Windows - за допомогою даної підсистеми виконуються 32 розрядні застосунки Windows (Win32), а також 16 розрядні застосунки Windows (Win16), застосунки MS DOS і консольні застосунки (Console). За підсистему Windows відповідає системний процес **Csrss.exe** і драйвер режиму ядра **Win32k.sys**;
  - ✓ POSIX (Portable Operating System Interface for UNIX - переносимий інтерфейс операційних систем UNIX) - підсистема для UNIX-застосунків. Починаючи з Windows Server 2003 R2 компонент, який реалізує цю підсистему, називається SUA (Subsystem for UNIX-based Applications). Компонент не встановлюється в Windows за замовчуванням.

Всі перераховані процеси користувальницького режиму (крім підсистеми POSIX1) для взаємодії з модулями режиму ядра використовують бібліотеки Windows DLL (Dynamic Link Library - динамічно підключається). Кожна DLL експортує набір Windows API функцій, які може викликати процес.

Windows API (Windows Application Programming Interface, WinAPI) - це спосіб взаємодії процесів користувацького режиму з модулями режиму ядра. WinAPI включає тисячі функцій і добре документований.

Основні Windows DLL наступні:

- ✓ **Kernel32.dll** - базові функції, в тому числі робота з процесами і потоками, управління пам'яттю і введенням-виведенням;
- ✓ **Advapi32.dll** - функції, в основному пов'язані з управлінням безпекою і доступом до реєстру;
- ✓ **User32.dll** - функції, що відповідають за управління вікнами і їх елементами в GUI застосунках (Graphical User Interface - графічний інтерфейс користувача);
- ✓ **Gdi32.dll** - функції графічного інтерфейсу користувача (Graphics Device Interface, GDI), що забезпечують малювання на дисплеї та принтері графічних примітивів і виведенням тексту.

Бібліотека Ntdll.dll експортує в більшості своїй недокументовані системні функції, реалізовані, в основному, в Ntoskrnl.exe. Набір таких функцій називається Native API ("рідний" API).

Бібліотеки Windows DLL перетворюють виклики документованих WinAPI функцій у виклики функцій Native API і перемикають процесор на режим ядра.

<u>Компоненти режиму ядра</u>

Диспетчер системних сервісів (*System Service Dispatcher*) працює в режимі ядра, перехоплює виклики функцій від Ntdll.dll, перевіряє їх параметри і викликає відповідні функції з Ntoskrnl.exe.

Виконавча система і ядро містяться в **Ntoskrnl.exe (NT Operating System Kernel** - ядро операційної системи NT).

Виконавча система (*Executive*) являє собою сукупність компонентів (так званих диспетчерами - manager), які реалізують основні завдання операційної системи:

- *диспетчер процесів (process manager)* - управління процесами і потоками;
- *диспетчер пам'яті (memory manager)* - управління віртуальною пам'яттю і відображення її на фізичну;
- *монітор контролю безпеки (security reference monitor)* - управління безпекою;
- *диспетчер введення виведення (I / O manager), диспетчер кеша (cache Manager), диспетчер Plug and Play (PnP Manager)* - управління зовнішніми пристроями і файловими системами;

- *диспетчер електроживлення (power manager)* - управління електроживленням і енергоспоживанням;
- *диспетчер об'єктів (object manager), диспетчер конфігурації (configuration manager), механізм виклику локальних процедур (local procedure call)* - управління службовими процедурами і структурами даних, які необхідні інших компонентів.

Ядро (Kernel) містить функції, що забезпечують підтримку компонентам виконавчої системи та здійснюють планування потоків, механізми синхронізації, обробку переривань.

Компонент Windows USER і GDI (Graphic Device Interface) відповідає за користувальницький графічний інтерфейс (вікна, елементи управління в вікнах - меню, кнопки і т. п., малювання), є частиною підсистеми Windows і реалізований в драйвері *Win32k.sys*.

Взаємодія диспетчера введення виведення з пристроями забезпечують драйвери (drivers) - програмні модулі, що працюють в режимі ядра, які мають максимально повною інформацією про конкретному пристрої.

Однак, і драйвери, і ядро не взаємодіють з фізичними пристроями безпосередньо - посередником між програмними компонентами режиму ядра і апаратурою є *HAL (Hardware Abstraction Layer)* - рівень абстрагування від устаткування, реалізований у Hal.dll. HAL дозволяє приховати від усіх програмних компонентів особливості апаратної платформи (наприклад, відмінності між материнськими платами), на якій встановлена операційна система.

Контрольні питання:

1. Які функціональні компоненти операційної системи вам відомі?
2. Що таке архітектура операційної системи?
3. Які вас відомі підходи до архітектури комп'ютерних систем?
4. У чому полягає механізм переривань?
5. Які особливості привілейованого режиму процесора?
6. Які основні модулі системи MS DOS?
7. Що таке завантажувач?
8. Які основні складові файлу конфігурації CONFIG.SYS MS DOS?
9. Які типи команд має операційної системи MS DOS?
10. Надайте характеристику ОС FreeBSD.
11. У чому полягають особливості ОС Windows?
12. Які компоненти режиму користувача ОС Windows?
13. Що таке Windows API?
14. Які основні Windows DLL?
15. Які компонети режиму ядра ОС Windows?

Тема 2. Підсистема управління оперативною пам’яттю
Лекція 3. Підсистема управління оперативною пам'яттю

Засоби апаратної підтримки управління пам’яттю Організація оперативної пам’яті. Функції ОС з управління пам’яті. Поняття віртуальної пам’яті. Диспетчер пам’яті. Фрагментація пам’яті. Логічна і фізична адресація пам’яті. Сегментація пам’яті та її особливості. Реалізація сегментації в архітектурі IA-32. Сторінкова організація пам’яті. Асоціативна пам’ять. Сторінково-сегментна організація пам’яті. Свопінг. Кеш-пам’ять.

Засоби апаратної підтримки управління пам’яттю. Інформація, яка вводиться користувачем з клавіатури або дисків, потрапляє в оперативну пам’ять. З оперативної пам'яті процесор бере програми, вихідні дані та записує до неї і результат. Оперативної називається тому, що процесорові практично не приходиться чекати при читанні даних з пам'яті або записувати у пам'ять, але дані в пам'яті зберігаються тільки поки комп'ютер включений. Таким чином процесор має доступ до будь-якої ділянки оперативної пам’яті. При вимиканні вміст оперативної пам'яті втрачається. Тому такий пристрій називають **пам'ять з довільним доступом або RAM-Random Access Memory**. RAM-пам’ять зазвичай задається з в Мегабайтах (МБ) і має типові величини – 1 МБ, 2 МБ, 4 МБ, 8 МБ, 16 МБ, 32 МБ, 64 Мб, 128 Мб, 256 Мб, 512 Мб, 1024 Мб [1, 5].

Оперативна пам'ять (RAM – Random Access Memory) – це масив кристалічних комірок, які здатні зберігати дані. Існує багато різних типів оперативної пам'яті, але з погляду фізичного принципу дії розрізняють *динамічну пам'ять (DRAM)* і *статичну пам'ять (SRAM).*

Є декілька видів оперативної пам’яті:

DRAM (Dynamic Random Access Memory) – динамічна оперативна пам’ять, що реалізується в модулях SIMM (*Single In-Line Memory Module*);

SDRAM (Synchronous DRAM)– синхронізована динамічна (швидкодіюча, дорога) – оперативна пам’ять, що реалізується в модулях DIMM (*Dual In-line Memory Module),* передає інформацію у високошвидкісних пакетах, який використовує високошвидкісний синхронізований інтерфейс;

SRAM (**Static Random Access Memory**) – статична оперативна пам’ять (**кеш-пам’ять**), це швидкодіюча оперативна пам’ять, що вирисовується в модулях кеш-пам’яті, вона також може працювати на тій же частоті, що і сучасні процесори;

Комірки динамічної пам'яті (DRAM) можна представити у вигляді мікроконденсаторів, здатних накопичувати заряд на своїх обкладинках. Це найбільш поширений і економічно доступний тип пам'яті.

Комірки статичної пам'яті (SRAM) можна представити як електронні мікроелементи – ***тригери****,* що складаються з декількох транзисторів. У тригері зберігається не заряд, а стан *(включений/виключений),* тому цей тип пам'яті забезпечує вищу швидкодію, хоча технологічно він складніше і, відповідно, дорожче.

Мікросхеми динамічної пам'яті використовують в якості основної оперативної пам'яті комп'ютера. *Мікросхеми статичної пам'яті використовують як допоміжну пам'ять* (так звану *кеш-пам'ять),* призначену для оптимізації роботи процесора.

Основна або оперативна пам'ять є одним з критичних ресурсів комп'ютерів, які в даний час забезпечують 32-х та 64-х розрядну адресацію. Завжди першою функцією будь-якої операційної системи є забезпечення розподілу основної пам'яті між конкуруючими процесами користувачів. Одним з методів управління оперативною пам'яттю є віртуалізація, реалізація якої здійснюється апаратною частиною механізму віртуальної пам'яті. Це означає, що адреса пам'яті, яка виробляється командою, інтерпретується апаратурою не як реальна адреса основної пам'яті, а як деяка структура, яка має поля, які обробляються різним чином. Тому пряма адресація означає, що адреса комірки може бути аргументом інструкцій процесора. Тому розглянемо режими роботи процесора.

Режими роботи мікропроцесора

Реальний режим. Це режим роботи перших 16-бітових мікропроцесорів. Наявність його обумовлена тим, що необхідно забезпечити в нових моделях мікропроцесорів функціонування програм, розроблених для старих моделей.

Захищений режим (protected mode). Означає, що паралельні обчислення можуть бути захищені програмно-апаратним шляхом. Дозволяє повністю використовувати всі можливості, що надаються мікропроцесором. Все сучасні багатозадачні ОС працюють саме в цьому режимі. Цей режим створений для роботи декількох незалежних програм. Для забезпечення спільної роботи декількох завдань необхідно захистити їх від взаємного впливу, взаємодія завдань повинна регулюватися. Програми, розроблені для реального режиму, не можуть функціонувати в захищеному режимі. (Фізична адреса формується по інших принципах.)

Режим віртуального 8086 (V86, VM86, іноді просто віртуальний режим) - режим адресації процесорів сімейства x86 сумісний з процесором Intel 8086. Перехід в цей режим можливий, якщо мікропроцесор вже знаходиться в захищеному режимі. Це підрежим захищеного режиму, який був першою спробою корпорації Intel впровадити в свої процесори технології апаратної віртуалізації. Вперше з'явився в процесорі 80386.

Організація оперативної пам'яті

Пам'ять комп'ютера, або основна пам'ять, або оперативна пам'ять, або ОЗП (оперативний запам'ятовуючий пристрій), або **пам'ять з довільним доступом або RAM-random access memory**) – це фізична пам'ять, до якої мікропроцесор має доступ по шині адреси. **RAM** є основною пам'яттю для тимчасового зберігання програм та даних. *Оперативна пам'ять організована як* одновимірний масив елементів пам'яті розміром в 1 байт. *Кожному байту відповідає унікальна адреса*

(номер), який називається фізична адреса. Діапазон значень фізичних адрес залежить від розрядності шини адреси мікропроцесора.

Для 16-розрядних процесорів i8086/8088 адресна шина 20 розрядна, для процесора 80286 – 24 розрядна, для процесорів 386DX, 486DX, Pentium, Pentium MMX – 32 розрядна, для процесорів Pentium Pro/II, AMD Athlon/Duron – 36 розрядна, для процесорів Pentium III/IV, AMD Athlon 4 / Duron 4 – 64 розрядна.

Для процесорів i8086/8088 діапазон значень адрес знаходиться в межах від *0 до 2^{16}* (або 64 Кб, 2^{16} – 1=65535), з іншого боку адресна шина має 20 розрядів, тобто максимально можлива адреса дорівнює FFFFFh(2^{20} -1 = 1 048 575), це і є розмір адресного простору ОП і становить 1 Мб пам’яті [1, 5].

Для процесорів 80286 адресна шина 24 розрядна, тобто максимально можлива адреса дорівнює FFFFFFh (2^{24} -1 = 16 777 215), тобто розмір адресного простору ОП - 16 Мб.

Для *i486 і Pentium діапазон* знаходиться в межах від *0 до FFFFFFFFh(2^{32} – 1, це становить4 294 967 295 = **4 Гб***). Для процесорів *Pentium Pro/II* цей діапазон ширше — *від 0 до FFFFFFFFFh(2^{36} – 1, це становить **64 Гб**).*

Для процесорів Pentium III/IV, AMD Athlon 4 / Duron 4 діапазон знаходиться в межах від *0 до FFFFFFFFFFFFFFFFh(2^{64} – 1).*

Механізм управління пам’яттю є повністю апаратним, тобто програма сама не може сформувати фізичну адресу пам’яті на адресній шині.

Оперативна пам’ять відіграє особливу роль – це ресурс, який потребує ретельного управління з боку мультипрограмної операційної системи. Розподілу підлягає оперативна пам’ять, яка не зайнята операційною системою Це обумовлено тим, що програма може виконаються тільки в тому разі, якщо вона знаходиться в пам'яті. Пам'ять розподіляється між користувацькими і системними програмами ОС.

Функції ОС по управлінню пам'яттю

До основних функцій ОС по управлінню пам'яттю належать:

- облік вільної і зайнятої пам'яті,
- виділення пам'яті процесам і її звільнення,
- витіснення кодів і даних процесів на диск, коли не вистачає пам'яті і повернення у відповідне місце пам’яті,
- налаштування адрес на конкретну область фізичної пам'яті,
- дефрагментація,
- захист пам'яті.

Типи адрес

Для ідентифікації команд програми і даних використовуються адреси.

Адреси підрозділяються на:

- *символьні імена,* їх присвоює програміст при написанні програми на алгоритмічній мові або на асемблері (наприклад, мітки);
- *віртуальні адреси*, їх формує транслятор, надаючи змінним та командам віртуальні (умовні) адреси, оскільки невідомо у яке місце оперативної пам’яті буде завантажена програма. За замовчуванням вважається, що програма буде розташована з нульової адреси;

- *фізичні адреси* відповідають номерам комірок оперативної пам'яті, в яких будуть розташовані команди і дані.

Існує *два основні типи представлення віртуальних адрес*:

- *лінійна,* при якій адреса початку завжди дорівнює нулю, а адреса є цілим числом, тобто це зсув відносно початку віртуального адресного простору
- *розподіл на сегменти*, тобто віртуальний адресний простір поділяється на сегменти, при якому адреса – *це пара чисел (n, m), де n- номер сегменту, m- зсув.*

Поняття віртуальної пам'яті. *Віртуальна пам'ять* - метод управління пам'яттю, що реалізується за допомогою апаратного та програмного забезпечення комп'ютера. Вона *відображає перетворення віртуальних адрес*, які використовуються програмами (процесами), *у фізичні адреси пам'яті комп'ютера*. Такі перетворення забезпечують захист пам'яті та відсутність прив'язки процесу до адрес фізичної пам'яті [6].

Основна пам'ять представляється у вигляді безперервного адресного простору або набору змішаних безперервних сегментів. Операційна система здійснює управління віртуальним адресним простором та співвідношенням оперативної пам'яті з віртуальною. Програмне забезпечення в операційній системі може розширити ці можливості, щоб забезпечити віртуальний адресний простір, який може збільшити обсяг оперативної пам'яті і таким чином мати більше пам'яті, ніж є в комп'ютері. Віртуальна пам'ять дозволяє модифікувати ресурси пам'яті, зробити обсяг оперативної пам'яті більше для того, щоб користувач розмістив туди якомога більше програм та реально заощадив час і підвищив ефективність своєї праці. Винахід віртуальної пам'яті зробив величезний внесок у розвиток сучасних технологій, суттєво полегшив роботу як професійного програміста, так і звичайного користувача для більш ефективного вирішення задач на комп'ютері.

Переваги віртуальної пам'яті. До основних переваг віртуальної пам'яті відносяться:

- позбавлення програміста від необхідності управління загальним простором пам'яті,
- підвищення безпеки використання програмного забезпечення за рахунок виділення пам'яті,
- можливість мати більше пам'яті, ніж це може бути фізично доступним на комп'ютері.

Властивості віртуальної пам'яті. Віртуальна пам'ять спрощує програмування застосунків:

- приховуючи фрагментацію фізичної пам'яті;
- усуваючи необхідність в програмі для обробки накладань у явному вигляді;
- коли кожний процес запускається у своєму власному виділеному адресному просторі, немає необхідності перемістити код програми або отримати доступ до пам'яті з відносною адресацією.

Віртуалізація пам'яті може розглядатися як узагальнення поняття віртуальної пам'яті. Майже всі реалізації віртуальної пам'яті поділяють віртуальний адресний простір (ВАП) на сторінки, блоки суміжних адрес віртуальної пам'яті.

При роботі машини з віртуальною пам'яттю використовуються методи сторінкової і сегментної організації пам'яті.

Поняття, пов'язані з віртуальними адресами. Сукупність віртуальних адрес складає *віртуальний адресний простір*. *Віртуальний адресний простір* (ВАП) визначається розрядністю комп'ютера. Для 32-розрядних – це максимум FFFFFFFF, що складає 4 Гб.

Призначений ВАП необхідний процесу для роботи. Його також називають образом процесу. Призначений ВАП може перевищувати фізичний обсяг пам'яті. На цьому заснований механізм віртуальної пам'яті.

ВАП і віртуальна пам'ять – це різні механізми для ОС. ОС може підтримувати ВАП, але механізм віртуальної пам'яті може бути при цьому відсутнім. Наприклад, у разі перевищення фізичної пам'яті над ВАП будь-якого процесу.

Системна і користувацька частини ВАП має по 2 Гб. На рис. 3.1 зображені дві типові структури віртуального адресного простору, які підтримуються 32-розрядними версіями Windows. (Параметр *increaseuserva* дозволяє виконуваним образам зі встановленим прапором обробки розширеного адресного простору використовувати адреси від 2 до 3 Гбайт) [7, 15].

Рис. 3.1. Типова структура адресного простору в 32-розрядних версіях Windows

64-розрядні версії Windows надають для процесів значно більше адресний простір: 128 Тбайт в Windows 8.1, Server 2012 R2 і пізніших системах. На рис. 3.2 зображено спрощене уявлення структури адресного простору в 64-розрядних системах. Слід враховувати, що ці розміри не визначаються архітектурними обмеженнями таких платформ. 64 біта адресного простору - це 2^{64} або 16 Ебайт (1

Ексабайт=10^{18} байт або квінтіліон) (1 Ебайт = 1024 Пбайт, 1 Петабайт=10^{15} байт або квадріліон) або 1 048 576 Тбайт (1 Терабайт=10^{12} або 2^{40}), але сучасне 64-розрядне обладнання обмежує його меншими значеннями.

На рис. 3.2 наведена типова структура адресного простору у 64-розрядних версіях Windows [7, 15].

Рис. 3.2. Типова структура адресного простору в 64-розрядних версіях Windows

Управління віртуальною пам’яттю здійснюється *диспетчером пам’яті*.

Крім управління віртуальною пам'яттю, диспетчер пам'яті надає базовий набір сервісних функцій, на основі яких будуються різні підсистеми середовища Windows. До числа таких сервісних функцій відносяться:

- файли, відображені в пам'ять (у внутрішній реалізації називаються об'єктами секцій);
- пам'ять з копіюванням при запису і підтримка застосунків, що використовують великий розріджений адресний простір.

Диспетчер пам'яті також надає засоби для виділення і використання великих обсягів фізичної пам'яті, яка може одночасно відображатися у віртуальний адресний простір процесу - наприклад, в 32-розрядних системах з більш ніж 3 Гбайт фізичної пам'яті.

Компоненти диспетчера пам'яті. Диспетчер пам'яті є частиною виконуючого середовища Windows, а отже, існує в файлі Ntoskrnl.exe. Це найбільший компонент виконуваної системи, що вказує на його важливість і складність. Диспетчер пам'яті складається з наступних компонентів:

1. Набір сервісних функцій виконуючої системи для виділення, звільнення віртуальної пам'яті й управління нею, більшість з яких доступні через Windows API або інтерфейси драйверів пристроїв режиму ядра.

2. Оброблювач некоректного перетворення і помилок доступу для обробки винятків управління пам'яттю, виявлених на апаратному рівні, і розміщення віртуальних сторінок в пам'яті від імені процесу.

3. Шість ключових функцій верхнього рівня, кожна з яких виконується в одному з шести потоків режиму ядра в процесі System:

3.1. Диспетчер набору балансування (KeBalanceSetManager, пріоритет 17). Викликає внутрішню функцію диспетчера робочих наборів (*MmWorkingSetManager*) кожну секунду, а також при падінні обсягу вільної пам'яті нижче деякого порога.
3.2. Управління підвантаженням процесу/стека (*KeSwapProcessOrStack,* пріоритет 23). Виконує підвантаженням і вивантаження стека процесів і потоків ядра.
3.3. Запис змінених сторінок (*MiModifiedPageWriter*, пріоритет 18). Записує «брудні» сторінки зі списку модифікацій до відповідних сторінкових файлів.
3.4. Запис відображених сторінок (*MiMappedPageWriter*, пріоритет 18). Записує «брудні» сторінки зі списку модифікацій на диск або в віддалене сховище.
3.5. Потік розіменування сегментів (*MiDereferenceSegmentThread,* пріоритет 19). Відповідає за скорочення кеша, а також за зростання і скорочення сторінкового файлу.
3.6. Потік обнулення сторінок (*MiZeroPageThread*, пріоритет 0). Заповнює нулями сторінки з вільного списку, щоб для задоволення майбутніх запитів обнулення був доступний кеш нульових сторінок.

Фрагментація пам'яті. Завдяки віртуальній пам'яті фізична пам'ять адресного простору процесу може бути *фрагментованою*, оскільки основний обсяг пам'яті, яку займає процес, більшу частину часу залишається вільним. Є так зване правило «дев'яносто до десяти», або правило локалізації, яке стверджує, що 90% звертань до пам'яті у процесі припадає на 10% його адресного простору. Адреси можна переміщувати так, щоб основній пам'яті відповідали тільки ті розділи адресного простору процесу, які дійсно використовуються у конкретний момент.

При цьому невикористовувані розділи адресного простору можна ставити у відповідність повільнішій пам'яті, наприклад простору на жорсткому диску, а в цей час інші процеси можуть використовувати основну пам'ять, у яку раніше відображалися адреси цих розділів. Коли ж розділ знадобиться, його дані завантажують з диска в основну пам'ять, можливо, замість розділів, які стали непотрібними в конкретний момент (і які, своєю чергою, тепер збережуться на диску). Дані можуть зчитуватися з диска в основну пам'ять під час звернення до них. У такий спосіб можна значно збільшити розмір адресного простору процесу і забезпечити виконання процесів, що за розміром перевищують основну пам'ять.

Однак якщо для більшості звернень до пам'яті система буде змушена дійсно звертатися до диска (який у десятки тисяч разів повільніший, ніж основна пам'ять), працювати із такою системою стане практично неможливо.

Ще однією проблемою є фрагментація пам'яті, яка виникає за ситуації, коли неможливо використати вільну пам'ять. Розрізняють ***зовнішню і внутрішню фрагментацію пам'яті***.

Зовнішня фрагментація зводиться до того, що внаслідок виділення і наступного звільнення пам'яті в ній утворюються вільні блоки малого розміру – *діри* (*holes*). Через це може виникнути ситуація, за якої неможливо виділити неперервний блок пам'яті розміру *N,* оскільки немає жодного неперервного вільного блоку, розмір якого *S>N,* хоча загалом обсяг вільного простору пам'яті перевищує *N.*

Внутрішня фрагментація зводиться до того, що за запитом виділяють блоки пам'яті більшого розміру, ніж насправді будуть використовуватися, у результаті всередині виділених блоків залишаються невикористовувані ділянки, які вже не можуть бути призначені для чогось іншого.

Логічна і фізична адресація пам'яті. Найважливішими поняттями концепції віртуальної пам'яті є логічна і фізична адресація пам'яті.

Логічна або віртуальна адреса - адреса, яку генерує програма, запущена на деякому процесорі. Адреси, що використовують інструкції конкретного процесора, є логічними адресами. Сукупність логічних адрес становить логічний адресний простір.

Фізична адреса - адреса, якою оперує мікросхема пам'яті. Прикладна програма в сучасних комп'ютерах ніколи не має справи з фізичними адресами. Спеціальний апаратний пристрій MMU (*memory management unit* - пристрій керування пам'яттю) відповідає за перетворення логічних адрес у фізичні (рис. 3.3). Сукупність усіх доступних фізичних адрес становить фізичний адресний простір. Отже, якщо в комп'ютері є мікросхеми на 128 Мб пам'яті, то саме такий обсяг пам'яті адресують фізично. Логічно зазвичай адресують значно більше пам'яті.

Рис. 3.3. Перетворення логічних адрес у фізичні

Методи розподілу пам'яті без використання дискового простору

Усі методи управління пам'яттю можуть бути розділені на два класи [16]:

- методи, які використовують переміщення процесів між оперативною пам'яттю і диском, тобто *без використання зовнішньої пам'яті*;
- методи, які не роблять цього, тобто *з використання зовнішньої пам’яті*.

Алгоритми розподілу пам'яті без використання зовнішньої пам'яті поділяються на:

- розподіл пам'яті фіксованими розділами;
- розподіл пам'яті динамічними розділами;
- розподіл пам'яті з переміщуваними розділами.

Розподіл пам'яті фіксованими розділами

Найпростішим способом управління оперативною пам'яттю є розділення її на декілька розділів фіксованої величини. Це може бути виконано уручну оператором під час старту системи або під час її генерації. Чергове завдання, що поступило на виконання, поміщається або в загальну чергу задач або в чергу до деякого розділу.

Розподіл пам'яті розділами змінної величини (динамічними)

При використанні даного методу пам'ять в початковий момент часу вважається вільною (за винятком пам'яті, відведеної під ОС). Кожному завданню (процесу), що знов поступає, виділяється необхідна нею пам'ять. Якщо достатній об'єм пам'яті відсутній, то завдання не приймається на виконання і стоїть в черзі. Після завершення завдання (процесу) пам'ять звільняється, і на це місце може бути завантажена інше завдання. Таким чином, в довільний момент часу оперативна пам'ять є випадковою послідовністю зайнятих і вільних ділянок (розділів) довільного розміру.

Функції ОС при реалізації даного методу управління пам'яттю є: :

- ведення таблиць вільних і зайнятих ділянок;
- пошук ділянки;
- завантаження і коректування таблиць вільних і зайнятих областей;
- після завершення завдання (процесу) коректування таблиць вільних і зайнятих областей.

У порівнянні з методом розподілу пам'яті фіксованими розділами даний метод є набагато гнучким, але йому властивий дуже серйозний недолік – високий рівень фрагментації пам'яті. Фрагментація - це наявність великого числа несуміжних ділянок вільної пам'яті дуже маленького розміру (фрагментів). Настільки маленького, що жодна з програм, що знов поступають, не може поміститися ні в одній з ділянок, хоча сумарний обсяг фрагментів може скласти значну величину, що набагато перевищує необхідний об'єм пам'яті.

Розподіл пам'яті з переміщуваними розділами

У цьому методі розробники спробували врахувати переваги і недоліки попередніх алгоритмів. Одним з методів боротьби з фрагментацією є переміщення всіх зайнятих ділянок у бік старших або у бік молодших адрес, так, щоб вся вільна пам'ять утворювала єдину вільну область - дефрагментація. Ця процедура називається "стисненням". Стиснення може виконуватися або при кожному завершенні завдання, або тільки тоді, коли для завдання, що знов поступило, немає вільного розділу достатнього розміру. У першому випадку потрібно менше обчислювальної роботи при коректуванні таблиць, а в другому - рідше виконується процедура стиснення. Оскільки програми переміщаються по оперативній пам'яті в ході свого виконання, то перетворення адрес з віртуальної форми у фізичну повинне виконуватися динамічним способом.

Такий метод був застосований в ранніх версіях OS/2. Недоліком цього методу є низька продуктивність.

Алгоритми розподілу пам'яті з використанням зовнішньої пам'яті

Для повного завантаження процесора можуть знадобитися іноді сотні інтерактивних завдань. Всі вони повинні бути розташовані в пам'яті, велика частина яких знаходиться в стані очікування. Логічно було б на час очікування, у разі браку фізичної пам'яті, витісняти їх на диск, а коли необхідно, повертати в пам'ять. Така підміна (віртуалізація) оперативної пам'яті дисковою пам'яттю істотно підвищує рівень мультипрограмування. Важливо, що всі дії з переміщення відбуваються автоматично, без участі програміста. Для віртуалізації застосовуються два основні підходи:

Свопінг – образ процесу вивантажується на диск і повертається в пам'ять цілком. Часто називається підкачкою.

Віртуальна пам'ять – образ процесу вивантажується на диск і повертається в пам'ять частинами (сегментами, сторінками...)

Реалізація віртуальної пам'яті представлена трьома класами:

- *сторінковий розподі;*
- *сегментний;*
- *сегментно-сторінковий розподіл.*

Сторінковий розподіл

При сторінковому розподілі віртуальна пам'ять поділяється на частини однакового і фіксованого для даної системи розміру, що називаються віртуальними сторінками. Вся оперативна пам'ять також поділяється на частини такого ж розміру, що називаються фізичними сторінками. Розмір сторінки дорівнюється ступеню двійки: 512, 1024, 2048, 4096 і так далі (числа кратні 256) (рис. 3.4) [15].

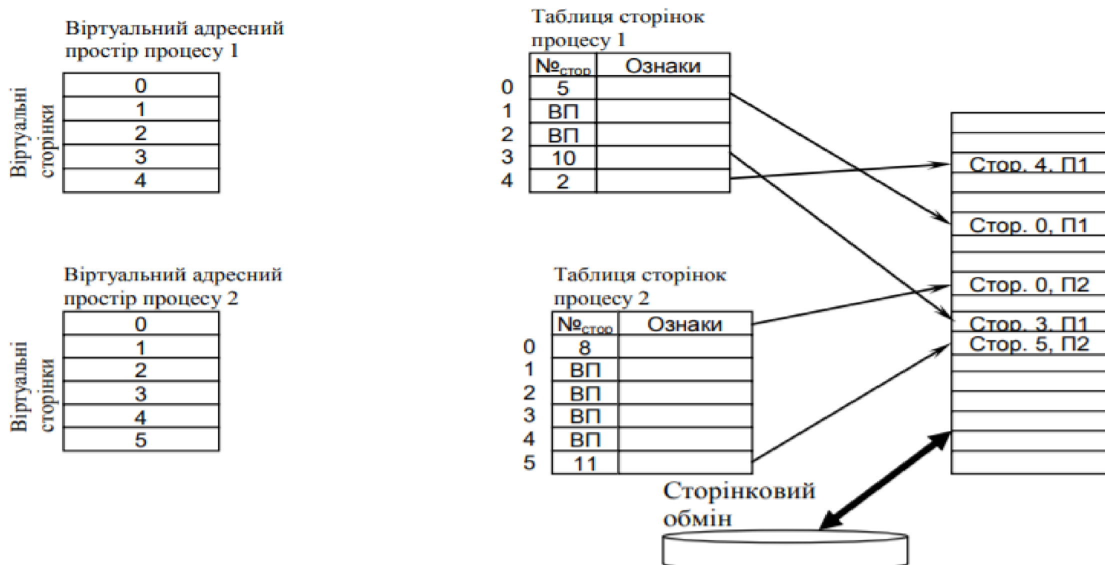


Рис. 3.4. Сторінковий розподіл пам'яті

Адреса сторінки входить в контекст процесу (це інформація про стан регістрів і програмного лічильника, режимів роботи процесора, покажчиків на відкриті файли, інформація про незавершені операції введення-виведення, коди помилок виконуваних даним процесом системних викликів, адреса сторінки і так далі).

Таблиця сторінок розміщується в оперативній пам’яті та складається з дескрипторів. *Кожний дескриптор містить*:

- номер фізичної таблиці;
- ознака присутності в ОЗП (формується апаратно);
- ознака модифікації (формується апаратно);
- ознака звернення (формується апаратно).

Віртуальна адреса, яка представлена парою (p, S_v),

- де p – порядковий номер віртуальної сторінки процесу (нумерація починається з 0);

- S_v – зсув в межах віртуальної сторінки.

Ця віртуальна адреса (p, S_v) перетворюється в фізичну адресу (n, S_f), де

- n – номер фізичної сторінки;

- S_f – зсув у фізичній сторінці.

Обсяг сторінки дорівнює ступеню 2^k, тоді зсув (s) можна отримати відділенням k – молодших розрядів у двійковому запису адреси. Старші розряди – це двійковий запис номера сторінки.

Наприклад, якщо розмір сторінки = 1Кб (2^{10}), то

$50718 = 101\ 000\ 111\ 001_2$, 10_2 – номер сторінки, а зсув відносно її початку $1\ 000\ 111\ 001_2$ байтів (рис. 3.5).

Рис. 3.5. Двійкове представлення адрес

На рис. 3.6 наведена схема перетворення віртуальних адрес у фізичні.

Апаратно з регістра отримується адреса таблиці (АТ) сторінок. На підставі номера сторінки Р і довжини запису L визначається адреса дескриптора (A=AT+P*L) [15].

З таблиці отримується номер фізичної сторінки N. До номера N приєднується зсув S. Розмір сторінок, (часто 4096) впливає на розмір таблиць, а це у свою чергу впливає на продуктивність. Для усунення цього недоліку ВАП може поділятися на розділи, а в кожному розділі формується своя таблиця сторінок. Цей варіант прискорює пошук.

Рис. 3.6. Схема перетворення віртуальних адрес у фізичні

Сегментний розподіл

При сторінковому розподілі віртуальний адресний простір поділяється на рівні частини механічно - сегменти, без урахування даних за своєю сутністю. В одній сторінці можуть одночасно опинитися код програми і вихідні дані. Такий підхід не дозволяє забезпечити роздільну обробку, наприклад захист, сумісний доступ і так далі. На рис. 3.7 наведена схема перетворення віртуальної адреси на фізичну при сегментному розподілі пам'яті [15].

Поділ віртуального адресного простору на сегменти здійснюється компілятором на основі на основі прийнятих у системі угод або вказівок програміста або за замовчуванням. Для 32-розрядної організації процесора максимальний розмір сегменту дорівнює 4 Гбайт, а діапазон віртуальних адрес для кожного сегмента становить 00000000_{16} … $FFFFFFFF_{16}$.

Віртуальна адреса – це пара чисел: номером сегмента та лінійна віртуальна адреса.

Розбиття адресного простору на частини відповідно своєї сутності усуває ці недоліки і називається сегментним розподілом, тобто сегмент – це частина

віртуального адресного простору довільного розміру. Приклади сегментів: код програми, масив початкових даних та ін.

На етапі створення процесу ОС створює таблицю сегментів процесу, аналогічну таблиці сторінок.

До недоліків сегментного розподілу можна віднести наступні:

- використання операції складання при формуванні фізичної адреси призводить до зниження продуктивності;
- надмірність. Оскільки сегмент в загальному випадку може бути більше сторінки, то одиниця обміну між ОЗП і диском є більшою, що приводить до уповільнення роботи.

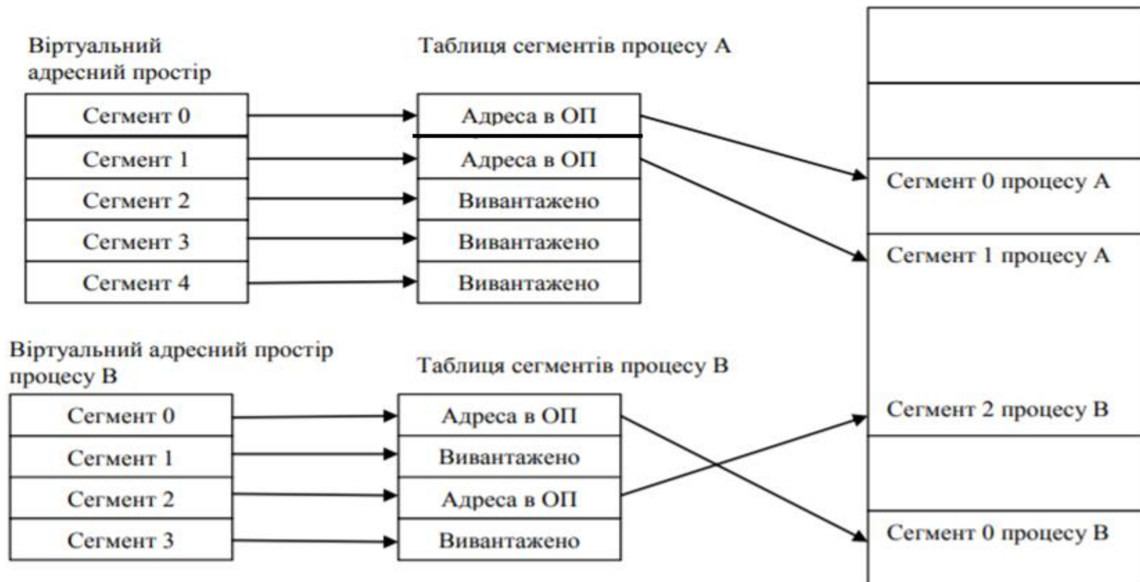


Рис. 3.7. Схема перетворення віртуальних адрес у фізичні при сегментному розподілі пам’яті

Сегментно - сторінковий розподіл

Цей метод є комбінацією сторінкового і сегментного механізмів управління пам'яттю і направлений на реалізацію переваг обох підходів. Віртуальна пам'ять поділяється на сегменти, а кожний сегмент - на віртуальні сторінки, які нумеруються в межах сегменту. Оперативна пам'ять поділяється на фізичні сторінки. Завантаження процесу виконується операційною системою посторінково, при цьому частина сторінок розміщується в оперативній пам'яті, а частина на диску. Для кожного сегменту створюється своя таблиця сторінок, структура якої повністю співпадає із структурою таблиці сторінок, використовуваної при сторінковому розподілі. Для кожного процесу створюється таблиця сегментів, в якій вказуються адреси таблиць сторінок для всіх сегментів даного процесу. Адреса таблиці сегментів завантажується в спеціальний регістр процесора, коли активізується відповідний процес.

Всі сучасні ОС використовують саме такий спосіб організації.

Для доступу до пам'яті необхідно:

1. Обчислити адресу дескриптора сегменту і прочитати його.

2. Обчислити адресу елементу таблиці сторінок цього сегменту і витягувати з пам'яті необхідний елемент.

3. До номера (адреси) фізичної сторінки приписати номер (адреса) комірки в сторінці.

Недоліком такого методу є затримка у доступі до пам'яті (у три рази більше, ніж при прямій адресації). Щоб уникнути цього вводиться кешування (кеш будується за асоціативним принципом).

В UNIX System V Release 4 реалізована сегментно-сторінкова модель пам'яті в її традиційному вигляді. Разом з механізмом управління сторінками використовується і механізм **свопінгу**, коли на диск виштовхуються всі сторінки будь-якого процесу. Свопінг застосовується в "передаварійних" ситуаціях, коли розмір вільної оперативної пам'яті зменшується до деякого заданого порогу, так що робота всієї системи дуже ускладнюється.

Кешування

Кеш - це пам'ять з більшою швидкістю доступу, яка призначена для прискорення звернення до даних, що містяться постійно в пам'яті з меншою швидкістю доступу (далі «основна пам'ять»). Кешування застосовується центральним процесором, жорсткими дисками, браузерами і веб-серверами.

Кеш складається з набору записів. Кожний запис асоційований з елементом даних або блоком даних (невеликої частині даних), який є копією елементу даних в основній пам'яті. Кожний запис має ідентифікатор, що визначає відповідність між елементами даних в кеші і їх копіями в основній пам'яті.

Коли клієнт кеша (центральний процесор управління, веб-сервер-браузер, операційна система) звертається до даних, перш за все досліджується кеш. Якщо в кеші знайдений запис з ідентифікатором, який співпадає з ідентифікатором затребуваного елементу даних, то використовуються елементи даних в кеші. Такий випадок називається попаданням кеша. Якщо в кеші не знайдено записів, що містять затребуваний елемент даних, то цей елемент читається з основної пам'яті в кеш і стає доступним для подальших звернень. Такий випадок називається промахом кеша. Відсоток звернень до кеша, коли в ньому знайдений результат, називається рівнем попадань або коефіцієнтом попадань в кеш.

У системах, що складаються з клієнтів і серверів, потенційно є чотири різні місця для зберігання файлів і їх частин: диск сервера, пам'ять сервера, диск клієнта (якщо є) і пам'ять клієнта. Найбільш доцільним місцем для зберігання всіх файлів є диск сервера. Він зазвичай має велику ємкість, і файли стають доступними всім клієнтам. Крім того, оскільки в цьому випадку існує тільки одна копія кожного файлу, то не виникає проблеми узгодження станів копій.

Проблемою при використанні диска сервера є продуктивність. Перед тим, як клієнт зможе прочитати файл, файл повинен бути переписаний з диска сервера в його оперативну пам'ять, а потім переданий по мережі в пам'ять клієнта. Обидві передачі займають час.

Значне збільшення продуктивності може бути досягнуте за рахунок кешування файлів в пам'яті сервера. Потрібні алгоритми для визначення, які файли або їх частини слід зберігати в кеш-пам'яті.

При виборі алгоритму повинні вирішуватися два завдання. По-перше, якими одиницями оперує кеш. Цими одиницями можуть бути або дискові блоки, або цілі файли. Якщо це цілі файли, то вони можуть зберігатися на диску безперервними областями (принаймні у вигляді великих ділянок), при цьому зменшується число обмінів між пам'яттю і диском а, отже, забезпечується висока продуктивність. Кешування блоків диска дозволяє ефективніше використовувати пам'ять кеша і дисковий простір.

По-друге, необхідно визначити правило заміни даних при заповненні кеш-пам'яті. Тут можна використовувати будь-який стандартний алгоритм кешування, наприклад, алгоритм LRU (*least recently used*), відповідно з яким витісняється блок, до якого найдовше не було звернення.

Кеш-пам'ять на сервері легко реалізується і абсолютно прозора для клієнта. Оскільки сервер може синхронізувати роботу пам'яті і диска, з погляду клієнтів існує тільки одна копія кожного файлу, так що проблема узгодження не виникає.

Хоча кешування на сервері виключає обмін з диском при кожному доступі, все ще залишається обмін по мережі. Існує тільки один шлях позбавитися від обміну по мережі - це кешування на стороні клієнта, яке, проте, породжує багато складнощів.

У більшості систем використовується кешування в пам'яті клієнта. Іншим місцем кешування є ядро. Третім варіантом організації кеша є створення окремого процесу призначеного для користувача рівня - кеш-менеджер.

Ряд моделей центральних процесорів (ЦП) мають власний кеш для того, щоб мінімізувати доступ оперативної пам'яті (ОЗУ), яка повільніше, ніж регістри. Кеш-пам'ять може давати значний виграш в продуктивності, у разі коли тактова частота ОЗУ значно менше тактової частоти ЦП. Тактова частота для кеш-пам'яті зазвичай не набагато менше частоти ЦП.

Існує **ієрархічна структура пам'яті** [15]:

- на верхньому рівні знаходяться *регістри процесора*, доступ до яких здійснюється найшвидше;
- далі йде *кеш-пам'ять*, обсяг якої може становити від 32 Кб до декількох мегабайтів;
- потім йде *основна пам'ять*, яка може вміщувати від 16 Мб до десятків гігабайтів;
- далі йдуть *магнітні диски*;
- потім *оптичні диски* для зберігання архівів.

У міру просування зверху вниз по ієрархії змінюються три параметри:

- *збільшується час доступу* – доступ до регістрів займає декілька наносекунд, до кеш-пам'яті дещо більше, до основної пам'яті - декілька десятків наносекунд, до дисків – 10 мкс, до оптичних дисків – секунди;
- *зростає обсяг пам'яті* – регістри можуть містити 128 Кб, кеш-пам'ять – декілька мегабайтів, основна пам'ять – десятки тисяч мегабайтів, магнітні диски – до десятків гігабайтів, магнітні стрічки та оптичні диски зберігаються окремо від комп'ютера, тому їх сукупний обсяг обмежується фінансовими можливостями власника;

- *збільшується кількість бітів,* які користувач отримує за долар – вартість обсягу основної пам’яті складає декілька доларів за мегабайт, магнітних дисків – декілька центів за мегабайт, оптичних дисків – декілька доларів за гігабайт).

Кеш-пам’ять має сукупність рядків (cache-lines), кожний з яких складається з фіксованої кількості адресованих одиниць пам’яті (байтів, слів), що зберігаються в основній пам’яті як комірки з послідовними адресами. Типовий розмір кеш-рядка 16, 64 128, 256 байтів.

Плоска модель пам'яті

Якщо вважати, що завдання складається з одного сегменту, який, у свою чергу, розбитий на сторінки, то фактично ми отримуємо тільки один сторінковий механізм роботи з віртуальною пам'яттю. Це підхід називається плоскою пам'яттю.

Переваги:

- при використанні плоскої моделі пам'яті спрощується створення і ОС, і систем програмування;
- зменшуються витрати пам'яті на підтримку системних інформаційних структур.

Плоска модель пам'яті – це метод організації адресного простору оперативної пам'яті обчислювальних пристроїв. У плоскій моделі код і дані використовують один і той же адресний простір. Для 16-бітових процесорів плоска модель пам'яті дозволяє адресувати 64 Кб оперативної пам'яті; для 32-бітових процесорів 4 ГБ, для 64-бітних - до 16 ексабайт (для amd64 розмір обмежений 256 ТБ.

У абсолютній більшості сучасних 32 - 64 розрядних операційних систем (для мікропроцесорів Intel) використовується плоска модель пам'яті *Flat*.

Контрольні питання:

1. Що таке оперативна пам’ять?
2. Які вам відомі види оперативної пам’яті?
3. Компоненти диспетчера пам'яті.
4. Що таке логічна адреса пам’яті?
5. Що таке фізична адреса пам’яті?
6. Які вас відомі методи розподілу пам'яті без використання дискового простору?
7. Що таке свопінг?
8. Що таке віртуальна пам’ять?
9. У чому полягає сегментний розподіл пам’яті?
10. У чому полягає сторінковий розподіл пам’яті?
11. Сутність сегментно-сторінкового розподілу пам’яті.
12. Що таке кеш-пам’ять?
13. Поясніть ієрархічну структуру пам’яті.
14. Що таке плоска модель пам’яті?

Тема 3. Підсистема організації задач, процеси, потоки

Лекція 4. Керування процесами і потоками

Базові поняття: процеси і потоки в сучасних ОС. Багатопотоковість. Складові елементи процесів і потоків. Потік ядра. Потік користувача. Стани процесів і потоків. Планування задач. Таблиці розподілу ресурсів. Таблиці розподілу процесів. Таблиці розподілу потоків. Керуючий блок процесу. Керуючий блок потоку. Образ процесу. Образ потоку. Організація перемикання контексту. Створення процесу і потоку. Синхронне та асинхронне виконання процесів. Взаємодія потоків. Тупики. Критична секція.

Базові поняття: процеси і потоки в сучасних ОС

В сучасній операційній системі одночасно виконуються код ядра, який належить до різних його підсистем, і код програм користувача. При цьому відбуваються різні дії: одні програми і підсистеми виконують інструкції процесора, інші зайняті введенням-виведенням, ще деякі очікують на запити від користувача або інших застосувань. Програма – це статична послідовність команд. Для опису виконання програмного коду в операційній системі використовуються дві основні абстракції ОС— процеси і потоки. Процес – це фундаментальний засіб розподілу роботи і ресурсів в ОС, який дозволяє ефективно використовувати процесор [1, 2].

Під *процесом* розуміють абстракцію ОС, яка об'єднує все необхідне для виконання однієї програми в певний момент часу (*процес – це програма у стадії виконання, тобто це програма та системні ресурси, які необхідні для її роботи*). Однозначна відповідність між програмою і процесом встановлюється тільки в конкретний момент часу: *один процес у різний час може виконувати код декількох програм, код однієї програми можуть виконувати декілька процесів одночасно.* Для операційної системи процес є засобом організації багатьох задач, які вона повинна виконувати. ОС виділяє кожному процесу порцію системних ресурсів і гарантує, що програма кожного процеса буде направлятися на виконання у певному порядку та своєчасно.

В різних ОС процеси реалізовані по різному. Процеси розрізняються:

- представленням (структурами даних);
- способами іменування та захисту;
- відношеннями між собою.

Зазвичай ОС містить блок коду, який керує створенням та видаленням процесів, а також відношенням між ними (M:1, 1:1, M:N). Цей код називається структурою процеса (process structure), реалізований як *диспетчер процесів (process manager).*

Для успішного виконання програми потрібні певні ресурси. До них належать:

- ресурси, необхідні для послідовного виконання програмного коду (передусім процесорний час);
- ресурси, що дають можливість зберігати інформацію, яка забезпечує виконання програмного коду (регістри процесора, оперативна пам'ять тощо).

Ці групи ресурсів визначають дві складові частини процесу:

- послідовність виконуваних команд процесора;
- набір адрес пам'яті (адресний простір), у якому розташовані ці команди і дані для них.

Виділення цих частин виправдане ще й тим, що в рамках одного адресного простору може бути кілька паралельно виконуваних послідовностей команд, що спільно використовують одні й ті ж самі дані. Необхідність розмежування послідовності команд і адресного простору підводить до поняття потоку. Процес розбивається на виконувані одиниці - потоки (threads) (один і більше потоків). Потік – це більш дрібна одиниця роботи процесора. Потоки дозволяють процесу паралельно виконувати різні частини його програми та ефективно використовувати процесор, особливо на багатопроцесорних комп’ютерах.

Потік (потік керування, нитка, thread) – *набір послідовно виконуваних команд процесора, які використовують загальний адресний простір процесу*.

Відмінність у поняттях процес і потік полягає у тому, що для виконання потоку потрібні усі види ресурсів – процесорний час, пам’ять, файли, пристрої введення-виведення тощо; для виконання процесу потрібні усі наведені вище ресурси, окрім процесорного часу.

Оскільки в системі може одночасно бути багато потоків, завданням ОС є організація перемикання процесора між ними і планування їх виконання. У багатопроцесорних системах код окремих потоків може виконуватися на окремих процесорах.

Можна дати ще одне визначення процесу. *Процесом називають сукупність одного або декількох потоків і захищеного адресного простору, у якому ці потоки виконуються.*

Захищеність адресного простору процесу є його найважливішою характеристикою. Код і дані процесу не можуть бути прямо прочитані або перезаписані іншим процесом; у такий спосіб захищаються від багатьох програмних помилок і спроб несанкціонованого доступу.

На відміну від процесів *потоки розпоряджаються загальною пам’яттю.* Дані потоку не захищені від доступу до них інших потоків за умови, що всі вони виконуються в адресному просторі одного процесу.

Захищений адресний простір процесу задає абстракцію виконання коду на окремій машині, а потік забезпечує абстракцію послідовного виконання команд на одному виділеному процесорі.

Багатопотоковість

Можливість розпаралелювання обчислень в рамках процесу на потоки підвищує ефективність ОС. Механізм розпаралелювання обчислень для одного застосування називається багатопотоковою обробкою (multithreading). Потоки процесу мають один адресний віртуальний простір. Розпаралелювання прискорює виконання процесу за рахунок відсутності перемикання ОС з одного адресного простору на інше, яке має місце при виконанні процесів. Програми стають логічніші. Паралелізм має місце у багатопроцесорних системах, під час виконання

операцій введення-виведення, під час взаємодії з користувачем, у розподілених системах.

Особливий ефект при цьому досягається в мультипроцесорних системах.

Складові елементи процесів і потоків

До елементів *процесу* належать:

- виконувана програма , яка має код і дані;
- захищений адресний простір (address space), тобто це набір адрес віртуальної пам'яті, який процес може використовувати;
- дані, спільні для всього процесу (ці дані можуть спільно використовувати всі його потоки);
- інформація про використання системних ресурсів (семафори – лічильник, який регулює кількість потоків, що використовують деякий ресурс; комунікаційний порт – логічна точка в системі, через яку пересилаються повідомлення між процесами обмінюються; секція – область пам'яті, яка спільно використовується; таймер, параметр реєстру – індексний ключ для посилання на записи у базі даних конфігурації системи; відкриті файли або пристрої введення – виведення, мережні з'єднання тощо);
- інформація про потоки процесу.

Потік містить такі елементи:

- стан процесора (набір поточних даних із його регістрів), зокрема лічильник поточної інструкції процесора;
- стек потоку (ділянка пам'яті, де перебувають локальні змінні потоку й адреси повернення функцій, що викликані у його коді).

Потік може породити інший потік – нащадок.

Розрізняють потік користувача і потік ядра.

Потік користувача — це послідовність виконання команд в адресному просторі процесу. Ядро ОС не має інформації про такі потоки, вся робота з ними виконується в режимі користувача. Засоби підтримки потоків користувача надають спеціальні системні бібліотеки; вони доступні для прикладних програмістів у вигляді бібліотечних функцій. Бібліотеки підтримки потоків зазвичай реалізують набір функцій, визначений стандартом POSIX (відповідний розділ стандарту називають POSIX.1b); тут прийнято говорити про підтримку *потоків POSIX.*

Потік ядра — це послідовність виконання команд в адресному просторі ядра. Потоками ядра управляє ОС, перемикання ними можливе тільки у привілейованому режимі. Є потоки ядра, які відповідають потокам користувача, і потоки, що не мають такої відповідності.

Стани потоків

Для **потоку** дозволені такі стани [2, 3]:

- *створення (new)* – потік перебуває у процесі створення;
- *виконання (running)* – інструкції потоку виконує процесор (у конкретний момент часу на одному процесорі тільки один потік може бути в такому стані);
- *очікування (waiting)* – потік очікує деякої події (наприклад, завершення операції введення-виведення); такий стан називають також заблокованим, а потік припиненим;
- *готовність (ready)* – потік очікує, що планувальник перемкне процесор на нього, при цьому він має всі необхідні йому ресурси, крім процесорного часу;
- *завершення (terminated)* – потік завершив виконання (якщо при цьому його ресурси не були вилучені з системи, він переходить у додатковий стан - у стан зомбі).

У деяких системах, коди і дані можуть відразу не поміщатися в пам’ять, а переписуватися в спеціальну область диска – область підкачки.

Можливі переходи між станами зображені на рис. 4.1.

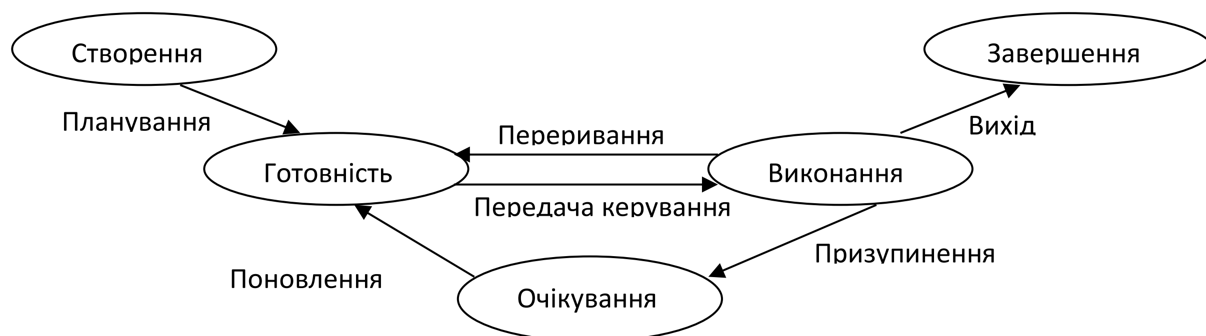


Рис. 4.1. Стани потоку

Перехід потоків між станами очікування і готовності реалізовано на основі *планування задач* або *планування потоків.* Під час планування потоків визначають, який з потоків треба відновити після завершення операції введення-виведення, як організувати очікування подій у системі.

Для здійснення переходу потоків між станами готовності та виконання необхідне *планування процесорного часу.* На основі алгоритмів такого планування визначають, який з готових потоків потрібно виконувати в конкретний момент, коли потрібно перервати виконання потоку, щоб перемкнутися на інший готовий потік тощо.

Опис процесів і потоків

Одним з основних завдань операційної системи є розподіл ресурсів між процесами і потоками. Такими ресурсами є, насамперед, процесорний час (його розподіляють між потоками під час планування), засоби введення-виведення й оперативна пам’ять (їх розподіляють між процесами).

Для керування розподілом ресурсів ОС повинна підтримувати структури даних, які містять інформацію, що описує процеси, потоки і ресурси. До таких структур даних належать:

- **таблиці розподілу ресурсів**: таблиці пам’яті, таблиці введення-виведення, таблиці файлів тощо;
- **таблиці процесів і таблиці потоків**, де міститься інформація про процеси і потоки, присутні у системі в конкретний момент.

Керуючі блоки процесів і потоків

Інформацію про процеси і потоки в системі зберігають у спеціальних структурах даних, які називають керуючими блоками процесів і керуючими блоками потоків. Ці структури дуже важливі для роботи ОС, оскільки на підставі їх інформації система здійснює керування процесами і потоками.

Керуючий блок потоку (Thread Control Block, ***TCB***) відповідає активному потоку, тобто тому, який перебуває у стані готовності, очікування або виконання. Цей блок може містити таку інформацію:

- ідентифікаційні дані потоку (зазвичай його ***унікальний ідентифікатор***);
- стан процесора потоку: ***користувацькі регістри процесора, лічильник інструкцій, покажчик на стек***;
- інформацію для ***планування*** потоків.

Таблиця потоків — це зв’язний список або масив керуючих блоків потоку. Вона розташована в захищеній області пам’яті ОС.

Керуючий блок процесу (Process Control Block, ***PCB***) відповідає процесу, що присутній у системі. Такий блок може містити:

- ідентифікаційні дані процесу (унікальний **ідентифікатор**, інформацію про інші процеси, пов’язані з даним);
- інформацію про потоки, які виконуються в адресному просторі процесу (наприклад, **покажчики на їх керуючі блоки**);
- інформацію, на основі якої можна визначити **права процесу на використання** різних ресурсів (наприклад, ідентифікатор користувача, який створив процес);
- інформацію з розподілу адресного простору процесу;
- інформацію про ресурси введення-виведення та файли, які використовує процес.

Образи процесу і потоку

Сукупність інформації, що відображає процес у пам’яті, називають *образом процесу* (process image), а всю інформацію про потік — *образом потоку* (thread image). *До образу процесу належать*:

- керуючий блок процесу;
- програмний код користувача;
- дані користувача (глобальні дані програми, загальні для всіх потоків);
- інформація образів потоків процесу.

Програмний код користувача, дані користувача та інформація про потоки завантажуються в адресний простір процесу. Образ процесу зазвичай не є безперервною ділянкою пам’яті, його частини можуть вивантажуватися на диск.

До образу потоку належать:

- керуючий блок потоку;
- стек ядра (стек потоку, який використовується під час виконання коду потоку в режимі ядра(;
- стек користувача (стек потоку, доступний у користувацькому режимі).

Схема розташування у пам’яті образів процесу і його потоків зображена на рис. 4.2 [2, 3].

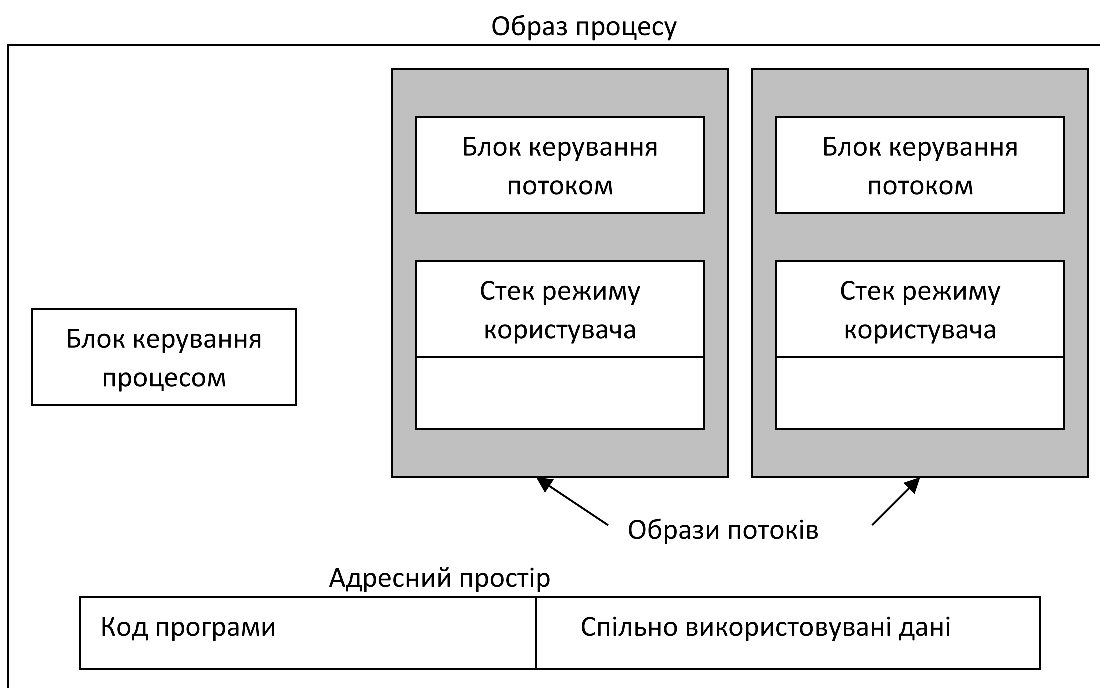


Рис. 4.2. Образи процесу і його потоків

Організація перемикання контексту

Кожний процес характеризується *контекстом* (набір регістрів, які потрібні процесу, файлів з даними, режими роботи процесора) та *дескриптором* (набір описувачів – ім’я процесу, його стан, привілеї, місце знаходження).

Для організації керування процесами і потоками найважливішим завданням ОС є передача керування від одного потоку до іншого зі збереженням стану процесора. Це називається перемикання контексту. Для перемикання контексту необхідно виконати такі операції:

- зберегти стан процесора потоку в деякій ділянці пам’яті (області зберігання стану процесора потоку);
- визначити, який потік слід виконувати наступним;
- завантажити стан процесора цього потоку з його області зберігання;
- продовжити виконання коду нового потоку.

Перемикання контексту здійснюється за допомогою засобів апаратної підтримки (використовується спеціальна ділянка пам'яті – сегмент стану задачі TSS).

Створення процесу та потоку

Процес створює диспетчер процесів, який визначає тип об'єкта (процес), атрибути процеса та надає системні сервіси.

Об'єкт процес має такі *атрибути*: ідентифікатор процесу, базовий пріоритет, час виконання, маркер доступу, лічильник введення-виведення, лічильник операцій віртуальної пам'яті, коди завершення).

Сервіси процеса: створити процес, відкрити процес, запитати інформацію про процес, завершити процес, виділити віртуальну пам'ять, читати – записувати у віртуальну пам'ять, захистити віртуальну пам'ять, скинути віртуальну пам'ять на диск.

В *Unix*-сумісних (*Linux*) системах процеси створює системний виклик *fork()*. У стандарті *Posix* потік створює функція *pthread_create()*. В Windows процес створює функція *CreateProcess()* [4, 7].

Процес створення включає також наступні дії:

- знайти програму на диску;
- перерозподілити оперативну пам'ять;
- виділити пам'ять новому процесу;
- переписати програму у виділену пам'ять;
- змінити деякі параметри програми.

Створити процес (рис. 4.3) означає створити описувач процесу (інформаційна структура, що містить відомості необхідні для управління цим процесом), а саме :

- сформувати характеристики контекст та дескриптор (виконує ОС);
- включити дескриптор у чергу (виконує ОС);
- переключити контекст зі «старого» на «новий» (виконується на апаратному рівні).

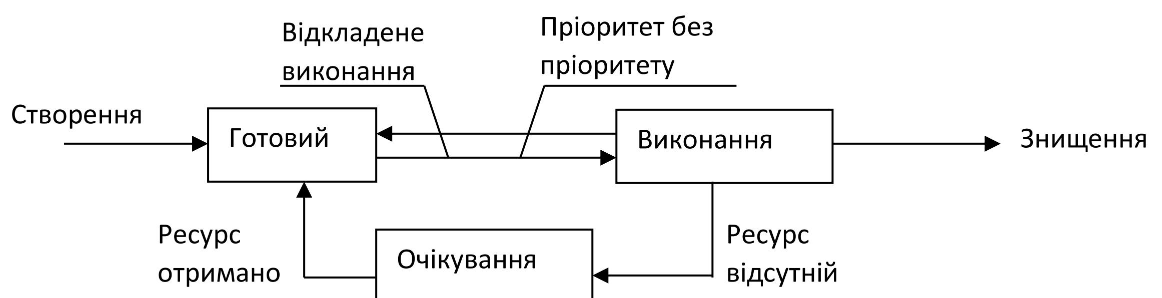


Рис. 4.3. Створення процесу

Для визначення порядку виконання потоків диспетчер ядра використовує систему пріоритетів: динамічні пріоритети задаються числом у діапазоні 1-15 (присвоюються потокам застосувань користувача), пріоритети реального часу – 16 - 31 (резервуються системою для критичних дій).

Розрізняють *наступні класи пріоритету процесів*:

- *реального* часу (real time, відповідає пріоритету потоку 24);
- *високий* (high, 3);
- *нормальний* (normal, 8);
- *невикористовуваний* (idle, 4).

Процес є нежиттєздатним, поки у нього немає потоку, який направляється на виконання. Якщо у процеса є потік, він може створити додаткові потоки. Потоки реалізовані у вигляді об'єктів, які створює та знищує диспетчер об'єктів.

Об'єкт потік має *тип об'єкта* (потік), *атрибути* об'єкта (клієнтський ідентифікатор, контекст потоку, динамічний пріоритет, базовий пріоритет, час виконання потоку, лічильник призупинок, порт завершення, код завершення), сервіси об'єкта (створити потік, відкрити потік, запитати інформацію потоку, поточний потік, завершити потік, запитати контекст, призупинити, відновити виконання, зареєструвати порт завершення).

Алгоритми планування процесів. Планування (диспетчеризація) процесів включає в себе вирішення наступних завдань:

- визначення моменту часу для зміни виконуваного процесу;
- вибір процесу на виконання з черги готових процесів;
- перемикання контекстів "старого" і "нового" процесів.

Перші два завдання вирішуються програмними засобами операційної системи, а останнє - засобами апаратної підтримки управління пам'яттю процесором.

Існує безліч різних алгоритмів планування процесів, які по різному вирішують перераховані вище завдання, переслідують різні цілі і забезпечують різну якість мультипрограмування.

У більшості ОС універсального призначення планування здійснюється *динамічно* (*on-line*), тобто рішення приймаються під час роботи системи на основі аналізу поточної ситуації. ОС не має ніякої попередньої інформації про завдання, які з'являються в випадкові моменти часу.

Статичний тип планування використовується в спеціалізованих системах, де набір одночасно виконуваних завдань визначено заздалегідь (наприклад, в системах реального часу). Тут рішення про планування приймається заздалегідь (off-line).

В різних ОС компоненти, що займаються плануванням, можуть називатися по-різному: *scheduler* - розпорядник, або планувальник, - в *Unix*; *dispatcher* - в *Windows*.

В багатьох операційних системах алгоритми планування побудовані на алгоритмах, засновані *на квантуванні*, та алгоритмах, засновані *на пріоритетах*.

Витісняючі і невитісняючі алгоритми планування. З найбільш загальних позицій - за принципом звільнення процесора активним процесом - існує два основних типи процедур планування процесів: витісняючі і невитісняючі.

Невитісняюча багатозадачність (non-preemptive multitasking) - спосіб планування процесів, при якому активний процес виконується до тих пір, поки він сам, за власною ініціативою, не віддасть керування планувальнику ОС для того, щоб той вибрав з черги інший готовий до виконання процес.

При невитісняючому програмуванні механізм планування розподілений між ОС і прикладними програмами, що створює проблеми як для користувачів, так і для розробників застосунків.

Витісняюча багатозадачність (preemptive multitasking) - спосіб, при якому рішення про переключення процесора з виконання одного процесу на виконання іншого приймається операційною системою, а не найактивнішим завданням.

При витісняючому мультипрограмуванні функції планування процесів цілком зосереджені в операційній системі.

Майже у всіх сучасних операційних системах, орієнтованих на високопродуктивне виконання застосунків (Unix, Windows NT/2000, OS/2, VAX/VMS), реалізовані витісняючі алгоритми планування процесів, в яких механізм планування задач цілком зосереджений в операційній системі. Програміст пише свій застосунок, не піклуючись про те, що воно буде виконуватися паралельно з іншими завданнями. Операційна система визначає момент зняття з виконання активної задачі, запам'ятовує її контекст, вибирає з черги готових задач наступну і запускає її на виконання, завантажуючи її контекст.

Алгоритми, засновані на квантуванні (класифікація за принципом зміни активного процесу в часі). Відповідно до алгоритмів, заснованих на квантуванні, зміна активного процесу відбувається, якщо вичерпано квант процесорного часу, відведеного даному процесу (або процес перейшов в стан очікування, або сталася помилка, або процес завершився і залишив систему).

Процес, який вичерпав свій квант, переводиться в стан готовності і чекає, коли йому буде надано новий квант процесорного часу, а на виконання відповідно до визначеного правила вибирається новий процес з черги готових.

Кванти, які виділяються, можуть бути однаковими для всіх процесів або різними. Кванти, що виділяються одному процесу, можуть бути фіксованої величини або змінюватися в різні періоди життя процесу.

Черги готових процесів також можуть бути організовані по-різному: за правилом «перший прийшов - перший обслуговується» (**FIFO** - *First In, First Out*) або за правилом «останній прийшов - перший обслуговується» (***LIFO*** *- Last In, First Out).*

Алгоритми, засновані на пріоритетах (класифікація за принципом вибору процесу на виконання з черги). Пріоритет - це число, що характеризує ступінь привілейованості процесу при використанні ресурсів обчислювальної машини, зокрема, процесорного часу. Чим вище пріоритет процесу, то вагоміші його привілеї та тим менше часу він буде проводити в чергах.

Пріоритет може виражатися цілим або дробовим, позитивним або негативним значенням. У деяких ОС прийнято, що більше число позначає більший пріоритет, в інших - навпаки (більше число означає менший пріоритет).

Пріоритет може призначатися директивно адміністратором системи, наприклад, в залежності від важливості роботи, або обчислюватися самою ОС за певними правилами.

Залежно від можливості зміни пріоритету протягом життя потоку розрізняються *динамічні і фіксовані пріоритети*. У системах з динамічними пріоритетами зміни пріоритету можуть відбуватися з ініціативи процесу, який звертається з викликом до операційної системи; або з ініціативи користувача, що виконує відповідну команду; або з ініціативи ОС в залежності від ситуації, що складається в системі.

Існує два різновиди алгоритмів пріоритетного планування: обслуговування з *відносними пріоритетами і обслуговування з абсолютними пріоритетами*.

В обох випадках вибір процесу на виконання з черги здійснюється однаково: вибирається процес, який має найвищий пріоритет. По-різному вирішується проблема визначення моменту зміни активного процесу.

В системах з відносними пріоритетами активний процес виконується до тих пір, поки він сам не покине процесор, перейшовши в стан очікування (або ж станеться помилка, або процес завершиться).

В системах з абсолютними пріоритетами виконання активного процесу переривається ще за однієї умови: якщо в черзі готових процесів з'явився процес, пріоритет якого вище пріоритету активного процесу. В цьому випадку перерваний процес переходить в стан готовності.

Змішані алгоритми планування (квантування з пріоритетами). В багатьох операційних системах алгоритми планування побудовані з використанням як квантування, так і пріоритетів. Наприклад, в основі планування лежить квантування, але величина кванта і/або порядок зміни процесів і вибору процесу з черги готових визначається пріоритетами процесів. На змішаних алгоритмах засновано планування в системах Windows NT і Unix System V Release 4. І в одній, і в іншій системі реалізована дисципліна витісняючої багатозадачності, заснована на використанні абсолютних пріоритетів і квантування.

Синхронне та асинхронне виконання процесів

Для коректної взаємодії процесів, окрім забезпечення надійного зв’язку щодо обміну інформацією між процесами, для організації правильного вирішення задачі необхідно координувати дії процесів. Наприклад, один процес потребує збільшити змінну і другий процес також потребує збільшити цю ж змінну, тобто треба синхронізувати роботу цих процесів. Синхронізація процесу означає здатність потоку добровільно призупинити своє виконання та чекати, поки не завершиться виконання деякої операції іншим потоком. Усі операційні системи, що підтримують багатозадачність або мультипроцесорну обробку, повинні надавати потокам засіб очікування того, що інший потік виконує щось інше. Наприклад, звільнити дискову пам’ять або закінчити записувати у спільно використовуваний буфер пам’яті. ОС повинна надати потоку можливість повідомити інші потоки про закінчення виконання операції. Отримавши таке повідомлення, потік який очікував, може продовжити виконання. Засоби очікування та повідомлення реалізовані як об’єкти синхронізації, за допомогою яких потік синхронізує своє виконання. До таких об’єктів синхронізації відносяться: процес, потік, файл, подія, пара подій, семафор, таймер, м’ютекс. За допомогою цих об’єктів потоки можуть координувати своє виконання, використовуючи різні правила для різних ситуацій.

В будь-який момент часу синхронізаційний об’єкт знаходиться в одному зі станів: вільний (закінчилося виконання останнього потоку) або зайнятий. Для синхронізації з об’єктом потік викликає один з системних сервісів очікування, який надає диспетчер об’єктів, і передає описувач даного об’єкта. Потік може чекати

один або декілька об'єктів, а також задати відміну очікування, якщо це очікування не закінчилося за певний проміжок часу.

До засобів синхронізації Win32 API належать функції очікування об'єкта WaitForSingObject() та функція очікування декількох об'єктів WaitForMultipleObjects().

У деяких ситуаціях час від часу операційній системі необхідно повідомляти потік про те, що він повинен виконати деяку дію. Іноді потік повинен виконати дію після того, як відбулася певна подія. Наприклад, нагадування про заплановану зустріч. У цьому випадку використовується механізм асинхронного виклика процедур (asynchronous procedure call, APC).

Взаємодія потоків

Потоки, які виконуються в рамках одного процесу, можуть виконуватися паралельно або взаємодіяти між собою. У випадку, коли потоки взаємодіють і використовують спільні дані без додаткових заходів синхронізації, виникає ситуація, яка називається *станом гонок* або *змаганням* (*race condition*). Наприклад, два клієнта банку спільно користуються одним і тим же рахунком і одночасно вносять кошти на рахунок. Можливі дві ситуації: обидва внески занесені на рахнок успішно або внесок одного з клієнтів втрачено. Спроби розв'язати подібну ситуацію викликали необхідність синхронізації потоків [4, 7].

Тупики

У деяких ситуаціях засоби синхронізації не дозволяють успішно координувати роботу процесів, виникають непередбачувані ускладнення. Наприклад, декілька процесів конкурують за володіння кінцевим числом ресурсів. Якщо ресурс є недоступним, ОС переводить даний процес у стан очікування. У випадку, коли потрібний ресурс утримується іншим очікуваним процесом, перший процес не може змінити свій стан. Така ситуація називається тупиком (*deadlock*). Наприклад, один процес потребує виконати наступні дії: зайняти принтер, звільнити магнітний диск, звільнити принтер, звільнити магнітний диск. Другий процес потребує точно таких же дій. Виникає ситуація тупика, тобто це взаємне блокування виділених пристроїв [4, 7].

Множина процесів знаходиться у тупиковій ситуації, якщо кожний процес з множини чекає подію, яку може викликати тільки інший процес цієї множини.

Тупики мають місце в системах керування базами даних, коли один процес заблокував записи, які необхідні іншому процесу, і навпаки.

Для запобігання тупиків необхідно забезпечити умови, які виключають можливість виникнення тупикових ситуацій. Для цього розроблені певні алгоритми (алгоритм банкіра).

Критична секція

При вивченні способів синхронізації важливим поняттям є поняття критичної секції (critical section). Критична секція – частина програми, виконання якої може призвести до виникнення гонок (*race condition*) для певного набору програм. Щоб виключити ефект гонок по відношенню до

певного ресурсу, необхідно організувати роботу таким чином, щоб в кожний момент часу тільки один процес знаходився в своїй критичній секції, пов'язаній з цим ресурсом, тобто треба забезпечити реалізацію взаємовиключення для критичних секцій програм. Реалізація взаємовиключень для критичних секцій означає, що критична секція виконуватися як атомарна, неподільна операція, тобто може виконуватися тільки один потік. На практиці використовується блокування – механізм, який не дозволяє більш як одному потокові виконувати код критичної секції. Використання блокування зводиться до двох дій: запровадження блокування (функція acquire_lock()) і зняття блокування (функція release_lock()). Для організації блокування в архітектурі IA-32 може бути використана спеціальна інструкція процесора, яку називають «перевірити і заблокувати» (Test & Set Lock, TSL).

Контрольні питання:

1. Що таке процес?
2. Що таке потік?
3. Складові процесу.
4. Складові потоку.
5. Що таке потік користувача?
6. Що таке потік ядра?
7. Стани потоків.
8. Що таке образ процесу?
9. Що таке образ потоку?
10. Сервіси процесу.
11. Сутність витісняючих і невитісняючих алгоритмів планування.
12. Сутність алгоритмів, заснованих на квантуванні.
13. Сутність алгоритмів, заснованих на пріоритетах.
14. Як здійснюється синхронізація процесів під час їх виконання?
15. Наведіть умови, за яких виникають тупики.
16. Що таке критична секція?

Лекція 5. Структури процесів і потоків в Windows, API функції

Процеси і потоки в Windows: фундаментальні концепції. Закритий віртуальний простір. Виконувана програма, Список відкритих дескрипторів. Контекст безпеки. Ідентифікатор процеса. Програмній потік. PEB (Process Environment Block - блок середовища процесу). Диспетчер завдань (Task Manager). Дерево процесів. Системний виклик Win32 API – функції CreateProcess. Структура EPROCESS (Executive Process, виконавчий процес). Структура ETHREAD (Executive Thread, виконавчий потік). Волокна. Зв’язок між завданнями, процесами, потоками і волокнами. Функція CreateThread створення потоку.

Процеси і потоки в Windows: фундаментальні концепції

Фундаментальні концепції

В Windows процеси є контейнерами для програм.

Контейнери - це технологія упаковки і запуску застосунків Windows і Linux в різних локальних середовищах і в хмарі. Контейнери надають невимогливе до ресурсів ізольоване середовище, яке спрощує розробку, розгортання і керування застосунками.

Контейнери містять віртуальний адресний простір, описувачі об'єктів режиму ядра, а також потоки [2, 3].

Процес – це контейнер для набору ресурсів, які використовуються для виконання програми. Програма - статична послідовність команд.

На верхньому рівні абстракції *процес Windows включає наступні компоненти*:

- **закритий віртуальний простір** – множина адрес віртуальної пам'яті, яка може використовуватися процесом;
- **виконувана програма**, яка визначає початковий код і дані і відображається у віртуальний адресний простір процесу;
- **список відкритих дескрипторів** для різних системних ресурсів (семафорів, об'єктів синхронізації портів, файлів і т. д.), які доступні для всіх програмних потоків в процесі;
- **контекст безпеки** - *маркер доступу* (*access token*), який ідентифікує користувача, групи безпеки, привілеї, стан віртуалізації UAC (*User Account Control*), сеанс і обмежене стан облікового запису користувача, пов'язаний з процесом, а також ідентифікатор контейнера застосунки і пов'язана з ним ізольована інформація;
- **ідентифікатор процеса** – унікальний ідентифікатор, який є частиною *ідентифікатора клієнта*;
- **принаймні один програмний потік (thread).**

Кожний процес має системні дані користувацького режиму, які називаються ***PEB (Process Environment Block - блок середовища процесу).*** PEB включає список завантажених модулів (EXE і DLL), область пам'яті з рядками оточення, поточний робочий каталог, а також дані для управління купами процесу.

Існують різні програми для перегляду (та зміни) процесів й інформації процесів.

Для перегляду інформації про процеси найчастіше використовується диспетчер завдань (*Task Manager*). Диспетчер завдань можна запустити чотирма способами:

- натиснути *Ctrl+Shift+Esc;*
- клацнути правою кнопкою миші на панелі завдань і вибрати команду *Диспетчер задач (Start Task Manager;*
- натиснути Ctrl+Alt+Del і клацнути на кнопці *Запустить диспетчер задач (Start Task Manager);*
- запустити виконуваний файл *Taskmgr.exe.*

Для повного представлення диспетчера завдань треба клацнути на кнопку *Подробнее* (*More Details*). За замовчуванням вибирається вкладка *Процессы (Processes)*.

На вкладці *Процессы* (Processes) виводиться список процесів, що складається з чотирьох стовпців: *ЦП (CPU), Пам'ять (Memory), Диск (Disk) і Мережа (Network).*

Щоб додати в список інші стовпці, треба клацніути правою кнопкою миші на заголовку. Також доступні стовпці *Process (Image) name*, *ИД процесса (Process ID), Тип (Type), Стан (Status), Издатель (Publisher) і Командная строка (Command Line).* Деякі процеси можна додатково розгорнути з виведенням інформації про видимі вікна верхнього рівня, створених процесом.

Щоб отримати ще більше інформації про процес, треба клацнути на кнопці *Подробнее (Details).* Також можна натиснути правою кнопкою миші на процесі і вибрати команду *Подробнее* (*Go to Details*), щоб переключитися на вкладку *Подробности (Details)* і вибрати цей конкретний процес.

Деякі найважливіші стовпці:

- *Потоки (Threads)* - в цьому стовпці виводиться кількість програмних потоків в кожному процесі. Це число зазвичай не менше 1, так як неможливо безпосередньо створити процес, який не містить жодного потоку. Якщо в списку присутній процес з 0 потоків, зазвичай це означає, що процес не вдається видалити через помилки в коді драйвера;
- *Дескриптори (Handles)* - в цьому стовпці виводиться кількість дескрипторів об'єктів ядра, відкритих програмними потоками, виконуваними в процесі.

Кожний процес також містить ідентифікатор свого батьківського процесу або створеного процесу. Для перегляду дерева процесів можна виконати команду *tlist /t.*

Створення процесу

Зазвичай процес створюється іншим процесом викликом Win32-функції **CreateProcess** (а також **CreateProcessAsUser** і **CreateProcessWithLogonW**). Створення процесу здійснюється в декілька етапів [2, 3].

На **першому етапі**, що виконується бібліотекою kernel32.dll в режимі користувача, на диску відшукується потрібний файл-образ, після чого створюється об'єкт "розділ" пам'яті для його проектування на адресний простір нового процесу.

На **другому етапі** виконується звернення до системного сервісу **NtCreateProcess** для створення об'єкта "процес". Формуються блоки *EPROCESS, KPROCESS* і блок змінних оточення *PEB*. Менеджер процесів ініціалізує в блоці процесу маркер доступу (копіюючи аналогічний маркер батьківського процесу), ідентифікатор і інші поля.

На **третьому етапі** в вже повністю проініціалізованому об'єкті "процес" необхідно створити первинний потік. Це, за допомогою системного сервісу **NtCreateThread**, робить бібліотека kernel32.dll.

Потім kernel32.dll посилає підсистемі Win32 повідомлення, яке містить інформацію, необхідну для виконання нового процесу. Дані про процес і потоці поміщаються, відповідно, в список процесів і список потоків даного процесу, потім встановлюється пріоритет процесу, створюється структура, яка використовується тією частиною підсистеми Win32, яка працює в режимі ядра, і т.д.

Нарешті, запускається первинний потік, для чого формуються його початковий контекст і стек, і виконується запуск стартової процедури потоку режиму ядра **KiThreadStartup**. Після цього стартовий код з бібліотеки C/C++ передає управління функції **main**() запускається програма.

Завершення процесу може бути здійснено різними способами, наприклад, за допомогою функцій **ExitProcess**, **TerminateProcess**. Однак, єдиним способом, що гарантує коректну очистку всіх ресурсів, є повернення управління вхідний функції первинного потоку.

Реалізація процесів

Внутрішній устрій процесів в ОС Windows

Кожний процес Windows представляється структурою **EPROCESS** (*Executive Process, виконавчий процес*). Крім багатьох атрибутів, які стосуються процесу, EPROCESS містить ряд інших взаємопов'язаних структур даних і вказівників на них. Наприклад, кожний процес містить один або кілька потоків, кожний з яких представлений структурою **ETHREAD** (Executive Thread).

Структура *EPROCESS* та більшість пов’язаних з нею структур даних існують у системному адресному просторі. Винятком є тільки блок *PEB (Process Environment Block,* процес оточуючого середовища), який існує в адресному просторі процесу (користувацькому), тому що він містить інформацію, доступну для коду користувацького режиму. Крім того, деякі структури даних процесу, які використовуються для управлінні пам'яттю (наприклад, список робочих наборів), дійсні тільки в контексті поточного процесу, оскільки зберігаються в системному просторі, що належить процесу.

Для кожного процесу, що виконує програму Windows, процес підсистеми Windows (Csrss) підтримує паралельну структуру з ім'ям **CSR_PROCESS** [2, 3].

Крім того, частина підсистеми Windows, що відноситься до режиму ядра (*Win32k.sys*), підтримує структуру даних рівня процесу W32PROCESS, яка створюється при першому виклику з потоку функції Windows USER або GDI (*Graphics Device Interface*), реалізованої в режимі ядра. Це відбувається відразу ж після завантаження бібліотеки *User32.dll*. Типові функції, які ініціюють завантаження цієї бібліотеки, - *CreateWindow(Ex) і GetMessage.*

Об’єкт *PROCESS* має декілька важливих структур *EPROCESS,* KPROCESS (Kernel Process) і *PEB*. Ключові поля структури EPROCESS зображена на рис. 5.1, структури KPROCESS – на рис. 5.2, структури PEB– на рис. 5.3 [2,3].

Блок PEB розміщується в адресному просторі режиму користувача, описуваного їм процесу. Він містить інформацію, необхідну для завантажувача образів, диспетчера купи і інших компонентів Windows, які повинні звертатися до нього з призначеного для користувача режиму; надавати доступ до всієї цієї інформації через системні виклики було б занадто витратно. Структури EPROCESS і KPROCESS доступні тільки з режиму ядра.

Купа - це діапазон зарезервованого адресного простору. Спочатку більшій його частини фізична пам'ять не передається. У міру того, як програма займає цю область під дані, спеціальний диспетчер, який керуючий купами (*heap manager*), посторінково передає їй фізичну пам'ять (з сторінкового файлу). А при звільненні блоків в купі диспетчер повертає системі відповідні сторінки фізичної пам'яті.

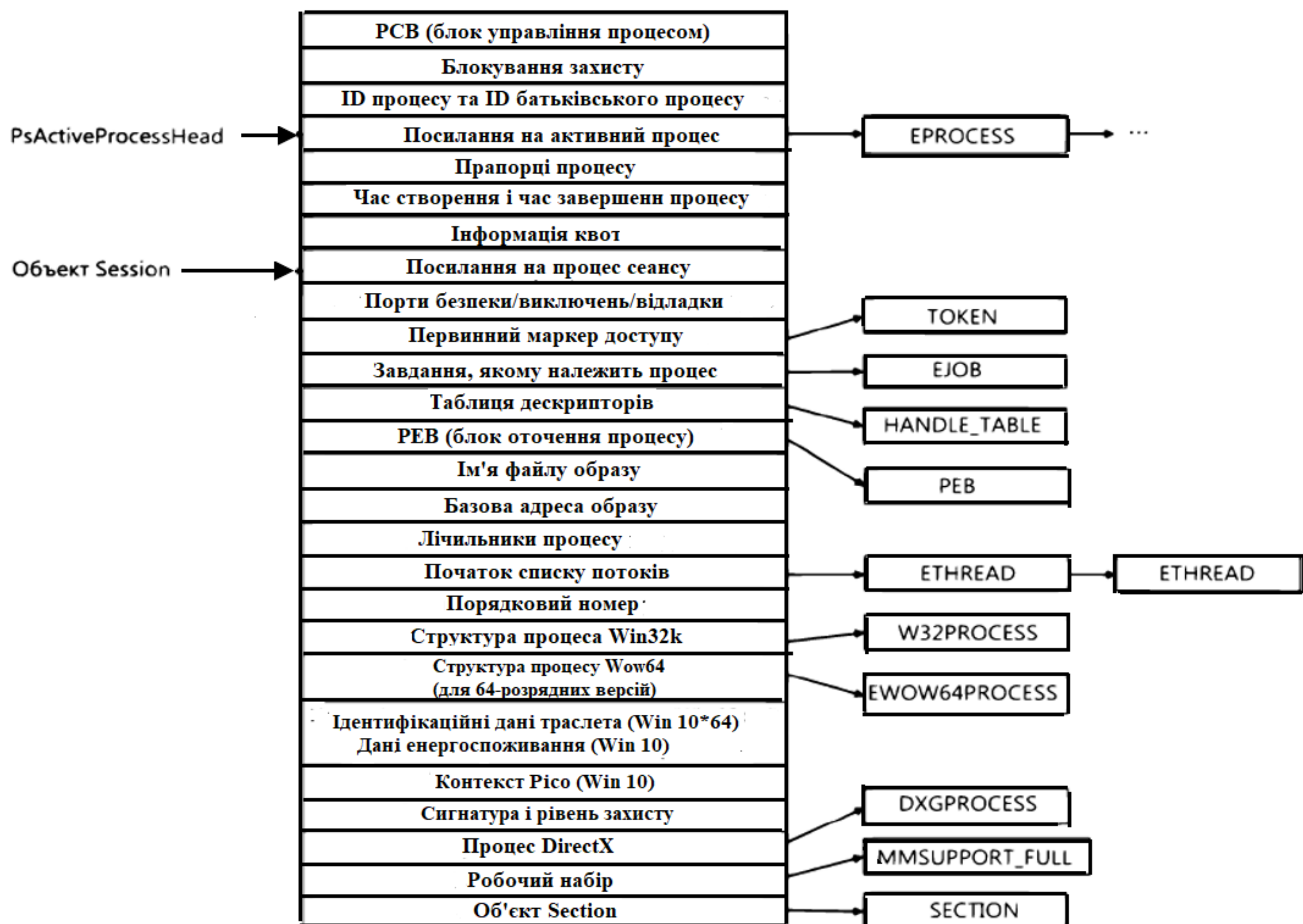


Рис. 5.1. Ключові поля структури *EPROCESS*

Рис. 5.2. Ключові поля структури *KPROCESS*

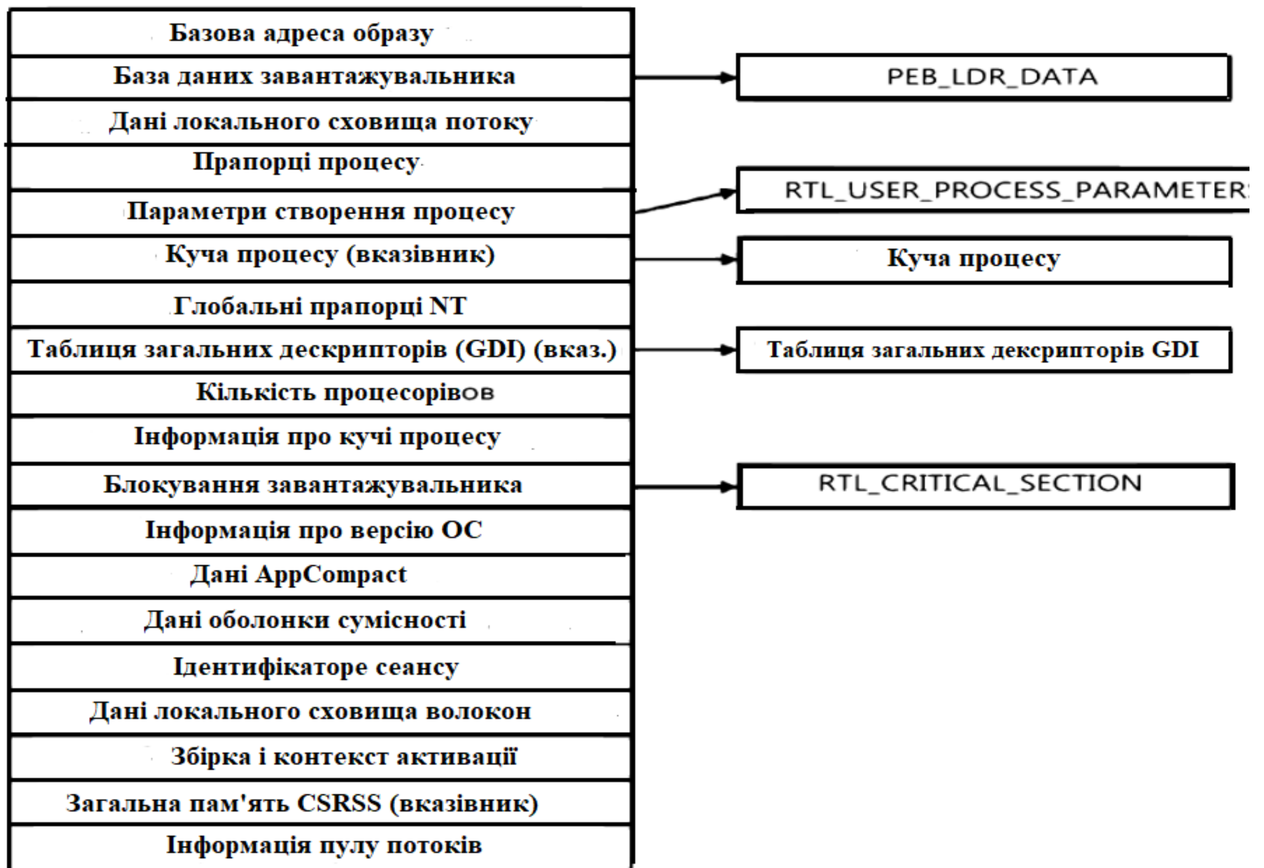


Рис. 5.3. Ключові поля структури PEB

Потоки. *Програмний потік (або просто потік)* - послідовність команд всередині процеса, яка планується Windows для виконання. Без потоків програма процесу не зможе виконуватися. Потік містить такі найважливіші компоненти:

- *вміст набору регістрів CPU*, що представляють стан процесора;

- *два стека:* один для програмного потоку, який повинен використовуватися при виконанні в режимі ядра, інший - для виконання в користувацькому режимі;

- закрита область пам'яті, яка називається *локальною пам'яттю потоку команд* (*TLS, Thread-Local Storage*); використовується підсистемами, бібліотеками часу виконання і динамічними бібліотеками DLL;

- унікальний ідентифікатор - *ідентифікатор потоку (TID, Thread ID);* є частиною внутрішньої структури ідентифікатора клієнта (*client ID).* Ідентифікатори процесу і потоку генеруються з одного простору імен, тому вони ніколи не перекриваються [2, 3].

Крім того, потоки іноді мають власний контекст безпеки, який часто використовується багато-серверними застосунками, які реалізують контекст безпеки клієнтів, що обслуговуються.

Регістри, стеки і закрита область пам'яті називаються контекстом потоку.

Потоки - це абстракції ядра для планування процесора в Windows. Кожному потоку присвоюється пріоритет (в залежності від значення пріоритету його процесу).

Потоки можуть бути ***аффінізірованними*** (*affinitized*), це означає, що кожний потік виконується тільки на певних процесорах. Це допомагає паралельним програмам, які працюють на багатоядерному мікропроцесорі або декількох процесорах розподіляти навантаження явним чином. Кожний потік має два окремих стека викликів: один для виконання в користувацькому режимі, інший для режиму ядра. Є також блок ***TEB*** (***Thread Environment Block*** - блок середовища потоку), який зберігає специфічні для потоку дані користувацького режиму, в тому числі **області зберігання для потоку** (***Thread Local Storage***) і поля для Win32, локалізації мови і культури, а також інші спеціальні поля, які були додані різними засобами.

Окрім PEB і TEB існує ще одна структура даних, яку режим ядра використовує спільно з усіма процесами, - **спільно використовувані дані користувача (user shared data**). Це сторінка, в яку ядро може ввести запис, а процеси користувацького режиму можуть з неї тільки читати. Вона містить деякі значення, які підтримуються ядром, такі як різні форми часу, інформація про версії, кількість фізичної пам'яті. Ця спільно використовувана сторінка застосовується виключно для оптимізації продуктивності, оскільки всі ці значення можна отримати і за допомогою системного виклику в режимі ядра.

Так як перемикання виконання між потоками вимагає участі планувальника ядра, ця операція може бути досить затратною, особливо якщо два потоки часто передають управління між собою. В Windows реалізовані два механізми для скорочення цих витрат: *волокна (fibers) і планування призначеного для користувача режиму (UMS, User Mode Scheduling).*

Волокна (нити). Волокна дозволяють застосункам планувати свої потоки виконання, а не покладатися на механізм пріоритетного планування, вбудований в

Windows. Волокна також часто називають «полегшеними потоками». Відносно планування волокна невидимі для ядра, тому що вони реалізуються в режимі користувача в *Kernel32.dll.* Щоб використовувати волокна, слід спочатку викликати функцію Windows *ConvertThreadToFiber*. Ця функція перетворює потік в працююче волокно. Надалі перетворене волокно може створювати додаткові волокна функцією *CreateFiber. (*Кожне волокно може мати власний набір волокон.) Однак, на відміну від потоків, волокно не починає виконуватися до того, як воно буде вручну вибрано викликом функції *SwitchToFiber*. Нове волокно залишиться активним, поки не завершиться або не викличе *SwitchToFiber* з вибором іншого волокна для виконання. Додаткова інформація щодо до опису функцій волокон є в документації *Windows SDK.*

Завдання. В Windows реалізовано розширення моделі процесу - так звані завдання. Головна функція **об'єкта завдання (job) - забезпечити можливість управління і виконання операцій з групами процесів як з єдиним цілим**. Об'єкт завдання дозволяє управляти деякими атрибутами і встановлює обмеження для процесів, пов'язаних із завданням. Також він зберігає основну облікову інформацію для всіх процесів, пов'язаних із завданням, і для всіх завдань, які були пов'язані із завданням, але встигли завершитися. В якомусь сенсі об'єкт завдання компенсує відсутність структурованого дерева процесів в Windows - проте у багатьох відношеннях він потужніший дерева процесів в стилі UNIX [2.3].

Завдання і волокна. Windows може об'єднувати процеси в завдання, які групують процеси, щоб обмежити потоки, які в них містяться, та використовувати спільне квотування ресурсів або ***маркер обмеженого доступу (restricted token),*** який не дозволяє потокам звертатися до багатьох системних об'єктів. Найважливішою властивістю завдань (в плані управління ресурсами) є той факт, що з того моменту, як процес виявився у завданні, усі створені (в цих процесах) потоками процеси також будуть перебувати в цьому завданні. Виходу немає. У повній відповідності зі своєю назвою завдання були призначені для таких ситуацій, які швидше нагадували пакетну обробку завдань, ніж звичайні інтерактивні обчислення.

Процес може перебувати всередині тільки одне завдання (максимум).

В Windows завдання використовуються для угруповання процесів, що виконують програми. Процеси, які містять застосунок, що виконується, повинні бути ідентифіковані операційною системою, щоб вона змогла управляти всім застосунком від імені користувача.

На рис. 5.4 показано зв'язок між завданнями (*jobs*), процесами (*processes*), потоками (*threads*) і волокнами (*fibers*). Завдання містять процеси. Процеси містять потоки. Але потоки не містять волокон. Зв'язок між потоками і волокнами зазвичай має тип «багато-до-багатьох».

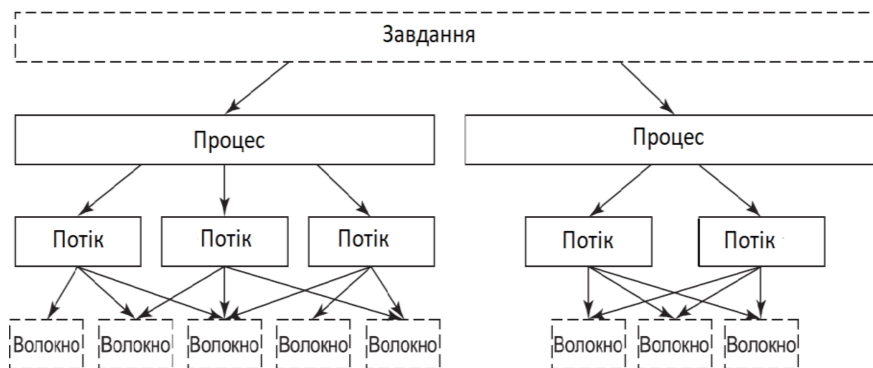


Рис. 5.4. Зв'язок між завданнями, процесами, потоками і волокнами

Завдання і волокна не обов'язкові: не всі процеси перебувають в завданнях або містять волокна. Волокна створюються шляхом виділення місця в стеку і структури даних «волокна» в користувацькому режимі (для зберігання регістрів і даних, пов'язаних з цим волокном). Потоки перетворюються у волокна, проте волокна можуть створюватися і незалежно від потоків. Такі волокна не будуть виконуватися до тих пір, поки волокно, яке вже виконується в потоці, не викличе явно *SwitchToFiber* (для запуску волокна). Потоки можуть спробувати переключитися на те волокно, яке вже виконується, так що програміст повинен передбачити синхронізацію (щоб уникнути цього явища).

Основною перевагою волокон є те, що витрати перемикання між волокнами набагато нижче, ніж перемикання між потоками. Для перемикання між потоками треба увійти в ядро і вийти з нього. Перемикання між волокнами зберігає і відновлює кілька регістрів (без усякої зміни режиму).

Для планування в режимі користувача використовують *пули потоків*. Пул потоків Win32 є надбудовою над моделлю потоків Windows, що надає більш вдалу абстракцію для певного типу програм. Ідея створення пулу полягає у тому, що для програми буде виділено обмежену кількість потоків та підтримка черги завдань, що вимагають виконання. Як тільки потік завершить виконання завдання, він бере з черги наступне завдання. Ця модель відокремлює питання управління ресурсами (скільки процесорів доступно і скільки потоків має бути створено) від моделі програмування (що собою являє завдання і як завдання синхронізуються). У Windows це рішення оформлено в пул потоків Win32, набір API-функцій для автоматичного управління динамічним пулом потоків і у відправленні йому завдань.

Програмісти бачать, як один Windows-потік фактично являє собою два потоки: той, що запускається в режимі ядра, і той, що запускається в режимі користувача.

Точно така ж модель є і в UNIX. Кожному з цих потоків виділяються його власний стек і його власна пам'ять для зберігання його регістрів, коли він не виконується. Два потоки стають одним потоком, тому що вони не працюють в один і той же час. Потік режиму користувача працює в якості розширення потоку режиму ядра, що запускається, тільки коли потік ядра на нього перемикається, повертаючись з режиму ядра в режим користувача. Коли потік режиму користувача хоче виконати системний виклик, зіткнувшись з помилкою відсутності сторінки

або витісненням, система входить в режим ядра і перемикається назад на відповідний потік ядра. Зазвичай неможливо перемикатися між потоками в режимі користувача без попереднього перемикання на відповідний потік режиму ядра, перемикання на новий потік режиму ядра, а потім перемикання на його потік режиму користувача.

Таким чином, *основні концепції, що використовуються для управління процесором і ресурсами:*

- завдання – це колекція процесів, у яких є загальні квоти і ліміти;
- процес – це контейнер для ресурсів;
- потік – це одиниця планування для ядра;
- волокно - це «легкий» потік, який повністю керований в просторі користувача;
- пул потоків – це модель програмування, орієнтована на застосування завдань;
- потік режиму користувача – це абстракція, що дозволяє перемикати потоки в режимі користувача.

Таким чином, потоки є концепцією планування, а не концепцією володіння ресурсами. Будь-який потік може звертатися до всіх об'єктів, які належать його процесу. Все, що для цього потрібно зробити, - використовувати значення описувача і зробити відповідний виклик Win32.

Виклики API для управління завданнями, процесами, потоками і волокнами

Нові процеси створюються за допомогою функції *CreateProcess* інтерфейсу Win32 API [4, 6, 7]. Ця функція має багато параметрів і опцій.

Функція ***CreateProcess*** створює новий процес і його первинний (головний) потік. Новий процес виконує вказаний виконуваний файл в контексті безпеки та викликаючого процеса. Якщо викликаючий процес представляє іншого користувача, новий процес використовує маркер доступу для викликаючого процесу, а не маркер запозичення прав. Щоб запустити новий процес в контексті системи безпеки користувача, позначеного маркером запозичення прав, використовуйте функцію ***CreateProcessAsUser*** або **CreateProcessWithLogonW.**

BOOL CreateProcess (
LPCTSTR *lpApplicationName*, // вказівник на ім'я виконуваного модуля
LPTSTR *lpCommandLine,* // вказівник на командний рядок
LPSECURITY_ATTRIBUTES *lpProcessAttributes,* //вказівник на атрибути безпеки процеса
LPSECURITY_ATTRIBUTES *lpThreadAttributes,* // вказівник на атрибути безпеки потоку
BOOL *bInheritHandles*, // вказівник на прапорець успадкування
DWORD *dwCreationFlags,* // прапорці створення
LPVOID lpEnvironment, // вказівник на новий блок середовища
LPCTSTR *lpCurrentDirectory,* // вказівник на ім'я поточного каталогу

LPSTARTUPINFO *lpStartupInfo*, // вказівник на структуру передустановки
StartupInfo
LPPROCESS_INFORMATION *lpProcessInformation* // вказівник на структуру
ProcessInformation інформації про процес
);

Параметри (*Опис параметрів функції можна подивитися в MSDN*):

lpApplicationName - рядок може задавати повний шлях та ім'я виконуваного модуля;

lpCommandLine - може бути NULL, в цьому випадку, функція використовує рядок, на яку вказує параметр *lpApplicationName.* Інакше – вказує на командний рядок;

lpProcessAttributes - вказівник на структуру SECURITY_ATTRIBUTES, яка визначає, чи буде повернений дескриптор на процес успадкованим для дочірніх процесів. Якщо цей параметр нульовий, то дескриптор не може бути успадкованим;

lpThreadAttributes - якщо нульовий, потік отримує дескриптор безпеки за замовчуванням.

bInheritHandles - вказує на те, чи буде процес успадковувати дескриптори викликаючого процесу;

dwCreationFlags -вказує додаткові прапорці, які контролюють клас пріоритету і створення процесу.

Значення яке повертається: якщо функція успішна, повертається ненульове значення. Якщо функція невдала, повертається нуль. Для отримання докладної інформації про помилку викликайте функцію *GetLastError.*

Зауваження. Функція *CreateProcess* призначена для запуску нової програми. Функції *WinExec* і *LoadModule* також працюють для створення процесу, але не пропонують таких можливостей, як *CreateProcess.*

CreateProcess окрім створення процесу також створює потоковий об'єкт. Потік створюється з ініціалізованим стеком, розмір якого описаний в заголовку образу виконуваного файлу. Потік починає виконання в точці входу цього образу.

Дескриптори нового процесу і нового потоку створюються з повними правами доступу. Для обох дескрипторів, якщо дескриптор безпеки не забезпечується, то дескриптор може бути використаний будь-якою функцією, яка запитує дескриптор об'єкта цього типу. При забезпеченні дескриптора безпеки перевірка доступу проводиться при будь-якому запиті на використання дескриптора. Якщо перевірка не гарантує доступ, то запитуючий процес не може використовувати дескриптор даного потоку.

Приклад 1. Створення процесу за допомогою функції CreateProcess.

/ * Цей фрагмент коду слугує для запуску програми MS Word та відкриття в ній текстового файлу *1.txt* за допомогою функції *CreateProcess* зі значенням за замовчуванням для більшості її параметрів. Для коректної роботи прикладу треба вказати правильний шлях до програми MS Word і текстового файлу *1.txt.* * /

```
char *cmdline=new char [1000];
```

```
STARTUPINFO si = {sizeof (si)};
PROCESS_INFORMATION pi;
if (CreateProcess("f:\\path\\winword.exe",
                strcpy (cmdline, "f:path\\winword.exe e:\\1.txt"),
                NULL, NULL, FALSE, 0, NULL, NULL, & i, &pi))
    printf("\n\nEnd of work function CreateProcess\n");
else
    printf("\n\nError working function CreateProcess!!!\n");
delete [] cmdline;
```

Приклад 2 програми створення процесу

```
#include <windows.h>
#include <stdio.h>
void main (VOID)
{
 STARTUPINFO StartupInfo;
 PROCESS_INFORMATION ProcInfo;
 TCHAR CommandLine [] = TEXT("sleep");

 ZeroMemory (& StartupInfo, sizeof (StartupInfo));
 StartupInfo.cb = sizeof (StartupInfo);
 ZeroMemory (& ProcInfo, sizeof (ProcInfo));

 if (! CreateProcess (NULL, // Не використовується ім'я модуля
    CommandLine, // Командний рядок
    NULL, // Дескриптор процесу не успадковується.
    NULL, // Дескриптор потоку не успадковується.
    FALSE, // Установка описувачів успадкування
    0, // Немає прапорів створення процесу
    NULL, // Блок змінних оточення батьківського процесу
    NULL, // Використовувати поточний каталог батьківського процесу
    & StartupInfo, // Вказівник на структуру STARTUPINFO.
    & ProcInfo) // Вказівник на структуру інформації про процес.
   )

printf  "CreateProcess failed.");

  // Чекати закінчення дочірнього процесу
  WaitForSingleObject (ProcInfo.hProcess, INFINITE);

  // Закрити описувачі процесу і потоку
  CloseHandle (ProcInfo.hProcess);
```

```
    CloseHandle (ProcInfo.hThread);
}
```

У наведеній програмі ім'я програми передається через другий параметр функції **CreateProcess**.

У прикладі в якості дочірньої програми використовується найпростіша команда *sleep*, завдання якої - витримати паузу тривалістю 10 секунд.

```
#include <windows.h>
#include <stdio.h>
void main (VOID)
{
printf ( "Дана програма буде спати протягом 10000 мс \ n ");
Sleep (10000);
}
```

Приклад 3

```
#include "stdafx.h"
#include "windows.h"
#include "iostream.h"
void main()
{
  STARTUPINFO cif;
  ZeroMemory(&cif,sizeof(STARTUPINFO));
  PROCESS_INFORMATION pi;
  if (CreateProcess("c:\\windows\\notepad.exe",NULL,
          NULL,NULL,FALSE,NULL,NULL,NULL,&cif,&pi)==TRUE)
  {
          cout << "process" << endl;
          cout << "handle " << pi.hProcess << endl;
          Sleep(1000);              // почекати
          TerminateProcess(pi.hProcess,NO_ERROR);// прибрати процес
   }
}
```

Функція ***CreateThread*** створює потік, який виконується в межах віртуального адресного простору викликаючого процесу. Щоб створити потік, який запускається у віртуальному адресному просторі іншого процесу, використовується функція ***CreateRemoteThread*** [4, 6, 7].

Синтаксис

HANDLE CreateThread (
  LPSECURITY_ATTRIBUTES *lpThreadAttributes*, // дескриптор захисту
  SIZE_T *dwStackSize*, // початковий розмір стека
  LPTHREAD_START_ROUTINE *lpStartAddress*, // функція потоку
  LPVOID *lpParameter*, // параметр потоку
  DWORD *dwCreationFlags*, // опції створення

LPDWORD *lpThreadId* // ідентифікатор потоку
);

Параметри:

lpThreadAttributes - аргумент визначає, чи може створюваний потік бути успадкований дочірнім процесом. Ми не будемо створювати дочірні процеси, тому ставимо NULL;

dwStackSize - розмір стека в байтах. Якщо передати 0, то буде використовуватися значення за замовчуванням (1 мегабайт).

lpStartAddress - адреса функції, яка буде виконуватися потоком. Тобто можна сказати, що функція, адреса якої передається в цей аргумент, є створюваним потоком. Ця функція повинна відповідати певним прототипу;

lpParameter - вказівник на змінну, яка буде передана в потік.

dwCreationFlags - прапорці створення. Можна відкласти запуск виконання потоку. Для запуску потоку відразу передаємо 0.

lpThreadId - вказівник на змінну, куди буде збережений ідентифікатор потоку. Якщо цей ідентифікатор не потрібен, передаємо NULL.

Приклад код виклику CreateThread:

HANDLE thread = CreateThread (NULL, 0, thread2, NULL, 0, NULL);

Описувач потоку зберігаємо у змінну *thread*.

Зверніть увагу на третій аргумент - адреса функції потоку, thread2 - ім'я функції, яка і буде другим потоком. Ось її код:

```
DWORD WINAPI thread2 (LPVOID t)
{
/ * Код другого потоку * /
return 0;
}
```

Функція потоку повинна відповідати наступним прототипу:

DWORD WINAPI ThreadProc (LPVOID lpParameter).

Контрольні питання:

1. Що містить блок середовища процесу?
2. Що містить блок середовища потоку?
3. Призначення ідентифікатора процесу.
4. Призначення контейнера.
5. Які складові контексту потока?
6. Що таке завдання?
7. Що таке волокна?
8. Що таке пул потоків?
9. Які основні концепції використовуються для управління процесором і ресурсами?
10. Яка функція створює процес?
11. Яка функція створює потік?

Тема 4. Регістри процесора

Лекція 6. Регістри процесора, доступні для програмування

Існують такі види основних видів пам'яті: оперативна (RAM – Random Access Memory), регістри (Registers), постійна (ROM – Read-only Memory), флеш-пам'ять (Flash RAM), процесорна (Cache Memory) та зовнішня. Всі вони відрізняються своїм призначенням та швидкодією. Процесорна надшвидка кеш-пам'ять (L1 та L2) та регістри розташовані всередині процесора, тому відрізняються високою швидкодією (наносекунди –мільйонні долі секунд). Тобто регістри – це область високошвидкісної пам'яті в процесорі. Регістри дуже важливі для будь-якого процесора. Вони не мають адрес, але мають імена.

Регістри даних /загального призначення

Регістри загального призначення є основними робочими регістрами асемблерних програм (рис. 6.1) [5, 15]. Їх відрізняє те, що до них можна адресуватися одним словом (16 бітів) або однобайтовим кодом (8 бітів). Лівий байт вважається старшим (High), а правий – молодший (low).

High						Біти				Low					
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Регістр AX (Accumulator)															
AH								AL							
Регістр BX (Base Register)															
BH								BL							
Регістр CX (Count Register)															
CH								CL							
Регістр DX (Data Register)															
DH								DL							

Рис. 6.1. Регістри загального призначення процесора i8086/i8088

Регістр AX =<AH:AL> - первинний акумулятор (Accumulator), який використовується у всіх операціях введення-виведення, в деяких операціях з рядками та в деяких арифметичних операціях. Наприклад, команди множення та ділення передбачають обов'язкове використання регістра AX(AL). Деякі команди генерують більш ефективний код, якщо вони мають посилання на регістр AX.

Регістр BX=<BH:BL> - базовий регістр (Base Register), єдиний з регістрів загального призначення, що використовується в індексній адресації. Крім того, регістр BX може використовуватися при обчисленнях.

Регістр CX=<CH:CL> є лічильником (Count Register**).** Він використовується для управління числом повторень циклів та для операцій зсуву вліво або вправо. Регістр CX також може використовуватися для обчислень.

Регістр DX=<DH:DL> - регістр даних (Data Register). Використовується в деяких операціях введення-виведення, в операціях множення та ділення 16-розрядних чисел сумісно з регістром AX.

Будь-який з цих регістрів може використовуватися для підсумовування і віднімання 8- або 16-розрядних величин.

З введенням 32-бітної архітектури в 80386 з'явилися 32-бітні EAX – EDX регістри. **E** від слова **extended** – розширений, 32-бітові регістри **EAX (акумулятор), EBX (база), ECX (лічильник), EDX (регістр даних)** можуть використовуватися без обмежень для будь-яких цілей часового зберігання даних, аргументів або результатів різних операцій. Назви цих регістрів походять від того, що деякі команди застосовують їх спеціальним чином: так, акумулятор часто використовується для зберігання результату дій, що виконуються над двома операндами, регістр даних в цих випадках одержує старшу частину результату, якщо він не уміщається в акумулятор, регістр-лічильник використовується як лічильник в циклах і строкових операціях, а регістр-база використовується при так званій адресації по базі. Молодші 16 бітів кожного з цих регістрів можуть використовуватися як самостійні регістри і мають імена (відповідно AX, BX, CX, DX), як в процесорах 8086, 80286 (рис. 6.2). Ці регістри використовуються в арифметико-логічному пристрої.

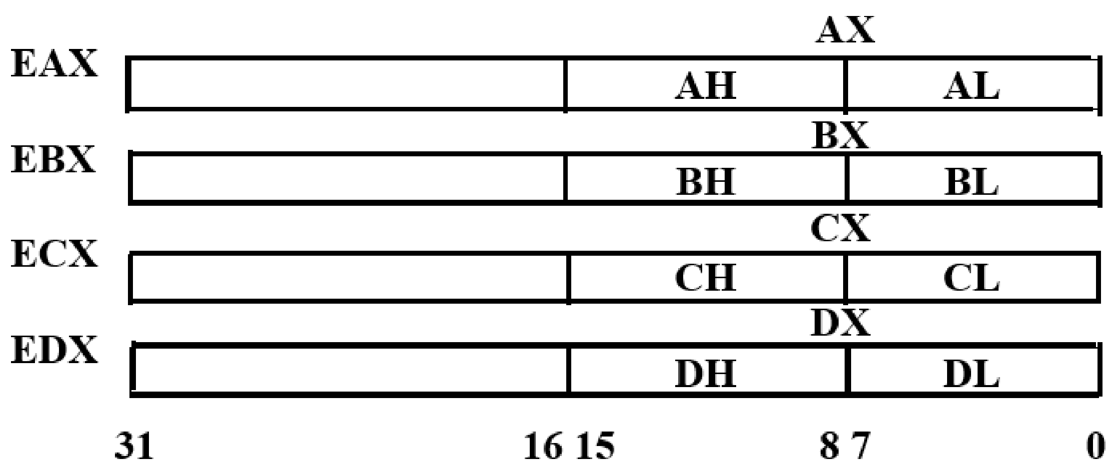


Рис. 6.2. Регістри загального призначення 32-бітної архітектури

Таким чином, в якості засобу для тимчасового зберігання даних усі регістри загального призначення (крім сегментних та вказівника стеку) є між собою рівноцінні. Між ти деякі команди вимагають для свого виконання використовувати тільки конкретні регістри. Наприклад, команда множення MUL потребує щоб один з множників був обов’язково у регістрі AX (або AL), а команда організації циклу LOOP виконує циклічний перехід CX раз.

З появою 64-х бітної архітектури, а саме мікропроцесорів **Intel 64** (EM64T / AMD64 / x86-64 / x64), які розроблялися фірмами Microsoft, Oracle, з’явилися і 64-бітові регістри – RAX, RBX, RCX, RDX, RSP, RBP, RSI, RDI, RIP (процесори AMD, рис. 6.3).

R означає Re-Extended (Accumulator / Base Register / Count Register / Data Register).

RAX		EAX
RBX		EBX
RCX		ECX
RDX		EDX
63	31	0

Рис. 6.3. Регістри загального призначення 64-бітної архітектури

Сегментні регістри

Процесори Intel *апаратно підтримують сегментну* організацію програми. Це означає, що *будь-яка програма складається з трьох сегментів: коду, даних і стека.* Відповідно і така організація пам'яті називається сегментною 5, 15].

Логічно машинні команди в архітектурі IA-32 (Intel Architecture, 32-bit або i386) побудовані так, що при вибірці кожної команди для доступу до даних програми або до стека неявно використовуються *адреси* з цілком певних сегментних регістрів. В залежності від режиму роботи процесора за їх вмістом визначаються адреси пам'яті, з яких починаються відповідні сегменти.

У програмній моделі IA-32 є шість *сегментних регістрів* **CS, DS, SS, ES, GS, FS**, що слугують для доступу до чотирьох типів сегментів:

- *сегмент коду* **CS** містить команди програми. Для доступу до цього сегменту слугує *регістр сегменту коду* (**Code Segment register**) **CS**. Він містить *початкову адресу сегменту коду з машинними командами*, до якого має доступ процесор (ці команди завантажуються в конвейєр процесора);
- *сегмент* даних **DS** містить дані, що оброблюються програмою. Для доступу до цього сегменту слугує регістр сегменту даних (**Data Segment register**) **DS**, який зберігає *адресу сегменту даних поточної програми*. Регістр сегменту даних плюс величина зсуву, визначена в команді, вказує на конкретну комірку в сегменті даних;
- *сегмент стека* **SS** є областю пам'яті (стеком), в якій програма тимчасово зберігає адреси та дані. Роботу із стеком процесор організовує за наступним принципом: останній елемент, записаний в цю область, вибирається першим. Для доступу до цієї області слугує регістр сегменту стека (**Stack Segment register**) **SS**, що містить *початкову адресу сегменту стека програми.* Ця адреса плюс зсув, який зберігається у вказівнику стека (stack pointer, SP), вказує на поточне слово у стеку [SS]+[SP];
- *додатковий сегмент даних* **ES** (**Extension** Data Segment registers) – *регістр сегмента розширення.* Неявно алгоритми виконання більшості машинних команд

припускають, що оброблювані ними дані розташовані в сегменті даних, адреса якого знаходиться в регістрі сегменту даних DS. Якщо програмі недостатньо одного сегменту даних, то вона має можливість задіяти ще три додаткові сегменти даних. Але на відміну від основного сегменту даних, адреса якого міститься в регістрі DS, при використанні додаткових сегментів даних їх адреси потрібно вказувати явно за допомогою спеціальних префіксів перевизначення сегментів в команді. Цей регістр використовується при роботі з символьними даними для виконання операцій над ними (рядковими даними) для управління адресацією пам'яті. У цьому випадку регістр ES пов'язаний з індексним регістром DI як <ES:DI> [ES]+[DI]. Якщо програма потребує використання сегменту ES, його необхідно ініціалізувати відповідним значенням початкової адреси;

• *додаткові сегментні регістри* **FS, GS** (extension data segment registers). введені вперше для процесор 80386 для засобів зберігання.

FS використовується для вказівки на інформаційний блок потоку (TIB) по процесам Windows. GS зазвичай використовується в якості вкзівника на локальне сховище потоку (TLS).

Розташування сегментів команд, даних, стеку у пам'яті наведено на рис. 6.4.

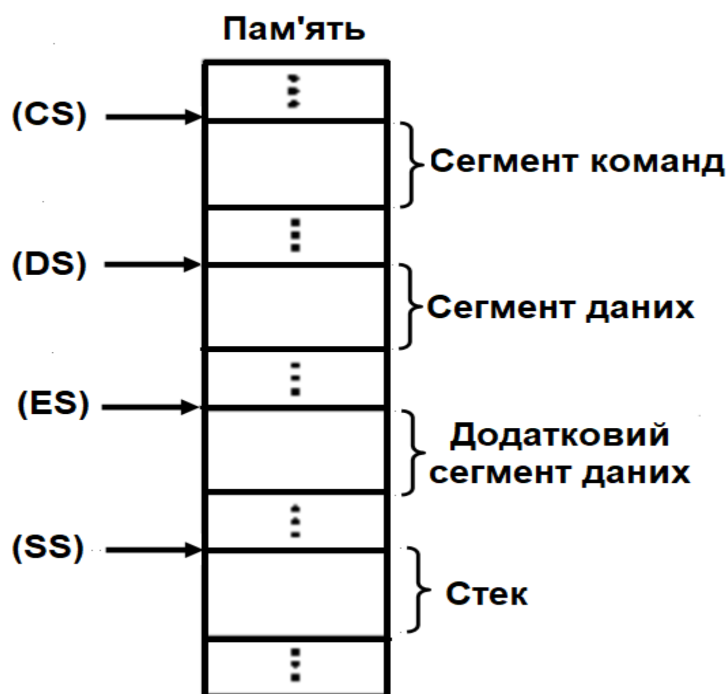


Рис. 6.4. Розташування сегментів у пам'яті

Регістри індексів

Регістри індексів ESI і EDI (їх молодші слова 16-розрядні регістри, відповідно SI і DI) використовується для індексованої адресації (рис. 6.5) [5, 15].

ESI (Source Index Register) **– регістр індексу джерела,** використовується у парі <DS:SI> для виконання операцій над рядками.

EDI (Destination Index Register) – **регістр індексу приймача,** також використовується у парі <ES:DI> для рядкових операцій.

Обидва індексних регістри **ESI**, **EDI** використовуються у операціях додавання і віднімання. Їх основне призначення полягає у зберіганні індексів (зсуву) відносно деякої бази (тобто початку масиву) при вибірці операндів з пам'яті.

Регістри **ESI**, **EDI** призначені для підтримки ланцюжкових операцій, тобто операцій, які виконуються послідовно.

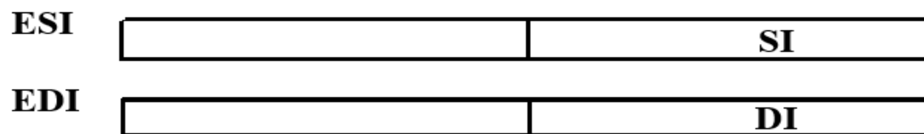


Рис. 6.5. Регістри індексів процесора 32-бітної архітектури

Регістри вказівників (EIP, ESP, EBP)

В архітектурі процесора на програмно-апаратному рівні використовується структура даних – **стек**. Сегмент – це область пам'яті, яка починається на межі параграфа (за адресою, кратною 16_{10}) , має розмір 64 Кбайт та може розташовуватися в будь-якому місці ОЗП (оперативного запам'ятовуючого пристрою). Реальний розмір сегменту такий, який необхідний для виконання програми. Для роботи зі стеком є спеціальні команди, які використовують наступні регістри (рис. 6.6) [5, 15]:

- **EBP** (*Base Pointer Register*) – регістр вказівника бази стеку, призначений для довільного доступу до даних в середині стеку. Він обробляє дані та адреси, що передає програма через стек ([SS]+[BP]). Він використовується у більшості арифметичних та логічних операціях або для тимчасового зберігання будь-яких даних.
- **ESP** (*Stack Pointer Register*) – регістр вказівника стека містить вказівник на вершину стека в поточному сегменті стека. Регістр SP зберігає значення зсуву, що вказує на поточне слово у стеку ([SS]+[SP]).

Наприклад, адреса початку сегмента SS – 4 BB30H, зсув у SP – 412H, тоді адреса наступної виконуваної команди це сума двох цих адрес = 4BF42H.

Регістри SI, DI, BP, SP на відміну від регістрів даних не допускають побайтову адресації.

Для визначення адреси наступної команди використовується регістр-вказівник команд EIP/IP.

- **EIP** (*Instruction Pointer*) **–** регістр вказівника команд містить зсув адреси наступної інструкції, яку треба виконати. Регістр IP пов'язаний регістром CS як CS:IP, тобто регістр IP вказує на поточну інструкцію всередині поточного кодового сегмента. Ця адреса плюс величина зміщення в командному покажчику (регістр IP) визначає адресу команди, яка повинна бути вибрана для виконання, - тобто IP пов'язаний з CS як CS:IP ([CS]+[IP]).

Наприклад, адреса початку сегмента у CS – 39D40H, зсув у IP – 524H , адреса наступної команди, яка повинна виконуватися – 3A054H. При виконанні кожної інструкції процесор змінює значення зсуву у регістрі IP. Таким чином, цей регістр постійно вказує на наступну команду, яка повинна виконуватися.

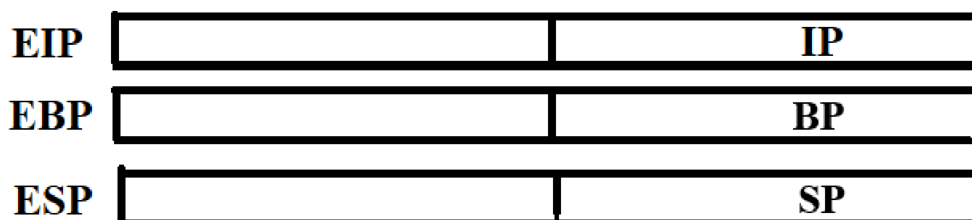


Рис. 6.6. Регістри вказівників

В літературі іноді до регістрів загального призначення, окрім регістрів EAX, EBX, ECX, EDX, відносять ще і регістри ESI, EDI, EBP, ESP.

Регістри стану і управління

Процесор містить два регістри, в яких постійно знаходиться інформація про стан як самого процесора, так і програми та команди, яку він в даний момент обробляє [5, 15]:

- регістр-**вказівник команд EIP/IP**;
- регістр **прапорців EFLAGS/FLAGS.**

За допомогою цих регістрів можна також обмеженим чином управляти станом процесора.

Регістр прапорців (**flag register**) EFLAGS/FLAGS відображає поточний стан процесора. Багато інструкції мітять в собі операції порівняння і математичні обчислення, які здатні змінити стан прапорів, а деякі інші умовні інструкції перевіряють значення прапорців стану, щоб перенести потік управління в інше місце.

Розрядність *регістра прапорців* (**flag register**) EFLAGS/FLAGS дорівнює 32(16) бітам. Окремі біти даного регістра мають певне функціональне призначення і називаються *прапорцями.* Молодша частина регістра EFLAGS/FLAGS повністю аналогічна регістру FLAGS процесора i8086 Цей регістр сигналізує про стан процесора та про те як виконалася арифметична або логічна команда.

Всі прапорці, розташовані в старшому слові регістра EFLAGS, мають відношення до управління захищеним режимом, тому розглянемо тільки регістр FLAGS (рис. 6.7). Значення прапорців наведено нижче.

Рис. 6.7. Регістр прапорців FLAGS

CF (carry flag) прапорець перенесення. Встановлюється в 1, якщо результат попередньої операції не помістився в приймачі і відбулося перенесення із старшого біта або якщо потрібно зайняти (при відніманні), інакше встановлюється в 0. Наприклад, після складання слова 0FFFFh і 1, якщо регістр, в який треба помістити результат, слово, в нього буде записано 0000h і прапор CF = 1.

PF (parify flag) прапорець парності (четности). Встановлюється в 1, якщо молодший байт результату попередньої команди містить парне число бітів, рівних 1; встановлюється в 0, якщо число одиничних біт непарне. (Це не те ж саме, що поділити на два. Число ділиться на два без залишку, якщо його наймолодший біт дорівнює нулю, і не ділиться, якщо він дорівнює 1.). Використовується в операціях введення-виведення.

AF (auxiliary carry flag) прапорець напівперенесення або допоміжного перенесення. Встановлюється в 1, якщо в результаті попередньої операції відбулося перенесення (або зайняли) з третього біта в четвертий. Цей прапор використовується автоматично командами двійково-десяткової корекції.

ZF (zero flaf) прапорець нуля. Встановлюється в 1, якщо результат попередньої команди нуль.

SF (sign flag) прапорець знаку. Цей прапор завжди дорівнює старшому біту результату.

TF (trap flag) прапорець тросування. Цей прапор був передбачений для роботи відладчиків, що не використовують захищений режим. Установка його в 1 призводить до того, що після виконання кожної команди програми управління тимчасово передається відладчику (викликається переривання 1 див. опис команди INT).

IF (interrupt flag) прапорець переривань. Установка цього прапорця в 1 призводить до того, що процесор перестає обробляти переривання від зовнішніх пристроїв (див. опис команди INT). Зазвичай його встановлюють на короткий час для виконання критичних ділянок коду.

DF ()direct flag) прапорець напряму. Цей прапорець контролює поведінку команд обробки рядків: коли DF = 0 рядки обробляються зліва направо, а коли DF=1 – навпаки, зправо наліво.

OF (overflow flag) прапорець переповнювання. Цей прапорець встановлюється в 1, якщо результат попередньої арифметичної операції над числами із знаком виходить за допустимі для них межі. Наприклад, якщо при складанні двох позитивних чисел виходить число із старшим бітом, рівним одиниці (тобто від'ємне) і навпаки.

Прапорці IOPL (рівень привілей введення-виведення) і **NT** (вкладене завдання) застосовуються у захищеному режимі.

При роботі з TASM, MASM директива **.386** формує 32-розрядний код, який може бути використаний при створенні *.COM та *.EXE файлів. При цьому треба використовувати турбодебагер td32 для того, щоб переглянути 32-розрядні регістри.

Контрольні питання:

1. Які вам відомі регістри загального призначення?
2. Які вам відомі сегментні регістри?
3. Для чого використовуються додаткові сегментні регістри?
4. Яке призначення регістру індексу джерела?
5. Яке призначення регістру індексу приймача?
6. Яке призначення регістру вказівника бази стека?
7. Яке призначення регістру вказівника стека?
8. Яке призначення регістру-вказівника команд і на що він вказує?
9. Яке призначення прапорця нуля?
10. Яке призначення прапорця переповнення?
11. Яке призначення прапорця перенесення?

Тема 5. Засоби компіляції та компонування

Лекція 7. Асемблери (призначення, основні поняття). Базові засоби розробки системних програм

Призначення. Основні поняття у мові програмування Асемблер: команда, псевдокоманда, макрокоманда. Інші директиви. Узагальнений формат команд. Основні функції транслятора з мови Асемблер. Одно- і дво прохідні асемблери. Типовий алгоритм роботи транслятора з мови Асемблер. Налагоджувачі. Призначення. Компонувальники. Засоби представлення об'єктних програм. Об'єднання програм підготовлених програм у єдиний виконуваний модуль.

Усі засоби розробки системних програм використовують мову асемблер:

- основні компоненти комп’ютерних ігор, ядра операційних систем реального часу, тобто ті складові, які потребують максимальної швидкості виконання;
- драйвери, програми, що працюють напряму з портами, звуковими та відеоплатами, тобто ті програми, які взаємодіють із зовнішніми пристроями;
- ядра багатозадачних операційних систем, програми, що переводять процесор у захищений режим, тобто програми, що використовують можливості процесора;
- віруси, антивіруси, програми від несанкціонованого доступу та інші, тобто програми, що використовують можливості операційної системи; та інші.

Для чого використовується асемблер? Він вирішує проблему продуктивності програми та доступу до апаратури. Можна написати значно більш швидкодіючу програму, ніж програма, написана мовою високого рівня. Для таких категорій прикладних програм як драйвери пристроїв, процедури BIOS швидкодія має надзвичайно важливе значення. Програми, призначені для обробки переривань та виключень операційних систем, а також контролери пристроїв вбудованих систем, які працюють в режимі реального часу, також пишуться тільки на асемблері. При обмеженні пам’яті написання програми на асемблері є єдиним виходом із ситуації. Знання асемблера дозволяє зрозуміти роботу комп’ютера.

Асемблер – це програма, яка перекладає текст з мови, яка зрозуміла людині, на мову, зрозумілій процесору, тобто це програма, яка перекладає мову асемблера в машинний код.

Разом з асемблером обов'язково повинна бути ще одна програма компоновник (*linker*), яка і створює виконувані файли з одного або декількох об'єктних модулів, одержаних після запуску асемблера. Крім цього для різних цілей можуть потрібно додаткові допоміжні програми - компілятори ресурсів, розширювачі DOS і тому подібне.

Асемблери та відповідні програми розробляють три провідні компанії: Microsoft (*masm* або *ml*), Borland (*tasm – Turbo Assembler*), *Watcom* (*wasm*) (див. табл.1). Важко говорити про те, продукція який з цих трьох компаній однозначно краще. З погляду зручності компіляції TASM краще підходить для створення 16-бітових програм для DOS, WASM для 32-бітових програм для DOS,

MASM для Windows. З точки зору зручності програмування розвиненість мовних засобів зростає у ряді WASM—MASM—TASM.

Таблиця 1. Асемблери і супутні програми

	Microsoft	**Borland**	**Watcom**
DOS, 16 біт	masm або ml, link (16 біт)	tasm tlink	wasm wlink
DOS, 32 біта	masm або ml, link (32 біта) і dosx link (16 біт) і dos32	tasm tlink wdosx або dos32	wasm wlink dos4gw, pmodew, zrdx або wdosx
Windows EXE	masm386 або ml, link (32 біта) rc	tasm tlink32 brcc32	wasm wlink wrc
Windows DLL	masm386 або ml, link (32 біта)	tasm tlink32 implib	wasm wlink wlib

TASM насправді містить два синтаксичних стандарти мови Асемблера – MASM та Ideal.

Існують й інші компілятори, наприклад, безкоштовно поширювані в мережі Internet. Це ***NASM*** (Netwide Assembler for Win32) - безкоштовний асемблер Intel86 з відкритим кодом, підтримує усю лінійку X86-процесорів, включаючи IA64 (Linux, MS DOS, Win32), але користуватися ними простіше, якщо ви вже знаєте турбо- або макроасемблер.

FASM (*Flat Assembler*) - асемблер, для архітектур IA-32 та x86-64, використовує синтаксис Intel. Відомий своєю швидкістю компіляції, оптимізацією розміру скомпільованого коду, портованістю на різні ОС та широкими можливостями препроцесора (макросами). FASM є вільним та відкритим програмним забезпеченням.

Безкоштовно поширюваний *GNU асемблер*, *gas*, взагалі використовує абсолютно несхожий синтаксис. Є асемблери: *SPARC*, *Motorola*.

У всіх програмах зустрічаються помилки. Для цього потрібен відладчик. Окрім пошуку помилок відладчики іноді застосовують і для того, щоб досліджувати роботу існуючих програм. Безумовно, *наймогутніший відладчик* на сьогодні - *SOFTICE від NuMega Software*. Це фактично єдиний відладчик для Windows 95/NT, що дозволяє досліджувати все - від ядра Windows до програм на C++, який підтримує одночасно 16- і 32-бітовий код і багато що інше. Інші

популярні відладчики, поширювані разом з відповідними асемблерами, *Codeview (MS), Turbo Debugger (Borland) і Watcom Debugger (Watcom)*.

Ще одна особливість асемблера, що відрізняє його від всієї решти мов програмування, можливість *дизасемблювання*. Тобто, маючи виконуючий файл, за допомогою спеціальної програми (*дизасемблера*) майже завжди можна отримати початковий (рос. исходный) текст на асемблері. Наприклад, можна дизасемблювати BIOS вашого комп'ютера і дізнатися, як виконується перемикання відеорежимів, або драйвер для DOS, щоб написати такий же для Windows. Дизасемблер не необхідний, але іноді виявляється зручно мати його під рукою. Кращі дизасемблери на сьогодні Sourcer від V Communications і IDA.

І нарешті, остання необов'язкова, але украй корисна *утиліта шістнадцятирічний редактор*. Такі редактори (*hiew, proview, Iview, hexit*) теж мають вбудований дизасемблер, так що можна, наприклад, відкривши в такому редакторі свою програму, подивитися, як скомпілювалася та або інша ділянка програми, поправити – будь-яку команду асемблера або змінити значення констант і тут же, без перекомпіляції, запустити програму, щоб подивитися на результат змін.

Рівень асемблера реалізується в комп’ютері шляхом *трансляції*. Для перекладу користувацьких програм з однієї мови на іншу розроблені спеціальні програми – транслятори. Мова, на якій з самого початку написана програма, називається вхідною, а мова, на якій вона транслюється – вихідною. Трансляція потрібна, коли є процесор для вихідної мови, але немає для вхідної. Тобто організовується новий рівень – спочатку транслюється програма, написана для вихідної мови, і створюється еквівалент програми на вихідній мові – об’єктна або двійкова виконавча, а потім отримана програма виконується. Ці два кроки не виконуються одночасно, а послідовно. Є інші мови – інтерпретатори, тобто за один крок виконується програма – оператор за оператором, наприклад, Java-код, Basic.

Транслятори поділяються на асемблер (вхідна мова є символічним представленням числової машинної мови) та компілятор (вхідна мова є мовою високого рівня, наприклад, С, а вихідна – числова машинна мова).

Мова асемблера – це мова, в якій кожний оператор відповідає рівно одній машинній команді, тобто є взаємно однозначна відповідність між машинними командами та операторами. Асемблер дає доступ програмісту до усіх об’єктів та команд машини. Такої можливості мова високого рівня немає.

При роботі з командним рядком для проведення трансляції необхідно щоб ім’я асемблерного файлу було не більше 8 символів (DOS).

Формат оператора в асемблері. Структура асемблерного оператора відображає структуру відповідної **машинної команди** [17, 18].

Асемблерні *оператори складаються з чотирьох полів:*

мітка операція(команда/директива) операнди ; коментар.

Мітки слугують символічними іменами для адрес пам’яті. Вони дозволяють по символічному імені отримати доступ де зберігаються команди та дані.

Кожна програма на мові Асемблер містить окрім машинних команд, які необхідно виконувати процесору, ще і директиви.

Директиви – спеціальні інструкції - *команди, які вказують самому асемблеру*, як організовувати різні секції програми, де розташовуються дані, а де команди, які дозволяють створювати макровизначення, вибирати тип використовуваного процесора, організовувати зв'язки між процедурами і т.д. Команди для асемблера називають **псевдокомандами** або **асемблерними директивами**.

У чому полягає різниця між командою асемблера і директивою? В процесі трансляції команда перетворюється на машинний код, директива не перетворюється на машинний код, а вказує транслятору виконати певні дії під час трансляції.

Мітка (якщо є), команда / директива і операнд (якщо є) розділяються принаймні одним символом пробілу або табуляції. Якщо команду або директиву необхідно продовжити на наступному рядку, то використовується символ «зворотний слеш»: «\». ***За замовчуванням Асемблер не розрізняє великі і малі літери.***

Приклади кодування:

```
Count db 1            ; Ім'я, директива, один операнд
mov eax, 0            ; Команда, два операнда
cbw                   ; команда
```

Коментарі у тексті програми починаються з символу «;». Де б ви не поставили цей символ, усі символи, які зустрічаються після нього, асемблером сприймаються як коментар. Коментар може бути пустим або складатися з будь-яких друкованих символів. Коментар може займати рядок

; Обчислення коефіцієнтів рівняння

або записуватися одразу після команди

ADD AX, BX ; *додавання до загальної суми*

Коментар не включається до складу генерованого коду.

В асемблері є певні імена, які зарезервовані в якості:

Інструкції (команди) – назви команди, наприклад, IMUL;

Директиви (SEGMENT - ENDS) – початок – кінець програмного коду, початок – кінець даних, початок – кінець макросу, початок – кінець процедури;

Оператори, які використовуються у складі виразів, наприклад, FAR, SIZE;

Наперед визначені символи - @Data, @Model, @Code, @Stack, що повертають інформацію вашій програмі.

Використання вище наведених зарезервованих імен призводить до помилок.

В якості ідентифікатора (це символ або ім’я, на яке можна посилатися в програмі, використовуються:

Ім’я у директиві для опису даних, наприклад, COUNTER DB 1;

Мітка – ім’я процедури або сегменту, або команди, наприклад,
MAIN PROC FAR
M30: ADD   BL, 20

Ідентифікатори можуть містити:
- букви латинського алфавіту великі та маленькі **A-Z**, **a-z**,
- цифри **0-9**, але не можна, щоб цифра була першим символом,
- спеціальні символи: знак питання (**?**), знак підкреслення (_), знак долара (**$**), знак комерційне ат (@), крапка(**.**), але не в якості першого символу.

Перший символ ідентифікатора повинен бути буквою або спеціальним символом, окрім крапки.

Наприклад,

	Ідентифікатор	Операція	Операнд	Коментар
Директива:	COUNT	DB	1	; ім’я
Інструкція:	L20:	MOV	AX, 1	; мітка

Для полегшення читаності асемблерних текстів прийнято, що мітка починається в першій позиції в рядку, команда в 17-ій (дві табуляції), операнди в 25-ій (три табуляції) і коментарі в 41-ій або 49-й. Якщо рядок складається тільки з коментаря, його починають з першої позиції.

Псевдокоманди визначення даних
(директиви ініціалізації та опису даних)

Дані можуть розташовуватися в ділянках пам’яті, які називаються сегментами. Зазвичай це або сегмент даних або сегмент коду. Сегменти описуються за допомогою директиви ***SEGMENT***.

Для ініціалізації простих типів даних в Асемблері використовують спеціальні директиви ***Dx***, які вказують компілятору на виділення певних обсягів пам’яті (див. табл. 2) [17, 18]. Для асемблера має значення тільки довжина комірки, в яку розміщуються дане.

Таблиця 2. Директиви асемблера для визначення простих типів даних

Довжина (біт)	Ініціалізація	Опис
8	***DB***	***BYTE***
16	***DW***	***WORD***
32	***DD***	***DWORD***
64	***DQ***	***QWORD***
80	***DT***	***TBYTE***

Мнемокоди директив ініціалізації даних Dx означають наступне:

DB (Define Byte) – визначити байт,

DW (Define Word) – визначити слово (2 байти),

DD (Define Double Word) – визначити подвійне слово (4 байти),

DF (Define Far word) – визначити вказівник дальнього слова, а саме визначити 6 байтів (адреса у форматі 16-бітний селектор: 32-бітне зміщення),

DQ (Define Quarter Word) – визначити зчетверене слово (8 байт),

DT (Define Ten Bytes) – визначити 10 байтів (80-бітні типи даних, що використовуються FPU).

Директиви даних можуть мати ім'я, у випадку коли є ім'я за цією адресою можна заносити якесь значення або витягувати дане. Регістр букв для імені та директив – байдужий.

Поле значення може містити одно або декілька чисел, рядків символів, які взяті в одинарні або подвійні лапки, операторів *?* та *DUP*, розділених комою. Всі встановлені таким чином дані опиняться у вихідному файлі, а ім'я змінної відповідатиме адресі першого з вказаних значень. Наприклад, набір директив:

```
text_string     db  'Hello student!'
number          dw  7
table           db  1,2,3,4,5,6,7,8,9,0Ah,0Bh,0Ch,0Dh,0Eh,0Fh
float_number    dd  3.5e7
```

заповнює даними 35 байтів. Перші 14 байтів містять ASCII-коди символів рядка «Hello student!», і змінна *text_string* вказує на першу букву в цьому рядку, так що команда

mov al, text_string

зчитує в регістр AL число 48h (код латинської букви H).

Директива *number* виділяє слово та присвоює початкове значення 7, символ *number* визначає адресу, за якою зберігається значення 7.

Директива *table* виділяє місце для 15 байтів, присвоює їм початкові значення від 1 до 0Fh, визначає символ *table* як адреса, за якою зберігається значення 1.

```
sym     db   26     ; Число26 займає 1 байт
wsim    dw   257    ; Число 257 займає 1 слово
hsym    dw   201h   ; Число 257 у 16-річній формі
dsym    dw   1000000001b ; Число 257 у двійковій формі
array1  db   10,12,30,25; Байтовий масив з 4-х членів
array2  dw   10,12,30,25; Масив слів з 4-х членів
ml      dd   40000000    ; 40 мільйонів у подвійному слові
mm      dd   0FFFFFFFFh; Максимально можливе ціле число у подвійному слові
```

Шістнадцятирічне число закінчується літерою *h*, ***якщо 16-річне число починається з літери, то перед ним обов'язково треба ставити 0 (нуль),*** щоб транслятор зрозумів, що це саме число, а не комірка пам'яті.

Наприклад, 0FFFF8h, 0A1h, 0C000h.

Якщо замість точного значення вказаний знак *?*, змінна вважається неініціалізованою і її значення на момент запуску програми може опинитися будь-яким. Якщо потрібно заповнити ділянку пам'яті даними, що повторюються, використовується спеціальний оператор DUP, що має формат лічильник DUP (значення). Наприклад, таке визначення:

table_512w dw 512 dup(?)

створює масив з 512 неініціалізованих слів, на перше з яких вказує змінна table_512w. Як аргумент в операторі DUP можуть виступати декілька значень, розділених комами, і навіть додаткові вкладені оператори DUP.

db 15 dup(30) створює 15 байтів зі значенням 30 кожний.

db 36 dup(1,2,3,4,5,6) виділяє 36 байтів зі значеннями від 1 до 6 у перших шести байтах, наступні шість значень також мають значення від 1 до 6, тобто асемблер повторює значення у дужках до тих пір, поки не будуть заповнені 36 байтів.

```
ss      db    256 dup (25); Масив з 256 байтів, який заповнений числом 27
nul     dw    500 dup(0)      ; Масив з 500 слів, заповнений нулями
dword dd      10 dup(3FFFFh)  ; Масив з 10 подвійних слів з числами 3FFFFh
```

Фрагменти тексту (символьні рядки) вводяться за допомогою оператора *db*, текст записується в одинарних або подвійних лапках:

```
sym          db 'A'
msg          db '~~~~Обережно, яма!~~~'
pname db "Петренко П.П."
stk          db 512 dup ('*')  ;   Символьний масив
```

Моделі пам’яті асемблера

Директива **SEGMENT** починає новий сегмент, директива **ENDS** завершує його. Дозволяється починати новий сегмент, потім починати сегмент даних, потім переходити знову до текстового сегменту і т.д.

Для управління сегментними регістрами (їх шість: сегмент коду, сегмент стека, сегмент даних та три додаткових сегменти даних) виникає потреба управляти директивами сегментації (кожний сегмент починається **SEGMENT** і закінчується **ENDS**). Для цього використовують директиву зазначення моделі пам’яті **MODEL**, яка визначає кількість та розмір сегментів 17, 18].

Директива **MODEL** повинна знаходитися перед будь-якою з директив оголошення сегментів (.DATA, .STACK, .CODE, SEGMENT). Параметр ***модель_пам’яті*** є обов'язковим.

.MODEL [модифікатор] модель_пам’яті [, угода_про_виклики] [, параметр_стека]

Основні моделі пам'яті асемблера, які використовують для MS DOS, WIN16, наступні моделі:

- **.MODEL TINY** (код, дані та стек розташовуються в одному і тому ж сегменті розміром до 64 Кб, поєднані в одну групу з іменем DGROUP, використовується при створення програм формату *.COM та написанні невеликих програм на асемблері);
- **.MODEL SMALL** (код займає один сегмент, дані та стек поєднані в інший сегмент, для їх опису можуть використовуватися різні сегменти, але поєднані в одну групу з іменем DGROUP; ця модель використовується для більшості програм асемблера). Дані та код при використанні цієї моделі адресуються як ***near*** (ближні);
- **.MODEL MEDIUM** (код займає декілька сегментів, по одному на кожний об'єднуючий програмний модуль, а усі дані – в одному, тому для доступу до даних використовується тільки зсув, а для виклику підпрограм застосовують команди дальнього виклику процедури);
- **.MODEL COMPACT** (код розташовується в одному сегменті, для зберігання даних використовуються декілька сегментів, для звернення до даних треба вказувати сегмент та зсув – дані дальнього типу *far*; для сегменту кода використовується ближня адресація *near*);
- **.MODEL LARGE** (код і дані можуть займати декілька сегментів, по одному на кожний об'єднуючий програмний модуль);
- **.MODEL HUGE** (введена у мову С, щоб зняти обмеження на довжину сегмента 64 КБ), ця модель еквівалентна моделі пам'яті *Larg;*.
- **.MODEL FLAT** (також як і для **TINY**, але використовуються 32-бітні сегменти, максимальний розмір сегмента, що містить і дані, і код, і стек, становить 4 Мбайт).

Для WIN32 (починаючи з i386) використовується .MODEL FLAT, в якій кількість та розмір сегмента обмежується можливостями процесора та материнської плати.

Для розробки програми для моделі *Flat* перед директивою *.model flat* слід записати одну з директив: **.386, .486, .586 або .686**. Бажано вказувати той тип процесора, який використовується в машині, хоча на машинах з Intel Pentium можна вказувати директиви *.386 і .486.* Операційна система автоматично ініціалізує сегментні регістри при завантаженні програми, тому модифікувати їх потрібно у тому випадку тільки, якщо необхідно змішувати в одній програмі 16- і 32-розрядний код. Адресація даних і коду є ближнею (*near*), при цьому всі адреси і вказівники є 32-розрядними.

Параметр **модифікатор** використовується для визначення типів сегментів і може приймати значення *use16* (сегменти обраної моделі використовуються як 16-бітові) або *use32* (сегменти обраної моделі використовуються як 32-бітові).

Параметр **угода_про_виклики** використовується для визначення способу передачі параметрів при виклику процедури з інших мов, в тому числі і мов високого рівня (C++, Pascal). Параметр може набувати таких значень:

C, BASIC, FORTRAN, PASCAL, SYSCALL, STDCALL.

При розробці модулів на асемблері, які будуть застосовуватися в програмах, написаних на мовах високого рівня, слід враховувати які угоди про виклики

підтримує та чи інша мову. Більш докладно угоди про виклики розглядаються при аналізі інтерфейсу програм на асемблері з програмами на мовах високого рівня.

Параметр **параметр_стека** встановлюється рівним NEARSTACK (регістр SS дорівнює DS, області даних і стек розміщуються в одному і тому ж фізичному сегменті, тобто SS повинен вказувати на DGROUP.) або FARSTACK (реєстр SS НЕ дорівнює DS, області даних і стека розміщуються в різних фізичних сегментах). За замовчуванням приймається значення NEARSTACK.

Інші директиви

Директива **ASSUME** повідомляє компілятору, що зазначений у ній сегмент необхідно зв'язати з іменем сегмента або групи. В якості сегментних регістрів застосовуються регістри CS, DS, ES, SS.

Директива **ALIGN** передає наступний рядок (зазвичай дані) за адресою, яка задана аргументом директиви. Наприклад, якщо поточний сегмент мітить 61 байт даних, то після виконання директиви *ALIGN 4* наступною адресою, що виділяється, буде адреса 64.

Директива **EQU** дає символічну назву деякому виразу. Наприклад, після виконання директиви BASE EQU 1000 символ BASE можна використовувати в програмі замість значення 1000.

Вираз, який слідує за директивою *EQU* , може містити декілька символів, поєднаних знаками арифметичних та інших операцій, наприклад:

LIMIT EQU 4*BASE+2000

Директиви **PROC** та **ENDP** визначають початок та кінець асемблер них процедур.

Директиви **MACRO** та **ENDM** визначають початок та кінець макроса**.**

Директиви **PUBLIC** та **EXTERN** визначають видимість символів, значення PUBLIC – забезпечує доступність символу для інших файлів, значення EXTERN – об'являє символ зовнішнім, який визначений в іншому файлі.

Директива **STRUC** дозволяє визначити структуру даних, яка не ініціалізується, а до неї в подальшому звертаються як до псевдокоманди.

Наприклад,

```
ім’я        struc
      поля
ім’я        ends
```

поля – будь-який набір псевдокоманд визначення змінних або структур. В подальшому для створення структури в пам'яті використовують псевдокоманду:

```
мітка:  ім’я   <значення>
```

Наприклад,

```
point    struc       ; Визначення структури
x         dw     0   ; Три слова зі значеннями
y         dw     0   ; за замовченням 0,0,0
z         dw     0
color     db     3 dup(?) ; та три байти
```

```
point     ends

cur_point  point    <1,1,1,255,255,255> ; Ініціалізація
             movx, cur_point.x     ; Звернення до слова "x"
```

Директива **INCLUDE** ім'я_файла включає в текст програми інший файл. Це необхідно, коли включені файли містять визначення констант, структур, макросів, необхідні для різних файлів.

Директива **END** помічає кінець програми.

Є й інші директиви.

Макрозасоби мови Асемблера. Це теж директиви**.** Макрозасоби дозволяють оперувати з набором команд як з однією командою. Використання макрозасобів дозволяє: спростити та скоротити вхідний текст програми, скористатися в програмі вже відлагодженими командами, створити свою бібліотеку макровизначень та скористатися розробленими бібліотеками макровизначень, вставляти їх в свою програму для прискорення програмування.

Структура простої програми:

```
Title Privet           ; Заголовок програми
.Model Small           ; Модель пам'яті
<Сегмент коду>               <Сегмент стеку>
<Сегмент даних>   або    <Сегмент даних>
<Сегмент стеку>              <Сегмент коду>
```

Схема розроблення програми на асемблері

Традиційно на ринку асемблерів для мікропроцесорів фірми Intel є два пакети:

- «Макроасемблер» MASM фірми Microsoft;
- Turbo Assembler TASM фірми Borland.

В цих пакетів багато загального. Пакет макроасемблера фірми Microsoft (MASM) отримав свою назву тому, що він дозволяв програмістові задавати *макровизначення (або макроси),* які являють собою іменовані групи команд. Вони мають властивість, що їх можна вставляти в програму в будь-якому місці, вказавши тільки ім'я групи у місці вставки. Пакет *Turbo Assembler (TASM*) цікавий тим, що має два режими роботи. Один з цих *режимів - це MASM*, підтримує всі основні можливості макроасемблера MASM. Інший режим, називається *IDEAL*, надає зручніший синтаксис написання програм, ефективніше використовує пам'ять при трансляції програми і має інші нововведення, що наближають компілятор асемблера до компіляторів мов високого рівня.

В ці пакети входять транслятори, компонувальники, відладчики та інші утиліти для підвищення ефективності процесу розробки програм на асемблері. Ми скористаємося тим, що транслятор TASM, працюючи в режимі MASM, підтримує

майже всі можливості транслятора MASM. Достатньо мати пакет асемблера фірми Borland — TASM 3.0 або вище.

Етап 1. Введення вхідного тексту програми (код програми)за допомогою текстового редактора. В Windows таким редактором може бути Блокнот (*Notepad*), в MS DOS – редактор edit.com. При виборі редактора слід враховувати, щоб він не повинен вставляти спеціальні символи форматування. ***Microsoft Word в якості редактора для асемблерних програм не підходить***. Є редактор *Asm Editor for Windows* (http://www.avtlab.ru). Файл, створений за допомогою текстового редактора, повинен мати розширення **.asm* (ім'я файлу <=8), в полі Тип файлу треба вказати Все файлы.

Вивести на екран повідомлення:

```
Title Prog
.Model Small
text    segment                 ; Початок сегмента команд
        assume  CS:text, DS:data  ;  Сегментний регістр CS вказує на
    сегмент
                                  команд
                        ; сегментний регістр DS - на сегмент даних
                        ; begin -  точка входу у програму
begin:  mov   AX, data          ; Адресу сегмента спочатку завантажуємо в
AX,
        mov   DS, AX            ;а потім переносимо з AX в DS
        mov   AH, 09h           ; Функція DOS 9h означає виведення на екран
        mov   DX, offset mesg;  Адреса повідомлення, що виводиться, повинно
                                  бути в DX
        int   21h               ; Виклик DOS - функція переривання
        mov   AH, 4Ch           ; Функція 4Ch - завершення програми
        mov   Al, 0             ; Код 0 - код успішного завершення
        int   21h               ; Виклик DOS
text    ends                    ; Кінець сегмента
data    segment                 ; Початок сегменту даних
mesg    db "NACHINAEM ! $"; Текст, що виводиться
data    ends                    ; Кінець сегменту даних
stk     segment stack           ; Початок сегменту стека
db 256 dup (0)                  ; Резервуємо 256 байт для стека
stk     ends                    ; Кінець сегменту стека
  end begin                     ; Кінець тексту програми з точкою входу
```

Етап 2. Трансляція програми. Для створення виконуваного модуля виконується трансляція програми. На етапі трансляції вирішується декілька задач:

- переведення команд асемблера у відповідні машинні команди;
- побудова таблиць символів;
- розширення макросів;

- формування файла лістингу та об'єктного модуля

Програма, яка вирішує ці задачі, називається асемблером. Результатом роботи асемблера є два файли – файл об'єктного модуля та файл лістингу.

Об'єктний модуль включає в себе представлення вхідної програми в машинних кодах та деяку іншу інформацію, потрібну для відладки та компоновки його з іншими модулями. Для виконання трансляції використовується пакет TASM.EXE (програма-асемблер) [17, 18]. В командному рядку ця програма запускається наступним чином:

TASM.EXE [ключі] ім'я_вхідного_файлу [, ім'я_об'єктного_файла] [, ім'я_файла_лістингу] [, ім'я_файла_перехресних_посилань].

Пам'ятати формат запуску в командному рядку асемблера *tasm.exe* необов'язково. Для отримання швидкої довідки про нього достатньо *запустити tasm.exe без параметрів*. В квадратних дужках позначені необов'язкові параметри. Обов'язковим є лише ім'я вхідного файла. Цей файл повинен знаходитися на диску, обов'язково мати розширення **.asm*. За іменем вхідного файла через кому можна задати імена об'єктного файла, лістингу та перехресних посилань. Якщо ці імена не задати, то відповідно ці файли не будуть створені.

Ключі – це режими роботи транслятора.

При запуску транслятора треба використовувати два ключа:

/la - виведення розширеного варіанту лістингу транслятора,

/zi – отримання повної інформації для відладчика.

Наприклад, **tasm /la /zi pr.asm**

Таким чином, **результатом роботи транслятора є створення трьох модулів**: ***.lst (лістингу), *.crf (таблиці перехресних посилань: таблиці символічних імен змінних,** які використовуються в програмі, та таблиці відносних посилань, в якій вказується в якому операторі визначено ім'я і де зустрічається), ****.obj*** (об'єктного).

Файл лістингу містить номер рядка тексту програми, код асемблера вхідної програми, а також розширену інформацію про цей код. Для кожної команди асемблера вказуються її машинний (об'єктний) код і зсув в кодовому сегменті. Крім того, в кінці лістингу TASM формує таблиці з інформацією про мітки і сегменти, що використовуються в програмі. Якщо є помилки або сумнівні ділянки коду, то TASM включає в кінець лістингу повідомлення про них. Якщо порівняти їх з повідомленнями, що виводяться на екран, то видно, що вони співпадають. Крім того, що дуже зручно, ці ж повідомлення включаються в текст лістингу безпосередньо після помилкового рядка.

Якщо вхідна програма містить макроси, то транслятор асемблера складається з двох частин – безпосередньо транслятора, який формує об'єктний модуль, та макроасемблера. Обробка програми на асемблері з використанням макрозасобів неявно здійснюється транслятором у дві фази. На першій фазі працює макроасемблер, тобто використовуються підстановки, це може стосуватися і даних і програмних кодів, і на виході макроасемблера програма буде містити символічні аналоги машинних команд процесора (це буде чистий асемблер, тобто вхідна

програма у внутрішньому представленні компілятора без макросів). А на другій фазі транслятора працює безпосередньо асемблер, який формує об'єктний код, що містить текст вхідної програми в машинному коді.

Таким чином, асемблер може прочитати програму двічі, кожне прочитання вхідної програми називається проходом, а транслятор, який читає вхідну програму двічі називається двопрохідним. На першому проході при першому проході у більшості асемблерів використовуються, принаймні, три таблиці: таблиця символічних імен (збираються та зберігаються всі визначення символів, у тому числі і мітки, таблиця директив (значення символів вже відомі), таблиця кодів операцій (найти код операції, визначити набір її операндів та розрахувати довжину команди), іноді ще використовується таблиця лібералів (константи, для яких асемблер автоматично резервує пам'ять). Також на першому проході зберігаються всі макровизначення і розширяються виклики у міру їх появи.

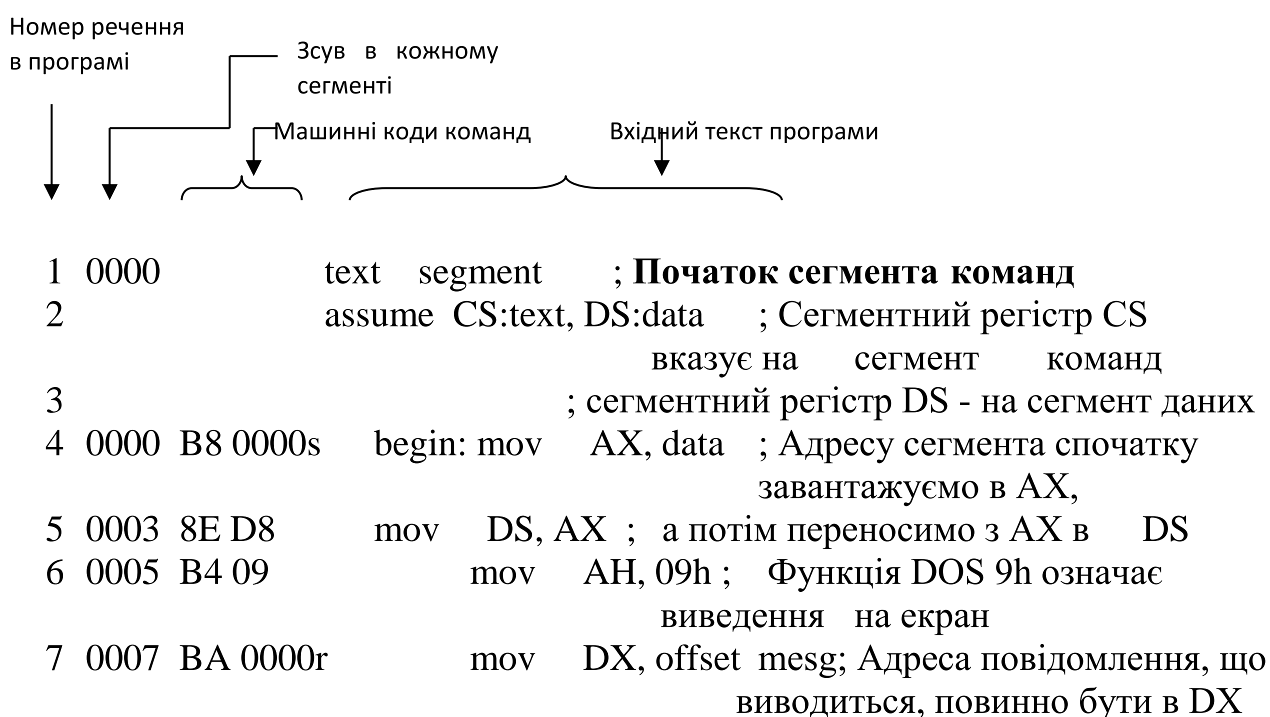
На другому проході асемблерна програма читається ще раз та перетворюється у проміжну форму, ця проміжна форма зберігається в таблиці. Потім відбувається другий прохід, але вже не по вхідній програмі, а по таблиці. Мета другого проходу – створити об'єктну програму, а також інформацію, яка необхідна для компоновки в один виконуючий файл.

Між тим, вхідна програма може містити помилки: не визначений символ, використовується невірний код операції, недопустиме застосування регістра, відсутній оператор END та інші. В цьому випадку помилки необхідно усунути, отримати об'єктний модуль, а потім переходити до наступного етапу – компоновки програми.

Результат трансляції:

Turbo Assembler Version 4.1 02/14/21 18:49:51 Page 1

Privet.asm




```
1  0000              text    segment       ; Початок сегмента команд
2                    assume  CS:text, DS:data     ; Сегментний регістр CS
                                           вказує на сегмент команд
3                                  ; сегментний регістр DS - на сегмент даних
4  0000  B8 0000s     begin: mov    AX, data   ; Адресу сегмента спочатку
                                                 завантажуємо в AX,
5  0003  8E D8        mov    DS, AX  ;  а потім переносимо з AX в   DS
6  0005  B4 09               mov     AH, 09h ;   Функція DOS 9h означає
                                           виведення   на екран
7  0007  BA 0000r            mov     DX, offset  mesg; Адреса повідомлення, що
                                              виводиться, повинно бути в DX
```

```
 8  000A  CD 21              int     21h ;      Виклик DOS – функція
                                                переривання
 9  000C  B4 4C              mov     AH, 4Ch ;  Функція 4Ch – завершення
                                                програми
10  000E  B0 00              mov     Al, 0   ; Код 0 - код успішного завершення
11  0010  CD 21         int       21h     ; Виклик DOS
12  0012                     text    ends       ; Кінець сегмента команд
     ↑
     └──── Розмір в байтах сегменту команд

13  0000                     data    segment       ; Початок сегменту даних
          Коди символів, що утворюють
          повідомлення
14  0000  4E 41 43 48 49 4E 41+     mesg    db     "NACHINAEM  !  $";  Текст,
                                                            що виводиться
15        45 4D 20 21 20 24
16  000D                     data    ends       ; Кінець сегменту даних
     ↑
Розмір в байтах сегменту даних

17  0000                     stk     segment stack   ; Початок сегменту  стека
18  0000  0100*(00)          db      256 dup     (0)  ; Резервуємо 256 байт для
                                                 стеку
19  0100                     stk     ends       ; Кінець сегменту стека
      ↑
   Розмір в байтах сегменту стека

20                           end begin    ;     Кінець тексту програми з
                                                точкою входу
```

```
Turbo Assembler    Version 4.1    02/14/21 18:49:51         Page 2
Symbol Table

Symbol Name                    Type Value

??DATE                         Text  "02/14/21"
??FILENAME                     Text  "Privet  "
??TIME                         Text  "18:49:51"
??VERSION                      Number 040A
@CPU                           Text  0101H
@CURSEG                        Text  STK
@FILENAME                      Text  PRIVET
@WORDSIZE                      Text  2
BEGIN                          Near  TEXT:0000
MESG                           Byte  DATA:0000
```

Groups & Segments	Bit	Size	Align	Combine	Class
DATA	16	000D	Para	none	
STK	16	0100	Para	Stack	
TEXT	16	0012	Para	none	

Етап 3. Створення завантажувального модуля – компоновка програми за допомогою програми *Tlink.exe*, результатом роботи цього етапу є створення модуля *ім'я_вхідного_файлу.exe*.

Головна мета цього етапу – перетворити код та дані в об'єктних модулях в їх виконуване відображення – виконуваний двійковий код. Введення етапу компоновки пов'язано з тим, що повинна бути можливість поєднати разом модулі, які написані на одній і тій же мові або на різних мовах. Формат об'єктного файла дозволяє при певних умовах поєднати декілька окремо відтрансльованих вхідних модулях в один модуль – виконати компоновку об'єктних модулів, тобто під'єднання до файлу основної програми файлів з підпрограмами та налаштувати зв'язки між ними. Компоновщик має ще і другу функцію – змінити формат об'єктного файла та перетворити його у виконуваний файл. Результатом компоновщика роботи є створення завантажувального файла з розширенням *.exe. Після цього операційна система може завантажити такий файл у пам'ять та виконати його. Запуск компоновщика (редактора зв'язку) здійснюється в командному рядку:

TLINK [ключі] список об'єктних модулів [, ім'я завантажувального модуля]

Ключі можна подивитися просто запустивши tlink без параметрів.

При запуску компоновщика треба використовувати два ключа:

/x – не створювати файл з розширенням **.map* (подавляється формування файлу лістингу компоновки, в якому відображається карта завантаження, без цього файлу можна обійтись),

/v – передає у завантажувальний файл символьну інформацію, яка дозволяє відладчику TD виводити на екран повний текст вихідної програми, включаючи мітки, коментарі та ін.

Список об'єктних модулів – це обов'язковий параметр, файли розділені пропуском або знаком плюс: **tlink /x /v prog+prog1+prog2** або вказати повний шлях до цих файлів.

Для отримання виконуваного модуля треба запустити ***tlink.exe /v prog.obj***, в результаті буде отриманий модуль ***prog.exe.***

Компоновщик поєднує окремі адресні простори об'єктних модулів і єдиний лінійний простір. Для цього виконуються наступні кроки:

1. Компоновщик будує таблицю об'єктних модулів та їх розмірів.
2. На основі цієї таблиці він приписує початкові адреси кожному об'єктному модулю.

3. Компоновщик знаходить усі команди, які звертаються до пам'яті, та додає до кожної з них константу перерозподілу, яка дорівнюється початковій адресі цього модуля.

4. Компоновщик знаходить усі команди, які звертаються до процедур, та вставляє в них адреси цих процедур.

На першому кроці будується таблиця об'єктних модулів, в якій задаються ім'я, довжина та початкова адреса кожного модуля. Наприклад,

Модуль	Довжина	Початкова адреса
A	400	100
B	600	500
C	500	1100
B	300	1600

Структура об'єктного модуля. Об'єктні модулі зазвичай складаються з шести частин:

1. Ідентифікація (містить ім'я модуля, дані про довжину різних частин модуля, дата асемблювання);
2. Таблиця точок входу (список символів, визначених в модулі разом з їх значеннями, тобто це символічні імена в директиві PUBLIC);
3. Таблиця зовнішніх посилань (список символічних імен, які використовуються в даному модулі, а визначені в інших модулях, тобто це зовнішні символи, які об'явлені в директиві EXTERN; є ще один список, який вказує які саме символічні імена використовуються тими чи іншими машинними командами, за допомогою цього списку компоновщик вставляє правильні адреси в команди, які використовують зовнішні імена, оскільки процедура може викликати інші процедури, імена яких об'явлені як зовнішні), іноді в комп'ютерах точки входу та зовнішніх посилань поєднують в одну таблицю;
4. Машинні команди та константи (ця частина об'єктного модуля завантажується в пам'ять для виконання, усі інші частини компоновщика відкидаються ще до початку виконання програми);
5. Словник перерозподілу пам'яті (до команд, які містять адреси пам'яті, повинна додаватися константа перерозподілу, оскільки компоновщик сам не може визначити які слова в частині 4 містять команди, а які константи);
6. Кінець модуля (ця частина містить вказівку на кінець модуля).

Більшості компоновщикам потрібно два проходи. На першому проході компоновщик зчитує усі об'єктні модулі та будує таблицю імен та розмірів модулів, а також глобальну таблицю символів, яка складається із усіх точок входу та зовнішніх посилань. На другому проході модулі по черзі зчитуються, перерозподіляються в пам'яті та компонуються.

Усі версії Windows підтримують динамічну компоновку. При динамічній компоновці використовується спеціальний файловий формат DLL (Dynamic Link Library – бібліотека динамічної компоновки). Бібліотеки динамічної компоновки можуть містити процедури, дані або все це разом узяте. Вони використовуються для того, щоб два і більше процесів могли розділяти процедури і дані бібліотеки. Більшість файлів мають розширення **.dll*, але можуть бути і інші розширення – для бібліотеки драйверів (driver libraries) **.drv*, для бібліотек шрифтів (font libraries) **.fon*. Типовими прикладами DLL – файлів є процедури сполучення (сопряжения) з бібліотекою системних викликів Windows та великими графічними бібліотеками. Використовуючи dll – файли, економиться простір в пам’яті та диску. Якщо б та чи інша бібліотека зв’язувалася статично з кожною програмою, яка її використовує, то треба було б включати цю бібліотеку в усі виконувані двійкові програми в пам’яті та на диску. А так використовується лише одна бібліотека.

Якщо б компілятор або асемблер зчитував декілька окремих вхідних процедур і зразу б переводив їх в готову програму на машинній мові, то при зміні одного оператора в одній вхідній програмі необхідно було заново транслювати всі вхідні програми. Коли ж кожна програма транслюється окремо, то транслювати треба тільки одну змінену процедуру, але заново треба скомпонувати усі об’єктні модулі. Компоновка відбувається значно швидше, ніж трансляція, тому при доопрацюванні програми трансляція і компоновка відбуваються швидко. Це дуже важливо, коли програма містить багато модулів.

Наступним обов’язковим етапом є етап відладки.

Етап 4. Відладка програми – цей етап здійснюється за допомогою програми ***Tdeburg.exe*** [17, 18].

На цьому етапі відповідно до алгоритму перевіряється правильність функціонування як окремих фрагментів коду, так і програми в цілому. Однак навіть успішне завершення відладки ще не є гарантією того, що програма буде правильно працювати з усіма можливими вхідними даними. Тому необхідне тестування програми, тобто перевірити її роботу з завідома некоректними вхідними даними. Для цього складаються тести. В разі необхідності треба повернутися до попередніх етапів.

Розрізняють такі категорії помилок в програмах:

- синтаксичні (виникають на етапі компіляції через помилки в конструкції мови програмування або просто неуважності);
- помилки компоновщика (виникають через невірну стиковку різних модулів);
- помилки при виконанні програм (ділення на нуль, невдала файлова операція, вихід індексу за межі масиву та ін.);
- логічні помилки (саме для пошуку такого типу помилок застосовується процес відладки).

Специфіка програм на асемблері полягає у тому, що вони інтенсивно працюють з апаратними ресурсами комп’ютера. Це означає, що необхідно постійно відслідковувати вміст певних регістрів та областей пам’яті. Тому для локалізації логічних помилок в програмах використовують програмні відладчики.

Відладка (Debug, Debugging) за термінологією IBM – це знаходження, локалізація та усунення помилкових операцій в програмах або відмовлень в комп'ютері.

Відладчики бувають двох типів:

- *інтегровані* (реалізовані у вигляді інтегрованого середовища, схожі на мови Turbo Pascal, Visual C ++ та ін.);
- *автономні* (реалізовані як окремі програми).

Найбільш популярними в країнах СНД є багаторежимні відладчики, що поставляються у складі різноманітних систем програмування: ***TurboDebugger*** фірми Borland, відладчик CodeViewer фірми Microsoft, ASD фірми Microsoft, Watcom і ряд інших.

Процес відладки в загальному випадку можна розділити на чотири етапи:

1. Виявлення помилки.
2. Пошук її місцезнаходження.
3. Визначення причини помилки.
4. Виправлення помилки.

Турбо відладчик можна використати для відладки будь-якої програми мовою C або C++ для компілятора сімейства Borland, Турбо Паскаля, Турбо Асемблера.

Турбо відладчик можна використати для рішення двох важких проблем процесу відладки: пошуку місця знаходження помилки та її причин. Турбо відладчик за допомогою спеціальних можливостей по затриманню виконання програми дозволяє досліджувати стан програми в будь-якій точці. Можна тестувати нові значення змінних, щоб побачити, як вони впливають на вашу програму. Ці можливості реалізуються за допомогою трасування, покрокового виконання, перегляду, змін й простежування.

Трасування: дозволяє виконувати програму по одному оператору, тобто послідовно крок за кроком виконується одна машинна інструкція (трасування в прямому напрямі).

Зворотне трасування можна виконувати програму у зворотному напрямку крок за кроком.

Покрокове виконання. Можна виконувати програму по одному оператору, але пропускати виклики процедур і функцій. Якщо є впевненість, що в процедурах і функціях немає помилок, то пропуск їхнього виклику збільшить швидкість відладки.

Перегляд: Можна в Турбо відладчику створити спеціальне вікно для показу всіляких речей - змінних, їхніх значень, точок зупинки, вмісту стека, файлів реєстрації, даних, файлів вихідних текстів, кодів ЦП, пам'яті, регістрів, інформації процесора, арифметики із плаваючою крапкою, виводу програми.

Перевірка. Можна в Турбо відладчику отримати вміст складних структур даних з вашої програми.

Зміна. Можна змінити вміст змінної (як локальної, так і глобальної) на нове значення.

Простежування. Можна виділити деякі програмні змінні й простежувати зміну їхніх значень в процесі роботи програми.

Вхідний модуль має бути від трансльований з ключем /zi:

Tasm /la /zi ім'я_вхідного модуля.asm

Ключ /la виводить повний лістингу транслятора, ключ /zi – дозволяє транслятору зберігати зв'язок символічних імен у програмі разом з їх зсувом в сегменті коду.

Редагування модуля повинно бути виконано з ключем /v :

Tlink /x /v ім'я_об'єктного_модуля.obj

Ключ /x не створює файл з розширенням *.map, ключ /v вказує на необхідність збереження відладочної інформації у виконуваному файлі.

В результаті відкривається вікно відладчика TD з назвою Module, в якому відображається вхідний текст програми на асемблері. У вікні відображається курсор виконання (у вигляді трикутника), що вказує на першу команду, яка має бути виконана. Цій команді передує мітка, яка є точкою входу в програму. В кінці вхідного тексту це ж ім'я записано як операнд в кінцевій директиві END.

Управління роботою виконується за допомогою системи меню. Є два таких меню: головне - знаходиться у верхній частині екрана (можна натиснути F10 і також буде викликано це меню) та контекстне (ALT+A10).

Запустити програму у відладчику можна в одному з чотирьох режимів:

- безумовне виконання;
- покрокове виконання;
- виконання до поточного положення курсора;
- виконання з встановленням точок переривання.

Режим безумовного виконання програми доцільно виконувати, коли треба подивитися на загальну поведінку програми. Для запуску програми у цьому режимі треба натиснути клавішу F9. В точках, де треба ввести дані, відладчик відповідно до логіки роботи програми буде вводити дані. Аналогічні дії будуть виконуватися і при виведенні даних. Для перегляду або введення цієї інформації можна відкрити вікно користувача (вибравши в меню команд Windows→User screen або натиснувши клавіші ALT+F5). Якщо програма працює вірно, то на цьому відладку можна і закінчити. Якщо виникли проблеми, то слід вибрати такі режими відладки:

Режим виконання програми до поточного положення курсора вибирається у тому випадку, коли необхідно перевірити правильність функціонування деякого фрагмента програми. Для активізації цього режиму треба встановити курсор на потрібний рядок програми та натиснути клавішу F4. Програма запуститься та зупиниться на відміченій команді, не виконавши її. Далі за необхідності можна перейти у покроковий режим.

У режимі виконання програми з встановленням точок переривання програма після запуску буде зупинятися у суворо визначених точках переривання (breakpoints). Перед виконанням програми необхідно встановити ці точки в програмі, після цього слід перейти до потрібного рядка та натиснути клавішу F2. Вибрані рядки підсвічуються. Встановлені раніше точки переривання можна прибрати – для цього треба знову перейти до потрібного рядка та натиснути F2. Після встановлення точок переривання програма запускається клавішею F9 (режим безумовного виконання). На першій ж точці переривання програма зупиниться.

Після цього можна подивитися стан процесора та пам'яті, а потім продовжити виконання програми. Це можна і зробити у покроковому режимі або до дійти до наступної точки переривання.

Режим виконання програми по крокам застосовується для детального вивчення її роботи. У цьому режимі виконання переривається на кожній машинній (асемблерній) команді. При цьому є можливість спостерігати результат виконання команд. Для активізації цього режиму треба натиснути клавішу F7 (Run → Trace into) або F8 (Run → Step over). Ці дві клавіші активізують покроковий режим, їх відмінність полягає у тому, що в потоці команд зустрічаються команди переходу в процедурі або на переривання. При натисканні F7 відладчик здійснює перехід до процедури або перериванню і зупиняється. При натисненні клавіші F8 виклик процедури або переривання відпрацьовується як одна команда, управління передається наступній команді програми. Крім роботи з вікном Module в цьому режимі корисно використовувати вікно CPU, яке викликається через головне меню View → CPU. Вікно CPU відображає стан процесора та складається з п'яти підпорядкованих вікон:

- у вікні з вхідною програмою в дизасемльованому вигляді представлена таж сама програма, що і у вікні Module, але уже в машинних кодах. Покрокову відладку можна виконувати прямо у цьому вікні. Рядок з поточною командою підсвічується;
- у вікні регістрів процесора (Registers) відображається поточний стан регістрів (за замовчуванням тільки регістрів процесора i8086). Щоб побачити регістри i486 або Pentium, треба задати режим їх відображення. Для цього треба клацнути правою кнопкою миші в області вікні регістрів та вибрати в контекстному меню команду Registers 32-bit-Yes;
- у вікні прапорців (Flag) відображається поточний стан прапорців процесора відповідно до їх мнемонічних назв;
- у вікні стека (Stack) відображається вміст пам'яті, яка виділена для стека. Адреса області стека визначається вмістом регістрів SS та SP;
- вікно дампа оперативної пам'яті (Dump) відображає вміст області пам'яті за адресою, яка формується з компонентів, вказаних в лівій частині вікна. У вікні можна побачити вміст довільної області пам'яті. Для цього в контекстному меню треба вибрати відповідну команду.

Вікно дампа CPU відображає видиму частину програмної моделі процесора. Деякі з підпорядкованих вікон можна вивести окремо, хоча зручніше працювати з вхідним текстом у вікні Module, між тим часто виникає потреба відслідковувати стан процесора за допомогою підпорядкованих вікон саме вікна CPU. Поєднати можливості вікон Module та CPU можна, вибравши в меню View імена потрібних підпорядкованих вікон CPU.

Перервати виконання програми в будь-якому з режимів можна натиснувши на клавішу Ctrl+F2.

За умови створення відлагоджених програм, наприклад п'яти програм, можна створити *пакетний файл make.bat* (кожний модуль транслюється окремо, а потім збирається редактором зв'язку):

```
Tasm /zi A1.asm
Tasm /zi A2.asm
Tasm /zi A3.asm
Tasm /zi A4.asm
Tasm /zi A5.asm
Tlink /v A1.obj A2.obj A3.obj A4.obj A5.obj
```

Контрольні питання:

1. Призначення асемблера.
2. Назвіть різновиди асемблерів.
3. Який формат оператора в асемблері?
4. Що таке директива?
5. У чому полягає різниця між командою асемблера та директивою?
6. Які правила побудови ідентифікатора в асемблері?
7. Назвіть директиви для визначення простих типів даних.
8. Що таке модель пам'яті?
9. Які моделі пам'яті вам відомі?
10. Яка модель пам'яті використовується для створення com-файлу?
11. Яка модель пам'яті використовується для створення exe-файлу?
12. Як здійснити трансляцію програми, які при цьому утворюються файли?
13. Як створити завантажувальний модуль?
14. Яка структура лістингу програми?
15. Яким чином здійснюється відладка програми?

Тема 6. Типи даних та засоби адресації до них
Лекція 8. Типи даних, засоби адресації до них

Подання даних у комп’ютерах. Двійкова, восьмирічна, десяткова, шістнадцятирічна системи числення, переведення з однієї системи у іншу.
Формат представлення базових даних (байт, слово, подвійне слово). Числовий діапазон беззнакових чисел, від’ємні числа, додатковий код, двійкові цілі числа зі знаком, цілі числа без знака, числа двійково-десяткової системи числення, числа з плаваючою комою). Регістрова адресація. Безпосередня адресація. Непряма адресація. Адресація по базі із зсувом.

Подання даних у комп’ютерах

Програми та дані в комп'ютері зберігаються у двійкової системі числення (форматі).

Система числення (с/с) - сукупність прийомів і правил для запису числових даних. Позиційна система числення - с/с, позиція цифр в якій впливає на зображення числових даних [17, 18].

Загальноприйнята форма запису числа є насправді не що інше, як скорочена форма запису розкладання по ступенях основи системи числення, наприклад

$130678_{10}=1*10^5+3*10^4+0*10^3+6*10^2+7*10^1+8*10^0$

Загальна формула надання чисел у позиційної системі числення:

$$D = n_0*m^0+ n_1*m^1+ n_2*m^2+. . . + n_k*m^k \qquad (1)$$

D – числове дане,
n - цифра на позиції і у числі,
m - основа системи числення (2, 8, 10, 16).

Десяткове представлення позначається d – Decimal, двійкове – b (Binary), шістнадцятирічне – h (Hexadecimal).

Основа системи числення - кількість символів (знаків), що використовуються в системі (десяткова, двійкова, восьмирічна, шістнадцятирічна).

$999_{(10)}=1111100111_2=1*2^0+1*2^1+1*2^2+0*2^3+0*2^4+1*2^5+1*2^6+1*2^7+1*2^8+1*2^{*9}=1+2+4+32+64+128+256+512$

$735_{(8)}=5*8^0+3*8^1+7*8^2$

$428_{(16)}=8*16^0+2*16^1+4*16^{*2}=8+32+1024$

Нижче в таблиці наведені перші 16 натуральних чисел, записані в десятковій, двійковій, восьмирічній і шістнадцятирічній системах числення (табл. 3).

Для представлення двійкового числа у восьмеричній формі необхідно розбити його цифри на групи з трьох цифр, кожну справа наліво і замінити кожну трійку цифр відповідною восьмеричною цифрою. Наприклад, число $1010110101111_2 =12657_8$, тому що

1 010 110 101 111
1 2 6 5 7.

Таблиця 3. Перші 16 чисел в десятовій, восьмерічній і шістнадцятирічній системах счислення

10	2	8	16
0	0000	0	0
1	0001	1	1
2	0010	2	2
3	0011	3	3
4	0100	4	4
5	0101	5	5
6	0110	6	6
7	0111	7	7
8	1000	10	8
9	1001	11	9
10	1010	12	A
11	1011	13	B
12	1100	14	C
13	1101	15	D
14	1110	16	E
15	1111	17	F

Зворотний перехід від восьмирічного представлення до двійкового здійснюється заміною кожної восьмеричної цифри відповідною трійкою двійкових.

Перехід від двійкового представлення до шістнадцятирічного і в зворотньому напрямі здійснюється відповідно до описаного для восьмирічної системи з тією лише різницею, що шістнадцятирічна цифра замінюється не трьома, а чотирма двійковими. Наприклад, $1111\ 1010\ 0100_2$=$FA4_{16}$

або 1111 1010 0100b= FA4h

$B5_{16}$ =$1011\ 0111_2$.або B5h=1011 0111b

Шістнадцятирічне представлення чисел зручно використовувати для позначення адрес розміщення даних у пам'яті, тобто адреси комірок оперативної пам'яті кодуються шістнадцятирічними числами..

При переведенні в десяткову систему счислення потрібно число розкласти по ступенях основи системи счислення**.**

Із шістнадцятирічної у десяткову:

$$(a_к\ a_{к-1}\quad a_1\ a_0)_{16} = a_к *16^{\,к} + a_{к-1} * 16^{\,к-1} + ...+a_1 *16 +a_0$$

$92C8_{16}=9*16_{10}^{3}+2*16_{10}^{2}+12*16_{10}^{1}+8*16_{10}^{0}=9*4096+2*256+12*16+8=36864+512+192+8=37576_{10}$

Із восьмирічної в десяткову:

$$(a_к\ a_{к-1}\ \ \ a_1\ \ a_0)_8 = a_к * 8^{к} + a_{к-1} * 8^{к-1} + ... + a_1 * 8 + a_0$$

$735_8 = 7*8_{10}^2+3*8_{10}^1+5*8_{10}^0=7*64+3*8+5*1=448+24+5=477_{10}$

Із двійкової у десяткову:

$110100101_2=1*2_{10}^8+1*2_{10}^7+0*2_{10}^6+1*2_{10}^5+ 0*2_{10}^4+0*2_{10}^3+1*2_{10}^2+0*2_{10}^1+ 1*2_{10}^0=$
$256+128+0+32+0+0+4+0+1=421_{10}$

Для переведення цілої частини числа з однієї системи числення в іншу необхідно послідовно число, записане в системі з основою p, ділити на основу нової системи числення q. Отримані залишки слід записати у зворотному порядку. Ділення виконується за правилами старої системи числення, а запис залишків - у новій системі числення.

Для переведення дробової частини з однієї системи числення в іншу необхідно послідовно дробову частину множити на основу нової системи числення, виділяючи цілі частини, які і будуть утворювати запис дробової частини числа в новій системі числення**.**

Приклади: перевести з десяткової у двійкову систему число

$999{,}35_{10}=1111100111{,}01011_2$

для цілої частини:

```
999 | 2
  1 | 499 | 2
        1 | 249 | 2
              1 | 124 | 2
                    0 | 62 | 2
                         0 | 31 | 2
                              1 | 15 | 2
                                   1 | 7 | 2
                                       1 | 3 | 2
                                           1 | 1
```

для дробової частини:

```
0, 35
    2
0,|70
  | 2
1,|40
  | 2
0,|80
  | 2
1,|60
  | 2
1,|20
```

Перевести 125_{10} у вісімкову систему числення:

125

```
_125 |8              _ 15 |_8
 120   15  - частка      8   1
   5       - остача      7
```

Результат: $175_8=1*8^2+7*8+5=64+56+5=125_{10}$

Перевести 125_{10} у шістнадцятирічну систему числення:

```
_125 |16               Результат 125₁₀=7D₁₆
 112   7  - частка
  13      - остача
```

Формат представлення базових даних

Комп'ютер оперує з великою кількістю самих різноманітних даних, які мають певний формат представлення. Це числові та нечислові. До нечислових відносяться оброблення текстів, управління базами даних, оброблення рядків, булеві дані, вказівники для оброблення машинних команд. Формати допустимих даних залежать від моделі комп'ютера та набору команд для їх обробки.

Поняття *тип даних* носить подвійний характер. З погляду розмірності процесор апаратно підтримує наступні основні типи даних [17, 18]:

Байт — вісім послідовно розташованих бітів, пронумерованих від 0 до 7, при цьому біт 0 є наймолодшим значущим бітом.

Слово — послідовність з двох байтів, що мають послідовні адреси. Розмір слова — 16 бітів; биті в слові нумеруються від 0 до 15. Байт, що містить нульовий біт, називається *молодшим байтом,* а байт, що містить 15-й біт, — *старшим.* Процесори Intel мають важливу особливість — молодший байт завжди зберігається за меншою адресою. *Адресою слова* вважається адреса його молодшого байта. Адреса старшого байта може бути використана для доступу до старшої половини слова.

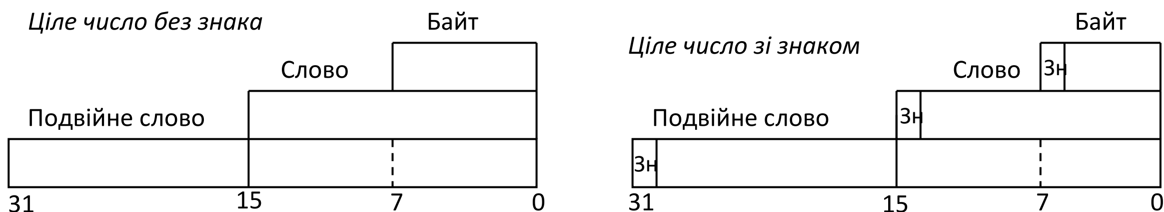
Подвійне слово — послідовність з чотирьох байтів (32 біти), розташованих за послідовними адресами. Нумерація цих бітів здійснюється від 0 до 31. Слово, що містить нульовий біт, називається *молодшим словом,* а слово, що містить 31-й біт, — *старшим словом.* Молодше слово зберігається за меншою адресою. *Адресою подвійного слова* вважається адреса його молодшого слова. Адреса старшого слова може бути використана для доступу до старшої половини подвійного слова.

Збільшене учетверо слово— послідовність з восьми байтів (64 біти), розташованих за послідовними адресами. Нумерація бітів проводиться від 0 до 63. Подвійне слово, що містить нульовий біт, називається *молодшим подвійним словом,* а подвійне слово, що містить 63-й біт, — *старшим подвійним словом.*

Молодше подвійне слово зберігається за меншою адресою. *Адресою збільшеного учетверо слова* вважається адреса його молодшого подвійного слова. Адреса старшого подвійного слова може бути використана для доступу до старшої половини почетвереного слова.

Основні типи даних процесора

Окрім трактування типів даних з погляду їх розрядності, процесор на рівні команд підтримує *логічну* інтерпретацію цих типів. (Зн означає знаковий біт).



Цілий тип із знаком — двійкове значення із знаком розміром 8, 16 або 32 біта.

Знак в цьому двійковому числі міститься в 7, 15 або 31 біті відповідно. *Нуль* в цих бітах в операндах *відповідає позитивному числу, а одиниця — від'ємному*. **Від'ємні числа зберігаються у зворотному додатковому коді.**

Числові діапазони для цього даних наступні:

- 8-розрядне ціле (Shortint) - 128 **(80h)** до +127 (**7F**h); -2^7 - $+2^7-1$, (діапазон від'ємних чисел від - 001 до -128) або необхідна пам'ять - 1байт;
- 16-розрядне ціле (Integer) - від -32 768 (**8000h)** до +32 767 **(7FFFFh)**;
  -2^{15} - $+2^{15}-1$, або необхідна пам'ять - 2 байти;
- 32-розрядне ціле (Longint) - від -2^{31} (**8001869Fh)** до $+2^{31}-1$**(7FFE7960h)** ;
  -2 147 583 648 +2 147 583 647, або необхідна пам'ять - 4 байти.

Цілий тип без знаку - двійкове значення *без знаку* розміром 8, 16 або 32 бита.

Числовий діапазон беззнакових чисел наступний:

- **байт** — від 0 до 255; (2^8) або **00h – FFh** → 1111 1111b →255d
- **слово** — від 0 до 65 535; (2^{16}) або **0000h – FFFFh** → 1111 1111 1111 1111b
- **подвійне слово** — від 0 до 2^{32} - 1. (2 147 583 647)

або **0000 0000h – FFFF FFFFh** → 1111 1111 1111 1111 1111 1111 1111 1111b

В обчислювальній техніці прийнято вважати **від'ємними** усі числа, в яких встановлено старший біт, тобто це **числа в діапазоні 8000h …FFFFh**. Позитивними вважаються числа зі скинутим старшим бітом, тобто це числа в діапазоні 0000h …7FFFh. Від'ємні числа записуються у додатковому коді.

Додатковий код – це таке число, яке треба додати до даного числа для одержання 0 у всіх його розрядах й 1 у розряді ліворуч від самого старшого (або це число, всі цифри в якому є доповненням даного числа до основи системи счислення).

Наприклад, у 10 с/с 934 66
66 934
1000

Щоб отримати зворотний додатковий код для двійкової системи счислення, треба інвертувати усі розряди та додати одиницю до молодшого розряду.

1010 0101+

1 **0110**

Таким чином, для зміни знаку числа виконуємо інверсію, тобто замінюємо в двійковому уявленні числа всі одиниці нулями і нулі одиницями, а потім додаємо 1. Наприклад, хай використовується змінна типу слово:
150 = 0096h = 0000 0000 1001 0110b
інверсія дає: 1111 1111 0110 1001b
+1 = 1111 1111 0110 1010b = 0FF6Ah
Перевіримо чи одержане число насправді -150: сума з +150 повинна бути дорівнюватися нулю:
+150 + (-150) = 0096h + FF6Ah = 10000h ;
Завдання. Запишіть числа: -37, -59, -172, -23, -33, -1, -254 у двійковій та шістнадцятирічній системах счислення у зворотному додатковому коді.

Таблиця 16-бітових чисел із зазначенням їх знакових та без знакових значень

16-річне представлення	Десяткове представлення: без знака	зі знаком	
0000h	00000	00000	Нуль
0001h	00001	+00001	Мінімальне позитивне число
0002h	00002	+00002	
0003h	00003	+00003	Діапазон позитивних чисел
…………			
7FFEh	32766	+32766	
7FFFh	32767	+32767	Максимальне позитивне число
8000h	32768	-32768	Максимальне від'ємне число
8001h	32769	-32767	
…………			Діапазон від'ємних чисел
FFFDh	65533	-00003	
FFFEh	65534	-00002	
FFFFh	65535	-00001	Мінімальне від'ємне число

Машинне слово можна записати в діапазоні 64 К різних чисел: 0000h ÷ FFFFh або 0 ÷ 65535. В обчислювальній техніці вважають від'ємними числами, в яких встановлено старший біт і знаходиться в діапазоні 8000h ÷ FFFFh. Позитивними числами вважаються такі, в яких скинуто старший біт і знаходяться в діапазоні 0000h ÷ 7FFFFh.

Числа з плаваючою комою

У процесорах Intel всі операції з плаваючою комою виконує спеціальний пристрій, FPU (*Floating Point Unit),* з власними регістрами і власним набором команд, що поставлялося спочатку у вигляді співпроцесора (8087, 80287, 80387, 80487), а починаючи з 80486DX – вбудовується в основний процесор.

Числовий процесор може виконувати операції з сімома різними типами даних, представленими в таблиці, – три цілих двійкових, один цілий десятковий і три типи даних з плаваючою комою.

Таблиця 4. Типи даних FPU

Тип даних	Біт	Кількість значущих цифр	Межі
Ціле слово	16	4	-32768 – 32767
Коротке ціле	32	9	$-2*10^{9}$ – $2*10^{9}$
Довге ціле	64	18	$-9*10^{18}$ – $9*10^{18}$
Упаковане десяткове	80	18	-99..99 – +99..99 (18 цифр)
Коротке дійсне	32	7	$1.18*10^{-38}$ – $3.40*10^{38}$
Довге дійсне	64	15-16	$2.23*10^{-308}$ – $1.79*10^{308}$
Розширене дійсне	80	19	$3.37*10^{-4932}$ – $1.18*10^{4932}$

Дійсні числа зберігаються, як і всі дані, у формі двійкових чисел. Двійковий запис числа з плаваючою комою аналогічна десятковим, тільки позиції праворуч від коми відповідають не діленню на 10 у відповідному ступені, а діленню на 2.

Логічні операції

Один з широко поширених варіантів значень, які може приймати один біт, це значення «истина» і «ложь», використовувані в логіку, звідки відбуваються так звані «логічні операції» над бітами. В асемблері використовуються чотири основні операції И (AND), ИЛИ (OR), исключающее ИЛИ (XOR) і отрицание (NOT), дія яких приводиться в таблиці:

Таблиця 5. Логічні операції

И	ИЛИ	исключающее ИЛИ	НЕ
0 AND 0 = 0 0 AND 1 = 0 1 AND 0 = 0 1 AND 1 = 1	0 OR 0 = 0 0 OR 1 = 1 1 OR 0 = 1 1 OR 1 = 1	0 XOR 0 = 0 0 XOR 1 = 1 1 XOR 0 = 1 1 XOR 1 = 0	NOT 0 = 1 NOT 1 = 0

Всі ці операції побітові, тому, щоб виконати логічну операцію над числом, треба перевести його в двійковий формат і виконати операцію над кожним бітом, наприклад:

96h AND 0Fh = 10010110b AND 00001111b = 00000110b = 06h

Типи даних процесора Pentium IV

Процесор Pentium IV підтримує двійкові цілі числа зі знаком, цілі числа без знака, числа двійково-десяткової системи числення, числа з плаваючою точкою. Процесор підтримує арифметичні команди, булеві операції, операції порівняння, для роботи з символами рядків.

Кодування інформації в комп'ютері

Символи (characters) в комп'ютері зберігаються у вигляді числового коду. В рамках Windows розроблений стандарт **ANSI** (American Standard Code for Information Interchange), який є однобайтною схемою кодування. Перші символи 0,..127 відповідають кодуванню ANSI. Сюди входять деякі управляючі коди (символ з кодом 0Dh – кінець рядка), розділові знаки, цифри (символи з кодами 30h–39h), великі (41h–5Ah) і маленькі (61h–7Ah) латинські букви

Друга половина 128, …255– національні алфавіти. В якості кодової сторінки використовується Code Page 1251. Для кодування ієрогліфів був придуманий стандарт кодування символів, що має фіксовану довжину представлення одного символу (16 біт або 2 байти) – **Unicode**, який дозволяє закодувати усі алфавіти світу. Цей кодовий набір застосовується в Windows NT. Існують ще так звані **розширені ANSI-коди**. Це послідовність двох символів по одному байту: у першому знаходиться нуль, у другому – **scan**-код. Для усіх цих стандартів розроблені відповідні таблиці. Наприклад, код символу A дорівнює 0041, L - 004C, у цьому можна впевнитися викликавши Таблицю символів.

Для підтримки кириличного алфавіту застосовують два варіанти таблиці кодування символів – кодову таблицю 866 для операційної системи MS DOS і кодову таблицю 1251 для графічної операційної оболонки Windows. . Інше розповсюджене кодування зветься КОИ-8 (код обміну інформацією, восьмизначний) мають широке поширення в комп'ютерних мережах на території СНД: для російського алфавіту це таблиця кодування KOI8-R, для українського KOI8-U.

Загальні операції над числами, які виконує процесор ПЕОМ

Процесор виконує набір арифметичних операції: додавання, віднімання, множення та ділення. Всі операції базуються на виконанні елементарних операцій двійкового додавання та зсувів даних що знаходяться у регістрах процесора.

Таблиця 6. Операції додавання.

Таблиця операції	Приклади	
0 + 0 = 0 0 + 1 = 1 1 + 0 = 1 1 + 1 = 0 (перенесення 1)	1010 +1001 10011	1101 +1001 10110

Віднімання.

Заміняється в машині операцією додавання із числом у додатковому коді.

Приклади.

Виконати операцію з отриманням позитивного результату:

937 - 148 = 786

Для цього процесору ПЕОМ необхідно одержати доповнення

(148) = 852 і замінити віднімання на додавання з доповненням.

+ 934 + 852 = #1## 786

наявність перенесення 1 у старший розряд вказує на те, що результат - число позитивне й отримане в прямому коді.

Виконати операцію з отриманням від'ємного результату:

154 - 388 = - 234

Для цього процесору ПЕОМ необхідно одержати доповнення

(388) = 612 та замінити віднімання на додавання з доповненням:

+ 154 + 612 = #x## 766

відсутність перенесення в старший розряд означає, що результат – від'ємне число та отримане число представлене в додатковому коді, тому потрібно визначити доповнення для результату:

доповнення (766) = - 234 - це від'ємний результат.

Аналогічні приклади для 2 с/с:

1010	Доповнення (0110) = інверсія (1001) +1 = 1010
- 0110	+1010
0100	#1# 0100
0011	Доповнення (1010) = інверсія (0101) +1 = 0110
- 1010	+0110
- 0111	#X# 1001
	доповнення (1001) = 0111

Процесор ПЕОМ має набір команд для роботи з числами у двійковою десяткової формі. В цьому випадку потрібно виконувати додаткові команди корекції результату для вірного надання кожної цифри результату.

Засоби адресації

Більшість команд процесора викликаються з аргументами, які в асемблері прийнято називати *операндами*. Наприклад: команда складання вмісту регістра з числом вимагає задання двох операндів вмісту регістра і числа.

Засобом, або режимом адресації, називають процедуру знаходження операнда для виконуваної команди. Якщо команда використовує два операнда, то для кожного з них повинен бути заданий спосіб адресації, причому режими адресації першого і другого операнда можуть як співпадати, так і відрізнятися.

Операнди команди можуть розташовані в різних місцях:

- безпосередньо в складі коду команди;
- в будь-якому регістрі;
- в комірці пам'яті.

В останньому випадку існує кілька можливостей зазначення адреси. Операнда. Строго кажучи, ***засоби адресації є елементом архітектури процесора***, відображаючи закладені в ньому можливості пошуку операндів [17, 18]. З іншого боку, різні засоби адресації певним чином позначаються в мові асемблера і в цьому сенсі є розділом мови.

Слід зазначити неоднозначність терміну «операнд» стосовно до програм, написаним на мові асемблера. Для машинної команди операндами є ті дані (по суті, двійкові числа), з якими вона має справу. Ці дані можуть, як уже зазначалося, перебувати в регістрах або в пам'яті. Якщо ж розглядати команду мови асемблера, то для неї операндами (або, краще сказати, параметрами) є ті позначення, які дозволяють спочатку транслятору, а потім процесору визначити місцезнаходження операндів машинної команди. Так, для команди асемблера

mov mem, AX

в якості операндів використовується позначення елементу пам'яті *mem*, а також позначення регістра *AX*. В той же час, для відповідної машинної команди операндами є вміст комірки пам'яті і вміст регістра. Було б правильніше говорити про операнди машинних команд і про параметри, або аргументи, команд мови асемблера.

В архітектурі сучасних 32-розрядних процесорів Intel передбачені досить витончені засоби адресації; в МП 86 засобів адресації менше. Ми розглядаємо режими адресації, які використовуються в МП 86.

В літературі по асемблера можна зустріти різні підходи щодо опису засобів адресації: не тільки назви цих режимів, але навіть і їх кількість можуть відрізнятися. Зрозуміло, засобів адресації існує в точності стільки, скільки їх реалізовано в процесорі; однак, режими адресації можна об'єднувати в групи за різними ознаками, від чого і створюється деяка плутанина, в тому числі і в кількості наявних режимів. Ми будемо дотримуватися поширеною, але не єдино можливою термінології.

1. Регістрова адресація.

Операнди можуть мати ім’я 8-, 16-, 32- регістрів та розташовуватися в будь-яких регістрах загального призначення і сегментних регістрах. В цьому випадку в тексті програми вказується назва відповідного регістра. Робота з регістрами не потребує звернення до пам’яті, тому ця інструкція найшвидше.

Операнд (байт / слово / подвійне слово) знаходиться в регістрі, назва якого вказується в команді. Цей засіб адресації застосовується до усіх програмно-адресованих регістрів процесора.

Наприклад:

mov ax, bx ;команда, що копіює в регістр AX вміст регістра BX
mov dx, word ; регістр у 1-му операнді
mov word, cx ; регістр у 2-му операнді
mov ds, ax ; вміст регістра ax (адреса) копіюється в сегментний регістр даних
sub eax, ebx ; значення регістра eax віднімається від значення регістра ebx, результат записується в eax

2. Безпосередня адресація.

Деякі команди (всі арифметичні команди, окрім ділення) дозволяють вказувати **один з операндів** константу або вираз **безпосередньо в команді**, наприклад, команда

mov ax, 2 *;* поміщає в регістр AX число 2 (0002),
mov ax, 0245H ; слово копіюється в AX.

Довжина безпосередньої константи не може перевищувати довжини, визначеної першим операндом, наприклад:

mov al, 0245H ; команда невірна, **al** має довжину 1 байт, а **0245H** – два байти.
mov DL, ‘!’ ; ASCII - код символу «!» заноситься в DL.
mov CX, limit; число, позначене *limit*, завантажується в CX.

Ця адресація застосовується, коли операнд довжиною у байт або слово знаходиться в асемблер ній програмі.

Команда *mov*, використана в останньому реченні, має два операнда: перший операнд визначається за допомогою реєстрової адресації, другий - за допомогою безпосередньої.

Важливим застосуванням безпосередньої адресації є пересилання відносних адрес (зсувів). Щоб вказати, що мова йде про відносну адресу даної комірки, а не про її вміст, використовується описувач *offset* (зміщення або зсув від початку сегменту до змінної в сегменті даних):

; Сегмент даних
mes db 'Привіт студентам!'; рядок символів

; Сегмент команд
mov DX, offset mes; адреса рядка записується в DX

У наведеному прикладі відносна адреса рядка *mes*, тобто відстань в байтах першого байта цього рядка від початку сегмента, в якому вона знаходиться, заноситься в регістр DX.

3. Пряма адресація.

В літературі ця адресація ще називається абсолютною.

Операнд знаходиться в пам'яті за адресою, який задається в команді.

У цьому форматі один з операндів посилається на область в пам'яті, а другий – на регістр. DS – це сегментний регістр, який за замовчуванням використовується для адресації даних в пам'яті у вигляді **DS:зсув**. *Пряма адресація іноді називається адресацією по зсуву.*

Наприклад, ADD BYTE_VAL, DL – додати значення з регістру до значення в пам'яті (байт)

MOV EBX, A ; сегментний регістр за замовчуванням DS

Якщо операнд - слово, що знаходиться в сегменті, на який вказує ES, із зсувом від початку сегменту 0001, то команда

mov ax, es:0001 помістить це слово в регістр AX.

В реальних програмах зазвичай для задання статичних змінних використовують директиви визначення даних, які дозволяють посилатися на статичні змінні не за адресою, а за іменем. Тоді, якщо в сегменті, вказаному в ES, була описана змінна word_var розміром в слово, можна записати розміром в слово, можна записати ту ж команду як

mov ax, es:word_var асемблер сам замінить слово «word_var» на відповідну адресу

Адресація відрізняється для реального і захищеного режимів. В реальному режимі зсув завжди 16-бітний, це означає, що ні безпосередньо вказаний зсув, ні результат складання вмісту різних регістрів в складніших методах адресації не можуть перевищувати границь слова. При програмуванні для Windows і в інших ситуаціях, коли програма запускатиметься в захищеному режимі, зсув не може перевищувати границь подвійного слова.

4. Непряма адресація.

Можна адресувати пам'ять у формі *сегмент:зсув*. Адресу операнда в пам'яті можна не вказувати безпосередньо, а зберігати в будь-якому регістрі. Для цього до процесорів 80386 використовуються основні (BX та BP) та індексні регістри (DI та SI), які зазначаються в квадратних дужках, що і визначає звернення до пам'яті. Непряма адресація, наприклад, [DI] вказує Асемблеру, що адреса пам'яті при виконанні програми зберігається в регістрі DI, а регістр BX використовується в поєднанні з регістрами DI і SI та пов'язані з DS як DS:BX, DS:DI, DS:SI та DS:SI – для обробки даних в сегменті даних. BP пов'язаний з SS як SS: BP для обробки даних у стеку.

Якщо перший операнд містить напряму адресу, другий – посилається на регістр або на безпосереднє значення. Якщо другий операнд містить непряму

адресу, то перший посилається на регістр. Квадратні дужки [BP], [BX], [DI], [SI]означають непрямий операнд, а процесор сприймає вміст регістра як адресу зсуву при виконанні програми. Наприклад:

DATA_VAL DB 50 ; Визначаємо байт

……

LEA BX, DATA_VAL ; Завантажуємо зсув в BX, адреса зсуву DATA_VAL

MOV [BX], CL ; Значення CL зберігаємо в комірці пам'яті, на яку вказує BX, тобто у в DATA_VAL

ADD EAX, [EBX] ; Адреса другого операнда знаходиться в регістрі ebx.

Для процесорів 386, 486, 586 адресу операнда дозволили прочитувати також і з EAX, EBX, ECX, EDX, ESI, EDI, EBP і ESP (але не з AX, CX, DX або SP безпосередньо; треба використовувати EAX, ECX, EDX, ESP відповідно або заздалегідь скопіювати зсув в BX, SI, DI або BP). Наприклад, *наступна команда поміщає в регістр AX слово з комірки пам'яті, селектор сегменту якої знаходиться в DS, а зсув в BX:*

mov ax, [bx]

Як і у разі прямої адресації, *DS використовується за замовчуванням*, але не у всіх випадках: якщо зсув беруть з регістрів ESP, EBP або BP, то як сегментний регістр використовується SS. В реальному режимі можна вільно користуватися всіма 32-бітними регістрами, треба тільки стежити, щоб їх вміст не перевищував границь 16-бітного слова.

Таким чином, абсолютна адреса формується виходячи з сегментної адреси в одному з сегментних регістрів та зсуву в регістрах BX, BP, SI або DI:

MOV AL, [BX] ; База – в DS, зсув – в BX

MOV AX, [BP] ; База – в SS, зсув – в BP

MOV AX, ES:[SI] ; База – в ES, зсув – в SI

5. Адресація по базі із зсувом (базова адресація).

Адреса операнда дорівнює сумі вмісту регістра і зміщення. Наприклад.

Комбінуються два попередніх методи адресації: наступна команда

mov ax, [bx+2]

поміщає в регістр AX слово, що знаходиться в сегменті, вказаному в DS, із зсувом на 2 більшим, ніж число, що знаходиться в BX. Оскільки слово займає рівно два байти, ця команда помістила в AX слово, безпосередньо наступне за тим, яке є в попередньому прикладі. Така форма адресації використовується в тих випадках, коли в регістрі знаходиться адреса початку структури даних, а доступ треба здійснити до будь-якого елементу цієї структури. Інше важливе застосування адресації по базі із зсувом - це доступ з підпрограми до параметрів, переданих в стеку, використовуючи регістр BP (EBP) в якості бази і номер параметра в якості зсуву. Інші допустимі форми запису цього способу адресації:

mov ax, [bp]+2

mov ax, 2[bp]

Виконавча адреса є сумою значень зсуву та вмісту регістра BP або BX, наприклад:
MOV AX, [BP+6]	; База – SS, зсув – BP+6
MOV DX, 8[BX]	; База – DS, зсув – BX+8

До процесора 80386 в якості базового регістра можна було використовувати тільки BX, BP, SI або DI і зсув міг бути тільки байтом або словом (із знаком). Починаючи з 80386 і старше, процесори Intel дозволяють додатково використовувати EAX, EBX, ECX, EDX, EBP, ESP, ESI і EDI, так само як і для звичайної непрямої адресації. За допомогою цього методу можна організовувати доступ до одновимірних масивів байт: зсув відповідає адресі початку масиву, а число в регістрі - індексу елемента масива, який треба зчитати. Якщо масив складається не з байтів, а із слів, треба помножати базовий регістр на два, а якщо з подвійних слів - на чотири. Для цього передбачений наступний спеціальний метод адресації.

6. Непряма адресація з масштабуванням.

Цей метод адресації повністю ідентичний попередньому, за винятком того, що за його допомогою можна прочитати елемент масиву слів, подвійних слів або учетверених слів, просто помістивши номер елемента в регістр:

```
mov     ax, [esi*2]+2
```

Множник, який може бути дорівнюватися 1, 2, 4 або 8, відповідає розміру елемента масиву – байту, слову, подвійному слову, зчетвереному слову відповідно. З регістрів в цьому варіанті адресації можна використовувати тільки EAX, EBX, ECX, EDX, ESI, EDI, EBP, ESP, але не SI, DI, BP або SP, які можна було використовувати в попередніх варіантах.

7. Адресація по базі з індексуванням.

У цьому методі адресації зсув операнда в пам'яті обчислюється як сума чисел, що містяться в двох регістрах, і зсуву, якщо він вказаний. Всі наступні команди – це різні форми запису однієї і тієї ж дії:

```
mov     ax, [bx+si+2]
mov     ax, [bx][si]+2
mov     ax, [bx+2][si]
mov     ax, [bx][si+2]
mov     ax, 2[bx][si]
```

В регістр AX розміщується слово з комірки пам'яті із зсувом, що дорівнює сумі чисел, які містяться в BX і SI, і числа 2. З шістнадцятибітних регістрів так можна складати тільки BX + SI, BX + DI, BP + SI і BP + DI, а з 32-бітових – всі вісім регістрів загального призначення. Так само як і для прямої адресації, замість безпосереднього зазначення числа можна використовувати ім'я змінної, яка задається однією з директив визначення даних. Зазвичай для двовимірних масивів BX вказує на стовпець, SI – на рядок. Так можна прочитати, наприклад, число з двовимірного масиву: якщо задана таблиця 10х10 байта, 2 – зсув її початку від початку сегменту даних (на практиці використовуватиметься ім'я цієї таблиці), BX=20, а SI=7, приведені команди *прочитають слово, що складається з сьомого і*

восьмого байт третього рядка. Якщо таблиця складається не з одиночних байтів, а із слів або подвійних слів, зручніше використовувати наступну, найбільш повнішу форму адресації.

Базово-індексна адресація передбачає використання для обчислення виконавчої адреси суми вмісту базового регістра та індексного, а також зсуву, що знаходиться в операторі, наприклад:

```
MOV BX, [BP][SI]          ; База – SS, зсув – BP+SI
MOV ES:[BX+DI], AX   ; База – ES, зсув – BX+DI
MOV AX, [BP+6+DI]     ; База – SS, зсув - BP+6+DI
```

8. Адресація по базі з індексуванням і масштабуванням.

Це найповніша схема адресації, в яку входять всі випадки, розглянуті раніше, як окремі. Повну адресу операнда можна записати як вираз, представлений на схемі, тобто для обчислення адреси використовується база, індекс, масштаб, зсув:

Сегмент	База		Індекс		Масштаб		Зсув
CS:	**EAX**		**EAX**		**1**		
SS:	**EBX**		**EBX**		**2**		
DS:	**ECX**		**ECX**		**4**		
ES:	**EDX**	+	**EDX**	*	**8**	+	**зсув**
FS:	**EBP**		**EBP**				
GS:	**ESP**		**ESP**				
	EDI		**EDI**				
	ESI						

Повна форма адресації

Зсув може бути байтом або подвійним словом. Якщо ESP або EBP використовуються як базовий регістр, селектор сегменту операнда береться за замовчуванням з регістра SS, в усіх інших випадках з DS.

Контрольні питання:

1. Що таке основа системи счислення?
2. Як представити двійкове число у восьмирічній системі?
3. Як здійснюється переведення числа з шістнадцятирічної системи у десяткову?
4. Як здійснюється переведення числа з двійкової системи у десяткову?
5. Яке правило переведення цілої частини числа з однієї системи счислення в іншу?
6. Яке правило переведення дробової частини числа з однієї системи счислення в іншу?
7. Які формати представлення базових даних вам відомі?
8. Що таке додатковий код?
9. Як зберігаються від'ємні числа у комп'ютері?
10. Які логічні операції застосовуються в асемблері?
11. Що таке засоби адресації?
12. Що таке регістрова адресація?
13. Що таке беспосередня адресація?
14. Що таке пряма адресація?
15. Що таке непряма адресація?
16. Що таке адресація по базі із зсувом?
17. Що таке непряма адресація з масштабуванням?
18. Що таке адресація по базі з індексуванням?
19. Що таке адресація по базі з індексуванням і масштабуванням?

Тема 7. Основні команди мови Асемблер
Лекція 9. Команди пересилання та обміну даними

Типи команд. Команда переміщення даних (MOV-переслати). Команда обміну даними (XCHG). Команда завантаження виконавчої адреси (LEA). Команда завантаження вказівна стеку з використанням регістрів DS або ES (LDS і LES). Побайтове пересилання зі стеком. (LASH і SAHF - завантажити /зчитати регістр AH в/з стека. PUSH і POP - записати /зчитати регістр в/з стека. PUSHF і POPF - переслати регістр прапорів RF

До найбільш поширених цілочисельних команд процесора Pentium IV відносяться [17, 18]:

- *команди визначення даних*;
- *команди переміщення даних* (MOV – переміщення в операнд, PUSH – переміщення в стек, POP – виштовхування слова зі стеку, XCHG – зміна містами операндів, LEA – завантаження дійсної адреси, CMOV – умовне переміщення);
- *арифметичні команди* (ADD – додавання, SUB – віднімання, MUL – множення без урахування знаку, IMUL – множення зі знаком, DIL – ділення без урахування знаку, IDV – ділення зі знаком, ADC – додавання з добавленням біта перенесення, SBB – віднімання з перенесенням в інший операнд, INC – інкремент (додавання 1 до операнду), DEC – декремент (віднімання 1 з операнду), DST – від'ємність (отрицание));
- *двійково-десяткові команди* (DAA – десяткова корекція, команди для корекції ASCII-коду: AAA - додавання, AAS - віднімання, AAM – множення, AAD – ділення);
- *логічні команди* (AND – логічне I, OR – логічне АБО, XOR – що виключає АБО, NOT – заміщення додаванням до 1);
- *команди звичайного та циклічного зсуву* (SAL/SAR – зсув вліво/вправо на ??? біт, SHL/SHR – логічний зсув вліво/вправо на ??? біт, ROL/ROR – циклічний зсув вліво/вправо на ??? біт, ROL/ROR – циклічний зсув по перенесенню на ??? біт);
- *команди тестування та порівняння* (TST – операнди логічної операції I, CMP – встановлення прапорців);
- *команди передачі управління* (JMP – перехід до адреси, Jxx – умовні переходи на основі прапорців, CALL – виклик процедури за адресою, RET – вихід з процедури, LOOPxx – продовження циклу до виконання певної умови, INN – програмне переривання, INTO – переривання, якщо встановлений біт переповнення);
- *команди обробки рядків* (LODS – завантаження рядка, SMOS – зберігання рядка, MOVS – переміщення рядка, CMPS – порівняння двох рядків, SCANS – сканування рядка);
- *команди для роботи з кодами умов* (SNC – встановлення біта в регістрі EFLAGS, CLC – скидання біта перенесення в регістрі EFLAGS, CMC –

доповнення біта перенесення в регістрі EFLAGS, STD – встановлення біта направлення в регістрі EFLAGS, CLD – скидання біта направлення в регістрі EFLAGS, STI – встановлення біта переривання в регістрі EFLAGS, CLI - скидання біта переривання в регістрі EFLAGS, PUSHFD – розміщення значення з регістра EFLAGS в стек, POPFD – виштовхування значення зі стеку в регістр EFLAGS, LAHF – завантаження AH з регістра EFLAGS, SAHF – зберігання AH в регістрі EFLAGS);

- інші команди (SWAP - зміна порядку проходження байтів, NOP – пуста операція, HLT – зупинення, IN- перенесення байта з порту в АЛ пристрій, OUT - перенесення байта з АЛ Пристрій в порт, WAIT – очікування переривання, інші).

Команди бувають різних типів, операндами у команді можуть бути:

- регістр ← регістр, регістр (джерелом операндів є регістри, результат зберігається також в регістрі);
- регістр ← регістр, пам’ять;
- регістр ← пам’ять, пам’ять;
- пам’ять ← пам’ять, пам’ять;
- пам’ять ← регістр, пам’ять;
- регістр ← пам’ять,
- пам’ять ← регістр.

9.1. Команди переміщення даних

9.1.1. MOVe operand Пересилання операнда

MOVe operand to/from system registers Пересилання операнда у системні регістри (або з них)

Схема команди: **MOV приймальник, джерело**

Призначення: пересилання даних між регістрами або регістрами та пам’яттю. Команда має обмеження:

- копіювання здійснюється з другого операнда у перший;
- значення другого операнда не змінюється;
- обидва оператора не можуть бути з пам’яті;
- лише один з операндів може бути сегментним, приймальником не може бути регістр CS; не можна пересилати сегментні регістри:

```
MOV  ES, DS
```

Треба розписати:

```
MOV  AX, DS
MOV  ES, AX
```

- не можна напряму ініціалізувати сегмент даних

```
DSEG SEGMENT

        MOV DS, DSEG
```

Треба розписати:

```
        MOV AX, DSEG
```

MOV DS, AX

- довжина обох операндів повинна бути однаковою.

Приклади застосування команди MOV з різним операндами

Переміщення між регістрами

MOV DX, ECX ; регістр – регістр

MOV ES, AX ; регістр – сегментний регістр

MOV BYTEFLD, DH ; регістр – пам'ять (байт)

MOV [DI], BX ; регістр – пам'ять (непряма адресація)

Переміщення безпосередніх даних

MOV CX, 40H ; безпосереднє значення – регістр

MOV BYTEFLD, 25 ; безпосереднє значення – пам'ять, напряму

MOV WORDFLD [BX] , 16H ; безпосереднє значення – пам'ять, непряма адресація

Переміщення даних з пам'яті:

MOV CH, BYTEFLD ; пам'ять – регістр, напряму

MOV CX, WORDFLD [BX] ; пам'ять – регістр, непряма адресація

Переміщення в сегментних регістрах:

MOV AX, DS ; сегментний регістр – регістр

MOV WORDFLD, DS ; сегментний регістр – пам'ять

Команда Mov має розширену можливість: для випадку, коли довжина операндів різна використовується директива значення типу:

Тип **PTR** *вираз*

Ця директива дозволяє отримати доступ до частини змінної.

Оператор *PTR* може використовуватися з елементами даних, мітками інструкцій. Він використовує специфікатори типів *Byte, Word, Dword, Tbyte* для явного зазначення типу *DB, DW, DD, DQ, DT* для змінних. Він також використовує специфікатори типів *NEAR*, *FAR, PROC* для явного зазначення відстані до мітки переходу. Таким чином, *тип* – це специфікатор типу, наприклад, BYTE, а *вираз* – це змінна або константа.

Наприклад,

```
BYTEA        DB    22H
             DB    35H
WORDA   DW   2672H       ; у пам'яті зберігається у вигляді 7226H
```

MOV BYTE PTR WORDA, 05; Розмістити 05 у перший байт WORDA - 2605

MOV AX, WORD PTR BYTEA; Розмістити в AX два байти (2235H) з BYTEA

При запису безпосереднього операнда в пам'ять треба вказувати його розмір директивою *BYTE PTR(mov [di], '$');*

```
m1      dword    12345678h
```

```
mov     ax,     word PTR m1     ; a=5678h
```

9.1.1.1. Додаткові команди передач

Починаючи з процесорів 386, 486 є додаткові команди, які дозволяють переслати операнд-джерело меншої розмірності в операнд-приймальник більшої розмірності (8→16/32, 16→32).

MOVZX (MOVe and Zero eXtension) *Пересилання з нульовим розширенням*

*Схема команди***: MOVZX приймальник, джерело**

Команда копіює вміст операнда джерела у більший за розміром регістр приймальника, тобто старші біти (розширення розмірності) доповнюються нулями. Ця команда використовується тільки при роботі з беззнаковими числами. Існує три варіанти цієї команди:

```
MOVZX     r16, r/m8        r – регістр, m – пам’ять
MOVZX     r32, r/m8
MOVZX     r32, r/m16
```

Наприклад,

```
    mov      bx, 0A69Bh
    movzx    eax, bx       ; eax=0000A69Bh
    movzx    edx, bl       ; edx=0000009Bh
    movzx    cx, bl  ; cx=009Bh
```

або з пам’яті:

```
.data
bt      db      9bh
wd      dw0a69Bh
.code
        movzx   eax, wd     ;   eax=0000A69Bh
        movzx   cx, bt      ;   cx=009Bh
        movzx   edx, bt     ;   edx=0000009Bh
```

MOVSX (MOVe and Sign eXtension) *Пересилання зі знаковим розширенням*

*Схема команди***: MOVSX приймальник, джерело**

Команда перетворює елемент зі знаком меншої розмірності у еквівалентний зі знаком більшої розмірності, тобто команда використовується при роботі зі знаковими числами. Існує три варіанти цієї команди:

```
MOVSX     r16, r/m8
MOVSX     r32, r/m8
MOVSX     r32, r/m16
```

Наприклад,

```
mov   bx, 0A69Bh
movsx  eax,  bx     ;     eax=FFFFA69Bh
movsx  edx, bl      ;     edx=FFFFFF9Bh
movsx   cx,  bl     ;     cx=FF9Bh
```

9.1.2. Команда обміну даними XCHG (eXCHanGe)

***Схема команди*: XCHG операнд_1,операнд_2**

Призначення: обмінює містами значення операнда_1 та операнда_2.

Формат: **XCHG** регістр/пам’ять, регістр/пам’ять

Обмін здійснюється між регістрами або між регістрами та пам’яттю.

mov bx, 0A69 h   bl=0A, bh=69h

Приклади: XCHG BL, BH ; обмін між регістрами

Результат команди   bx=690A h

```
WORDQ      DW ?
……………………………
XCHG CX, WORDQ ; обмін між регістром і пам’яттю (слово)
S DB -100
…………….
XCHG s, DL ; обмін між пам’яттю і регістром
```

9.1.3. Команда завантаження дійсної адреси (зсуву) LEA (*Load Effective Address*), **ініціалізація регістрів адресами зсуву**

***Схема команди:* LEA приймальник, джерело**

Алгоритм роботи команди залежить від діючого режиму адресації (use16 або use32): якщо use16, то в регістр приймальник завантажується 16-бітний зсув операнда джерела; якщо use32, то в регістр приймальник завантажується 32-бітний зсув операнда джерела.

В якості приймальника виступають індексні регістри SI або DI, в які завантажується адреса або адресний вираз (джерело). Для того, щоб задати адресу, можуть бути використані будь-які засоби адресації. Стан прапорців не змінюється

Ця команда є альтернативою оператору асемблера *offset*. На відміну від *offset* команда *lea* більш гнучко організовує адресацію операндів.

Приклад 1:

```
LEA SI, UR;   (аналог MOV SI, OFFSET UR)
LEA DI, [BX]+[SI]+20;        (результат обчислення завантажується в DI)
```

Приклад 2: завантажити в регістр BX адресу п’ятого елемента масиву *mas*

```
      .data
  MAS   DB   10 DUP (0)
      .CODE
…………….
      MOV   DI, 4
      LEA   BX,  MAS[di]
;  або
      LEA  BX,  MAS[4]
```

Приклад 3:

```
DATATBL    DB 25 DUP (?)   ; таблиця з 25 байтів
BYTEFLD       DB ?             ; один байт
………………………………………
        LEA BX, DATATBL   ; завантажити адресу зсуву таблиці
```

```
MOV BYTEFLD, [BX] ;    перемістити перший байт з  DATATBL
```
Аналогом цієї команди є
```
MOV  BX, OFFSET DATATBL    ;   завантажити адресу зсуву
```

Приклад 4:
```
        .data
  mat dw   1, 5, 7, 12, 1, 3
        .code
…………….
        mov si, 0
        mov bx, offset mat
        mov ax, bx [si]              ;1-й елемент
        add ax, bx [si+2]            ;1-го і 2-го елементів
```

9.1.3.1. Команди завантаження пар регістрів вмістом області пам'яті – завантаження сегментного регістра (DS, ES) вказівником з пам'яті

LDS/LES (Load pointer into ds/es segment register)

LDS приймальник, джерело

LES приймальник, джерело

Призначення – отримати повний вказівник у вигляді сегментної складової та зсуву.

Ці команди дозволяють одночасно завантажити сегментний регістр (DS або ES) та один з регістрів пам'яті. **Джерело** адресує початок області пам'яті, з якої виконується завантаження. Перші два байти з цієї області завантажуються в **приймальник**, наступні два байти завантажуються в сегментний регістр (**DS –** для **LDS** та **ES** для **LES).**

Наприклад:
```
ADR1 DB     1, 2, 3, 4, 5
            LDS BX, DWORD PTR ADR1
```
/* 0102030405   DWORD, перші два байти в BX, другі 2 байти в DS */

Після виконання команди (BX)=0201H, (DS)=0403H.
```
lds bx, [bp+4]   /* ds→bp, bx→4 */
les di, [bp+10]     /* es→bp, di→10 */
```

Для 80386 і більше є команди завантаження сегментних регістрів **LSS, LGS, LFS.**

Наприклад: **LSS EDX, FWORD PTR ADR1**

LFS DI, [BP+DI]

Тобто, для 32-бітної архітектури (use32) завантажити перші чотири байти з *комірки пам'яті джерело* в 32-розрядний регістр, вказаний *операндом приймач*. Наступні два байта в області джерело повинні містити сегментну складову, або селектор, деякої адреси; вони завантажуються в регістр ds/es/fs/gs/ss.

Таким чином, за допомогою даних команд в парі регістрів *ds/es/fs/gs/ss* і приймач виявляється повна адреса деякої комірки пам'яті. Цю обставину можна

використовувати, наприклад, при роботі з ланцюжковими командами, де існують жорсткі угоди на розміщення адрес оброблюваних рядків

9.1.4. Команди роботи зі стеком працюють у парі: ***PUSH*** (***PUSH* onto stack** – переміщення операнда в стек (зберігання інформації у стеку), **POP** – виштовхування слова зі стеку (відновлення інформації зі стеку).

Стек – це ділянка пам'яті, організована для зберігання та відновлення даних за принципом «перший зайшов, останній вийшов» (LIFO – last in, first out). Регістр SP (Stack Pointer) використовується для роботи зі стеком, він завжди вказує на вершину стеку, тобто на зсув останнього елемента у стеку або у сполученні <SS:SP> - на адресу цього елемента. Усі операції у стеку можна проводити тільки з одним елементом, який знаходиться на верхівці стеку та був введений у стек останнім. Стек широко застосовується для тимчасового зберігання даних та адрес, для передачі параметрів у процедури.

Команда PUSH розміщує в стек слово або подвійне слово для подальшого використання. Регістр SP (*Stack Pointer*) – вказівник стеку вказує адресу слова (або подвійного слова), що знаходиться в даний момент на вершині стеку.

Синтаксис: PUSH джерело

Символьний код: *PUSH регістр/пам'ять*

PUSH безпосереднє значення (80286)

Стек заповнюються навпаки – в бік менших адрес, тому команда PUSH спочатку зменшує значення вказівника стеку на 4 для подвійного слова <ESP>=<ESP>-4 або на 2 - для слова <SP>=<SP>-2 в залежності від значення атрибута розміру адреси - use16 або use32 та переміщує слово або подвійне слово з операнда у вершину стека. Операндом може бути регістр загального призначення, сегментний регістр або елемент пам'яті.

На прапорці ця команда не впливає.

Команда **POP (Pop Word / Doubleword Off Stack)** відновлює слово або подвійне слово зі стеку, яке попередньо було занесено у стек. Регістр SP вказує на положення поточного слова на вершині стеку. Команда POP після відновлення збільшує значення SP на 2 або ESP на 4.

Синтаксис: POP приймальник

Символьний код: POP регістр/пам'ять

Наприклад:

```
my_proc proc    near
        push    ax
        push    bx
;тіло процедури, в якій змінюється вміст регістрів ax та вх
...
        pop     bx
        pop     ax
        ret
endp
```

Можна і так push bx

pop ax /*тобто значення *вх* записується в регістр *ах* */

Не можна використовувати команду MOV ES, DS, треба скористатися стеком:

```
PUSH DS
POP   ES
```

І навпаки MOV DS, ES

```
PUSH ES
POP   DS
```

Команди для зберігання та витягування зі стеку регістру прапорців

PUSHF ;розташування у вершині стеку (ss:sp) вмісту регістра прапорців *flags*.

POPF ; витягування зі стеку слова та відновлення його у регістр прапорців *flags*

Додаткові команди

PUSHA (Push All)) ; Занести у стек усі 16 бітні регістри загального призначення *ax, cx, dx, bx, sp, bp, si, di.*

POPA (PUSH All **) ;** Відновити зі стеку *di, si, bp, sp, bx, dx, cx, ax.*

PUSHAD Занести у стек усі 32 бітні регістри *eax, ecx, edx, ebx, esp, ebp, esi, edi.*

POPA ; Відновити зі стеку *edi, esi, ebp, esp, ebx, edx, ecx, eax.*

Контрольні питання:

1. Команда переміщення даних – призначення та синтаксис.
2. Які обмеження має команда переміщення даних?
3. Призначення директиви PTR.
4. Які вам відомі додаткові команди передач?
5. Команда обміну даних – призначення та синтаксис.
6. Команда завантаження дійсної адреси – призначення та синтаксис.
7. Команди завантаження пар регістрів вмістом пам’яті – надайте характеристики.
8. Команди роботи зі стеком – надайте характеристики.

Лекція 10. Арифметичні операції

Команди складання: команда ADD, команда з використанням переносу при складанні 32-розрядних чисел ADC, збільшення значення операнда на 1 - INC. Команди віднімання SUB, SBB (віднімання з займанням прапорця переносу), DEC (зменшення значення операнда на 1). Команда множення двох цілих двійкових чисел без урахування знаку MUL. Команда множення двох цілих двійкових чисел з урахуванням знаку IMUL, особливості роботи. Команда беззнакового ділення DIV. Команда знакового цілочисельного ділення IDIV, особливості роботи.

Арифметичні команди

10.1. Команда складання ADD (*ADDition*)

***Синтаксис*: ADD приймальник, джерело**

Символьний код: *ADD регістр/пам'ять, регістр/пам'ять/безпосереднє значення* [17,18].

Призначення: складання двох операндів джерела та приймальника розмірністю *байт, слово або подвійне слово*, записати результат складання за адресою першого операнда, встановити прапорці.

Ця операція є коректною при використанні операндів: регістр-регістр, регістр-пам'ять, пам'ять-пам'ять, пам'ять-безпосереднє значення.

Логіка роботи команди: ***<приймальник>=<приймальник>+<джерело>***

Операнди повинні мати однаковий розмір. Результат записується на місце приймальника.

Виконання команди додавання впливає на стан прапорців:

CF (*Carry*)*– перенесення* (зі старшого знакового розряду; наприклад, для байта в 9-му розряді при виконанні команди додавання з'явилася 1),

PF (*Parity*)*– парність*,

AF (*Auxiliary*)*- допоміжне перенесення*,

ZF (*Zero*)*– нуль*,

SF (*Sign*)*– мінус*,

OF (*Overflow*)*– переповнювання.*

Команда **ADD** використовується для складання *двох цілочисельних операндів*. Якщо результат складання виходить за межі першого операнда (виникає переповнювання), то врахувати цю ситуацію слід шляхом аналізу прапорця CF і подальшого можливого застосування команди ADC. Наприклад, складемо значення в регістрі AX і області пам'яті CHH. При складанні слід врахувати можливість переповнювання.

```
ADD AX, CHH      ;  додати значення з області пам'яті CHH до регістру AX
ADD AX, 5        ; AX=AX+5
ADD DX, CX       ; DX=DX+CX
```

ADD DX, CL ; Помилка – різний розмір операндів.

ADD EBX, DBLWORD ; Додати подвійне слово з пам'яті до регістру.

ADD BL, 10 ; Додати 10 до молодшої частини регістра BX.

10.2. Команда складання двох операндів з урахуванням перенесенням з молодшого розряду ADC (*ADDition with Carry*)

***Синтаксис*: ADC приймальник, джерело.**

Символьний код: *ADC регістр/пам'ять, регістр/пам'ять/безпосереднє значення*

Впливає на прапорці AF, CF, OF, PF, SF, ZF

Результат заноситься у перший операнд, в залежності від результату встановлюються прапорці.

Логіка роботи команди:

<приймальник>=<приймальник>+<джерело>+<CF>

Операндами можуть бути байти, слова, числа зі знаком або без знаку.

Команда ADC використовується *при складанні багаторозрядних двійкових чисел*. Її можна використовувати як самостійно, так і спільно з командою ADD.

При сумісному використанні команди ADC з командою ADD складання *молодших байтів/слів/подвійних слів здійснюється командою ADD, а вже старші байти/слова/подвійні слова складаються командою ADC, що враховує перенесення з молодших розрядів в старші.* Тобто команда ADC додає вміст прапорця перенесення CF (0 або 1) до першого операнда - приймальника, а потім додає до приймальника другий операнд – джерело. Таким чином, команда ADC значно розширює діапазон значень чисел, що складаються.

Наприклад, при виконанні байтової операції 250 + 10

```
   1111 1010
+  0000 1010
 1 0000 0100
 ↑
 9-тий біт попадає у прапорець перенесення CF.
```

Приклад 1.

За допомогою *ADC* можна складати числа, розрядність яких більше розрядності регістру 64-бітне і 64-бітне (як 32:32 + 32:32). Приклад: 16-бітне додавання (8: 8):

```
1234=00000100 11010010 (al:bl)
367=00000001 01101111 (cl:dl)
mov al, 00000100b      ; старший байт 1234
mov bl, 11010010b      ; молодший байт 1234
mov cl, 00000001b      ; старший байт 367
mov dl, 01101111b      ; молодший байт 367
add bl, dl             ; 1(cf) 01000001(bl) - 9й біт записується  в cf
                        11010010b
                        01101111b
                      1 01000001b       молодші байти
```

```
adc al, cl ; 0(cf) 00000101(al) + 1 біт переносу = 00000110
                              00000100b
                              00000001b
                              00000101
                            +        1
                              00000110        старші байти
; 00000110 01000001 = 1601                    результат
```

Приклад 2.

Виконаємо додавання значення в регістрі *ax* і комірці пам'яті *chislo*. При додаванні варто врахувати можливість переповнення.

```
chislo  dw     2015
rez     dd     0
...
   add  ax, chislo      ;        (ax)=(ax)+chislo
   mov  word ptr rez, ax;        слово з AX перенести в слово rez
   jnc  dop_sum  ;          перейти на dop_sum, якщо немає переносу,
(результат не вийшов за розрядну сітку)
   adc  word ptr rez+2, 0;       розширити результат, для врахування
   ;                             перенесення в старший розряд
dop_sum:
...
```

Приклад 3.

```
  mov      ax, CFA3h
  sub      dx, dx
  add      ax, 5426h        ;DX=0000h, AX=23C9h, CF=1
                 CFA3h
                 5426h
               1 23C9h
  adc      dx, 0            ;DX=0001h, AX=23C9h            DX:AX
```

Приклад 4.

```
.data
sl1    dd     01fe544fh
sl2    dd     005044cdh
elderREZ      db     0 ;       ;      для обліку перенесення зі старшого розряду
результату
rez    dd     0
.code
...
  mov    ax, sl1
  add    ax, sl2 ;          складання молодших слів доданків (слагаемых)
```

```
                544fh
                44cdh
                991bh   rez
mov   rez, ax
mov   ax, sl1+2
adc   ax, sl2+2 ;                складання старших слів доданків плюс cf
                01feh
                0050h
                024eh   rez+2
mov   rez+2, ax
adc   rez, 0 ; врахувати можливе перенесення          024e991bh
```

10.3. Команда обміну та складання XADD (eXchange and ADD)

Синтаксис: XADD приймальник, джерело

Символьний код: XADD регістр/пам'ять, регістр

Впливає на прапорці AF, CF, OF, PF, SF, ZF

Команда використовується для виконання операції обміну та складання двох операндів. Вміст першого операнда – приймальника копіюється в другий операнд – джерело, виконується складання двох операндів і сума розміщується в операнд приймальник.

```
mov   al, 08h
mov   bl, 01h
xadd  al, bl  ; а) al=01h, bl=08h б) al=09h, bl=08h
```

10.4. Команда віднімання SUB (SUBtract)

Синтаксис: SUB операнд_1, операнд_2

Символьний код: *SUB регістр/пам'ять, регістр/пам'ять/безпосереднє значення*

Впливає на прапорці AF, CF, OF, PF, SF, ZF

Команда призначена *для віднімання цілочисельних операндів або для віднімання молодших частин значень багатобайтних операндів*.

Операнди повинні мати однаковий розмір.

Віднімання в процесорі реалізовано за допомогою додавання. Процесор змінює знак другого операнда на протилежний, а потім складає два числа.

Тобто віднімання здійснюється за методом складання з двійковим доповненням: для другого операнда встановлюється додатковий код (біти інвертуються +1), а потім відбувається складання з першим операндом. *Операнд_2 віднімається від операнда_1, результат записується в операнд_1.*

Команда SUB діє як ***приймальник = приймальник – джерело***

Виконання команди віднімання впливає на стан прапорців:

CF – перенесення (зі старшого знакового розряду; наприклад, для байта в 9-му розряді при виконанні команди додавання з'явилася 1), PF – парність, AF - допоміжне перенесення, ZF – нуль, SF – мінус, OF – переповнювання.

При відніманні прапорець CF діє як ознака зайняття.

Приклад 1

```
sub bl, 10;  з регістру bl відняти значення 10, результат занести в bl
sub ax, bx       ;ax=ax-bx
sub bx, cl               ; Помилка – різний розмір операндів.
```

Приклад 2

```
mov ax, 100
mov bx, 60
sub ax, вх              ;ax=40 (ах-вх) , вх=60
```

Приклад 3

```
mov dl, '8'
mov dh, '0'
sub dl, dh              ; dl=8 (перетворення ASCII-кода  в цифру)
```

Приклад 4 32-х бітні операнди

```
mov eax, 1000000
mov ebx, 60000
sub eax, ebx            ;eax=40000
```

10.5. Команда віднімання із зайняттям (заемом) SBB (SuBtract with Borrow) або віднімання з перенесенням

Синтаксис: SBB операнд_1, операнд_2

Символьний код: SBB регістр/пам'ять, регістр/пам'ять/безпосереднє значення

Команда призначена для виконання цілочисельного віднімання старших частин значень багатобайтних операндів з урахуванням можливого попереднього зайняття при відніманні молодших частин значень цих операндів, коли виконувалося попереднє віднімання командами SBB та SUB (за станом прапорця перенесення CF).

Команда ***SBB*** спочатку віднімає вміст прапорця CF з операнда_1 та віднімає з операнда_1 операнд_2: ***приймальник = приймальник - джерело - перенесення***.

Впливає на прапорці AF, CF, OF, PF, SF, ZF

```
;виконати віднімання 64-бітних значень: vich_1-vich_2
vich_1  dd     2 dup (0)
vich_2  dd     2 dup (0)
rez     dd     2 dup (0)
...
;ввести значення в поля vich_1 та vich_2 :
```

;молодший байт за молодшою адресою
...

```
mov    eax, vich_1
sub    eax, vich_2        ;відняти молодші половинки чисел
mov    rez, eax           ;  молодша частина результату
mov    eax, vich_1+4
sbb    eax, vich_2+4
                          ;відняти старші половинки чисел
mov    rez+4, eax         ;старша частина результату
```

10.6. Команда множення двох цілих двійкових чисел без урахування знаку MUL (MULtiply)

Команда перемножує два цілих числа без знаку.

Синтаксис: **MUL множник_1**

Символьний код: MUL регістр/пам'ять.

Команда MUL сприймає старший біт в якості біта даних , а не як біт знака.

Алгоритм роботи команди залежить від формату операнда команди і вимагає явної вказівки місцеположення тільки одного співмножника, який може бути розташований в пам'яті або в регістрі (перший співмножник). Місцеположення другого співмножника фіксовано і залежить від розміру першого співмножника:

- якщо операнд, вказаний в команді, — *байт*, то другий співмножник повинен розташовуватися в **AL**;
- якщо операнд, вказаний в команді, — *слово*, то другий співмножник повинен розташовуватися в **AX**;
- якщо операнд, вказаний в команді, — *подвійне слово*, то другий співмножник повинен розташовуватися в **EAX**.

Результат множення поміщається також у фіксоване місце, яке визначається розміром співмножників:

- при перемноженні байтів результат поміщається в AX;
- при перемноженні слів результат поміщається в пару DX:AX (молодші розряди - в AX, старші - в DX) і встановлюються прапорці переповнення и перенесення;
- при перемноженні подвійних слів результат поміщається в пару EDX:EAX (молодші розряди - в EAX, старші - в EDX) і встановлюються прапорці переповнення и перенесення;

Контролювати розмір результату зручно, використовуючи прапорці CF або OF (прапорці AF, PF, SF, ZF –не визначено). Після виконання команди MUL прапорці CF і OF дорівнюються 0, якщо старша половина множення дорівнюється нулю; інакше обидва ці прапорці дорівнюються 1.

Множення і ділення це одні з найповільніших операцій процесорів сімейства 80x86 (особливо 8086 і 8088). Множення і ділення на константи, що часто зустрічаються, швидше відбувається при зсуві розрядів.

Наприклад,

```
mn_1   db    15
mn_2   db    25
...
      mov   al, mn_1
      mul   mn_2
```

```
MUL  BX              ; множення регістру BX на  регістр AX без знаку
MUL MEM_BYTE   ; Якщо MEM_BYTE визначена як DB - множення
                          ;цієї комірки на регістр AL без знаку
```

10.7. Команда множення двох цілих двійкових чисел з урахуванням знаку IMUL (Integer MULtiply)

Команда виконує цілочисельне множення операндів з урахуванням їх знакових розрядів. Команда сприймає старші(перші ліворуч) біти чисел в якості знаків (0 – позитивне число, 1 – негативне число). Для виконання цієї операції необхідно наявність двох співмножників. Розміщення і задання їх місцеположення в команді залежить від форми вживаної команди множення, яка, у свою чергу, визначається моделлю мікропроцесора. Так, для мікропроцесора і8086 можлива тільки одноперандна форма команди, для подальших моделей мікропроцесорів додатково можна використовувати двох- і трьохоперандні форми цієї команди.

Синтаксис: **IMUL множник_1**
IMUL множник_1, множник_2
IMUL результат, множник_1, множник_2

Символьний код: IMUL регістр/пам'ять
IMUL регістр, безпосереднє значення (для 80286 і вище)
IMUL регістр, регістр, безпосереднє значення (для 80286 і вище)
IMUL регістр, регістр/пам'ять (для 80386 та вище)

Алгоритм роботи залежить від форми команди (одноперандна, двохоперандна або трьохоперандна):

1. Як і для команди MUL**,** вважається, що 2-й співмножник розташовується в регістрі AL (для операнда в команді - байта), в AX (для операнда в команді – слова), в EAX (для операнда –подвійне слово). *Результат множення для команди з одним операндом розташовується в AX* (при перемноженні двох байтів), в пару DX:AX (при перемноженні слів), в пару EDX:EAX (при перемноженні подвійних слів).

Наприклад,

```
IMUL DL          ; перемножити  DL на AL зі знаком
IMUL MEM_WORD; перемножити вміст комірки пам'яті на AX зі
                            ;знаком
```

Три інших формати використовують будь-який 16- або 32-розрядний регістр загального призначення, розміри елементів даних повинні бути однаковими.

2. Перший операнд (регістр) містить співмножник (множимое), саме в ньому з'являється результат множення, 2- операнд – множник (множитель), який має безпосереднє значення.

Наприклад, IMUL DX, 456 ; перемножити DX на 456

IMUL BX, 32

IMUL CX, 50

3. Перший операнд (регістр) вказує, де повинен знаходитися результат множення; другий операнд (регістр або адреса пам'яті) містить співмножник (множимое), третій операнд містить безпосереднє значення.

Наприклад, IMUL CX, DX, 56

IMUL EBX, подвійне_слово_у_пам'яті, 10

4. Перший операнд (регістр) містить співмножник (множимое), саме у ньому з'являється результат; другий операнд (регістр або адреса пам'яті) - множник (множитель). Наприклад,

IMUL DX, слово_у_пам'яті

IMUL EBX, EDX

Команда *imul* встановлює в нуль прапорці OF і CF, якщо розмір результату відповідає регістру призначення. Якщо ці прапори відмінні від нуля, то це означає, що результат дуже великий для відведених йому регістром призначення рамок і необхідно вказати більший за розміром регістр для успішного завершення даної операції множення.

Інші прапорці AF, PF, SF, ZF є невизначеними.

10.8. Команда беззнакового ділення DIV **(DIVide unsigned)**

Команда призначена для ділення двох двійкових беззнакових чисел.

Синтаксис: **DIV дільник (делитель)**

Символьний код: DIV регістр/пам'ять

Алгоритм роботи:

Для команди необхідно задати два операнди – ділене (делимого) і дільника (делителя), беззнакове ділене ділить націло дільник. Ділиме задається неявно і розмір його залежить від розміру дільника, який вказується в команді:

- якщо дільник розміром в *байт*, то *ділиме* повинно бути розташовано в регістрі AX. Після операції *частка* (частное) поміщається в **AL**, а *залишок* — в **AH**;
- якщо дільник розміром в *слово*, то *ділиме* повинно бути розташовано в парі регістрів DX:AX, причому *молодша частина* ділимого знаходиться в AX. Після операції *частка* поміщається в **AX**, а *залишок* — в **DX**;
- якщо дільник розміром в *подвійне слово*, то *ділиме* повинно бути розташовано в парі регістрів EDX:EAX, причому *молодша частина* ділимого знаходиться в EAX. Після операції *частка* поміщається в **EAX**, а *залишок* — в **EDX**.

Наприклад, DIV BH ; байт
DIV CX ; слово
DIV ECX ; подвійне слово

Ділення на нуль викликає переривання.

10.9. Команда знакового цілочисельного ділення IDIV (Integer DIVide signed)

Команда призначена для ділення двох двійкових чисел зі знаком, ділить знакове ділиме націло на знаковий дільник. Команда IDIV сприймає в якості знака старші (перші ліворуч) біти (1- від'ємне число, 0 – позитивне число). Ділення на нуль викликає переривання.

Синтаксис: **IDIV дільник (делитель)**

Символьний код: IDIV регістр/пам'ять

Алгоритм роботи: Для команди необхідно задати два операнди - ділиме і дільник. Ділиме задається неявно, і розмір його залежить від розміру дільника, місцезнаходження якого вказується в команді:

- якщо *дільник розміром в байт*, то ділине повинно бути розташовано *в регістрі AX*. Після операції частка поміщається в AL, а залишок - в AH;
- якщо *дільник розміром в слово*, то ділине повинно бути розташовано **в парі** *регістрів DX:AX*, причому молодша частина ділимого знаходиться в ax. Після операції частка поміщається в AX, а залишок - в DX;
- якщо *дільник розміром в подвійне слово*, то ділине повинно бути розташовано *в парі регістрів EDX:EAX*, причому молодша частина діленого знаходиться в eax. Після операції частка поміщається в EAX, а залишок – в EDX.

Залишок завжди має знак діленого. Знак частки залежить від стану знакових бітів (старших розрядів) ділимого і дільника.

Впливає на стан прапорців AF, CF, OF, PF, SF, ZF.

Для збільшення розрядності знакового ділимого використовується інструкція CWD (*Convert Word to Doubleword*) та MOVSX - команди перетворення слова у подвійне слово.

Наприклад, MOV AX, WORD1 ; розташувати слово в AX
CWD ; розширити до подвійного слова в DX:AX

MOVSX DBWORD, DX; слово в DX перетворюється у подвійне

```
;ділення слів
	mov	ax, 1045 ; ділине
	mov	bx, 587  ; дільник
	cwd		 ; розширення дільника dx:ax
	idiv	bx	 ; частка в ax, залишок в dx
```

10.10. Команди вирівнювання довжини даних

При виконанні арифметичних операцій операнди мають бути однієї довжини. Для вирівнювання довжини числових даних зі знаком використовуються команди:

- **CBW** (Convert Byte to Word) перетворює байт у слово;
- **CWD** (Convert Word to Double) розширює слово до подвійного слова для процесора 8086;
- **CWDE** перетворює слово у подвійне слово для процесора 80386;
- **CDQ** (Convert Double to Quad) перетворює подвійне слово у четверне слово для процесора 80386.

Команда **CBW** значення регістра **AL** записує у слово в регістрі **AX** з урахуванням знаку (старшого розряду регістра **AL**). Наприклад,

```
        mov ax,1124h
        cbw             ;AX=0024h
```

Команда CWD перетворює слово в регістрі AX у подвійне слово, яке зберігається в регістрах DX і AX (у DX – старші, в AX– молодші розряди).

```
        mov  ax,-134
        cwd             ;DX:AX=FFFFFF7Ah
```

Команда **CWDE** перетворює слово в регістрі **AX** у подвійне слово, яке зберігається в регістрі **EAX**.

```
        mov  ax,40h
        cwde            ;EAX=00000040h
```

Команда **CDQ** перетворює подвійне слово в регістрі **EAX** в подвійне слово, яке зберігається в регістрах **EAX** та **EDX** (в **EDX** – старші, в **EAX**– молодші розряди).

```
        mov  eax,34h
        cdq             ;EDX:EAX=0000000000000034h
```

10.11. Команда збільшення операнда на одиницю INC (INCrement operand by 1)

Команда збільшує значення операнда у пам’яті або у регістрі на 1.

Синтаксис: **INC операнд**

Команда використовується для збільшення значення байта, слова, подвійного слова в пам’яті або у регістрі на одиницю. При цьому команда не впливає на прапорець CF.

```
inc     ax
```
;збільшити значення в ах на 1

10.12. Команда зменшення операнда на одиницю DEC (DECrement operand by 1)

Синтаксис: **DEC операнд**

Команда використовується для зменшення значення байта, слова, подвійного слова в пам'яті або у регістрі на одиницю. При цьому команда не впливає на прапорець CF.

```
mov      al, 9
...
dec      al       ; al=8
```

10.13. Команда порівняння двох операндів CMP (Compare operands)

Синтаксис: CMP операнд1,операнд2

Виконується віднімання: операнд1 – операнд2. В залежності від результату встановити прапорці (значення операндів не змінюється).

```
len   equ   10
...
        cmp   ax, len
        jne   m1    ; перехід, якщо (ax)<>len
        jmp   m2    ; перехід, якщо (ax)=len
```

10.14. Команда логічного виключення XOR

Синтаксис: XOR приймальник, джерело

Виконує операцію логічного виключення АБО над операндами. Біт результату дорівнює 1, якщо значення відповідних бітів операндів різні. Якщо однакові – біт результату дорівнює 0. Операція використовується для інвертування або порівняння певних бітів операндів

```
;змінити значення біта 0 регістра al на зворотне
   xor   al, 01h
```

Контрольні питання:

1. Характеристика команди складання без врахування знаку.
2. Вимоги до операндів команди складання.
3. Надайте характеристику команді складання з урахуванням перенесення.
4. Як працює команда обміну та складання?
5. Команда віднімання без врахування знаку.
6. Як працює команда віднімання з врахуванням знаку?
7. Які особливості має команда множення без врахування знаку?
8. Які особливості має команда множення з врахуванням знаку?
9. Особливості роботи команди беззнакового ділення.
10. Як працює команда цілочисельного ділення?
11. Надайте характеристку командам вирівнювання довжини даних.
12. Як працює команда інкремент?
13. Як працює команда декремент?
14. У чому полягають особливості команди порівняння даних?
15. Як працює команда логічного виключення?

Лекція 11. Команди передачі управління

Безумовний перехід JMP. Команди умовного переходу зручно застосовувати для перевірки різних умов (JA, JAE, JB, JBE, JC, JE, JZ, JG, JGE, JL, JLE, JNA, JNAE, JNB, JNBE, JNC, JNE, JNG, JNGE, JNL, JNLE, JNO, JNP, JNS, JNZ, JO, JP, JPE, JPO, JS, JZ). Виклик процедур (безумовний перехід). Приклади.

Безумовний перехід JMP (JuMP)

Синтаксис: **jmp мітка**

Використовується в програмі для організації безумовного переходу як всередині поточного сегмента, так і поза його межі. При певних умовах у захищеному режимі команда **jmp** може використовуватися для переключення задач.

Алгоритм роботи команди. Команда jmp в залежності від типу свого операнда змінює вміст або тільки одного регістра EIP або обох CS та EIP [17, 18]:

- якщо операнд є мітка у поточному сегменті команд, асемблер формує машинну команду, операнд якої є значення зі знаком, що є зсувом переходу відносно наступної за **jmp** команди. При такому переході змінюється тільки регістр eip/ip;
- якщо операнд у команді jmp – символічний ідентифікатор комірки пам'яті, асемблер вважає, що в ній знаходиться адреса, за якою треба передати управління;

Виконання команди не впливає на прапорці. Цю команду використовують для виконання ближніх та дальніх безумовних переходів.

Перехід, якщо виконана умова JCC (Jump if condition)
Перехід, якщо CX/ECX дорівнює нулю (JCXZ/JECXZ)
(Jump if CX=Zero/ Jump if ECX=Zero)

Синтаксис: **jcc мітка**
jcxz мітка
jecxz мітка

Використовується для переходу всередині сегмента команд в залежності від деякої умови.

Алгоритм роботи команди jcc. Перевірка стану прапорців в залежності від коду операції:

- якщо умова виконується, перейти до комірки, позначеної операндом;
- якщо умова не виконується, передати управління наступній команді.

Алгоритм роботи команди jcxz/jecxz. Перевірка умови чи дорівнює нулю вміст регістру *ecx/cx*.

- якщо умова виконується, то вміст регістру *ecx/cx* дорівнює нулю, перейти до комірки, позначеної у операнді;

- якщо умова не виконується, то вміст регістру *ecx/cx* не дорівнює нулю, передати управління наступній команді, що стоїть після jcxz/jecxz.

Команди умовного переходу зручно застосовувати для перевірки різних умов. Букви СС в команді ***JCC*** у скороченому вигляді описують умови переходу і мають значення:

E – Equal (дорівнює),
N – Not (ні, не дорівнює),
G – Great (більше, використовується для чисел зі знаком),
L – Less (менше, використовується для чисел зі знаком)**,**
A – Above (вище, більше, використовується для чисел без знаку)**,**
B – Below (нижче, менше, використовується для чисел без знаку)**,**
C – Carry Flag (прапорець перенесення, встановлюється в 1, якщо результат попередньої операції не вміщається в приймачі і сталося перенесення зі старшого біта або якщо потрібно зайняти (при відніманні). Інакше встановлений в 0),
O – **Overflow Flag** (прапорець переповнення, встановлюється в 1, якщо результат попередньої арифметичної операції над числами зі знаком виходить за допустимі для них межі. Наприклад, якщо при складанні двох позитивних чисел виходить число зі старшим бітом, що дорівнює одиниці, тобто негативне і навпаки),
S - **Sign Flag (**прапорець знака, цей прапорець завжди дорівнює старшому біту результату),
P - Parity Flag (прапорець парності, встановлюється в 1, якщо молодший байт результату попередньої команди містить парну кількість бітів, які дорівнюються 1. Якщо кількість одиниць в молодшому байті є непарною, то цей прапорець дорівнює 0).

Перелік команд умовного переходу, відповідні прапорці та умови переходу наведені в табл. 7.

Таблиця 7. Перелік команд умовного переходу

Команда	Стан прапорців, які перевіряються	Умови переходу
JA	CF = 0 и ZF = 0	якщо вище (>)
JAE	CF = 0	якщо вище або дорівнює (> / =)
JB	CF = 1	якщо нижче (<)
JBE	CF = 1 або ZF = 1	якщо нижче або дорівнює (< / =)
JC	CF = 1	якщо перенос
JE	ZF = 1	якщо дорівнює
JZ	ZF = 1	якщо нуль
JG	ZF = 0 и SF = OF	якщо більше

JGE	SF = OF	якщо більше або дорівнює
JL	SF <> OF	якщо менше
JLE	ZF=1 або SF <> OF	якщо менше або дорівнює
JNA	CF = 1 и ZF = 1	якщо не вище (не більше)
JNAE	CF = 1	якщо не вище або дорівнює (не більше / =)
JNB	CF = 0	якщо не нижче (не менше)
JNBE	CF=0 и ZF=0	якщо не нижче або дорівнює (не менше / =)
JNC	CF = 0	якщо немає переносу
JNE	ZF = 0	якщо не дорівнює
JNG	ZF = 1 или SF <> OF	якщо не більше
JNGE	SF <> OF	якщо не більше або дорівнює
JNL	SF = OF	якщо не менше
JNLE	ZF=0 и SF=OF	якщо не менше або дорівнює
JNO	OF=0	якщо немає переповнення
JNP	PF = 0	якщо кількість одиничних бітів результата є непарною (непарний паритет)
JNS	SF = 0	якщо знак плюс(знаковий (старший) біт результата дорівнює нулю)
JNZ	ZF = 0	якщо немає нуля
JO	OF = 1	якщо переповнення
JP	PF = 1	якщо кількість одиничних бітів є парною (парний паритет)
JPE	PF = 1	теж що і JP (парний паритет)
JPO	PF = 0	теж що і JNP (непарний паритет)
JS	SF = 1	якщо знак мінус (знаковий (старший) біт результата дорівнює 1)
JZ	ZF = 1	якщо нуль

Логічні умови «більше» та «менше» відносяться до порівнянь цілочисельних значень зі знаком, умови «вище» та «нижче» - до порівнянь цілочисельних значень без знака. Є декілька мнемонічних позначень однієї і тієї ж команди. Це пояснюється тим, що для мікропроцесора і8086 команди умовного переходу могли здійснювати тільки короткі переходи в межах -128 до +127, починаючи від наступної команди. Починаючи з мікропроцесора і386, ці команди могли виконувати будь-які переходи в межах поточного сегмента команд. Це стало можливим шляхом введення в систему команд мікропроцесора додаткових команд. Для переходу між сегментами треба комбінувати команди умовного переходу та команду безумовного переходу jmp.

Застосування команд *jcxz/jecxz*:

Команда	Стан прапорців в eflags/flags	Умови переходу
JCXZ	не впливає	якщо регістр CX=0
JECXZ	не впливає	якщо регістр ECX=0

Команду jcxz/jecxz зручно використовувати для організації цикла та ланцюжкових команд, які використовують регістр *ecx/cx*. Ця команда виконує тільки ***близькі переходи в межах -128 до +127 байт***, починаючи від наступної команди. Її використовують для попередньої перевірки лічильника циклу в регістрі cx для того, щоб обійти цикл, якщо його лічильник є нульовим.

Наприклад,

```
        jcxz    m1      ;обійти цикл, якщо cx=0
 cycl:
        ; деякий цикл
        loop    cycl
        ............
m1:     ...
```

Іноді спільно з інструкцією JMP використовується операція SHORT –

jmp SHORT мітка.

Для вказівки на цільову мітку інструкція JMP зазвичай використовує 16-бітовий зсув. *Операція SHORT вказує Турбо Асемблеру, що потрібно використовувати не 16-бітовий, а 8-бітовий зсув* (що дозволяє заощадити в інструкції JMP один байт). Наприклад, останній фрагмент програми можна переписати так, що він стане на два байти коротше:

```
        mov   ax,1
        jmp   SHORT AddTwoToAX
 AddOneToAx:
        inc   ax
        jmp   SHORT AXIsSet
 AddTwoToAX:
        inc   ax
 AXIsSet:
        .
        .
```

Недолік використання *операції SHORT (короткий)* полягає в тому, що короткі переходи *можуть здійснювати передачу управління на мітки, віддалені від інструкції JMP не далі, чим на 128 байтів*, тому в деяких випадках Турбо Асемблер може повідомляти вам, що мітка недосяжна за допомогою короткого переходу. До того ж операцію SHORT має сенс використовувати для посилань вперед, оскільки для переходів назад (на попередні мітки) Турбо Асемблер автоматично використовує короткі переходи, якщо на мітку можна перейти за допомогою короткого переходу, і довгі інакше.

Асемблер, зустрічаючи в програмі мітку, автоматично приписує їй тип Near (близький). Якщо за допомогою команди Jmp треба передати управління від команди Jmp далі більше, чим на 128 байтів, є перехід дальнього типу на мітку. Директива *LABEL* дозволяє надати тип мітці FAR**.**

Інструкцію JMP можна використовувати для переходу в інший сегмент коди, завантажуючи в одній інструкції і регістр CS, і регістр IP. Для цього використовують мітку дальнього переходу з описом FAR.

Ім'я мітки LABEL FAR

Наприклад, в програмі:

```
CSeg1     SEGMENT
        ASSUME   CS:Cseg1
        .
        .
FarTarget         LABEL   FAR
        .
        .       .
CSeg1     ENDS
        .       .
        .
CSeg2     SEGMENT
        ASSUME   CS:CSeg2
        .          .
        jmp   FarTarget  ; перехід дальнього типу
        .          .
CSeg2     ENDS
        .       .
```

виконується перехід дальнього типу.

Якщо необхідно, щоб мітка примусово інтерпретувалася, як мітка дальнього типу, можна використовувати операцію FAR PTR:

jmp FAR PTR ім'я мітки.

Викорстовуючи FAR PTR, виконується перехід на мітку дальнього типу в одному і тому ж сегменті.

Наприклад, у фрагменті програми:

```
        .
        jmp   FAR PTR NearLabel
        nop   /*немає операції*/
 NearLabel:
        .       .
```

виконується перехід дальнього типу на мітку NearLabel, хоча ця мітка знаходиться в тому ж сегменті коди, що і інструкція JMP.

Нарешті, ви можете виконати перехід за адресою, записаною в регістрі або в змінній пам'яті. Наприклад:

```
        .
        mov   ax, OFFSET TestLabel
        jmp   ax
        .
  TestLabel:
        .
```

Тут виконується перехід на мітку TestLabel, так само, як і в наступному фрагменті:

```
      .
      .DATA
JumpTarget      DW   TestLabel
      .
      .CODE
      .
      jmp   [JumpTarget]
      .
TestLabel:
      .
```

Виклик процедур (безумовний перехід)

Процедура або підпрограма – це основна функціональна одиниця декомпозиції деякої задачі. Програмний сегмент може розбиватися на частини директивами визначення підпрограм (процедур). Процедура являє собою групу команд для вирішення конкретної задачі. Процедура має засоби отримання управління у цю точку, хоча процедура описана однократно. ЇЇ можна викликати у будь-якому місці програми багатократно. Для опису послідовності команд у вигляді процедури на мові Асемблер використовують дві директиви **PROC** та **END**. Процедура повинна мати ім’я, яке всключається в обидві директиви. У сегменті коду процедури можуть розташовуватися послідовно, а можуть бути вкладена одна в іншу. Для виклику процедури треба скористатися командо **CALL**. Існує декілька різновидів команди виклику процедури CALL. Всі команди виклику є безумовними і використовуються для передачі управління процедурі. Внутрішньо сегментний виклик **NEAR CALL** (близький, обсяг команд менше 64 Кб) використовується для передачі управління процедурі, що знаходиться в тому ж сегменті. Міжсегментний виклик **FAR CALL** використовується для передачі управління процедурі, що знаходиться в іншому сегменті або навіть і іншому програмному модулі.

Команда виклику процедури : **CALL ім’я процедури** [17, 18].

Команда CALL за ідеєю роботи аналогічна команді JMP, але додатково ця команда запам’ятовує у стеку ще і точку повернення.

Логіка роботи команди CALL для випадку Near-виклику процедури:

Зменшується адреса в SP (одне слово) SP=SP-2.

PUSH IP ; зберігається вміст IP в стеку

<IP>=<IP>+ *зсув до потрібної процедури*

В стеку зберігається точка програми, до якої повинне бути здійснене повернення з процедури, яка здійснювала виклик. Адреса початку процедури розміщується в IP, очищається черга підготовлених до виконання інструкцій в процесорі.

Інструкція RET (Retn, Return) виконує повернення з процедури типу Near, виконує зворотні дії: забирає зі стеку (операція POP) старе значення IP (знову очищає чергу інструкцій), збільшує адресу в *SP* на 2 - *SP=SP+2.*

Логіка роботи команди CALL для випадку Far-виклику процедури:

PUSH IP

PUSH CS

< CS>=<Code2>

<IP>=<IP>+ *зсув до потрібної процедури*

Code2 – це ім'я другого сегмента, куди саме треба передати управління для Far-переходу.

Викликається процедура, яка може розташовуватися в іншому сегменті. Виклик процедури типу Far розташовує у стек значення з CS та IP. Відповідна інструкція RET забирає їх зі стеку.

Організація програми з викликом процедури типу Near

```
TITLE   A5CALL   Виклик процедури
.MODEL SMALL
.STACK
………………………….
.DATA
…………………………..
.CODE
A2MAIN    PROC    FAR    ;  Головна процедура
          CALL  D10      ;   Виклик процедури D10
……………………………………………..
          MOV AX,  4C00H;    Завершити роботу
          Int   21 h
A2MAIN     ENDP
…………………………………………………………
D10           PROC   NEAR
          CALL   C10
……………………………………………………….
          RET   ;   Повернутися у точку виклику
D10           ENDP
……………………………………………………………………
C10            PROC  NEAR
```

```
……………………………………………………
C10           ENDP
…………………………………………………..
END    A2MAIN

Або
A2MAIN:
            Mov ax,  @data
            Mov dx,  ax
            …………………..
            Call   D10
            …………………
            Mov ax,   4c00h
            Int  21h
        D10  PROC
        ……………………….
            Call B10
            …………..
            Ret
        D10 ENDP
        B10  Proc
        ………………..
               Ret
         B10  ENDP
 End   A2MAIN
```

За аналогією роботи JMP-оператора, коли треба передати управління на мітку Lb1, яка знаходиться у другому сегменті (для Far-переходу), у цьому випадку, з точки зору компілятора змінюється не тільки вміст регістр IP, а йвміст сегментного регістра CS. Для цього треба скористатися директивами Extern та Public. Наприклад:

```
………………………………..
Code1   Segment
Extern   Lb1:Far
…………………………………………
JMP Code2:Lb1
……………………………………….
Cod1   Ends
Code2   Segment
Public  Lb1
……………………………………..
Lb1:
……………………………………….
Code2 Ends
```

В цьому випадку логіка роботи команди JMP наступна:
< CS>=<Code2>
<IP>=<IP>+ ***зсув до потрібної процедури***

Або

```
A10MAIN PROC  FAR
……………………………………
            CALL  T1
            CALL   B10
            CALL  D10
            ……………………
A10MAIN ENDP
T1              PROC  NAER
             PUSHA
             …………..
             POPA
             RET
T1               ENDP
B10             PROC  NAER
             PUSHA
             …………..
             POPA
             RET
B10             ENDP
D10             PROC  NAER
             PUSHA
             …………..
             POPA
             RET
D10             ENDP
END       A2MAIN
```

Контрольні питанння:

1. Особливості роботи команди безумовного переходу.
2. Який алгоритм роботи команд умовного переходу?
3. Наведіть основні значення, які описують умови переходу?
4. Що означають логічні умови «більше» та «менше»?
5. Що означають логічні умови «вище» та «менше»?
6. Яка команда виконує близькі переходи?
7. Які особливості виконання команд безумовного переходу в інший сегмент?
8. Які особливості команди виклику процедури для ближнього виклику?
9. Які особливості команди виклику процедури для дальнього виклику?

Лекція 12. Організація циклів

Управління циклом по CX - LOOP. Використання оператора loop для роботи з масивами. Робота з одновимірним масивом. Робота з двовимірним масивом.

Управління циклом по CX LOOP *(LOOP control by register cx)*

Синтаксис**: loop мітка**

Команда призначена для організації циклу з лічильником у регістрі CX.

Алгоритм роботи [17, 18]*:*

- виконати декремент вмісту регістра *ecx/cx* (зменшити на 1);
- аналіз регістра *ecx/cx*: якщо *ecx/cx*=0, передати управління наступній команді, яка записана за командою loop ; якщо *ecx/cx*=1, передати управління команді, мітка якої зазначена в операнді loop.

Команда не впливає на стан прапорців. Кількість повторень цикла задається значенням значенням в регістрі *ecx/cx* перед входженням в послідовність команд, які утворюють тіло циклу.Для запобігання виконання цикла при нульовому значенні *ecx/cx* треба використовувати команду jecxz/jcxz.

Наприклад,
```
        mov     cx,10
    …………………………..
        jcxz    m1
   cycl:
   ;тіло циклу
loop   cycl
   m1:
```

Приклад: Підрахунок нульових байтів з використанням команди **loop**
```
prg_10_3.asm
.model small
.stack 100h
.data
len equ 10   ; кількість елементів в mas
mas db 1,0,9,8,0,7,8,0,2,0
.code
 start:
mov ax, @data
mov ds, ax
mov cx, len ; довжину поля mas в cx
xor ax,ax
xor si,si
jcxz exit ;  перевірка cx на 0, якщо 0, то вихід
 cycl:
cmp mas[si], 0   ;порівняти наступний елемент mas з 0
jne ml        ;якщо не дорівнює 0, то на ml
```

```
inc al              ;в al – лічильник нульових елементів
  ml:
inc si              ;перейти до наступного елемента
loop cycl
 exit:
mov ax, 4c00h
int 21h         ;повернення управління операційній системі
end start
```

*Використання оператора **loop** для роботи з масивами*

Одновимірний масив. Такий масив є вектором і він являє собою послідовне розташування елементів у пам'яті. Для локалізації потрібного елемента одновимірного масиву потрібно знати його індекс. *Асемблер немає засобів для роботи з масивом як зі структурою даних*, тому для використання елемента масива *необхідно обчислити його адресу*.

Для обчислення адреси і-го елемента одновимірного масиву можна використати формулу: $\mathbf{a_i = A + i*len}$,

де А – адреса першого елемента масива розмірністю n,

i – індекс (i=0, … n-1),

len – розмір елемента масива в байтах.

При такому визначенні можна не вказувати на тип елемента масива.

Приклад використання організації циклу для виведення на екран текстового символьного масиву, який заповнено кодами алфавітно-цифрових та псевдографічних символів.

Для цього необхідно скористатися *Таблицею символів.*

Як відомо для кодування текстової інформації прийнятий міжнародний стандарт ASCII (*American Standard Code for Information Interchange*). За цим стандартом:

- всі символи знаходяться в спеціальній таблиці,
- кожний символ займає 1 байт,
- всього 256 символів,
- кожний символ має свій внутрішній код, який співпадає з порядковим номером символу в таблиці,
- перша половина цієї таблиці (символи з номерами з 0 по 127) стандартна, вона містить цифри, латинські літери та інші необхідні символи.
- у другій половині таблиці (символи з номерами з 128 по 255) знаходяться букви національних алфавітів і додаткові символи. Ця частина таблиці є різною для різних країн і різних кодових таблиць, встановлених в операційній системі.

Програма **Таблица символов** (табл. 12.1), як окремі символи або цілі послідовності символів будь-якого шрифту, з низки встановлених до Windows, можуть бути скопійовані до Буферу обміну і потім використані в будь-якій іншій програмі. Знаходиться ця програма в підгрупі **Служебные** групи **Стандартные** головного меню Windows. Після запуску програма показує всі символи деякого шрифту. Вибрати інший шрифт можна у полі **Шрифт**. Якщо клацнути по символу.

він буде відображений у збільшеному масштабі. Виділивши символ або мишею, або клавіатурою, клацаємо по кнопці **Выбрать**, після чого цей символ з’являється у полі **Копировать**. Так само можна додати до цього поля й інші символи. Вибрану послідовність символів переводимо до Буферу обміну за допомогою кнопки **Копировать**.

Якщо вибрано якийсь символ, в нижній рядку вікна наводиться значення коду цього символу в кодуванні Юнікод (в шістнадцятковому вигляді). Якщо ж до того це спеціальний символ, який не можна набрати на клавіатурі, в правому куті цього рядка наводиться значення коду в десятковій системі. Цей символ можна вставити до документу, якщо за натиснутої лівої клавіші **<Alt>** та увімкненому режимі **Num Lock** набрати код на цифровій клавіатурі.

Таблиця 8. Таблиця символів

Код	Символ	Код	Символ	Код	Символ	Код	Символ	Код	Символ	Код	Символ
0		43	+	86	V	128	А	171	л	214	╓
1	☺	44	,	87	W	129	Б	172	м	215	╫
2	☻	45	-	88	X	130	В	173	н	216	╪
3	♥	46	.	89	Y	131	Г	174	о	217	┘
4	♦	47	/	90	Z	132	Д	175	п	218	┌
5	♣	48	0	91	[	133	Е	176	░	219	█
6	♠	49	1	92	\	134	Ж	177	▒	220	▄
7	•	50	2	93	]	135	З	178	▓	221	▌
8	◘	51	3	94	^	136	И	179	│	222	▐
9		52	4	95	_	137	И	180	┤	223	▀
10		52	5	96	`	138	К	181	╡	224	р
11	♂	54	6	97	a	139	Л	182	╢	225	с
12	♀	55	7	98	b	140	М	183	╖	226	т
13		56	8	99	c	141	Н	184	╕	227	у
14	♫	57	9	100	d	142	О	185	╣	228	ф
15	☼	58	:	101	e	143	П	186	║	229	х
16	►	59	;	102	f	144	Р	187	╗	230	ц
17	◄	60	<	103	g	145	С	188	╝	231	ч
18	↕	61	=	104	h	146	Т	189	╜	232	ш
19	‼	62	>	105	i	147	У	190	╛	233	щ
20	¶	63	?	106	j	148	Ф	191	┐	234	ъ
21	§	64	@	107	k	149	Х	192	└	235	ы
22	▬	65	A	108	l	150	Ц	193	┴	236	ь

23	↕	66	B	109	m	151	Ч	194	┬	237	э
24	↑	67	C	110	n	152	Ш	195	├	238	ю
25	↓	68	D	111	o	153	Щ	196	─	239	я
26	→	69	E	112	p	154	Ъ	197	┼	240	Е
27	←	70	F	113	q	155	Ы	198	╞	241	ё
28	∟	71	G	114	r	156	Ь	199	╟	242	Є
29	↔	72	H	115	s	157	Э	200	╚	243	є
30	▲	73	I	116	t	158	Ю	201	╔	244	І
31	▼	74	J	117	u	159	Я	202	╩	245	ї
32		75	K	118	v	160	а	203	╦	246	У
33	!	76	L	119	w	161	б	204	╠	247	ў
34	"	77	M	120	x	162	в	205	═	248	°
35	#	78	N	121	y	163	г	206	╬	249	·
36	$	79	O	122	z	164	д	207	╧	250	·
37	%	80	P	123	{	165	е	208	╨	251	√
38	&	81	Q	124	|	166	ж	209	╤	252	№
39	'	82	R	125	}	167	з	210	╥	253	¤
40	(	83	S	126	~	168	и	211	╙	254	■
41	)	84	T	127	⌂	169	й	212	╘	255	
42	*	85	U			170	к	213	╒		

Символ f в таблиці має номер 102.

Символ Б має номер 129.

Символи англійських а А і російські а А мають однаковий вигляд, але внутрішні коди у них різні - 65 і 128.

Символ 0 (нуль) має внутрішній код 48.

Символ пробіл має внутрішній код 32.

Великі і маленькі букви мають різні ASCII-коди.

Якщо уважно проаналізувати *Таблицю символів кодами алфавітно-цифрових та псевдографічних символів,* то ці символи мають коди від 32 (пропуск) до 254 (суцільний квадрат). Такий масив можна створити в операторі db:

```
symbols  db  32, 33, 34, 35, 36, 37, 38, 39,…,
```

однак це можна зробити програмним способом:

```
; Цикли
text     segment            ; Початок сегмента
assume   CS:text, DS:data
begin:   mov   AX, Data     ; Ініціалізація сегментного
         mov   DS, AX       ;регістра DS
      ; Підготовка організації циклу
         mov CX, 223        ; Кількість циклів
         mov SI,0     ;Індекс   елемента   в   масиві,   що
заповнюється
         mov AL, 32         ; Код першого символа пропуска
; Початок циклу (складається з 4-х команд)
```

```
fill:mov symbols[SI], AL; Черговий  код  заноситься  в  байт
масива
    inc  AL          ; Створюємо код наступного символа
    inc  SI          ; Пересуваємося на 1 байт у масиві
    loop fill        ; Команда циклу з CX кроків
; Виведення на екран символьного масиву
    mov  AH, 40h     ; Функція виведення на екран
    mov  BX, 1           ; Дескриптор екрана
    mov  CX, 223     ;  Кількість  символів,  які  виводяться
на екран
    mov  DX,  offset  symbols;  Адреса  повідомлення,  яке
виводиться
    int 21h          ; Виклик DOS
; Успішне завершення програми
    mov  AX, 4C00h
    int 21h
text ends                ; Кінець сегмента коду
data segment         ; Початок сегмента даних
; Поля даних програми
symbols  db  223 dup (‘*’)    ; Масив, який заповнюється
data ends                ; Кінець сегмента даних
stk  segment stack       ; Початок сегмента стеку
    db 256 dup(0)        ; Стек
stk ends             ; Кінець сегмента стеку
    end begin            ; Кінець тексту програми
```

Приклад знаходження суми елементів одновимірного масиву.

```
.model small
.stack 100h
.data
  matr dw 4, 7, 12, 8, 3, 2, 5, 1, 9, 3
  summa  dw  ?
.code
start:
    mov ax, @data
    mov ds, ax
    xor si, si       ; обнулення зсуву індексу від початку
масиву
    mov cx, 10
    xor ax, ax
    mov summa, ax        ; обнулення суми
    mov bx, offset matr  ; зсув до елементу масива
M1:    mov ax, [bx][si]      ;  записуємо  в  ах  елемент
матриці
   add summa, ax     ; накопчуємо суму
```

```
    add si, 2           ; перехід до наступного елемента
    loop M1
    mov av, 4c00h
    int 21h
end start
```

Двовимірний масив. Представлення двовимірних масивів є більш складним. Ми вважаємо його матрицею, структура зберігання даних - вектор. Програміст повинен сам для себе інтерпретувати як вибирати елемент з матриці по рядкам або по стовпцям. При цьому швидше змінюється останній елемент індексу. Наприклад, масив A розмірністю n×m, де 0<i<n-1, 0<j<m-1:

a_{00} a_{01} a_{02} a_{03}

a_{10} a_{11} a_{12} a_{13}

a_{20} a_{21} a_{22} $\mathbf{a_{23}}$

a_{30} a_{31} a_{32} a_{33}

Цей масив в пам'яті фізично має таке представлення як вектор:

a_{00} a_{01} a_{02} a_{03} a_{10} a_{11} a_{12} a_{13} a_{20} a_{21} a_{22} a_{23} a_{30} a_{31} a_{32} a_{33}

Номер конкретного елемента масива в пам'яті визначається адресною функцією, виходячи із значення його індексів: $\mathbf{a_{ij} = n*i+j}$,

де i – номер рядка, j – стовпця.
Наприклад, n=4, m=4, i=0,..3, j=0,…3,
$a_{23} = n*i+j = 4*2+3=11$,
де 4 – кількість елементів в рядку, 2 – номер рядка, j – стовпця.

Наприклад, масив розміром 3*5 можна представляти як двовимірну структуру (i=0, 1, 2; j=0, 1, 2, 3, 4)

a_{00} a_{01} a_{02} a_{03} a_{04}
a_{10} a_{11} a_{12} a_{13} a_{14}
a_{20} a_{21} a_{22} $\mathbf{a_{23}}$ a_{24}

DATA_ARRAY DW 3 DUP (5 DUP(?))

Звернення до елементу, наприклад, (2,3), це 3-й рядок (i=2), 4-й стовпець (j=3) здійснюємо таким чином:

2*5=10 (номер рядка множимо на кількість елементів в рядку), додаємо номер елемента у рядку: 10+3=13.

Тобто в цьому випадку потрібний елемент має номер 13 за умови, що нумерація стовпців та рядків починається з 0. Елементи визначалися як слова, однак адресація ведеться побайтно, тому значення слід подвоїти, тобто 13-й елемент починається з 26-го байта. Звернутися до цього елемента можна так:

```
MOV      BX, 20             ; Рядок - початкова адреса (2*5)
MOV      DI, 6              ; Стовпець - зсув (3*2)
```

```
ADD      DATA_ARRAY[BX+DI], AX ; Додаємо до елемента масива
;значення з AX 26-й байт
```

Як правило, номер елемента у масиві визначається в результаті обчислень, а не вказується прямо у тексті програми.

Наприклад, значно зручніше розглядати масив як одновимірну структуру (виконаємо обнулінння масиву DATA_ARRAY):

```
   MOV  CX, 15                ; Число елементів
   MOV  BX, 0                 ; Зсув першого елемента
L10:  MOV  DATA_ARRAY[BX], 0  ;  Обнуляємо елемент
   ADD  BX, 2             ; Додаємо наступне слово у масиві
   LOOP L10               ; Повторити 15 разів
```

Для роботи з матрицею треба мати:

- початковий зсув першого байту матриці (записуємо його в регістр вх – mov bx, offset matr),

- зсув до елементів матриці (регістр si).

Наприклад, маємо матрицю 3*4 з типом даних dw:

4 7 8 3
1 2 4 4
6 2 1 7

Треба порахувати суму елементів в рядку. Запишемо основні команди коду.

```
    mov si, 0        ; xor si, si
    mov ryadok, 3  ;       лічильник рядків матриці
    mov stovp, 4          ; лічильник стовпців матриці
strok:                    ; обробка рядків
      mov cx, 4           ; стовпці
      xor ax, ax
      mov summa, ax       ; обнулення суми
  stovp:
        mov bx, offset matr           ; mov bx, 0
        mov ax, [bx][si]              ; [bx+si] зсув по si
        add summa, ax
        add si,2                      ; перехід до наступного елемента
  loop stovp
      add si, 2
      dec ryadok
      mov cx, ryadok
loop strok
```

Контрольні питання:

1. Наведіть алгоритм роботи команди управління циклом?
2. Які особливості оператора *loop* для роботи з одновимірним масивом?
3. Яке представлення має двовимірний масив в оперативній пам’яті?
4. Як визначається адреса конкретного елемента двовимірного масиву?

Лекція 13. Команди для роботи з рядковими даними, логічними операціями та зсувами

Робота з рядковими даними. Пересилання рядків - MOVS. Завантаження рядків - LODS. Запис рядків у пам'ять - STOS. порівняння рядків. Порівняння рядків - CMPS. Сканування рядка - SCAS. Використання префікса повторення REP. Логічні операції. Зсуви.

Оброблення рядкових даних. Команди MOVS, LODS, STOS, CMPS, SCAS

Для переміщення або порівняння строкових даних (наприклад, сортування назв в алфавітному порядку) використовуються інструкції [17, 18]:

MOVS – перемістити один байт ***MOVSB***, слово ***MOVSW***, подвійне слово з однієї комірки пам'яті в іншу ***MOVSD***;

LODS – завантажити з пам'яті байт в AL ***LODSB***, слово – в AX ***LODSW***, подвійне слово – в EAX ***LODSD***;

STOS – зберігти значення з AL ***STOSB***, AX ***STOSW***, EAX у пам'яті ***STOSD***;

CMPS – порівняти байти ***CMPSB***, слова ***CMPSW***, подвійні слова в пам'яті ***CMPSD***;

SCAS – порівняти вміст AL ***SCASB***, AX ***SCASW***, EAX з вмістом елемента даних у пам'яті ***SCASD***.

Інструкції для роботи з рядковими даними використовують пари регістрів ES:DI, DS:SI. Для рядків з непарною кількістю байтів вибирають операції, що обробляють по одному байту, для рядків з парною кількістю байтів використовують команди для оброблення слова або подвійного слова (якщо довжина рядка ділиться на 4). Регістри SI, DI містять конкретні величини зсувів, за якими розташовані у пам'яті дані. SI пов'язується з DS (для звернення до сегменту даних) - DS:SI, а DI пов'язується з ES (для звернення до додаткового сегменту) - ES:DI.

З цими інструкціями застосовується **префікс REP.** За наявності префікса REP рядкова команда виконується багатократно та може обробляти будь-яку кількість (вказується в регістрі CX) байтів, слів, подвійних слів.

Прапорець направлення (*Direction Flag, DF*) визначає у якому напряму будуть виконуватися операції, які послідовно повторюються.

Для обробки рядка зліва направо використовують інструкцію ***CLD (Clear DF)*** для встановлення DF=0.

Для обробки рядка з права наліво використовують інструкцію ***STD (Set DF)*** для встановлення DF=1.

REP – повторити інструкцію, поки CX не стане рівним 0.

Інструкція MOVS: переміщення рядкових даних

Інструкції MOVSB, MOVSW, MOVSD за наявності префікса **REP** та занесеної у регістр CX кількості можуть переміщувати вказану кількість символів. Адреса рядка призначення вказується в регістрах ES:DI , адреса рядка джерела вказується в регістрах DS:SI. В залежності від значення DF інструкція MOVS зменшує або збільшує в регістрах DI, SI на 1 при переміщенні байта, на 2 – слова, на 4 – подвійного слова.

Приклад 1: копіювати 20 байтів з одного рядка у інший

При роботі з рядками обов’язково слід ініціалізувати сегментні регістри DS, ES.

```
.data
        Send_str  db 20 dup (‘*’)   ; Поле джерело
        Recv_str  db 20 dup (‘ ’)   ; Поле приймальник
.code
Start:
    Mov AX, @data          ; Адреса сегменту даних
    Mov DS, AX             ; Заносимо цю адресу у DS
    MOV ES, AX             ; і ES
    CLD                    ; Встановити DF=0
    Mov CX, 20             ; Довжина
    LEA SI, Send_str       ; Завантажити адресу рядка джерела
    LEA DI, Recv_str       ; Завантажити адресу рядка приймальника
    REP MOVSB              ; Копіювання
```

Префікс повторення виконує рядкову команду в циклі. Нижче наведені команди, які є еквівалентними командам, наведеними в ланцюговій команді, але з використанням регістрів edi та esi:

```
    jecxz  m_end       ;перехід, якщо cx=0
m_loop:
     mov al,[esi]
     mov [edi],al
     inc esi
     inc edi
     loop   m_loop
m_end:
```

Приклад 2: перемістити 12 слів:

```
MOV  CX, 12                ; Кількість слів
LEA  DI, RECV_STR          ; Адреса RECV_STR (у ES:DI)
LEA  SI, SEND_STR          ; Адреса SEND_STR (у DS:SI)
REP  MOVSW                 ; Переміщуємо 12 слів
```

Інструкція LOADS: завантажити рядок

Ця інструкція переміщує з пам’яті в AL байт - **LOADSB**, в AX – слово **LOADSW**, в EAX – подвійне слово **LOADSD**. Адреса, з якої виконується завантаження, зберігається у парі DS:SI. В залежності від стану прапорця DF операція зменшує або збільшує значення SI на 1 при переміщенні байта, 2 – слова, 4 – подвійного слова.

Приклад 1: перемістити вміст string1 у string2, але так щоб string2 містила string1 у зворотному порядку

```
String1     db    ‘Student’
String2     db    7 dup (20H)        ; 7 пропусків
……………………………………..
     CLD                       ; Зліва направо
     MOV  CX, 7
     LEA  SI, String1          ; Адреса string1 (DS:SI)
     LEA  DI, string2+6               ; Адреса кінця string2 (ES:DI)
L20:  LODSB                    ; Завантажити символ в AL
     MOV  [DI], AL             ; Зберігти його в string2
     DEC  DI                   ; Перейти до попереднього символу в
                              ; string2
     LOOP   L20                ; Повторити процедуру 7 разів
```

Приклад 2: Реверсування рядка. Обробка початкового рядка виконується з права наліво. Регістр esi, що використовується в команді lodsb, встановлюється на останньому байті початкового рядка.

```
string1   db   "Hello";
string2   db    dup 10("0")

     mov ecx, 5              ; довжина рядка
     lea esi, string1
     add esi, ecx
     dec esi
     lea edi, string2
     std
m_beg:
     lodsb
     mov byte ptr [edi], al
     inc edi
     loop m_beg
     cld
```

Інструкція STOS: зберігання рядка

Інструкція зберігає вміст AL - **STOSB**, AX - **STOSW**, EAX - **STOSD** відповідно у байті, слові, подвійному слові у пам'яті. Адреса у пам'яті завжди мітиться у парі ES:DI. В залежності від стану DF інструкція STOS збільшує або зменшує значення в DI на 1 – для байта, на 2 – для слова, на 4 – для подвійного слова. STOS разом з REP використовується для заповнення області пам'яті певним значенням.

Приклад: розташувати послідовно 6 слів (пропуски) в рядок string1 з 12 символів:

```
CLD                  ; Зліва направо
Mov  AX, 2020H       ; Розмістити
Mov  CX, 06                  ; 6 слів по 2 пропуски
LEA  DI, string1     ; в рядок string1 (адреса в ES:DI)
REP STOSW
```

Приклад: сформувати рядок, який має вигляд 1010… (довжина 20 байтів), дублюючи зразок з використанням команди STOSW:

```
str1 db 20 dup(?)
     mov ax, "01"    ; зворотня послідовність байтів
     cld
     mov ecx, 10
     lea edi, str1
     rep stosw
```

Інструкція CMPS: порівняння рядків

Інструкція CMPS порівнює вміст одного елемента даних (за адресою DS:SI) з вмістом другого (за адресою ES:DI) – порівнює ASCII- коди символів. В залежності від стану прапорця DF CMPS збільшує або зменшує значення в SI, DI на 1 – для байта, на 2 – для слова, на 4 – для подвійного слова. Операція закінчується, коли CX=0.

Для побайтного порівняння використовується інструкція ***REPE CMPSB.***
Відповідно для порівняння слів - ***REPE CMPSW.***
Для порівняння подвійних слів - ***REPE CMPSD.***

Приклад: порівняти два рядки «STUD» і «STAD». Порівняння виконується побайтно зліва направо.

```
MOV CX, 4
LEA  DI, string3     ; ES:DI
LEA  SI, string2     ; DS:SI
```

```
REPE   CMPSB            ; порівняти два рядка
JE  Vykhid              ; Дорівнює, пропускаємо
…….                     ; Не дорівнює
```

Інструкція SCAS: перегляд рядка

Інструкція SCAS переглядає рядок щодо пошуку вказаного значення. Послідовно порівнюються комірки, адреси яких визначаються парою ES:DI, з вмістом AL - **SCASB**, AX - **SCASW**, EAX - **SCASD**. В залежності від значення DF SCAS збільшує або зменшує значення DI на 1 – для байтів, 2 – для слів, на 4 - для подвійних слів. Ця команда є корисною, коли треба у тексті знайти або фрагмент або символ.

Приклад, знайти у рядку символ R

```
String1        db “Integrator”
………………………………
CLD                 ; Перегляд зліва направо
Mov  AL, ‘r’        ; Пошук ‘r’ у рядку
Mov  CX, 10
LEA  DI, string1    ; ES:DI
REPE   SCASB        ; CX=0, DI=6
JE  Vykhid          ; Знайшли
……………………        ; Не знайшли
```

Приклад: Сканування і заміна символа. У рядку замінити перший знайдений символ '+' на символ '-'

```
str1 db '+Iv+an+Iva+nov+'
   cld
   mov ecx, 15       ; довжина рядка
   mov al, '+'
   lea edi, str1
   repne scasb       ; скануємо рядок, поки не знайдемо символ
                     ; або не закінчиться рядок
   jecxz m_not       ; символ не знайдено
   ; символ знайдено, edi вказує на наступний символ
   mov byte ptr [edi-1],' -' ...
```

Приклад: в рядку замінити усі символи '+' на символи '-'.

```
string1 db  "+Iv+an+Iva+nov+";
………

   xor ebx, ebx
   cld
   mov ecx, 15              ; кількість елементів
```

```
    mov al, '+'
    lea edi, string1
  m_beg:
      or al, 0              ; скидання zf
      repne scasb
      jz m_replace          ; zf=1 - знайдено входження
      jecxz m_end           ; ecx=0 – весь рядок переглянутий
      jmp m_beg
  m_replace:
      mov byte ptr [edi-1], '-'
      jmp m_beg
  m_end:
```

Приклад: Обчислити кількість нульових елементів масиву

```
mas db 0, 1, 0, -2, 34, 0, 0, 7, -1, 0
result db ?
…………………
    cld
    xor ebx, ebx            ; кількість нульових елементів
    mov ecx, 10             ; кількість елементів
    mov eax, 0
    lea edi, mas
m_beg:
    repne  scasd
    jz   m_inc              ; zf = 1 – знайдено входження
    jecxz   m_end           ; ecx = 0 - весь масив переглянуто
    jmp   m_beg
m_inc:
    inc  ebx
    jmp   m_beg
m_end:
    mov result, ebx
```

Команди логічних операцій

Такі команди дозволяють виконувати логічні операції, представлені булевими операторами.

AND операнд1, операнд2 (*кон'юнкція, логічне множення*)

Команда порозрядно виконує операцію «*І*» над бітами операндів, результат записується на місце першого операнда.

OR операнд1, операнд2 (*диз'юнкція, логічне складання*)

Команда порозрядно виконує операцію «*АБО*» над бітами операндів, результат записується на місце першого операнда.

XOR операнд1, операнд2 (*виключення АБО*, тобто складання за *mod* 2)

Команда порозрядно виконує операцію логічного виключення «АБО» над бітами операндів, результат записується на місце першого операнда.

TEST операнд1, операнд2 (*перевірка)*

Команда порозрядно виконує операцію логічного множення над бітами операндів. Стан операндів залишається незмінним, змінюється тільки прапорці *zf, sf, pf*. Це дає можливість аналізувати стан окремих бітів операнда без зміни їх стану.

NOT операнд (*логічне заперечення*)

Команда виконує порозрядне інвертування (змінює значення на зворотне) кожного біта операнда. Результат записується на місце операнда.

Команди AND, OR, XOR, TEST впливають на арифметичні прапорці таким чином:

OF, CF	завжди в “0”
SF, ZF, PF	визначаються розміром
AF	не визначене
NOT	не змінює прапорці

Призначення команд

AND - для виділення першого або більше бітів в операнді1 або скидання у “0” групи бітів

OR - можна встановити потрібний біт в “1”

XOR - проінвертувати біт (у операнд2 необхідно біт встановити у “1”, інші - у “0”)

TEST - для перевірки стану бітів

Починаючи з 386 процесора команди роботи з бітами:

AND -> BTR перевірити біт і скинути

OR -> BTS перевірити біт і встановити

XOR -> BTC перевірити біт і інвертувати його

TEST -> BT (bit test) перевірити біт

Команди зсувів

Команди цієї групи забезпечують маніпуляції над окремими бітами операндів, але іншим способом, ніж логічні команди, розглянуті вище.

Всі команди зсувів переміщають біти в полі операнда вліво або вправо залежно від коду операції.

Всі команди зсувів мають однакову структуру:

код_операції операнд, лічильник_зсувів

Кількість зсунутих розрядів (*лічильник_зсувів)* розташовується на місці другого операнда і може задаватися двома способами:

- *статично*, що припускає задання фіксованого значення за допомогою безпосереднього операнда;
- *динамічно*, що означає занесення значення лічильника зсувів в регістр cl перед виконанням команди зсуву.

Значення лічильника зсувів може знаходитися у діапазоні від 0 до 255. Але насправді це не зовсім так. В цілях оптимізації мікропроцесор сприймає тільки значення *п'яти молодших бітів* лічильника, тобто значення знаходяться в діапазоні від 0 до 31. В останніх моделях мікропроцесора, у тому числі і в мікропроцесорі Pentium, є додаткові команди, що дозволяють робити 64-розрядні зсуви.

За принципом дії команди зсувів можна розділити на два типи:

- *команди лінійного зсуву;*
- *команди циклічного зсуву.*

Команди лінійного зсуву

Команди лінійного зсуву діляться на два підтипи:

- команди логічного лінійного зсуву;
- команди арифметичного лінійного зсуву.

До команд *логічного лінійного зсуву* відносяться наступні:

SHL операнд, лічильник_зсувів (*Shift Logical Left*) - логічний зсув ліворуч.

Вміст операнда зсувається вліво на кількість бітів, визначених значенням лічильника_зсувів. Праворуч (у позицію молодшого біта) вписуються нулі.

Команду *SHL* зручно використовувати для множення цілочисельних операндів без знаку на ступені 2. Це найшвидший спосіб множення; помножити вміст ax на 16 (2 в ступені 4).

Приклад1:

```
mov  ax, 17
shl  ax, 4                    ; ax=272
```

Приклад 2:

```
bcd_dig dw    0905h       ; опис неупакованого BCD-числа 95
        mov  ax,   bcd_dig  ; пересилання
        shl   ah,  4          ; зсув вліво на 4 розряди: ah=90h
        add   al,  ah   ; складання для отримання результату: al=95h
```

SHR операнд, зсувається **лічильник_зсувів** (*Shift Logical Right*) — логічний зсув праворуч.

Вміст операнда зсувається вправо на кількість бітів, визначених значенням лічильника_зсувів. Зліва (у позицію старшого, знакового біта) вписуються нулі.

Команди арифметичного лінійного зсуву відрізняються від команд логічного зсуву тим, що вони особливим чином працюють із знаковим розрядом операнда.

Команду SHR можна використовувати для ділення цілочисельних операндів без знаку на ступені 2. Наприклад:

```
mov   cl,  4
shr   eax,  cl           ;(eax) розділити на 2 в ступені 4
```

SAL операнд, лічильник_зсувів (*Shift Arithmetic Left*) - арифметичний зсув ліворуч.

Вміст операнда зсувається вліво на кількість бітів, визначених значенням лічильника_зсувів. Праворуч (у позицію молодшого біта) вписуються нулі. Команда SAL **не зберігає знаку**, але *встановлює прапор cf у разі зміни знаку* черговим бітом, що висувається. В іншому команда SAL повністю аналогічна команді shl.

Команду SAL зручно використовувати для множення цілочисельних операндів без знаку на ступінь 2. Це найшвидший спосіб такого множення; помножити вміст ах на 16 (2 в ступені 4):

```
mov   ax, 17
sal     ax,  4
```

SAR операнд, лічильник_зсувів (*Shift Arithmetic Right*) - арифметичний зсув праворуч.

Вміст операнда зсувається вправо на кількість бітів, визначених значенням лічильника_зсувів. Зліва в операнд вписуються нулі. Команда SAR **зберігає знак**, відновлюючи його після зсуву кожного наступного біта.

Команду SAR можна використовувати для ділення цілочисельних операндів зі знаком на ступені 2.

Наприклад:

```
mov    ax, 88
;(ax) поділити на 2 у другому ступені, тобто на 4
sar    ax, 2                    ;  ax=22
```

Команди циклічного зсуву

До команд *циклічного* зсуву належать команди, що зберігають значення зсунутих бітів. Є два типи команд циклічного зсуву:

- команди простого циклічного зсуву;
- команди циклічного зсуву через прапорець перенесення cf.

До команд *простого циклічного* зсуву відносяться:

ROL операнд, лічильник_зсувів (Rotate Left) — циклічний зсув ліворуч.

Вміст операнда зсувається вліво на кількість бітів, визначених операндом лічильника_зсуву . Зсунуті вліво біти записуються в той же операнд справа.

Наприклад:

```
; поміняти місцями половинки регістра eax:
```

```
mov   ax,  0ffff0000h
mov   cl,  16
rol    eax,  cl          ;eax=0000ffffh
```

ROR операнд, лічильник_зсувів (*Rotate Right*) — циклічний зсув праворуч. Вміст операнда зсувається вправо на кількість бітів, визначених операндом лічильник_зсувів. Зсунуті праворуч біти записуються в той же операнд зліва.

```
;помістити чотири  молодших біта ax на місце старших бітів:
ror    ax,  4
```

Модифікація прапорців при зсувах

- AF не визначене,
- CF містить значення останнього висунутого біта,
- при однобітних зсувах OF=1, якщо зміна знаку, інакше не визначено,
- циклічні зсуви впливають тільки на OF і CF,
- SF, ZF і PF модифікується відповідно до результатів.

Починаючи з мікропроцесора 486 для маніпуляції з бітовими рядками довжиною до 64 бітів, використовуються команди зсуву подвійного слова:

Подвійний зсув праворуч (*Shift Left Double word*):

SHRD приймальник, джерело, кількість_зсувів

Подвійний зсув ліворуч (*Shift Right Double word*):

SHLD приймальник, джерело, кількість_зсувів

Контрольні питання:

1. Наведіть команди для оброблення рядкових даних.
2. Які команди вказують напрям оброблення рядка?
3. Характеристка команди переміщення рядкових даних в залежності від типу даних?
4. Як працюють команди завантаження рядка в залежності від типу даних?
5. Характеристка команди зберігання рядка в залежності від типу даних?
6. Особливості роботи команди порівняння рядків?
7. Особливості роботи команди перегляду рядка в залежності від типу даних?
8. Надайте характеристику командам логічних операцій.
9. Надайте характеристику командам зсуву?
10. Як працюють команди лінійного зсуву?
11. Як працюють команди арифметичного зсуву?
12. Як працюють команди циклічного зсуву?

Тема 8. Макрозасоби мови Асемблер
Лекція 14. Макрозасоби мови Асемблер IBM PC

Основні поняття: макровизначення, макрогенератор, макророзширення, макропроцесор. Складові макрозасобів. Породження міток і умовна макрогенерація. Варіанти розташування макровизначення. Організація циклу у макровизначенні. Вкладені макроси. Блоки повторень. Макрооператори.

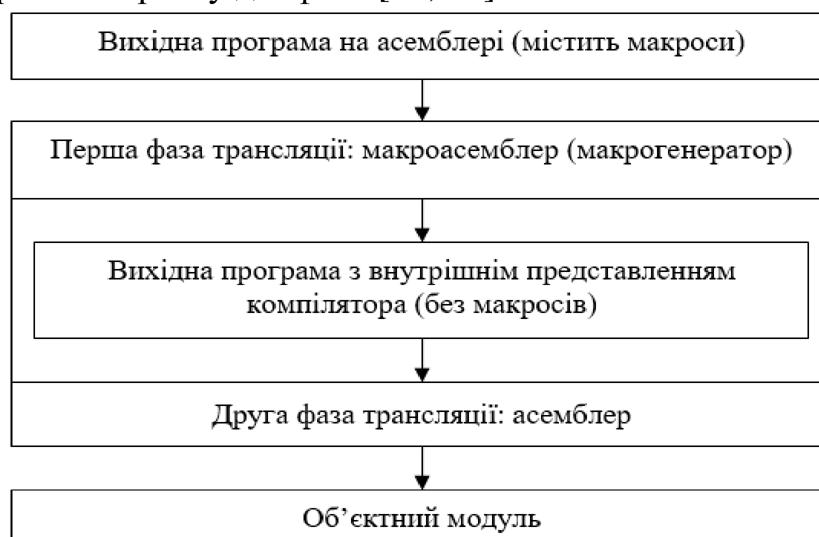
Основні поняття. Макровизначення та макрокоманда

Макрозасоби мови Асемблера IBM PC є одним з найпотужніших засобів цієї мови, яку ще називають МАКРОАСЕМБЛЕРОМ за здатність оперувати з набором команд як з однією командою. Тобто ділянки програми з однаковою структурою, що часто повторюються, можна оформити у вигляді макровизначень (макрокоманд, макросів), що характеризуються довільними іменами та списками формальних параметрів. Після створення макровизначення, коли в програмі з'являється команда, яка містить ім'я макровизначення та список формальних параметрів, відбувається генерація всього тексту макророзширення і усі формальні параметри замінюються фактичними. Змінюючи формальні параметри, можна при незмінній структурі макророзширення змінювати окремі його елементи.

Використання макросів спрощує вихідний текст програми, прискорює виконання програми, дозволяє скористатися розробленими бібліотеками макровизначень. Однак, використання макросів має і недоліки: макроси зберігаються у вигляді вихідних файлів та при підключенні до програми їх необхідно асемблювати, макроси можуть приховувати результати роботи команд.

З макросами активно працює мова С на рівні опису функцій за допомогою препроцесорної директиви **#define** та мова С++ - на рівні **inline**-функцій. Однак ці мови накладають на макроси суттєві обмеження, чого немає в асемблері.

Обробка програми на асемблері з використанням макрозасобів неявно здійснюється транслятором у дві фази [17, 18]:



Макроасемблер в загальній схемі трансляції програми TASM

Макрозасоби Асемблера IBM PC включають в себе три складові [17, 18]:

1. **Макровизначення (макрос) –** набір команд, який містить опис якоїсь дії або алгоритму. *Макрос повинен знаходитися на самому початку програми, до визначення сегментів.*

Синтаксис *макровизначення*:

ім'я_макроса MACRO [список_формальних_аргументів]
;; тіло макровизначення
Endm

2. **Макрокоманда –** коротке посилання на макровизначення (виклик макроса):
ім'я_макроса MACRO [список_формальних_аргументів]

3**. Макророзширення** (макропідстановка, макровставка) – вставка замість макрокоманди макроса з заміною формальних параметрів на фактичні (якщо вони є).

Макровизначення може простим та вкладеним, тобто містити у собі інше макровизначення. Рівень вкладання макровизначень може бути будь-яким, з одного макроса можна викликати інші макроси.

Наявність двох символів ;; в коментарях – це макровизначення і означає ознаку подавлення виведення коментарю у лістингу макророзширення.

Псевдооператори EQU та „=" (дорівнює)

До простих засобів макроасемблера відноситься псевдооператори EQU та „=".

Ці псевдооператори призначені для присвоєння деякому виразу символічного імені або ідентифікатора. При трансляції макроасемблер підставить замість нього відповідний вираз (це можуть бути константи, імена міток, символічні імена та рядки в апострофах).

*Синтаксис псевдооператора **equ**:*

ім'я_ідентифікатора **equ** рядок або числовий_вираз

Синтаксис псевдооператора "=":

ім'я_ідентифікатора = числовий_вираз

Псевдооператор "=" може використовуватися тільки з числовими виразами, тоді як псевдооператори EQU може працювати і з текстовими виразами.

Псевдооператор **equ** зручно використовувати для налаштування програми на конкретні умови виконання, заміни складних в позначенні об'єктів, які багатократно використовуються в програмі, більш простими іменами.

Наприклад:

```
.model   small
stack   256
mas_size  equ  10                    ; розмірність масива
akk   equ  ax                        ; перейменувати регістр
mas_elem   equ   mas[bx][si]         ; адресувати елемент масива
.data
     ; опис масива з 10 байт:
     mas    db        mas_size dup (0)
.code
   mov      akk,  @data              ; фактично mov ax, @data
   mov      ds,   akk                ; фактично mov ds, ax
...
   mov   al,   mas_elem              ; фактично — mov al, mas[bx][si]
```

Псевдооператор “=” зручно використовувати для визначення простих абсолютних математичних виразів.

```
.data
    adr1     db        5 dup (0)
    adr2     dw        0
    len=43
    len=len+1                  ; можна через попередній вираз
    len=adr2-adr1
```

Директива злиття рядків catstr:

Ідентифікатор   catstr рядок_1, рядок_2, ...

значенням цього макроса буде новий рядок, який поєднує зліва направо послідовності рядків **рядок_1, рядок_2,...**

В якості поєднаних рядків можуть бути вказані імена раніше визначених макросів.

Наприклад:

```
pre       equ      Привіт,
name      equ      < Юля>
privet catstr pre, name          ; privet= “Привіт, Юля”
```

Директива виділення підрядка в рядку substr:

ідентифікатор substr рядок, номер_позиції, розмір

значенням цього макроса буде частина вказаного рядка, яка починається з позиції з номером *номер_позиції* та довжиною, що вказана у *розмірі*.

Якщо треба отримати тільки залишок рядка, то треба вказати з якої позиції без позначення розміру.

Наприклад:

```
 ; продовження попереднього фрагмента:
 privet      catstr pre, name            ; privet="Привіт, Юля"
 name        substr privet, 7, 4         ; name="Юля"
```

Існує три варіанти де повинні розташовуватися макровизначення:

1. *На початку тексту програми* до сегмента коду та даних. Цей варіант використовується тоді, коли визначені користувачем макрокоманди є актуальними в межах однієї програми.

2. *В окремому файлі*.

Такий варіант підходить при роботі з декількома програмами однієї проблемної області. Для того, щоб зробити доступними макровизначення у конкретній програмі, слід записати директиву **include ім'я_файл**у.

Наприклад:

```
.model      small
include     show.inc
; в це місце буде вставлено текст файлу show.inc
    ...
```

3. В макробібліотеці.

Універсальні макрокоманди, які часто використовуються в програмах користувача, (наприклад, фрагменти програмної затримки, призупинення програми до натискання клавіші, перетворення двійкових чисел у символьну форму) доцільно записати в макробібліотеку.

Макробібліотека являє собою файл з текстами макровизначень, які записуються у цей файл, як у текст програми. Файл макробібліотеки може мати будь-яке ім'я і розширення, наприклад, MYMACRO.MAC. В програмі залишаються тільки макровиклики. Включати макрокоманди з цієї бібліотеки в програму також можна за допомогою директиви **include** (наприклад**,** *include mymacro.mac*). Після цього у програмі можна використовувати будь-які макрокоманди з цієї макробібліотеки.

Недоліком двох останніх способів є той факт, що у вихідний текст програми включаються абсолютно усі макровизначення. Для усунення цього недоліку використовується директива **purge**, в якості операндів через кому слід перелічити імена макрокоманд, які не повинні включатися в тіло програми.

PURGE ім'я_макроса відміняє визначений раніше макрос (не підтримується WASM). Ця директива часто застосовується відразу після INCLUDE, програми, що включила в текст, файл з великою кількістю готових макровизначень.

Наприклад:

```
 ...
 include  iomac.inc
 purge  _outstr, _exit
         ...
```

В цьому прикладі у вихідний текст програми перед початком компіляції TASM замість рядка **include iomac.inc** вставить рядки з файлу **iomac.inc**, однак у ньому будуть відсутні макровизначення **_outstr** та **_exit**.

Приклади створення та використання макрокоманд

```
;prg_3_1.asm з макровизначеннями
init_ds    macro

; Макрос налаштування ds на сегмент даних
          mov  ax, data
          mov  ds, ax
endm

out_str  macro  str
; Макрос виведення рядка на екран
; На вході — рядок, що виводиться
; На виході – повідомлення на екрані
          push     ax
          mov      ah, 09h
          mov      dx, offset str
          int      21h
          pop      ax
endm

clear_r  macro  rg
; Очистка регістра rg
          xor  rg, rg
endm

get_char   macro
; введення символу
; символ, що  вводиться, знаходиться в al
          mov  ah, 1h
          int   21h
endm

conv_16_2   macro
; макрос перетворення символу шістнадцятирічної цифри
; в її двійковий еквівалент в al
          sub      dl, 30h
          cmp      dl, 9h
          jle      $+5       ; перейти, якщо менше або дорівнює
          sub      dl,7h
endm
```

```
exit  macro
; макрос кінця програми
        mov     ax,4c00h
        int     21h
endm

data    segment para public 'data'
message         db      'Введіть дві шістнадцятирічні цифри (букви A,B,C,D,E,F - великі): $'
data    ends

 stk    segment         stack
        db      256 dup('?')
stk     ends

code    segment para public 'code'
        assume cs:code, ds:data, ss:stk
main:           ; точка входу у програму
main    proc
        init_ds
        out_str  message

        clear_r  ax
        get_char
        mov     dl, al
        conv_16_2
        mov     cl, 4h
        shl     dl, cl  ;  логічний зсув dl на 4 біти вліво
        get_char
        conv_16_2
        add     dl, al
        xchg    dl, al  ; результат в al
        exit
main    endp
code    ends
end    main            ; кінець програми з точкою входу main
```

Приклад макровизначення (багатократно у стеку треба зберігати вміст трьох регістрів, але для кожного конкретного випадку імена та їх порядок відрізняються:

```
Push3 macro a, b, c
        Push a
        Push b
        Push c
Endm
```

Поява у вхідному тексті команди push3 DX, ES, BP призведе до генерації фрагмента:

```
Push DX
Push ES
Push BP
```

Виклик макровизначення з параметрами:

```
s_mov   macro   register1,register2
        push    register1
        pop     register2
        endm
```

Для організації циклу у макровизначенні мітку треба оголосити за допомогою оператора **local**

LOCAL мітка... перераховує мітки, які застосовуватимуться усередині макроозначення, щоб не виникало помилки «мітка вже визначена» при використанні макросу більше одного разу або якщо та ж мітка присутня в основному тексті програми (у WASM директива LOCAL дозволяє використовувати макрос з мітками кілька разів, але не дозволяє застосовувати мітку з тим же ім'ям в програмі). Операнд для LOCAL мітка або список міток, які використовуватимуться в макросі.

```
;   виведення на екран рядка з 10 зірочок, який може слугувати
;   розділювачем у фрагменті тексту в екранному кадрі
starts      macro
local       outpt
        mov  CX, 10          ; лічильник циклу
        mov  AH,  02h    ;  функція виведення символу
        mov  DL, ‘*’         ;  символ *
outpt:      int   21h        ;  виклик DOS
        loop   outpt
```

В тексті програми можна включати декілька макровикликів:

```
……………….
        starts
……………….
        starts
……………….
```

Приклад вкладених макросів. У попередньому прикладі зірочки виводяться не з нового рядка, а від поточного положення курсору, тому доцільно включити в макрос до та після виведення зірочок переведення курсора на початок наступного рядка екрану. Для цього слід записати макрос:

```
crlf    macro
        mov  AH, 02h       ;  функція виведення символу
        mov  DL,  13               ;   код повернення рядка
        int      21h           ;   виклик DOS
        mov   DL,  10      ;   код переведення рядка
        int      21h           ;   виклик DOS
endm
```

Тепер макровизначення stars можна модифікувати таки чином:

```
stars   macro
local   outpt
crlf                  ; вкладений макровиклик
mov  CX,  10          ;  лічильник циклу
mov   AH,  02h        ;  функція виклику символу
mov   DL,  ‘*’        ;  символ
outpt: int  21h              ;  виклик DOS
loop outpt            ;  цикл з CX кроків
crlf                  ;  вкладений макровиклик
endm
```

Наступний важливий засіб, що використовується в макровизначеннях, *директиви умовного асемблювання.* Наприклад: напишемо макрос, що виконує множення регістра *AX* на число, причому, якщо множник ступінь двійки, то множення виконуватиметься швидшою командою зрушення вліво.

```
fast_mul   macro      number
    if        number eq 2
        shl      ax, 1            ; Множення на 2, зсув вліво на біт
    elseif     number eq 4
        shl      ax, 2            ; Множення на 4, зсув вліво на 2 біти
    elseif     number eq 8
        shl      ax, 3                      ; Множення на 8
          ...             ; Аналогічно аж до:
    elseif     number eq 32768
        shl      ax, 15           ; Множення на 32768
    else
          mov      dx, number             ; Множення на число, що не є
          mul      dx                     ; ступенем двійки.
    endif
    endm
```

Цей макрос можна записати за допомогою засобу, що постійно використовується в макросах, блоків повторень.

Блоки повторень

Простий блок повторень **REPT** (не підтримується WASM) виконує асемблювання ділянки програми задане число раз. Наприклад, якщо потрібно створити масив байтів, що проініціалізував значеннями від 0 до 0FFh, це можна зробити шляхом повтору псевдокоманди DB таким чином:

```
mas0_256   macro
hexnumber=0
hextable       label   byte            ; Ім'я масиву
        rept    256                    ; Початок блоку
        db      hexnumber              ; Ці два рядки асемблюються
        hexnumber=hexnumber+1          ; 256 разів.
endm
```

Блоки повторень, так само як макроозначення, можуть викликатися з параметрами. Для цього використовуються директиви **IRP** і **IRPC**:

```
irp       параметр,<значение1,значение2...>
   ...
endm

irpc      параметр, рядок
   ...
endm
```

Блок, описаний директивою *IRP*, викликатиметься стільки раз, скільки значень вказано в списку (у кутових дужках), і при кожному повторенні буде визначена мітка з ім'ям *параметр*, рівна черговому значенню із списку. Наприклад, наступний блок повторень збереже в стек регістри AX, BX, CX і DX:

```
irp       reg, <ax, bx, cx, dx>
   push     reg
endm
```

Директива IRPC (FORC в WASM) описує блок, який виконується стільки раз, скільки символів містить вказаний рядок, і при кожному повторенні буде визначена мітка з ім'ям *параметр*, рівна черговому символу з рядка. Якщо рядок містить пропуски або інші символи, відмінні від дозволених для міток, вона повинна бути поміщена в кутові дужки.

Наприклад:

```
irp   ini, <1, 2, 3, 4, 5>
      db  ini
endm
```

Макрогенератором буде сгенеровано наступне макророзширення:

```
db 1
db 2
db 3
db 4
```

Наприклад, наступний блок задає рядок в пам'яті, розташовуючи після кожного символу рядки атрибут 0Fh (білий символ на чорному фоні), так що цей рядок згодом можна буде скопіювати прямо у відеопам'ять.

```
irpc     character, <рядок символів>
    db       ’&character&’, 0Fh
 endm
```

В даному прикладі використовуються амперсанди, щоб замість параметра character було підставлено його значення навіть усередині лапок.

Амперсанд це один з макрооператорів спеціальних операторів, які діють тільки усередині макроозначень і блоків повторень.

Макрооператори

Макрооператор & (амперсанд) потрібен для того, щоб параметр, переданий як операнд макровизначенню або блоку повторень, замінювався значенням до обробки рядка асемблером. Так, наприклад, наступний макрос виконає команду PUSH EAX, якщо його викликати як PUSHREG A:

```
pushreg   macro    letter
      push     e&letter&x
endm
```

Іноді можна використовувати тільки один амперсанд на початку параметра, якщо не виникає неоднозначностей. Наприклад, якщо передається номер, а потрібно створити набір змінних з іменами, що закінчуються цим номером:

```
irp      number, <1, 2, 3, 4>
      msg&number db       ?
endm
```

Макрооператор <> (кутові дужки) діє так, що весь текст в дужках розглядається як текстовий рядок, навіть якщо він містить пропуски або інші роздільники. Цей макрооператор використовується при передачі текстових рядків як параметри для макросів. Інше часте застосування кутових дужок - передача списку параметрів вкладеному макровизначенню або блоку повторень.

Макрооператор ! (знак оклику) використовується аналогічно кутовим дужкам, але діє тільки на один наступний символ, так що, якщо цей символ кома або кутова дужка, він все одно буде переданий макросу як частина параметра.

Макрооператор % (відсоток) вказує, що текст, що знаходиться за ним, є виразом і повинен бути обчислений. Зазвичай це потрібно для того, щоб передавати як параметр в макрос не сам вираз, а його результат.

Макрооператор ;; (дві крапки з комою) - початок макрокоментаря. На відміну від звичайних коментарів текст макрокоментаря не потрапляє в лістинг і в текст програми при підстановці макросу. Це заощадить пам'ять при асемблюванні програми з великою кількістю макроозначень.

Директива ***EXITM*** (не підтримується WASM) виконує передчасний вихід з макроозначення або блоку повторень.

Наприклад, наступне макроозначення не виконує ніяких дій, тобто не буде розширено в команди процесора, якщо параметр не вказаний:

```
pushreg     macro     reg
        ifb     <reg>
        exitm
        endif
          push     reg
        endm
```

Контрольні питання:

1. Призначення макрозасобів Асемблера.
2. Які складові макрозасобів Асемблера?
3. Що таке макровизначення?
4. Що таке макрокоманда?
5. Що таке макророзширення?
6. Які бувають макровизначення?
7. Що таке псевдооператори?
8. Синтаксис директиви злиття рядків.
9. Синтаксис директиви виділення підрядка в рядку.
10. Наведіть варіанти розташування макровизначення.
11. Як повинна оголошуватися мітка в макровизначення для організації циклу?
12. Призначення блоків повторень.
13. Призначення макрооператора.

Лекція 15. Резидентні програми MS DOS

Специфіка резидентних програм

TSR - програмами (*Terminate and Stay Resident*). Передача управління резидентній програмі. Переривання IRET (*Interrupt RETurn*). Структура резидентної програми. Приклади резидентних програм.

Багато програм, що забезпечують функціонування обчислювальної системи (драйвери пристроїв, програми стиснення або шифрування даних, русифікатори, інтерактивні довідники та ін.), повинні постійно перебувати в пам'яті, швидко реагувати на запити або якісь події, що відбуваються в обчислювальній системі. Такі програми називаються резидентними.

Резидентна програма - це програма, що постійно знаходиться в оперативній пам'яті ЕОМ. Інакше такі програми називають ***TSR- програмами (Terminate and Stay Resident)*** [1, 5]**.**

Резидентна програма може бути як типу .COM, так і .EXE, однак, з огляду на постійний дефіцит основної пам'яті, такі програми частіше виконують типу .COM. Будь-яку програму можна зробити резидентною, але найчастіше - це * COM програма.

Для того щоб використовувати програму, яка вже знаходиться в пам'яті, їй необхідно передати управління. Специфіка передачі управління резидентній програмі полягає в тому, що програма, яка викликає, і програма, яку викликають (резидентна) завантажуються і запускаються незалежно одна від одної, тому необхідні спеціальні заходи для того, щоб повідомити адресу резидентної програми програмі, яка її викликає.

Передати управління резидентної програмі можна трьома способами:

- Викликати її командою CALL як звичайну процедуру (підпрограму). Однак для цього необхідно після завантаження резидентної програми дізнатися її розташування в пам'яті за допомогою будь-якої службової програми, наприклад, ***mi*** *(memory information);*
- Використовувати будь-яке апаратне переривання (наприклад, переривання від таймера) для періодичної передачі управління резидентній програмі;
- Використовувати програмне переривання. Для цього резидентна програма повинна відповідним чином встановити вектор програмного переривання, який буде використаний для її виклику.

Для користувача в MS-DOS зарезервовані вектори **60h - 66h**, а також **F1h - FFh**.

В цьому випадку резидентна програма повинна завершуватися командою повернення з переривання **IRET (Interrupt RETurn**) - Повернення з переривання - *iretd*.

Призначення: використовується в тій точці програми обробки переривання, звідки необхідно повернути управління перерваній програмі.

Алгоритм роботи:

Робота команди залежить від режиму роботи мікропроцесора:

- в реальному режимі команда **iret** послідовно витягує зі стеку і потім відновлює в мікропроцесорі вміст наступних регістрів: **eip/ip, cs, eflags/flags**. Далі перервана програма продовжується з точки переривання;
- в захищеному режимі дії команди залежать від стану прапорця *NT* (вкладеного завдання) в регістрі прапорців:

якщо NT=0, то виробляються дії по поверненню управління перерваної програми, при цьому характер цих дій залежить від співвідношення рівнів привілейованості перерваної програми і програми обробки переривання;

в разі NT=1 виробляються дії по переключенню завдань.

Команду *iret* необхідно застосовувати для відновлення збережених командою *int* регістрів прапорців, вказівника команд і сегментного регістра коду. Число цих команд в програмі обробки переривання повинна відповідати кількості точок виходу з неї. Команда *iretd* використовується в старших моделях мікропроцесорів для вилучення зі стека і відновлення 32-бітових регістрів.

Адресу резидентної програми можна передати прикладній програмі також в області даних BIOS, призначеної для зв'язку програм (40h: F0h - 40h: FFh). У цій же області прикладна програма може передавати адреси масивів даних, які повинні бути передані резидентній програмі, а також отримувати адреси масивів даних, які повертаються резидентній програмі.

Резидентна програма після завантаження її в пам'ять фактично стає частиною операційної системи, тому до неї відноситься і така властивість MS-DOS, як нереєнтерабельність (тобто вона не має властивості повторної входимості). Це пов'язано з тим, що MS-DOS розроблялася, як однозадачна операційна система, і в ній використовуються внутрішні робочі області, які можуть бути зіпсовані при спробі паралельного виконання декількох процесів. Практичним наслідком цієї властивості є той факт, що резидентна програма не може використовувати велику частину функцій MS-DOS і BIOS. Ці функції може використовувати ініціалізуюча частина резидентної програми, так як в момент завантаження резидентна програма ще не є частиною операційної системи.

Після першого завантаження резидентної програми в пам'ять необхідно заборонити всі наступні подібні спроби, так як перезавантаження може призвести до більш-менш великих неприємностей. Стежити за цим повинна сама резидентна програма.

Структура резидентної програми

Резидентна програма складається, як правило, з двох частин (рис. 15.1):

- резидентної секції (яка зазвичай розташовується спочатку);
- ініціалізуючої (яка зазвичай розташована в кінці).

При першому запуску резидентна програма завантажується в пам'ять цілком, і управління передається ініціалізуючій секції, яка перевіряє, чи не перебуває вже резидентна секція цієї програми в пам'яті. Якщо така програма вже присутня, виводиться відповідне повідомлення і подальше виконання програми припиняється без наслідків.

Якщо такої програми немає в пам'яті, виконуються наступні дії:

- налаштовуються всі необхідні вектори переривань (при цьому можуть встановлюватися нові вектори і модифікуватися старі);
- якщо необхідно, заповнюються всі області вказівників адрес передачі управління і даних;
- програма налаштовується на конкретні умови роботи (можливо задані в командному рядку при запуску резидентної програми);
- завершується виконання ініціалізуючої частини за допомогою функції *31h* переривання DOS *int 21h* або за допомогою переривання DOS *int 27h*. При цьому резидентна секція програми, розмір якої ініціалізуюча секція передає DOS, залишається в пам'яті.

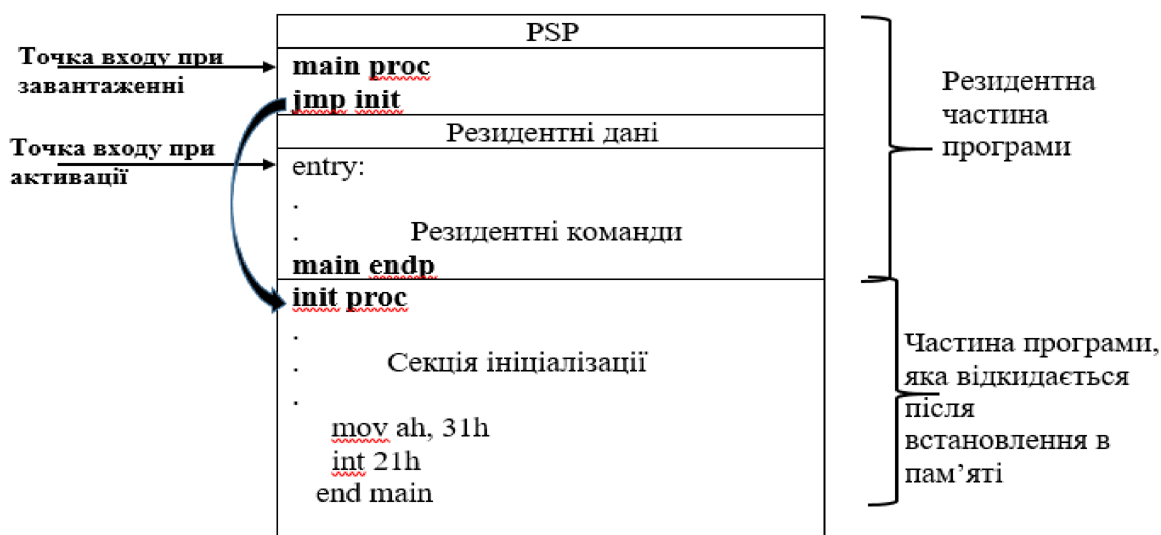


Рис. 15.1. Структура резидентної програми

Слід зазначити, що *найважливішою функцією ініціалізуючої секції резидентної програми є вказівка операційній системі* DOS *розміру залишеної резидентної секції програми*. Якщо для завершення ініціалізуючої секції використовується переривання DOS *int 27h,* в регістрі *dx* вказується розмір резидентної секції в байтах. При цьому слід мати на увазі, що в цей розмір входять також 100h байтів префікса програмного сегмента *PSP*. Зрозуміло, що за допомогою цього переривання DOS не можна залишити в пам'яті програму, більше 64 кілобайт. Якщо для завершення ініціалізуючої секції використовується функція *31h* переривання DOS *int 21h*, в регістрі dx вказується розмір резидентної секції (з урахуванням *PSP*) в параграфах (1 параграф=16 байтам).

Для визначення розміру резидентної секції в параграфах обчислюється вираз (*size+100h+0Fh)/16* де:

size - розмір резидентної секції в байтах.

Додатковий доданок 0Fh (десяткове 15) у натуральному вираженні необхідно для того, щоб кількість параграфів, які відводяться, було округлено в більшу сторону. В іншому випадку буде відтятий кінець програми, менший параграфа.

Раніше вже було сказано про те, що ініціалізуюча секція розташовується в кінці програми. Таке розташування призводить до того, що після завершення ініціалізуючої секції займана ним пам'ять звільниться, так як вона не входить в зазначений розмір і розташована після резидентної секції.

Приклад структури резидентної програми типу .COM

```
Code SEGMENT
assume cs:Code, ds:Code
org 100h
------------------------------------------------------
main proc far
      jmp init ; Перехід на секцію ініціалізації
      ; Дані та змінні резидентної секції
      ..................
      entry: ; Текст резидентної секції
      ..................
main endp
------------------------------------------------------
size equ $-main ; Розмір резидентної секції в байтах
------------------------------------------------------
init proc near ; Ініціалізуюча секція
     ; Текст ініціалізуючої секції
     ; Обчислення (size + 10Fh)/16 -> DX    або (init-main+10Fh)/16
     mov DX, (size+10F)/16       ; розмір в параграфах
     mov ax, 3100h            ; Функція 31h – завершити і залишити у пам’яті
     int 21h
init endp
------------------------------------------------------
Code ends
END main
```

Пояснення до прикладу структури резидентної програми:

1. Передбачається, що великі та малі літери транслятором не розрізняються (за замовчуванням так і є).

2. Процедура main оголошена як дальня, так як в ній знаходиться текст резидентної секції, управління якою може передаватися тільки за допомогою дальнього переходу або виклику.

3. У тексті резидентної секції повинна бути передбачена команда повернення у викликаючу прикладну програму. Це може бути команда IRET, якщо резидентна програма викликається за допомогою програмного переривання *int*, це може бути просто RET, якщо резидентна програма викликається, як підпрограма командою CALL, це може бути і що-небудь більш екзотичне - фантазії програмістів немає меж.

4. Процедура init оголошена як ближня, так як викликаюча процедура знаходиться в тому ж сегменті.

Взаємодія елементів резидентної програми

Будь-яка резидентна програма має, принаймні, дві точки входу. При запуску з клавіатури *.COM програми управління завжди передається на перший байт після PSP (IP=100h). Тому практично завжди першою командою резидентної програми є команда JMP, яка передає управління на початок секції ініціалізації.

Після обробки функції DOS 31h програма залишається в пам'яті в пасивному стані. Для того, щоб активізувати резидентну програму, їй потрібно передати управління. Найбільш часто для цього використовується механізм апаратних чи програмних переривань. У цьому випадку в процесі ініціалізації необхідно заповнити відповідний вектор адресою точки входу в програму (*entry*). Адреса *entry* утворює другу точку входу в програму, через яку здійснюється її активізація. Резидентна секція повинна закінчуватися командою виходу з переривань - *iret*.

Звернення до резидентної програмі

Для звернення до резидентної програмі можна використовувати область даних BIOS, призначену для зв'язку між процесами (40h:F0h - 40h:FFh). Ця область не використовується операційною системою, тому використання її для виклику резидентної програми начебто не віщує нічого несподіваного.

Так воно і є, якщо розробник однієї (або не однієї) резидентної програми, що вже перебуває в пам'яті, не використав ту ж область для тих же цілей. (До речі, це відноситься і до всіх інших способів звернення до резидентної програмі.) Ми не будемо розглядати таку можливість, хоча і в цьому випадку є простір для творчості.

Отже, є область розміром 16 байтів, яка може бути використана за бажанням. Так як повна адреса, необхідна для дальнього виклику або переходу вимагає чотирьох байтів, в цій області можна розмістити 4 таких адреси. Це може бути адреса входу в резидентну секцію і 3 адреси, що вказують на таблиці даних, розташованих десь ще в пам'яті. Можна використовувати цю область таким чином - адреса точки входу в резидентну секцію (4 байти) і 12 байтів безпосередніх даних. Можуть бути різні проміжні варіанти.

Розглянемо варіант з двома адресами - адресою точки входу в резидентну секцію і адресою таблиці параметрів (*tabl_param*) в сегменті даних прикладної програми, яка повинна бути передана резидентній програмі.

Для забезпечення взаємодії ініціалізуюча секція резидентної програми записує в слово 40h:F0h - зміщення точки входу в резидентну секцію (наприклад, **offset entry**), в слово 40h:F2h - вміст сегментного регістра CS.

Прикладна програма для виклику резидентної програми повинна, наприклад, налаштувати сегмент розширення на початок області даних BIOS (ES=40h) і виконати команду дальнього виклику **call dword ptr es:0F0h**

Зазвичай, резидентна програма повинна бути оголошена дальньою процедурою і завершуватися відповідною командою дальнього повернення RET (втім, її можна явно зробити дальньою - RETF).

Для передачі резидентній програмі адреси таблиці параметрів прикладна програма повинна записати в слово **40h:F4h** - зсув початку таблиці параметрів в

сегменті даних прикладної програми (**offset tabl_param**), а в слово **40h:F6h** - поточний вміст сегментного регістру DS.

Резидентна програма для отримання цих даних повинна помістити в будь-який регістр, наприклад **SI**, зсув початку таблиці з **40h:F4h**, а в сегментний регістр, наприклад в **DS**, сегментну адресу з **40h:F6h**, після чого резидентна програма отримує доступ до самих даних.

Послідовність команд в резидентній програмі може бути такою:

```
mov es, 40h         ; ES на початок області даних BIOS
mov bx, 0F4h
mov si, es:[bx]     ; SI=offset tabl_param
mov bx, 0F6h
mov ax, es:[bx] ; AX – сегм. адрес tabl_param
mov ds, ax
mov ax, [si]        ; Перше слово даних
mov bx ,[si+2]      ; Друге слово даних і т.п.
```

Не слід забувати зберігати в резидентній програмі усі регістри, які використовуються нею, і відновлювати їх перед виходом з програми. При цьому слід обережно користуватися стеком для збереження регістрів, так як системний стек не дуже великий, а створювати власний стек в резидентній програмі не завжди доцільно. Можна зберігати регістри в спеціально відведених для цього робочих змінних.

Більш зручно використовувати для виклику резидентної програми один з вільних векторів переривання (вектори 60h - 66h, а також F1h - FFh). Ініціалізуюча секція резидентної програми повинна помістити свою адресу в один з вільних векторів, наприклад, F1h:

```
mov ax,  0
mov es,  ax
mov es:0F1h*4,  offset entry          ;   Адреса вектора F1h
mov es:0F1h*4+2,  cs
```

В результаті цієї послідовності команд у векторі *F1h* виявиться адреса точки входу в резидентну програму. Для виклику резидентної програми в цьому випадку досить використовувати команду *int 0F1h*. В цьому випадку резидентна програма, як і всі програми обробки, повинна завершуватися командою повернення з переривання *IRET*. Адреси таблиць параметрів можна передавати у попередній спосіб, а можна і через інші вільні вектори переривань.

Захист від повторного завантаження

Для захисту від повторного завантаження резидентної програми в пам'ять ініціалізуюча секція повинна зробити деякі дії по виявленню власної резидентної

секції в пам'яті, а резидентна секція повинна відповідним чином відповісти на ці дії.

Для здійснення цих дій можна використовувати мультиплексне переривання DOS **int 2Fh.**

Функції **C0h - FFh** цього переривання зарезервовані для користувача.

В DOS прийнято, що переривання 2Fh повертає в регістрі AL наступні стани резидентної програми:

0 - програма не встановлена, але її можна встановити;

1 - програма не встановлена, і її не можна встановити;

FFh - програма встановлена.

При помилці повинен бути встановлений прапорець переносу CF, а в регістрі AX слід повернути код помилки. Для того щоб резидентна секція програми реагувала на переривання 2Fh, в неї слід включити обробник відповідних функцій цього переривання. Для нормальної роботи цього обробника ініціалізуюча секція повинна встановити новий вектор переривання 2Fh, зберігши при цьому старий вектор у внутрішній змінній. Новий обробник переривання 2Fh повинен виконати все, що йому належить, а після цього викликати старий обробник цього переривання. У наведеному нижче прикладі резидентної програми використаний саме цей спосіб захисту від повторного завантаження.

Іншим способом захисту від повторного завантаження є використання спеціального коду для індикації наявності резидентної програми в пам'яті. Спеціальний ідентифікуючий код поміщається у заздалегідь визначене місце в пам'яті або у заздалегідь визначене місце в резидентній секції програми. Якщо код поміщається у певне місце в пам'яті (наприклад, на місці вектора переривання 60h), при ініціалізації перевіряється наявність цього коду в цьому місці. Якщо є код в наявності, завантаження програми не проводиться.

Якщо ідентифікуючий код (сигнатура) поміщається у певному місці резидентної секції, ініціалізуюча секція перевіряє наявність цього коду за адресою точки входу в програму (вона знає, як визначити адресу точки входу) і по положенню цього коду відносно точки входу (це вона теж знає). Виявлення коду тягне за собою відмову від завантаження програми.

Приклади резидентних програм

1. Написати резидентну програму, яка перехоплює переривання ***int 5 (Print Screen)*** *і замість роздруківки екрану на кожне натискання клавіші* ***PrtSc*** *змінює колір рамки екрану. Рамка повинна приймати циклічно один з 16 кольорів. Програма не повинна дозволяти завантажити себе повторно. При спробі повторного завантаження програма повинна виводити попередження.*

ВКАЗІВКИ:

- Передавати управління старому обробникові переривання не треба.
- Для фарбування рамки екрану слід використовувати під функцію 01h функції 10h переривання 10h (ax=1001h, bx=колір).
- На початку програми слід не забути записати в DS значення CS.

• Для перевірки наявності резидентної програми в пам'яті використовувати функцію FFh переривання 2Fh.

```
Assume CS: Code, DS: Code
Code SEGMENT
org 100h
resprog proc far
        mov ax, cs
        mov ds, ax
        jmp init
color db 0
old_int2Fh_off  dw ?
old_int2Fh_seg  dw ?
msg db 'Драйвер вже встановлений $'
; Новий обробник переривання 2Fh
new_int2Fh proc far
      cmp ax,  0ff00h
      jz installed
      jmp  dword  ptr  cs:old_int2Fh_off
      installed:    mov ax,   0ffh
      iret
new_int2Fh endp
; Новий обробник переривання 5
new_int5 proc far
      mov bh, color
      inc color
      mov ax,  1001h
      int 10h
      iret
new_int5 endp
resprog endp
ressize   equ $-resprog ;   Розмір в байтах резидентної частини
init proc near
      ; Перевірка наявності резидентної програми в пам’яті
      mov ax,  0ff00h
      int 2fh
      cmp ax,  0ffh
      jnz first_start           ; Не встановлена
      lea dx,  msg              ; Виведення повідомлення про те,
      mov ah,  09 h             ; що драйвер вже завантажений
      int 21h
      ret
```

```
        first_start:    mov ax,    2505h        ; Функція 25h, вектор 5
        lea dx,   new_int5
        int 21h                                 ; Запис нового вектора 5
        mov ax,    352fh                ; Збереження старого вектора переривання
2Fh
        int 21h
        mov cs:old_int2fh_off,   bx
        mov cs:old_int2fh_seg,   es
        lea dx,new_int2Fh                       ; Запис нового вектора переривання 2Fh
        mov ax,    252fh
        int 21h
        ; Завершення програми, залишаючи резидентну частину в пам’яті
        mov dx,    (ressize+10fh)/16
        mov ax,     3100h
        int 21h
    init endp
  Code ENDS
  END resprog
```

2. Написати резидентну програму, що записує вміст екрану у символьному режимі у файл. Програма повинна аналізувати прапорець активності DOS і не повинна допускати повторного завантаження в пам'ять.

*За ключем /**u** програма повинна розвантажуватися з пам'яті зі звільненням займаного нею місця. Зауваження: наведена нижче програма нормально працює лише під DOS до версії 5.0, так як в більш пізніх версіях інакше відбувається робота з клавіатурою.*

```
    Code SEGMENT
    Assume CS: Code, DS: Code
    org 100h
    resprog proc far
        mov ax, cs
        mov ds, ax ;    DS = CS
        jmp init ;        Перехід на ініціалізуючу секцію

    num dw 0                    ; Кількість  скинутих екранів
    old_int8_off dw ?            ; Адреса старого обробника
    old_int8_seg dw ? ; переривання таймера 8h
    old_int5_off dw ?           ; Адрес старого обробника
    old_int5_seg dw ? ; переривання 5h
    old_int2F_off dw ?          ; Адрес старого обробника
    old_int2F_seg dw ?          ; мультиплексне переривання 2Fh
```

```
        adr_psp dw ?                ; Адреса PSP
        vbuf  dw  0b000h            ; Сегментна адреса відеобуфера
        handle dw ?                 ; Дескриптор файла
        buf db 2050 dup(0) ;        Буфер для даних екрана
        mes  db   'Disk error$'
        filename db   'filescr&.txt',0      ; Специфікація кінцевого файла
        iniflag db 0                        ; Прапорець запиту на виведення екрана у
файл
        outflag db 0                ; Прапорець початку виведення у файл
        _crit d ?                           ; Вказівник на прапорець активності DOS
    ; Новий обробник переривання 2Fh
      new_int2F proc far
          cmp ax,  0ff00h
          jz installed
          jmp dword  ptr  cs:old_int2F_off            ; Перехід на старий обробник 2Fh
          installed:    mov ax,    0ffh               ; "Програма в пам’яті"
          iret
      new_int2F endp


    ; Новий обробник переривання 1ch
    new_int8 proc far
          push ax
          push bx
          push cx
          push dx
          push si
          push di
          push ds
          push es
          mov ax,  cs
          mov ds,  ax
          cmp iniflag,  0
          jz exit8                  ; Немає запиту
          test outflag,   0ffh
          jnz exit8                 ; Файл вже виводиться
          jnz exit8                 ; DOS зайнята
          les bx, _crit             ; Завантаження вказівника з використанням регістра
ES
          test byte ptr es:[bx],  0ffh
          jnz exit8                 ; DOS зайнята
          ; iniflag=1, outflag=0, crit=0
```

```
        mov outflag,  0ffh

        call writef                    ; Виведення буфера у файл
        exit9: pop es
        pop ds
        pop di
        pop si
        pop dx
        pop cx
        pop bx
        pop ax
        iret
    new_int8 endp
    ; Новий обробник переривання 5
    new_int5 proc far
        mov cs:iniflag,  0ffh
        iret
    new_int5 endp
; Запис відеобуфера у файл
    writef proc near
        mov ax,  cs
        mov ds,  ax
        mov ax,  0e07h
        int 10h
        mov ax,  vbuf                  ; Початок відеобуфера (сегмент)
        mov es,   ax
        mov si, 0
        lea di,  buf
        mov dx,  25                    ; Число рядків
        cld
        trans1: mov cx,  80            ; Число символів у рядку
        trans: mov al,  es:[si]
        mov [di],   al
        inc si
        inc si
        inc di
        loop trans
        mov byte ptr [di],  0dh
        inc di
        mov byte ptr [di],  0ah
        inc di
        dec dx
```

```
  jnz trans1
; Створення файла
test word ptr num,  0ffffh          ; Скинуто екранів 0 ?
jnz sdwig                           ; Перехід на зсув вказівника
mov word ptr num,  2050
mov ah,  3ch                        ; Функція створення файла

mov cx,  0                          ; Без атрибутів
lea dx,  filename                   ; Адреса специфікації файла
int 21h
jc noform
mov handle,  ax                     ; Створення дескриптора файла
jmp write
sdwig:     mov ax,    3d01h         ; Відкриття файлу з записом
lea dx,  filename
int 21h
jc noform
mov handle,   ax
mov ax,   4200h                     ; Установка вказівника файла
mov bx,   handle
mov cx,   0
mov dx,   num
add word ptr num,   2050
int 21h
 ; Запис файла
  write:    mov ah,  40h            ; Функція запису в файл
  mov bx   ,handle                  ; Дескриптор файла
  mov cx,   2050                    ; Довжина масиву, що записується
  lea dx  ,buf                      ; Адреса масиву, що записується
  int 21h
  jc noform
 ;Закриття файла
  mov ah,   3eh                     ;Функція закриття файла
  mov bx,   handle                  ; Дескриптор файла
  int 21h
  jmp prend
  noform:   mov ah,   9
            mov dx,    offset mes
            int 21h
  prend:    mov outflag     ,0
  mov iniflag,       0
  ret
```

```
writef endp
resprog endp
    ressize equ $-resprog              ; Розмір в байтах резидентної частини
init proc near
    ; Перевірка ключа /u
    mov bx,   80h
    mov cx,    [bx]                    ; Кількість символів у командному рядку
    inc bx                             ; Початок командного рядка
    cmd: mov al,    [bx]
    cmp al,    20h
    jz cmd1                            ; Пропуск
    cmp al   ,'/'
    jnz cmd2                           ; Не ключ
    cmp byte ptr [bx+1] ,        'u'
    jnz cmd2                           ; Не u
    ; Звільнення блока пам’яті
    ; Перевірка завантаженості
    mov ax,   0ff00h
    int 2fh
    cmp ax,    0ffh
    jz uninst                          ; Програма в пам’яті
    lea dx,   msgno                    ; Виведення повідомлення про те,
    mov ah,  9                         ; що програми немає в пам’яті
    int 21h
    int 20h
    uninst:          call set_int      ; Відновлення  векторів переривань
    int 20h
    cmd1:   inc bx
    loop cmd
    cmd2:     mov ax,    0ff00h        ; Перевірка завантаженості
    int 2fh
    cmp ax,   0ffh
    jnz first_start                    ; Не встановлена
    lea dx,  msg                       ; Виведення повідомлення про те,
    mov ah,   9                        ; що драйвер вже завантажений
    int 21h
    int 20h
    first_start:   mov ax,   3505h     ; Створення старого вектора переривань 5
    int 21h
    mov cs:old_int5_off,  bx
    mov cs:old_int5_seg,  es
    mov ax,    2505h                   ; Функція 25h, вектор 5
```

```
        lea dx,  new_int5
        int 21h                          ; Запис нового вектора 5
        mov ax,   352fh                  ; Збереження старого вектора переривань 2Fh
        int 21h
        mov cs:old_int2F_off,   bx
        mov cs:old_int2F_seg,   es
        lea dx,     new_int2F            ; Запис нового вектора переривань 2Fh
        mov ax,  252fh
        int 21h
        mov ax,    351ch                 ; Збереження старого вектора переривань 8
        int 21h
        mov cs:old_int8_off,    bx
        mov cs:old_int8_seg,     es
        lea dx,    new_int8          ; Запис нового вектора переривання 8
        mov ax,    251ch

        int 21h
        mov ah,    34h               ; Запис вказівника на прапорець критичної секції
DOS
        int 21h
        mov word ptr _crit,   bx
        mov word ptr _crit[2],   es
    ; Визначення адреси відеобуфера
        mov ah,  0fh                 ; Функція отримання відеорежима
        int 10h
        cmp al,  7
        jz ini1
        mov vbuf,   0b800h
        ini1: lea dx,   msg2
        mov ah,   9
        int 21h
  ; Завершення програми, залишаючи резидентну частину в пам’яті
        mov dx,   (ressize+10fh)/16
        mov ax,   3100h
        int 21h
  init endp
  set_int proc near
        mov ax,    3505h
        int 21h                          ; ES – сегментна адреса PSP резидента
        mov adr_psp,   es
; Відновлення старого вектора 2Fh
        push ds
```

```
        mov dx,  es:old_int2F_off
        mov ax,  es:old_int2F_seg
        mov ds,  ax
        mov ax,   252fh                    ; Установка старого вектора 2Fh
        int 21h
        mov dx,   es:old_int8_off
        mov ax,   es:old_int8_seg
        mov ds,   ax
        mov ax,   251ch                    ; Установка старого вектора 8
        int 21h
        mov dx,    es:old_int5_off
        mov ax,    es:old_int5_seg
        mov ds,    ax
        mov ax,    2505h                   ; Установка старого вектора 5
        int 21h
        pop ds
        mov ah,  9
        lea dx,   msg1
        int 21h
        mov es,    adr_psp
        mov ah,    49h                     ; Звільнення пам’яті
        int 21h
        ret
    set_int endp
msg db 0dh,  0ah, 'Програма вже в пам’яті',0dh,0ah,'$'
msgno db 0dh,  0ah,' Програми немає в пам’яті',0dh,0ah,'$'
msg1 db 0dh,  0ah, 'Програма вивантажена, 0dh, 0ah, '$'
msg2 db 0dh,  0ah
db 'Програма для запису вмісту символьного', 0dh, 0ah
db 'екрана у файл FILESCR&.TXT.', 0dh, 0ah
db 'ALESOFT (C) Roshin A. 1994.', 0dh, 0ah
db 'Для копіювання натисніть PrtSc.', 0dh, 0ah
db У файл можна записати не більше 32 екранів.',0dh, 0ah
db 'Для вивантаження програми слід набрати'
db ' filescr /u', 0dh, 0ah
db 0dh, 0ah,' $'
Code ENDS
END resprog
```

Контрольні питання:

1. Що таке резидентна програма?
2. Який тип файлу повинна мати резидентна програма?
3. Наведіть способи передачі управління резидентній програмі?
4. Які зарезервовані вектори використовуються для користувача?
5. Наведіть алгоритм роботи команди повернення з переривання?
6. Після завантаження в пам'ять резидентної програми частиною чого вона стає?
7. Наведіть складові резидентної програми.
8. Що входить до резидентної частини програми?
9. Що таке секція ініціалізація?
10. Скільки точок входу має резидентна програма?
11. Яка область даних BIOS використовується для звернення до резидентної програми?
12. В який діапазон області пам'яті необхідно записати адресу таблиці параметрів прикладної програми для передачі її резидентній програмі?

Тема 9. Підсистема управління зовнішніми пристроями
Лекція 16. Організація операцій введення зовнішніми пристроями у середовищі MS DOS

Введення з клавіатури засобами файлової функції. Введення з клавіатури засобами DOS (відповідні функції).
Основні функції для роботи у текстовому режимі, робота з відеосторінками. Робота у графічному режимі засобами BIOS. Керування зображенням курсору на екрані. Типові операції: установка курсору в позиції, посимвольне виведення на екран, виведення рядків на дисплей.

Для організації операцій введення-виведення введено поняття **переривання** (**interrupt**) та створена **підпрограма обробки переривань** (ISR – **interrupt service routine**), яка аналізує подію, що обробляє операційна система. Процесор, отримавши від операційної системи спеціальний сигнал (сигнал переривань), призупиняє виконання поточної команди та передає управління операційній системі. Операційна система визначає яка відбулася подія (чи натиснута клавіша на клавіатурі або на мишці, чи попав символ в область відеопам'яті дисплея та ін.) та реагує на цю подію, викликавши відповідну підпрограму обробки переривань. Виклик потрібного переривання відбувається по команді **INT** (*INTerrupt* - переривання). Існує багато різних вимог для вказівки операційній системі яку дію треба виконати (введення або виведення) та на якому пристрої це необхідно виконати. Програма (а також і програміст) при цьому може і не знати як це робити. Треба просто сформувати потрібну інформацію та згенерувати необхідний тин переривань.

Розглянемо основні переривання та вимоги для виведення інформації на екран та введення даних з клавіатури.

Команди обробки переривань INTx

Будь-яка операція введення виведення в DOS може бути реалізована через певну підпрограму обробки переривань (ISR). Для виклику цих підпрограм використовують наступні команди сімейства **INTx**:

INTO
INT 3
INT *Номер переривання*

Логіка роботи команди INTO:

Якщо прапорець переповнення **OF=0**, то ніякі додаткові дії не виконуються і продовжується виконання програми.

Якщо прапорець переповнення **OF=1**, то ініціюється команда **INT 4**.

Логіка роботи команди INT:

1. Записати в стек регістр прапорців EFLAGS (FLAGS).
2. Записати в стек адресу повернення:
   - вміст сегментного регістру CS;
   - вміст вказівник команд EIP (IP).

3. Скинути в нуль прапорець IF (та TF, якщо виконується відладка або викликано переривання INT 3).

4. Передати управління програмі обробки переривань із зазначенням номера Номер переривання. Для цього використовується спеціальна системна область пам’яті – **таблиця векторів переривань** (див. Загальну схему розподілу пам’яті в MS DOS).

Загальна схема розподілу пам'яті в MS DOS

Адреси (Hex)	Назва та опис
Стандартна пам'ять (640 К)	
00000-003FF	Таблиця векторів переривань (256 чотирьох байтних адрес)
00400-004FF	Область даних ROM BIOS (BIOS DATA Area)
00500-00xxx	Область DOS (DOS Area)
00xxx-9FFFF	Пам'ять для програм користувачів (User RAM) <=638 K
9FC00-9FFFF	Розширення BIOS DATA Area для PS/2 миші
Верхня пам'ять - UMA (384K)	
A0000-BFFFF	Відеопам'ять (Video RAM)
C0000-DFFFF	Блоки зовнішнього коду ROM (по 2K)
E0000-EFFFF	Вільна область, іноді SYSTEM BIOS
F0000-FFFFF	System BIOS або flash-ROM на системній платі

5. Забезпечити виконання необхідних дій.
6. Відновити зі стеку адресу повернення та прапорці, продовжити виконання перерваної програми.

Всі необхідні екранні та клавіатурні операції можна виконати, використовуючи відповідні функції команди **INT 10h** (**виведення на дисплей** - відеосервіс) та **INT 16h (введення з клавіатури)**, які передають управління безпосередньо в BIOS (Basic Input Output System – базова система введення-виведення). BIOS будь-якого комп’ютера сімейства i86 знаходиться в ПЗП (постійний запам’ятовуючий пристрій), це одна або дві мікросхеми, які встановлені на системній (материнській) платі. Сучасні комп’ютери мають ще і додаткові ПЗП BIOS, які розташовані на картах розширень (наприклад, відеокартах), які містять альтернативні ISR для обробки виведення на екран.

Для виконання деяких більш складних операцій існує переривання більш вищого рівня **INT 21h**, яке спочатку передає управління в DOS. Наприклад, при введенні з клавіатури необхідно підраховувати введені символи, перевіряти на максимальне число символів та перевіряти на символ Enter. Переривання **DOS INT 21h** виконує багато з цих додаткових обчислень, а потім автоматично передає управління в BIOS.

Таким чином, команда **INT** перериває обробку програми, передає управління програми в DOS для виконання певних дій, а потім повертає управління в перервану програму для продовження подальших обчислень.

Слід мати на увазі, що при реалізації операцій введення-виведення через переривання BIOS на даному комп’ютері багато функцій можуть модифіковані,

тому результат, на який ви розраховували при відладці програми на вашому комп'ютері, можна не отримати на іншому. Оскільки підпрограми обробки переривань в процесі своєї роботи можуть псувати регістри, тому до виклику переривань треба в стеку зберігти потрібні регістри, а потім відновити їх і продовжити виконання програми.

16.1 ВВЕДЕННЯ З КЛАВІАТУРИ

Операційна система надає декілька способів введення даних з клавіатури - засобами сервісного переривання **MS DOS 21h** та засобами **BIOS 16h** [17, 18]:

MS DOS 21h – звернення до клавіатури за допомогою файлової функції **3Fh** переривання **21h**;
– використання групи функцій введення-виведення з діапазона **01h … 0Ch**;

BIOS – посимвольне введення з використанням переривання INT 16h.

Щоб звернутися до потрібної функції DOS, слід виконати наступні дії:

- завантажити регістри, відповідно опису функції;
- підготувати необхідні буфери, рядки ASCII та ін.;
- розмістити номер функції DOS в регістр **AH**. Якщо використовується функція з під функцією, то номер під функції зазвичай розташовується в регістр **AL;**
- викликати переривання DOS INT 21h.

Наприклад, для завершення програми з конкретним кодом *Code* та передачі управління DOS треба виконати функцію 4Ch:

```
MOV     AL, Code
MOV     AH, 4Ch
INT     21h
```

Використання файлової функції 3Fh переривання 21h

При використанні цієї функції використовується дескриптор 0, закріплений за стандартним пристроєм введення – клавіатурою. Число введених символів вказується в регістрі CX., введення закінчується лише тоді, коли натиснута клавіша Enter, незалежно від того скільки введено символів (але не більше запланованого). Доцільно завантажити в регістр CX. Максимальну довжину рядка (80 байт). В регістрі AX повертається реальне число введених байтів, при цьому враховуються також і два байти, які надходять при натисненні клавіші Enter.

Вхід: AH=3Fh, BX=дескриптор, CX=кількість введених символів

Вихід: AX=реальне число введених байтів

Наприклад, **Введення з клавіатури рядка за допомогою файлової функції 3Fh:**

```
title VVOD S KLAVATYRU
        .model small            ; Модель пам'яті для EXE
        .stack  100h            ; сегмент стека розміром 256 байт
        .code
```

```
; В сегменті команд
; Виведемо на екран рядок-запит
 start:                              ; start – точка входу в програму
      mov ax, @data
      mov ds, ax
      mov     AH, 09h
          mov   DX, offset msg
          int     21h
; Запит на введення
          mov   AH, 3Fh      ;  Файлова функція введення
          mov   BX,  0        ;  Дескриптор клавіатури
          mov   CX,  25       ;  Максимальне число символів
          mov dx,  offset inbuf  ;  або   lea dx, [inbuf]   адреса буфера введення
          int     21h
          mov cx, ax
          sub cx, 2             ; фактично введено  (без кодів 13, 10)
      .data
; В сегменті даних
msg      db      'VVEDIT PRISVUSCHE: $'
inbuf            db      256  dup('*'), '$'
          end  start
```

В цьому прикладі на екран виводиться рядок-запит для користувача, після чого функція 3Fh очікує введення рядка, який завершується натисненням клавіші Enter. При введенні рядка (до натискання Enter) рядок можна редагувати: стирати введені символи та заміняти новими. Після натискання клавіші Enter введений рядок разом з кодами 13 та 10 (повернення рядка та переведення каретки) надходять в програмний буфер *inbuf*, де його можна побачити за допомогою відладчика.

Використання групи функцій введення-виведення з діапазона 01h … 0Ch

Другий спосіб отримання даних з клавіатури в програму забезпечує більш різноманітні можливості. Символ можна вводити таким чином, щоб він відображався на дисплеї при введенні (це називається *ехо-супровід*), або вводити таким чином, що його не видно (без *ехо-супроводу).* Для цього використовуються функції (див. табл. 10).

Таблиця 10. Функції введення з групи функцій введення-виведення

Функція	Призначення
01h	Введення символа з ехом та відпрацювання Ctrl-Break (Ctrl+C), вихідний регістр <AL>=ASCII код символа, <AH>=ScanCode
06h	Пряме по символьне введення-виведення через консоль
07h	Введення символа без еха та без відпрацювання Ctrl-Break, вихідний регістр <AL>=Символ

08h	Введення символа без еха з відпрацюванням Ctrl-Break, вихідний регістр <AL>=Символ
0Ah	Введення рядка в буфер клавіатури з ехом, адреса буфера <DS:DX>, в першому байті буфера записується довжина буфера
0Bh	Перевірка стану стандартного пристрою введення, вихідний регістр <AL>=0FFh, якщо буфер клавіатури є пустим, <AL>=0, якщо буфер непустий
0Ch	Очищення кільцевого буфера клавіатури та введення однієї з функцій, вхідний регістр <AL>=номер потрібної функції

Буфер клавіатури

Область даних BIOS за адресою **40:1EH** містить *буфер клавіатури*. Це дозволяє вводити до 15 символів ще до того, як програма запитає введення з клавіатури. Коли ви натискаєте клавішу, процесор клавіатури автоматично генерує *скан-код* (унікальний номер, призначений клавіші) і викликає переривання BIOS INT 09H.

Обробник INT 09H отримує з клавіатури скан-код, перетворює його у відповідний символ ASCII та розташовує його в область буфера клавіатури. Після цього переривання BIOS INT 16H (сама низькорівнева операція роботи з клавіатурою) зчитує символ з буфера та передає його виконуючій програмі. Буфер клавіатури організований за принципом кільця і має довжину 16 символів (15символів + Enter). Символ в буфері має 16 бітів (ScanCode клавіші в AH плюс ASCII-код символа в AL), див. Таблицю скан-кодів.

Функції 01h, 06h, 07h, 08h при кожному виклику програми вводять в програму один символ з кільцевого буфера (у випадку необхідності введення рядка символів ці функції треба вводити в циклі). Ці функції розрізняються наявністю чи відсутністю відображення символа на екрані (еха), а також реакцією на введення клавіш Ctrl+C. Функції 07h, 08h дають можливість вводити дані тайком від оточуючих (наприклад, пароль або ключ). При виконанні функцій 01h та 08h DOS перевіряє кожний введений символ, якщо у вхідному потоці зустрічається код Ctrl+C (03h), то програма аварійно завершується. Функції 06h та 07h пропускають цей код пропускають.

Функції 01h, 07h, 08h призначені тільки для виведення, функція 06h реалізує як по символьне введення з клавіатури, так і виведення символів на екран. Режим роботи цієї функції задається в регістрі AL: код FFh означає введення, будь-який інший код призводить до виведення на екрані символа, який відповідає цьому коду.

Приклад введення символа у циклі з клавіатури: ввести масив, який складається з 7 чисел:

```
        .data
        mas1 db 7 dup (0)
        cr_lf db 0ah, 0dh, ‘$’
……………………………………
        .code
……………………………………
```

```
            mov di, 0          ; обнуляємо лічильник
   cicle:   mov ah, 1          ;  зчитуємо один символ
            int 21h
            cmp al, ‘0’ ; код повинен бути в інтервалі 30H – 39H або 30h, або 48 –
десятковий код 0
            jnb num1                 ; (>=)
            jmp cicle
   num1: cmp al, ‘9’                 ; 39H
            jbe num2           ; (<=)
            jmp cicle
   num2: sub al, 30h                 ; перетворюємо у цифру
            mov mas1[di], al         ; записуємо цифру у масив
            mov dx, offset cr_lf     ; переведення на наступний рядок
            mov ah, 09h
            int 21h
            inc di
            cmp di, 7
            jne cicle
```

Функція 0Ah передає в буфер рядок, довжина якого не повинна перевищувати 254 символа, введення закінчується на тисканням клавіші Enter. Символи, що вводяться, відображаються на екрані, при введенні Ctrl+C відбувається аварійне завершення програми.

При використанні функції *0Ah* символ записується в буфер клавіатури, структура якого має вигляд: L – довжина буфера. Введений рядок починається з 3-го байту буфера клавіатури (див. табл. 11).

Таблиця 11. Буфер клавіатури

Перший байт	Другий байт	Інші L-байтів	
Загальна довжина буфера	Кількість фактично введених в буфер символів	actL введених символів	Вільні
L	actL	actL<=L	

Мовою Асемблера **буфер можна записати двома способами** (для L=30 байтів):

```
; Другий та інші 30 байтів заповнюються пропусками
Buffer  Db     30, 31 dup (' ')       ; 31 – це 30+символ Enter
```

Або з використанням директиви **Label**:

```
; ----------- структура буфера -----------
Buffer          Label          BYTE
L         db     30            ; максимальна довжина
actL      db     ?             ; реальна довжина
field            db     30  dup  (?)  ; інформаційне поле буфера
```

Приклад1 :

```
; Введення імен з клавіатури
      Parlist Label byte     ; Список параметрів імен
      Maxlen db 20           ; Максимальна довжина імені
      Actulen db ?           ; Кількість введених символів
; Реальна довжина імені 20 символів, однак символ Enter (09h) теж символ, тому поле імені 21 символ
      Kbname db 21 dup (‘ ‘)  ; Введене ім’я
      ……………………………………………
      Push AX
      Push DX
; Запити на введення з клавіатури
      Mov AH, 0AH  ;
      Lea DX, Parliast       ; Адреса структури
      Int 21H
      Pop DX                 ; Відновлення значень
      Pop AX                 ;   регістрів
      …………………………..
   ; Вивести ім’я
      Mov AH, 09h
      Lea DX, kbname         ; Завантажити адресу введеного імені
      Int 21H
```

Функція 0Bh дозволяє перевірити наявність в кільцевому буфері символів, які очікують на введення. За наявності символів програма вилучає їх з буфера однією з функцій введення, якщо символів немає, програма продовжує виконуватися. Таку функцію застосовують в програмах, що носять циклічний характер, якщо треба забезпечити управління ходом виконання програми з клавіатури термінала. В кожному кроці циклу перевіряється стан кільцевого буфера, якщо оператор натиснув будь-яку клавішу, програма аналізує введений код і здійснює вихід з циклу; якщо буфер пустий – продовжується циклічне виконання.

Функція 0Ch призначена для організації введення з попередньою очисткою кільцевого буфера, тобто спочатку буфер очищається, а потім очікує введення символа з клавіатури. Якщо випадково були введені коди, то вони втрачаються. Зазвичай ця функція використовується в програмі безпосередньо слідом за функцією виведення на екран символьного рядка з пропозицією оператору вводити дані. В результаті з кільцевого буфера вилучається все „сміття” від випадкових натиснень, а в програму вводиться оператором тільки, що було після запиту.

Приклад 2: Введення рядка символів

```
: Структура буфера
      Buffer Label Byte
      MaxL db 11             ; Максимальна довжина
      ActL db ?                       ; Реальна довжина
```

```
        Field db 11 dup (?)     ; Поле буфера
    ; Реальна довжина 10 символів, оскільки ENTER (09H) теж символ
……………………………………………
        Call readString         ; Введення рядка символів
……………………………………………………………..
        Call readKey            ; Введення символів без Ехо-супроводження
…………………………………………………………………
; Введення рядка символів
; Вхід: Buffer – буфер клавіатури
; Вихід: CX – фактичне число введених символів,
        DX – адреса інформаційної частини буфера
        Actl – реальна довжина буфера
readString proc
        push ax
        lea dx, buffer
            mov ah, och         ; очистка буфера
            mov al, 0ah         ; читання рядка у буфер
        int 21h
        xor ch, ch
        mov ch, actl            ; реальна кількість введених символів
        add dx, 2 ; встановлення вказівника на інформаційну частину буфера,
тобто переведення на 3-й байт клавіатури або
                    ; або  xor  di, di          mov di, 2
                    ; або  mov si, 2
                    ; або  mov dx, 2
                    ; або  mov al, array [di][si]
        pop ax
        ret
readString endp
; Введення символа без ехо-супроводу
            ;   Вхід: HI
        ;  Вихід: AL – символ, що вводиться
        readKey  proc
                 push   ax           ; збереження регістра AX
                 mov    ah,  8       ; читання символа
                 int    21h
                 pop    ax           ; відновлення регістра AX
                 ret
        readKey  endp
```

Додаткові функціональні клавіші та скан-коди

Додаткові функціональні клавіші, наприклад, *F1, Home*, викликають виконання будь-якої операції, а не відображення символа на екрані. В апаратній частині системи ніщо не пов’язує окремі клавіші з певними діями. Тільки програміст визначає, що, наприклад, натискання клавіші *Home* повинно

перемістити курсор на початок рядка, а натискання клавіші *END* призводить до переміщення курсору в кінець рядка. Можна запрограмувати виконанння будь-яких інших дій при натисненні цих клавіш.

Кожній клавіші відповідає відповідний скан-код [17, 18].

Скан – коди – це номери, що починаються з 01 (скан-коду клавіші Esc). Таблиця Scan-кодів функціональних клавіш містить клавіші *F1-F10, Shift-F1 - Shift-F10, Ctrl-F1 - Ctrl-F10, Alt-F1 - Alt-F10,*

Alt –A - Alt-Z та інші комбінації клавіш, а також їх *HEX* та *DEC* коди. Тобто в *AH* записується скан-код клавіші, а в AL для усіх функціональних клавіш заноситься 00h. За допомогою скан-кодів програма може використовувати функцію BIOS 10H переривання INT 16H та запитати введення одного символа. Операція повертає різні результати в залежності від того, яка була натиснута функціональна клавіша або клавіша символа. Наприклад, для символа літери A операція повертає в AX два елементи даних: AH – скан-код літери A (1Eh) та AL - ASCII – код для літери A (41h). При натисканні клавіші *Del* операція розмістить в AX такі два елементи: AH – скан-коди для клавіші **Del** (53H), в AL – 00H.

При натисканні <**ESC**> AH=01h, AL=1B, <**Enter**> AH=09h, AL=00h.

Скан-код клавіші **F1** – 2Bh, **F2** – 3Ch, **F3** – 3Dh, **Del** – 53h, ↓ - 50h, ← - 4Bh, → - 4Dh, ↑ - 48h.

Приклад 3: Управління програмою за допомогою функціональних клавіш

```
        ;В сегменті команд
        ;Виведемо на екран рядок-запит
            mov  AH, 09h
            mov  DX, offset reqst
            int  21h
        ;Фільтрація розширених кодів ASCII
            mov  AH, 08h          ; Функція введення символа без еха
again:      int  21h
            cmp  AL, 0            ; Молодший байт = 0?
            jne   again           ; Ні, повторювати введення символа
            mov   AH, 08h         ; Так, введемо старший байт
            int  21h
            cmp   AL, 3Bh         ; Натиснута клавіша F1?
            je    f1              ; Так, на відповідний фрагмент
            cmp   AL, 3Ch         ; Натиснута клавіша F2?
            je    f2              ; Так, на відповідний фрагмент
            cmp   AL, 3Dh         ; Натиснута клавіша F3?
            je   f3               ; Так, на відповідний фрагмент
            jmp   again           ; Натиснуто незаплановане
f1:         ; Фрагмент, що виконується по команді F1
            mov   AH, 09h
            mov   DX,  offset  msg_f1
            int  21h
            jmp   go              ; На виконання програми
```

```
 f2:          ;Фрагмент, що виконується по команді F2
              mov   AH, 09h
              mov   DX,  offset msg_f2
              int   21h
              jmp      go              ;  На  виконання програми
f3:           ; Фрагмент,   що виконується по команді F3
              mov      AH, 09h
              mov      DX,  offset msg_f3
              int      21h
go:           ; Продовження програми   (для нас  -  завершення)
;В сегменті даних
          reqst      db        'Введіть  команду   (F1,   F2  або  F3) : ', 13, 10, '$'
msg_f1       db        'Введена команда F1$'
msg_f2       db        'Введена  команда  F2$'
msg_f3       db        'Введена  команда  F3$'
```

Введення з клавіатури з використанням переривання BIOS 16h

Функції BIOS зчитування даних з клавіатури розраховані на певний тип клавіатури з певним числом клавіш. Розрізняють звичайну клавіатуру (83/84 – key) та дві розширені (101/102- key та 122- key)

Робота з клавіатурою на рівні BIOS (INT16h) дозволяє зчитувати двохбайтні коди, що надходять в кільцевий буфер (ASCII-код і скан-код) та аналізує слово прапорців клавіатури (натискання клавіш Ctrl, Alt, Shift та ін.). Функція 00h дозволяє однією дією отримати повний двохбайтний код натиснутої клавіші або комбінації клавіші. З цього коду можна отримати скан-код та значущу частину ASCII-коду (наприклад, функціональної клавіші). При виконанні цієї функції програма зупиняється в очікуванні натискання клавіші (це називається синхронною функцією).

1. Читання символа без ехо-супроводу:

MOV AH, 0; для клавіатури (83/84 – key)
INT 16h

MOV AH, 10h; для клавіатури (101/102 – key)
INT 16h

MOV AH, 20h; для клавіатури (122 – key)
INT 16h

На виході ця функція повертає регістри:
AL - ASCII_Символ, 0 або префікс scan-кода
AH – ScanКодФункціональної Клавіші

Розширені ASCII-коди повертаються тими клавішами (функціональними, комбінаціями клавіш, які не можуть бути представлені стандартними кодами

ASCII. **Розширений ASCII-код зберігається у двох байтах** (**перший байт завжди містить нуль, а другий – scan-код).** В таблиці наведено scan у двох системах числення – десятковій, шістнадцятирічній та натисненню якої клавіші він відповідає.

Якщо натискається будь-яка функціональна клавіша, то в регістрі AL буде міститися нуль, тому для виконання такого переривання треба обов’язково перевіряти вміст регістра AL:

```
        MOV         AH, 0
        INT         16h
        CMP         AL, 0
        JZ          ScanCode
………………………………………
        ScanCode:
        ; Обробка ScanCode (вміст регістра AH)
```

Наприклад, користувач при натиснення клавіші **Home** (скан-код 47H) встановлює курсор в 0-стовпець та 0-й рядок.

```
        Mov AH, 10H       ; Запитати введення з клавіатури
        Int 16H           ; Викликати обробник переривання
        CMP AL, 00H       ; Функціональна клавіша Del для цифрової
клавіатури?
        JE L30            ; Так, пропустити
        CMP AL, 0EH       ; Функціональна клавіша Del для аналогової
                          : клавіатури?
        JNE stop          ; Ні, вихід
 L30:   CMP AH, 47H       ; Скан-код клавіші Home?
        JNE stop          ; Ні, вихід
        Mov AH, 02H       ; Запитати встановлення курсора?
        Mov BH, 00        ; в рядок та стовпець
        Mov DX, 00        ;       00:00
        Int 10H           ; Викликати обробник переривань
```

2. Визначення наявності введеного символа

MOV AH, 1 ; для клавіатури (83/84 – key)
INT 16h

MOV AH, 11h ; для клавіатури (101/102 – key)
INT 16h

MOV AH, 21h ; для клавіатури (122 – key)
INT 16h

На виході ця функція повертає регістри:
AL - ASCII_Символ

AH – ScanКодФункціональноїКлавіші та скидає прапорець нуля (ZF=0), якщо в буфері є зчитаний символ, ZF=1, якщо буфер пустий.

3. Запис символа у буфер клавіатури

Буфер клавіатури організований по принципу кільця та має довжину 16 байтів. У ньому символ має довжину 16 бітів (ScanCode клавіші та власне символ ASCII).

```
MOV   AH, 5      ; Помістити символ у буфер клавіатури
MOV   CH, ScanКодФункціональноїКлавіші
MOV   CL, ASCII_Символ
INT   16h
```

На виході ця функція повертає регістри:

AL - 00, якщо операція виконана успішно,

AH – 01h**,** якщо буфер клавіатури переповнений

4. Визначення поточного стану клавіатури

```
MOV      AH, 2       ; для клавіатури (83/84 – key)
INT 16h

MOV      AH, 12h     ; для клавіатури (101/102 – key)
INT 16h

MOV      AH, 22h     ; для клавіатури (122 – key)
INT 16h
```

На виході ця функція повертає регістри:

AL - байт стану клавіатури 1,

AH – байт стану клавіатури 2 (тільки для функцій 12h та 22h)**.**

Байт стану клавіатури 1 завжди знаходиться в оперативній пам'яті:

Біт 7 – клавіша Ins включена,
Біт 6 - клавіша CapsLock включена,
Біт 5 - клавіша NumLock включена,
Біт 4 - клавіша ScrollLock включена,
Біт 3 - клавіша Alt включена (ліва для функцій 12h, 22h),
Біт 2 - клавіша Ctrl (будь-яка) натиснута,
Біт 1 - Ліва клавіша Shift натиснута,
Біт 0 - Права клавіша Shift натиснута.

Байт стану клавіатури 2 завжди знаходиться в оперативній пам'яті:

Біт 7 – клавіша SysRq натиснута,
Біт 6 - клавіша CapsLock натиснута,
Біт 5 - клавіша NumLock натиснута,
Біт 4 - клавіша ScrollLock натиснута,
Біт 3 – Права клавіша Alt натиснута,
Біт 2 - Права клавіша Ctrl натиснута,
Біт 1 - Ліва клавіша Alt натиснута,

Біт 0 - Ліва клавіша Ctrl натиснута.

Виведення інформації на екран

При виконанні операцій введення-виведення, а саме, організації доступу до файлів, каталогів, відповідних пристроїв операційна система використовує ідентифікатор файлу або пристрою (дескриптор файлу або пристрою), який являє собою 16-бітне число та ініціалізується системою таким чином:

0 : STDIN – стандартний пристрій введення (зазвичай це клавіатура),
1: STDOUT –- стандартний пристрій виведення (зазвичай це екран),
2 : STDERR – пристрій виведення повідомлень про помилки (завжди екран),
3 : STDAUX – послідовний порт (зазвичай COM1),
4 : PRN – паралельний порт (зазвичай LPT1),
тобто функції читання / запису застосовуються і до файлу, і до пристроїв.

Для виведення даних на екран операційна система надає засоби DOS та BIOS.
До засобів DOS відносяться:

- звернення до екрану за допомогою файлової функції 40h переривання 21h;
- використання групи функцій введення-виведення діапазону 01h…..0Ch переривання 21h.

До засобів BIOS відносяться виведення на екран за допомогою переривання INT 10h.

DOS (переривання 21h) підтримує тільки текстовий монохромний режим, тобто не підтримує колір, також не надає засобів очистки екрана та позиціювання курсору. Виведення за допомогою функцій DOS здійснюється тільки рядок за рядком, при виведенні рядків зображення на екрані автоматично прокручується вверх. BIOS дозволяє реалізувати усі можливості відео системи: виведення кольорових символів, встановлення відеорежимів, переключення відео сторінок, завантаження шрифтів і т.ін. **Розглянемо засоби DOS – виведення на екран у текстовому режимі.**

♦ **Функція DOS 40h. Виведення даних у файл або в пристрій.**

Універсальна функція виведення даних з буфера користувача в сегменті даних у файл або на пристрій, дескриптор якого указується в регістрі ***BX***. Дескриптор 1, закріплений за стандартним пристроєм виведення, забезпечує перенаправлення виведення. Значення регістра ***CX*** визначає число байтів, які повинні бути виведені, а пара регістрів *DS:DX* вказує адресу даних, що виводяться. Наступні коди розглядаються як управляючі: *08h* (повернення на крок), *0Ah* (переведення рядка), *0Dh* (повернення каретки) і деякі інші призводять до виконання відповідних ним дій. Після завершення виведення при *CF = 0* регістр *AX* містить число дійсно виведених байтів, а при *CF =1* – код помилки, що повертається (5 або 6) [17, 18].

Вхід: AH=40h
BX=1 для STDOUT або 2 для STDERR
DS:DX = адреса початку рядка
CX = довжина рядка

Вихід: CF = 0
AX = кількість записаних символів

Приклад 1.

```
Out_Area    DB    20 DUP(?)
        mov ah, 40h          ; Запит функції 40h
        mov bx, 01           ; Дескриптор дисплея
        mov cx, 20           ; Число байт, що пересилаються
        lea dx, [Out_Area]   ; Адреса буфера для повідомлення, що
                               виводиться
        int 21h              ; Виклик DOS
```

Приклад 2.

```
; В сегменті команд
        mov   AH, 40h            ; Функція виведення
        mov   BX, 1              ; Дескриптор екрана
        mov   CX, msglen         ; Довжина рядка
        mov   DX, offset msg     ; Адреса рядка
        int   21h                ; Виклик DOS
; В сегменті даних
    msg   db    “programa pochinae svoju robotu”, 13, 10
    msglen=$ - msg               ; Довжина рядка = поточна адреса мінус
    адресу початку рядка
```

- **Функція 02h. Виведення одиночного символу**

Виводить символ, що знаходиться в регістрі *DL,* на екран, після чого курсор пересувається на одну позицію вправо. *Для виведення рядка функцію слід використовувати в циклі.* Допускається перенаправлення виведення. Виконує обробку *<Ctrl/C> (Ctrl-Break)* при введенні цієї комбінації з клавіатури перед виведенням кожного 64-го символу. Ця функція виводить і управляючі *ASCII-символи* з кодами *07h, 08h, 09h, 0Ah, 0Dh*. Символ з кодом *07h (bell,* дзвінок*)* викликає звуковий сигнал, з кодом *08h (backspace, забити символ)* – повертає курсор на одну позицію вліво, з кодом *09h (tab,* табуляція*)* – зміщує курсор на одну позицію управо, кратну *8*. Дії управляючих клавіш з кодами *0Ah* і *0Dh* розглядалися раніше.

Вхід: AH = 02h, Int 21h.
DL = ASCII – код символу

Вихід: *AL = код останнього записаного символу (дорівнює DL, крім випадку, коли DL = 09 (табуляція), тоді повертається значення 20h).*

Використання даної функції розглянемо на прикладі процедури переходу на новий рядок.

```
NewLine PROC
        push ax
        push dx
        mov ah, 2         ;Запит функції 02h
        mov dl, 13        ;Повернення каретки
```

```
        int 21h            ;Виклик DOS
        mov dl, 10         ;Переведення рядка
        int 21h            ;Другий виклик DOS
        pop dx
        pop ax
        ret
NewLine         ENDP
```

Якщо в ході роботи цієї функції була натиснута комбінація клавіш Ctrl-Break, викликається переривання 23h, яке за замовчуванням здійснює вихід з програми.

Приклад виведення на екран ASCII – символів: 16 рядків по 16 символів у рядку.

```
; Виведення на екран усіх ASCII - символів 16 рядків по 16 символів
; vuvasc.asm
        .model tiny
        .code
        org    100h                ; Початок com - файла
 start:
        mov    cx, 256             ; Вивести 256 символів
        mov    dl, 0               ; Перший символ - з кодом 00
        mov    ah, 2               ; Функція виведення символа
cloop:  int    21h          ; Виклик DOS
        inc    dl                  ; Збільшуємо dl на 1 - наступний символ
        test   dl, 0fh             ; Якщо dl не кратний 16,
        jnz    continue_loop       ; продовжити цикл
        push   dx                  ; Інакше: зберігти поточний символ,
        mov    dl, 0dh             ; Вивести CR
        int    21h
        mov    dl, 0ah             ; Вивести LF
        int    21h
        pop    dx                  ; Відновити поточний символ
continue_loop:
        loop cloop                 ; продовжити цикл
        ret                        ; Завершення com - програми
        end start
```

Якщо ми скористаємося перенаправленням, то можна результат роботи програми направити у файл, а не на екран, наприклад: *vuvasc.com > test.txt*, в якому буде уся таблиця.

- **Функція 06h. Пряме виведення одиночного символу без перевірки на Ctrl-Break**

Вхід : AH = 06h
 DL = ASCII – код символе (окрім 0FFh)
Вихід : AL=код записаного символе (копія DL)
Не оброблює управляючі символи та не перевіряє на натиснення Ctrl-Break, функція використовується значно рідше.

- **Функція 09h. Виведення рядка на екран**

Вхід : AH = 09h

DS:DX = адреса рядка, який закінчується символом $ (24h)

Вихід: AL = 24h (код останнього символе)

Функція виводить цілий рядок.

Виведення на екран засобами BIOS

Функції BIOS використовуються для формування кольорових інформаційних кадрів, переключення відеорежимів, завантаження шрифтів та інших дій. Основні функції відеосервісу викликаються через переривання **10h.** BIOS дозволяє переключати екран у різні текстові та графічні режими. Режими відрізняються один від одного роздільною здатністю (для графічних) і кількістю рядків і стовпчиків (для текстових), а також кількістю можливих кольорів.

Екраном вважається прямокутна область екрана з базовими координатами [17, 18]:

<**Лівий верхній кут**>**→ CX (CH = x – рядок, CL =y - стовпець)**

За замовченням CX=0

<**Правий нижній кут**>**→ DX (DH=x – рядок, DL=y - стовпець)**

За замовчуванням DOS встановлює текстовий режим роботи екрана. У цьому випадку DX=184Fh (24 та 79 – весь екран), кількість рядків 0-24, стовпчиків 0-79.

Значення координат (x, y) залежать від режиму роботи дисплея та типу відеоадаптера.

Встановлення та запит відеорежиму

INT 10h, AH=00h ; Установить відеорежим

Вхід : AH=00h

AL=номер режима в молодших 7 бітах

```
mov AH, 0
mov AL, НомерВідеорежиму
int 10h
```

Встановлення в одиницю старшого (8-го) біта означає очистку екрана.

Номери текстових відеорежимів: 0, 1, 2, 3, 7 (див. табл. 12).

Таблиця 12. Текстові відеорежими

Регістр AL (номер відеорежиму)	Роздільна здатність екрана	Режим, кількість кольорів, що підтримуються	Номери відеосторінок
0	40×25	Чорно-білий	0-7
1	40×25	Кольоровий, 16 кольорів	0-7
2	80×25	Чорно-білий	0-3
3	80×25	Кольоровий, 16 кольорів	0-3
7	80×25	Монохромний	0-3

AH = 00H Установка відеорежима.

Вхід: AL=відеорежим

AL Тип Формат Кольорів Адаптер Адрес

0 текст 40 x 25 16/8 CGA, EGA b800
1 текст 40 x 25 16/8 CGA, EGA b800
2 текст 80 x 25 1 6/8 CGA, EGA b800
3 текст 80 x 25 16/8 CGA, EGA b800
4 графіка 320 x 200 4 CGA, EGA b800
5 графіка 320 x 200 4 CGA, EGA b800
6 графіка 640 x 200 2 CGA, EGA b800
7 текст 80 x 25 3 MA, EGA b000
0dh графіка 320 x 200 16 EGA a000
0eh графіка 640 x 200 16 EGA a000
0fh графіка 640 x 350 3 EGA a000
10h графіка 640 x 350 4/16 EGA a000
0bh,0ch (резервується для EGA BIOS)

Для відеоадаптера стандарту VESA BIOS *Extention* у режимі з високою роздільною здатністю використовується функція 4Fh.

```
MOV AH, 4Fh      ; Встановити розширений відеорежим дисплея
MOV AL, 02       ; Режим SVGA
MOV BX, Номер відеорежима
INT 10h
```

Номер відеорежиму задається в молодших 13 бітах. Встановлення в одиницю старшого (15-го) біта означає, що екран не очищується. Кількість кольорів, які підтримуються, визначається ємністю відеопам’яті.

Текстові режими, які можна викликати з використанням цієї функції такі:
80×60 (регістр BX, режим 108h), 132×25 (регістр BX, режим 109h),
132×43 (регістр BX, режим 10Ah), 132×50 (регістр BX, режим 10Bh),
132×60 (регістр BX, режим 10Ch).

Функції BIOS для роботи у текстовому режимі наведені у табл. 13.

Таблиця 13. Функції BIOS для роботи у текстовому режимі

Функції	Призначення
02h	Встановити позицію курсору
03h	Отримати позицію курсору
05h	Встановити відеосторінку
06h	Ініціалізувати або прокрутити вікно вверх
07h	Ініціалізувати або прокрутити вікно вниз
08h	Прочитати символ та атрибут в позицію курсору
09h	Вивести символ та атрибут в позицію курсору
0Ah	Вивести символ в позицію курсору
0Eh	Вивести символ в режимі телетайпа
1003h	Переключити біт мерехтіння / яскравість
13h	Вивести рядок в режимі телетайпа

Управління положенням курсору
MOV AH, 2 ; **Встановити положення курсору**
MOV BH, *Номер відеосторінки*
MOV DH, *Номер рядка*
MOV DL, *Номер стовпця*
INT 10h

MOV AH, 3 ; **Читання поточного положення та розмір курсору**
MOV BH, *Номер відеосторінки*
INT 10h

На виході ця функція повертає регістри **CX, DX.**
DH – номер рядка поточної позиції,
DL – номер стовпця поточної позиції,
CH, CL – верхня та нижня лінії сканування відповідно.

Функція 05h переключає відеосторінки. Якщо відеосистема знаходиться у текстовому режимі, то текстові сторінки (0…7), якщо встановлений графічний режим, то переключаються графічні сторінки (0…1).

Виведення символів на екран у текстовому режимі (пряма робота з відеопам’яттю)

Все, що зображено на моніторі: графіка, текст, одночасно присутні в пам’яті, яка вбудована у відеоадаптер. Для того, щоб зображення з’явилося на моніторі, його необхідно записати у пам’ять відеоадаптера. Для цього відводиться спеціальна область пам’яті, звернення до якої здійснюється по заданим фізичним адресам, а саме: для текстових режимів текстова пам’ять включає 8 відеосторінок обсягом 32 Кб і починається вона для адаптерів EGA, VGA, SVGA з абсолютної адреси 0B800h:0000h, закінчується - 0B800h:FFFFh. Кожна сторінка займає 4 Кб. Адреса початку 0-ї відеосторінки співпадає з адресою усієї відеопам’яті, тобто 0B800h, 1-ша сторінка починається з сегментної адреси B900h, 2-га сторінка – з адреси BA00h і т.д. Уся відеопам’ять доходить до сегментної межі C000h.

При включенні комп’ютера активною, тобто видимою, стає нульова відеосторінка. Зміна відеосторінок здійснюється викликом функції 05h переривання 10h BIOS. Будь-який символ, який записується у відеопам’ять, зразу відображається на екрані у вигляді кольорового символу на одному із знакомісць.

Кожний символ займає у відеопам’яті поле з 2-х байт (логічна організація текстової пам’яті) [17, 18]:

B800h:00	B800h:01	B800h:02	B800h:03
Символ	Атрибут	Символ	

Знакомісце 0 Знакомісце 1

Молодші байти (парні) усіх полів відводяться під коди ASCII-символів, які відображаються, старші (непарні) байти – під їх атрибути. Атрибути вказують колір символу та колір фону, а також чи не є символ мигаючим.

Структура атрибута символу у текстовому режимі роботи дисплея:

0, 1, 2 біти – колір символу; 3 біт – це інтенсивність: 1 – яскравий колір символу; *4, 5, 6 біти - колір фону (фон),* 7 біт – біт мерехтіння (символ мерехтить за замовчуванням) або яскравий фон.

Кодування кольорів фону та символу наведено нижче (стандартна кольорова палітра EGA).

Опції фону: 00h – чорний, 01h – синій, 02h – зелений, 03h – бірюзовий,
04h – червоний, 05h – фіолетовий, 06h – коричневий, 07h – сірий.

Опції кольору символу: 00h – чорний, 01h – синій, 02h – зелений,
03h – бірюзовий, 04h – червоний, 05h – фіолетовий, 06h – коричневий,
07h – сірий, 08h – темно-сірий, 09h – яскраво-синій, 0Ah – світло-зелений,
0Bh – світло-бірюзовий, 0Ch – світло-червоний, 0Dh – світло-фіолетовий,
0Eh – жовтий, 0Fh – білий.

Таким чином, будь-яка програма може вивести текст на екран простим пересиланням даних без використання функцій DOS, BIOS.

У відеопам'яті перші 80 двохбайтових полів відповідають першому рядку, наступні 80 полів – другому рядку і т.д., тобто коди повернення каретки та переведення рядка не є управляючими символами, а коди символів розташовуються у тому місці відеопам'яті, яке відповідає відповідному рядку.

Для отримання доступу до відеопам'яті треба занести в один з сегментних регістрів даних сегментну адресу відеопам'яті. Після цього, вказуючи ті чи інші зсуви, можна виконати запис у будь-яке місце відеопам'яті. За допомогою базових та індексних регістрів вказують зсув. Слід враховувати, що кожний рядок екрана містить 80 символів, символ потребує 2 байти, тобто для того, щоб задати зсув від початку відеопам'яті до *центру екрану*, треба записати:

$$80*2*12 + 40*2,$$

до останнього знакомісця на екрані – 80*2*25 – 2.

При роботі з відеопам'яттю символи, що виводяться, засилаються у парні байти відеопам'яті.

Приклад 1 прямого програмування відеопам'яті:

```
  ; Налаштування сегментного регістру ES на сторінку 0 відеопам'яті
        mov   AX, 0B800h; Сегментна адреса відеопам'яті
        mov   ES, AX
        mov   BX, 0              ; Зсув до початку екрана
        mov   SI, 80*2*12+40*2 ; Зсув до центру екрану
        mov   DI, 80*2*25-2      ; Зсув до кінця екрану
; Засилаємо у самий початок відеопам'яті «рожицу», код 01 у молодшому байті
  ; У старшому – атрибут: F – яскраво-білий, 0 – на чорному фоні
```

```
        mov   word ptr ES:[BX], 0F01h
; Засилаємо у центр екрану нуль – код 30, з атрибутом синій – 1
; на бірюзовому фоні - 3
        mov   word ptr ES:[SI], 3130h
; Засилаємо в кінець екрану зірочку – код 0F,
; з атрибутом червоний (4) на жовтому фоні (E)
        mov   word ptr ES:[DI], 0E40Fh
        mov   AH, 01h       ; Зупинка програми
        int 21h             ; Для перегляду результату
```

Приклад 2 Запис рядка у відеопам'ять

```
; Налаштуємо сегментний регістр DS на сегмент даних
        mov   AX, @DATA
        mov   DS, AX
; Налаштуємо регістр ES на 0 сторінку відеопам'яті
mov ES, 0B800h
……………………………………………….
; Перешлемо у відеобуфер рядок символів, для цього налаштуємо
; регістри SI, DI, CX
        mov SI, offset msg         ; SI = зсув джерела
        mov DI, 80*2*12+37*2  ; DI = зсув приймальника
        mov CX, msglen             ; CX = кількість байтів, що пересилається
        cld              ; очистка прапорця напрямку DF=0, можна скористатися
```

командою movs для роботи з рядковими даними для їх пересилання у пам'ять, ця операція виконується зліва на право і збільшує DI, SI на 1 для байта, на 2 – для слова, на 4 – для подвійного слова

```
; команда rep        рядкова інструкція – повторити рядок вказану кількість
; префікс rep виконує інструкцію стільки разів, скільки вказано у регістрі CX
rep    movsb        ; переміщує у циклі байти (movsw – слово, movsd – подвійне
                    ; слово)
; зупиняємо програму
mov AH, 01h
int 21h
; дані         10h – переведення рядка, 11h – вертикальна табуляція,
;              0Eh -висунути
msg    db 10h, 0Eh, 'T', 0Eh, 'e', 0Eh, 'c', 0Eh, 't', 0Eh, 11h, 0Eh
msglen=$-msg
```

Приклад 3 Формування рядка для запису у відеопам'ять кольорового тексту для символів з одним і тим же атрибутом

```
; Налаштуємо сегментні регістри ES, DS
mov   AX, @DATA
mov   DS, AX
mov   AX, 0B800h        ; Сегментна адреса відеопам'яті
```

```
mov   ES, AX
mov DI, 80*2*5          ; Початковий зсув на екрані
; Заносимо байт за байтом на екран, вставляючи між ASCII-кодами байти атрибута
mov CX, msglen          ; Довжина рядка, що обробляється
mov SI, offset msg;     ; Зсув рядка
cld                     ; DF=0
mov AH, 31h             ; Атрибут – синій по бірюзовому
load:  lodsb            ; Завантажили з пам’яті у AL наступний символ
stosw                   ; Зберігаємо вміст AX – це символ та атрибут у відеопам’ять
loop load               ; Повторюємо msglen – разів
; Зупиняємо програму
mov AH, 01h
int 21h
; Дані
msg db ‘Звернення до функцій DOS і BIOS здійснюється ‘
    db ‘за допомогою механізму програмних переривань. Для цього треба’
    db ‘налаштувати певним чином регістри загального призначення’
    db ‘та виконати команду програмного переривання INT з відповідним номером’
    db ‘користувач активізує потрібну функцію DOS, BIOS’
msglen=$-msg
```

Команда lodsb на кожному кроці циклу зсуває адресацію на 1 байт, тобто на наступний символ рядка, що виводиться. Команда stosw виводить у відеопам’ять ціле слово (символ і атрибут), зсуває адресацію на 2 байти та переміщується до наступного знакомісця.

Приклад 4 Виведення у відеопам’ять тексту з атрибутом за замовчуванням

```
; Налаштуємо сегментні регістри ES, DS
mov   AX, @DATA
mov   DS, AX
mov   AX, 0B800h; Сегментна адреса відеопам’яті
mov   ES, AX
mov SI, offset msg; Зсув рядка
mov DI, 160*24    ; Нижній рядок екрану
mov CX, 5         ; Довжина тексту
ss:   lodsb       ; Завантажили з пам’яті у AL наступний символ
      stosb       ; Виведемо на екран
      inc DI      ; До наступного знакомісця
      loop ss     ; Цикл по усім символам рядка msg
; Зупиняємо програму
mov AH, 01h
int 21h
; Дані
```

msg db ‘STOP!’

Для виведення символів на екран у текстовому режимі використовують такі функції:

MOV AH, 8 ; **читання символу та його атрибута у поточній позиції ; курсору**
MOV BH, *Номер відеосторінки*
INT 10h
На виході функція повертає регістри:
AH – атрибут символу,
AL – ASCII – код символу.

MOV AH, 9 ; **Вивести символ з заданим атрибутом**
MOV BH, *Номер відеосторінки*
MOV AL, *ASCII – код символу*
MOV BL, *Атрибут символу*
MOV CX, *Число повторень символу*
INT 10h

MOV AH, 0Ah ; **Вивести символ з поточним атрибутом**
MOV BH, *Номер відеосторінки*
MOV AL, *ASCII – код символу*
MOV CX, *Число повторень символу*
INT 10h

Функція виводить на екран будь-який символ, але в якості атрибута символу використовується атрибут, який мав символ, що знаходився раніше у даній позиції.

За допомогою цих двох функцій на екран можна вивести будь-який символ, включаючи CR та LF.

MOV AH, 0Eh **; Вивести символ з поточним атрибутом у режимі телетайпу**
MOV BH, *Номер відеосторінки*
MOV AL, *ASCII – код символу*
INT 10h
Ця функція інтерпретує символи CR, LF, BS (08), BELL (07) як управляючі.

Виведення рядка символів
MOV AH, 13h
MOV AL, *Режим виведення ;* у нульовому біті режим виведення задається прапорцем: біт = 0 – розташувати курсор в кінець рядка після виведення (рядок виводиться по одному байту на символ),
біт =1 рядок містить і символи і атрибути (по два байти на символ)
MOV CX, *Кількість символів, що виводиться*
MOV BL, *Атрибут виведення*; якщо рядок містить тільки символи
MOV DH, *Номер рядка початку виведення*

MOV DL, *Номер стовпця початку виведення*
LEA ES:BP, *Адреса початку рядка у пам'яті*
INT 10h

Очистка і прокрутка екрана

MOV AH, 6 ; прокрутка екрана вверх
MOV AL, *Кількість рядків прокрутки* ; 0 – екран заповнюється пропусками
MOV BH, *атрибут екрану* ; 7 – чорно-білий
MOV DH, *Номер рядка нижнього правого кута ; 24 – увесь екран у режимах 0-3, 7*
MOV DL, *Номер стовпця нижнього правого кута ; 79 – увесь екран у режимах 2, 3, 7*
MOV CH, *Номер рядка верхнього лівого кута*
MOV CH, *Номер стовпця верхнього лівого кута*
INT 10h

MOV AH, 7 ; прокрутка екрана вниз
MOV AL, *Кількість рядків прокрутки* ; 0 – екран заповнюється пропусками
MOV BH, *атрибут екрану* ; 7 – чорно-білий
MOV DH, *Номер рядка нижнього правого кута ; 24 – увесь екран у режимах 0-3, 7*
MOV DL, *Номер стовпця нижнього правого кута ; 79 – увесь екран у режимах 2, 3, 7*
MOV CH, *Номер рядка верхнього лівого кута*
MOV CH, *Номер стовпця верхнього лівого кута*
INT 10h

Приклад: вивести на екран фразу червоними літерами по бірюзовому фону

```
; Виведення на екран кольорових символів засобами BIOS
                .model small
                .stack 100h
                .data
                msg db 'AVARIYA!'
                .code
 start:
                mov ax, @data
                mov ds, ax
; Виконуємо початкове настроювання регістрів
                mov cx, 8  ; кількість символів, що виводиться
                mov dl, 36 ; початкова позиція рядка на екрані
                mov si, offset msg      ; зсув у рядку текста
 output:
; Позиціонуємо курсор
                mov ah, 02h      ; Функція встановлення курсору
```

```
                mov bh, 0  ; Відеосторінка
                mov dh, 12          ; Рядок
                int 10h             ; переривання bios
; Виводимо символ
                mov ah, 09h         ; Функція виведення символу
                mov al, [si]        ; Символ
                mov bl, 34h         ; Атрибут
                push cx             ; Зберігаємо cx
                mov cx, 1           ; Коефіцієнт повторення
                int 10h
                pop cx              ; Відновлення cx
                inc si              ; Зсув по рядку текста
                inc dl              ; Зсув по екрану
                loop output         ; Цикл
                mov ax, 4c00h       ; Повернення  в середовище DOS
                int 21h
                end start
```

Графічні відеорежими
Робота з VGA-режимами

У графічному режимі зображення на екран виводиться пікселів або кольорових цяток. Піксель – це найменша одиниця растрового зображення, яка отримується за допомогою графічних систем виведення інформації (комп’ютерні монітори, принтери і т.д. Роздільна здатність такого пристрою визначається горизонтальним та вертикальним розмірами зображення, яке ми бачимо, у пікселях.

Функція 00 переривання BIOS 10h дозволяє переключатися у деякі графічні режими. Ці відеорежими є стандартними і підтримуються усіма відеоадаптерами, починаючи з VGA (див. табл. 14) [17, 18].

Таблиця 14. Основні графічні режими VGA

Номер режима	Роздільна здатність	Кількість кольорів
11h	640*480	2
12h	640*480	16
13h	320*200	256

Для відеоадаптера EGA – номер режима 10h, який забезпечує виведення на екран графічного зображення з роздільною здатністю 640*350 цяток та широко використовується при роботі з відеоадаптерами.

Зображення з використанням пікселів працює більш повільніше, ніж прямий запис у відеопам’ять, однак з точки зору сумісності для різних режимів та відеоадаптерів є більш надійним. У графічних режимах функція BIOS використовується для запису пікселя заданого кольору у будь-яку сторінку відеопам’яті.

Функція BIOS - виведення пікселя 0Ch переривання 10h має вигляд:

На вході: AH = 0Ch ; запис пікселя
AL = номер кольора;
BH = номер сторінки;
CX = координата X;
DX = координата Y;

Зображення малюємо по цяткам (в BIOS не передбачено програмних засобів виведення будь-яких геометричних фігур та замальовування областей екрану). Для виведення на екран кольорової цятки (пікселя) використовується функція 0Ch переривання 10h. Ця функція потребує занесення в регістр AL коду кольору, в BH – номер відеосторінки, в CX – х-координати цятки, яка виводиться, в діапазоні 0,…349, а в DX – у-координати цятки в діапазоні 0,…639.

Приклад.1. Виведення на екран горизонтальної прямої

```
; Встановимо графічний режим EGA
mov AH, 00h        ; (1) Функція завдання режиму
mov AL, 10h        ; (2) Графічний режим EGA
int 10h            ; (3) Виклик BIOS
; Намалюємо пряму лінію циклі по координаті х
mov SI, 150        ; (4) Початкова х-координата
mov CX, 300        ; (5) Число точок по горизонталі
line: push CX      ; (6) Збережемо його в стеці
mov AH, 0Ch        ; (7) Функція виведення пікселя
mov AL, 4          ; (8) Колір червоний
mov BH, 0          ; (9) Відеосторінка
mov CX, SI         ; (10) X-координата (змінна)
mov DX, 175        ; (11) Y-координата (константа)
int 10h            ; (12) Виклик BIOS
inc SI             ; (13) Інкремент х-координати
pop CX                     ; (14) Відновимо лічильник кроків
loop line          ; (15) Цикл з CX кроків
; Зупинимо програму для спостереження результату її роботи
mov AH, 08h        ; (16) Функція введення з клавіатури без ехо
int 21h            ; (17) Виклик DOS
; Переключився відеоадаптер назад у текстовий режим
mov AH, 00h        ; (18) Функція, яка задає режим
mov AL, 03h        ; (19) Текстовий режим
int 10h            ; (20) Виклик BIOS
```

У реченнях 1 ... 3 за допомогою функції 00h переривання BIOS 10h здійснюється перемикання відеоадаптера у графічний режим. Оскільки номер режиму заноситься у байтовий регістр AL, всього може існувати 256 різних текстових і графічних режимів, з яких на сьогоднішній день використовуються (апаратурою різних фірм) близько ста. Режим 10h забезпечує виведення графічного

зображення 16 кольорами з роздільною здатністю 640*350 цяток і широко використовується при роботі з відеоадаптерами EGA і VGA.

Зображення малюється по точках (в BIOS не передбачено програмних засобів виведення будь-яких геометричних фігур або хоча б ліній, як немає і засобів зафарбовування областей екрану). Для виведення на екран кольорової точки (пікселя) використовується функція 0Ch переривання 10h. Ця функція потребує занесення в регістр AL коду кольору, в BH - номера відеосторінки, в CX - x-координати виведеної точки в діапазоні 0 ... 349, а в DX - y-координати точки в діапазоні 0 ... 639. Оскільки регістр CX використовується, як лічильник кроків в циклі, для зберігання x-координати зарезервований регістр SI.

Пряма горизонтальна лінія в прикладі 1 малюється шляхом виклику функції 0Ch в циклі, на кожному кроці якого значення y-координати залишається незмінним - 175 (речення 11), а значення x-координати збільшується на одиницю (речення 13).

Після завершення циклу формування зображення в програмі передбачена зупинка (речення 16 і 17) для того, щоб користувач міг, залишаючись в графічному режимі, проаналізувати результати роботи програми. Для зупинки програми використовується функція DOS 08h - введення одного символу з клавіатури. Функція 08h, як уже зазначалося, не відображає введений символ на екрані і тим самим не спотворює графічне зображення. Натискання будь-якої клавіші (крім управляючих - *Ctrl, Alt, Shift* та ін.) відновлює виконання програми.

В кінці розглянутого фрагмента передбачено перемикання відеоадаптера в стандартний текстовий режим з номером 03h (речення 18 ... 20). Якщо таке перемикання не виконати, відеоадаптер залишиться в графічному режимі, що може перешкодити правильному виконанню прикладних програм.

Розглянемо коротко параметри виклику функції 0Ch переривання 10h. У регістр BH заноситься номер відеосторінки, на яку виводиться дана точка. Графічний адаптер EGA забезпечує зберігання і відображення двох графічних сторінок. За замовчуванням видимою (активною) є сторінка 0, проте малювати зображення можна як на видимій, так і на невидимій сторінці. Для перемикання сторінок передбачена функція 05h переривання 10h.

В регістр AL заноситься код кольору цятки. В кожний момент часу зображення на екрані може містити лише 16 кольорів. Цей набір кольорів, що виводяться на екран (колірна палітра), задається програмно і може бути змінений. При завантаженні комп'ютера встановлюється стандартна палітра.

Приклад 2. Виведення на екран горизонтальної прямої в режимі 103h SVGA

```
; Встановимо графічний режим 103h SVGA
mov AH, 4Fh          ; (1) Функція виклику Video BIOS Extention
mov AL, 02h          ; (2) Підфункція установки режиму
mov BX, 103h         ; (3) Графічний режим SVGA 800x600x256
int 10h              ; (4) Переривання BIOS
cmp AH, 0            ; (5) В разі помилки
jne errmes1          ; (6) Перейти на виведення повідомлення
; Встановимо в регістрі 150 таблиці кольорів
```

```
; зелений колір максимальної яскравості
mov AH, 10h              ; (13) Функція управління регістрами палітри
mov AL, 10h              ; (14) Підфункція установки регістра кольорів
mov BX, 150              ; (15) Номер регістра таблиці кольорів (0-255)
mov DH, 0                ; (16) Інтенсивність червоного кольору (6 біт)
mov CH, 63               ; (17) Інтенсивність зеленого кольору (6 біт)
mov CL, 0                ; (18) Інтенсивність синього кольору (6 біт)
int 10h                  ; (19) Переривання BIOS
; Намалюємо пряму лінію в циклі по координаті х
mov SI, 0                ; (20) Початкова х-координата
mov CX, 800              ; (21) Кількість цяток по горизонталі
line: push CX            ; (22) Збережемо його в стеці
mov AH, 0Ch              ; (23) Функція виведення пікселя
mov AL, 150              ; (24) Колір зелений
mov BH, 0                ; (25) відеосторінки
mov CX, SI               ; (26) х-координата (змінна)
mov DX, 300              ; (27) у-координата (константа)
int 10h                  ; (28) Виклик BIOS
inc SI                   ; (29) Інкремент х-координати
pop CX                   ; (30) Відновлення лічильника кроків
loop line                ; (31) Цикл з СХ кроків
; Зупинимо програму для спостереження результатів її роботи
mov AH, 08h              ; (32) Функція введення з клавіатури без ехо
int 21h                  ; (33) Виклик DOS
; Переключимо відеоадаптер назад у текстовий режим
mov AX, 3                ; (34) Установка текстового режиму
int 10h                  ; (35) Виклик BIOS
jmp output               ; (36) Безумовний перехід на мітку
errmes1: mov AH, 09h     ; (37) Функція виведення повідомлення
mov DX, offset message1  ; (38) Зміщення (адреса) повідомлення
int 21h                  ; (39) Виклик DOS
output:                  ; (40) Завершення програми
```

```
;Зупинимо програму для спостереження результатів її роботи
        mov   AH,    08h          ; (32) Функція введення з клавіатури без еха
        int      21h              ;(33) Виклик DOS
; Перемикаємо відеоадаптер назад у текстовий режим
        mov   AX,   3             ;(34)Установка текстового режиму
        int       10h             ;(35) Виклик DOS
        jmp       output          ; (36) Безумовний перехід на мітку
errmes1: mov   AH,      09h       ;  (37)Функція вививедення повідомлення
        mov      DX,   offset message1; (38)  Зсув (адрес) повідомлення
        int      21h              ;(39) Виклик DOS
output:                           ; (40)Завершення програми
```

Контрольні питання:

1. Як працює функція 40h виведення даних у файл або пристрій?
2. Як працює функція виведення одиночного символу 02h?
3. Як здійснюється виведення на екран засобами BIOS?
4. Які вам відомі засоби введення даних з клавіатури?
5. Як працює файлова функція 3Fh для введення рядка з клавіатури?
6. Призначення групи функцій 01h…0Ch?
7. Що таке буфер клавіатури?
8. Наведіть способи заповнення буферу клавіатури.
9. Що таке скан-коди та функціональні клавіші?
10. Як здійснюється введення з клавіатури з використанням переривання BIOS 16h?
11. Які засоби виведення інформації на екран?
12. Як працює функція 40h виведення даних у файл або пристрій?
13. Як працює функція 02h виведення одиночного символу?
14. Як здійснюється виведення на екран засобами BIOS?
15. Які вам відомі текстові відеорежими?
16. Наведіть функції BIOS для роботи у текстовому режимі.
17. Як розташовується символ у відеопам'яті?
18. Які вам відомі графічні режими?

Тема 10. Файлові системи
Лекція 17. Робота з файлами

Файли, правила надання назв файлів, типи структур файлів. Файлова система сімейства Windows NTFS. Таблиця файлів MFT. Атрибути в записах MFT. Приклад файлової системи - MS DOS. Формат елемента каталогу. Атрибути файлів. Функції для роботи з файлами.

Файли є логічними інформаційними блоками, створюваними процесами. На диску зазвичай містяться тисячі або навіть мільйони не залежних один від одного файлів. Файлами управляє операційна система.

Структура файлів, їх імена, доступ до них, їх використання, захист, реалізація і управління ними є основними питаннями розробки операційних систем.

З позиції користувача найбільш важливим аспектом файлової системи є її уявлення, тобто що собою являє файл, як файли іменуються, яким захистом володіють, які операції дозволено проводити з файлами і т. д.

Імена файлів. Файл є механізмом абстрагування. Він надає спосіб збереження інформації на диску і подальшого її зчитування, який повинен захистити користувача від подробиць про спосіб та місце зберігання інформації, а також деталей фактичної роботи дискових пристроїв.

Напевно, найбільш важливою характеристикою будь-якого механізму абстрагування є спосіб управління об'єктами та їх іменування, тому, що стосується файлової системи слід розглянути питання імен файлів. Коли процес створює файл, він присвоює йому ім'я. Коли процес завершується, файл продовжує існувати, і до нього по цьому імені можуть звертатися інші процеси.

Файлова система організована у вигляді дерева каталогів, тому потрібен спосіб зазначення імен файлів. Найчастіше для цього використовуються два методи. У першому методі кожному файлу надається *абсолютне ім'я (повне ім'я),* що складається зі шляху від кореневого каталогу до файлу.

Наприклад, ім'я */usr/ast/mailbox* означає, що кореневий каталог містить підкаталог *usr*, який, в свою чергу, містить підкаталог *ast*, в якому міститься файл *mailbox*. Абсолютні імена файлів завжди починаються з назви кореневого каталогу і є унікальними іменами. В системі UNIX компоненти шляху поділяються символом «слеш» - /. В системі Windows роздільником слугує символ «зворотний слеш» - \. В системі MULTICS цим роздільником служила кутова дужка ->. У цих трьох системах одне і те ж ім'я буде виглядати наступним чином:

Windows	\usr\ast\mailbox
UNIX	/usr/ast/mailbox
MULTICS	> usr> ast> mailbox

Якщо в якості першого символу в імені файлу використовується роздільник, то незалежно від символу, використовуваного в цій якості, шлях буде абсолютним.

Іншим різновидом імені є *відносне ім'я*. Воно використовується спільно з поняттям робочого каталогу (званого також *поточним каталогом*).

Користувач може визначити один каталог як поточний, і тоді всі імена файлів стануть розглядатися відносно робочого каталогу і не будуть починатися з кореневого каталогу.

Наприклад, якщо поточним робочим каталогом буде */usr/ast*, то до файлу, що має абсолютне *ім'я /usr/ast/mailbox*, можна буде звертатися, просто вказуючи *mailbox*. Інакше кажучи, команда UNIX

cp /usr/ast/mailbox/mailbox.bak

і команда *cp mailbox mailbox.bak*

роблять одне і те ж, якщо робочим каталогом є /*usr/ast.*

Сучасною файловою системою сімейства Windows є ***NTFS (New Technology File System)***, яка була розроблена спеціально для версії Windows NT. Починаючи з Windows XP і закінчуючи Windows 10 та Windows Server 2019 за замовчуванням встановлюють цю файлову систему більшість виробників комп'ютерів, вона істотно збільшує безпеку і функціональність Windows.

NTFS використовує 64-бітові дискові адреси і теоретично може підтримувати дискові розділи розміром до 2^{64} байт (проте деякі міркування обмежують цей розмір до більш низьких значень).

Імена файлів в NTFS обмежені 255 символами і зберігаються до кодуванні *Unicode*, що дозволяє в тих країнах, де не використовується латинський алфавіт (наприклад, Греції, Японії, Індії, Росії та Ізраїлі), писати імена файлів на своїй мові.

Структура файлової системи NTFS

Кожний том NTFS (наприклад, дисковий розділ) містить файли, каталоги, бітові масиви й інші структури даних. Кожний том організований як лінійна послідовність блоків (які в термінології компанії Microsoft називаються *кластерами*), причому розмір блоків для кожного тому фіксований (в залежності від розміру тому він може змінюватися від 512 байт до 64 Кбайт). Більшість дисків NTFS використовує блоки розміром 4 Кбайт - це компроміс між застосуванням великих блоків (для ефективної передачі даних) і використанням маленьких блоків (для зниження внутрішньої фрагментації). Посилання на блоки робляться з використанням зміщення від початку томи (за допомогою 64-бітових чисел) [1, 5].

Головна структура даних кожного тому - це **MFT** (*Master File Table - головна таблиця файлів)*, яка є лінійною послідовністю записів фіксованого розміру (1 Кбайт). Кожний запис MFT описує один файл або один каталог. Він містить атрибути файлу (такі, як його ім'я і тимчасова мітка), а також список дискових адрес (де розташовані його блоки). Якщо файл дуже великий, то іноді доводиться використовувати два або більше записів MFT (щоб розмістити в них список всіх блоків). В цьому випадку перший запис в MFT називається основним записом (*base record*), вказує на додаткові записи в MFT.

Сама MFT також є файлом і в цій іпостасі може бути розміщена в будь-якому місці томf (таким чином усувається проблема наявності дефектних секторів на першій доріжці). Більш того, при необхідності цей файл може рости (до максимального розміру в 2^{48} записів).

Перші 16 файлів носять службовий характер і недоступні ОС, називаються *метафайлами*, причому перший файл - сам MFT (рис. 17.1). Ці 16 елементів мають

суворо фіксоване положення і мають копію в середині диска. Решта частини MFT можуть перебувати в довільних місцях диска. Метафайли знаходяться в кореневому каталозі NTFS тома, їх імена починаються з $.

$MFT - сам MFT;

$MFTMirr - копія 16 записів в середині томи;

$LogFile - файл підтримки операцій журналювання;

$Volume - Службова інформація - мітка тому, версія файлової системи і т.д.;

$AttrDef - список стандартних атрибутів файлів томи;

$. - кореневої каталог;

$Bitmap - карта вільного місця тому;

$Boot - завантажувальний сектор;

$Quota - файл з правами користувачів на використання дискового простору (починаючи з NTFS 5.0);

$Upcase - таблиця відповідності великих і великих літер в іменах файлів.

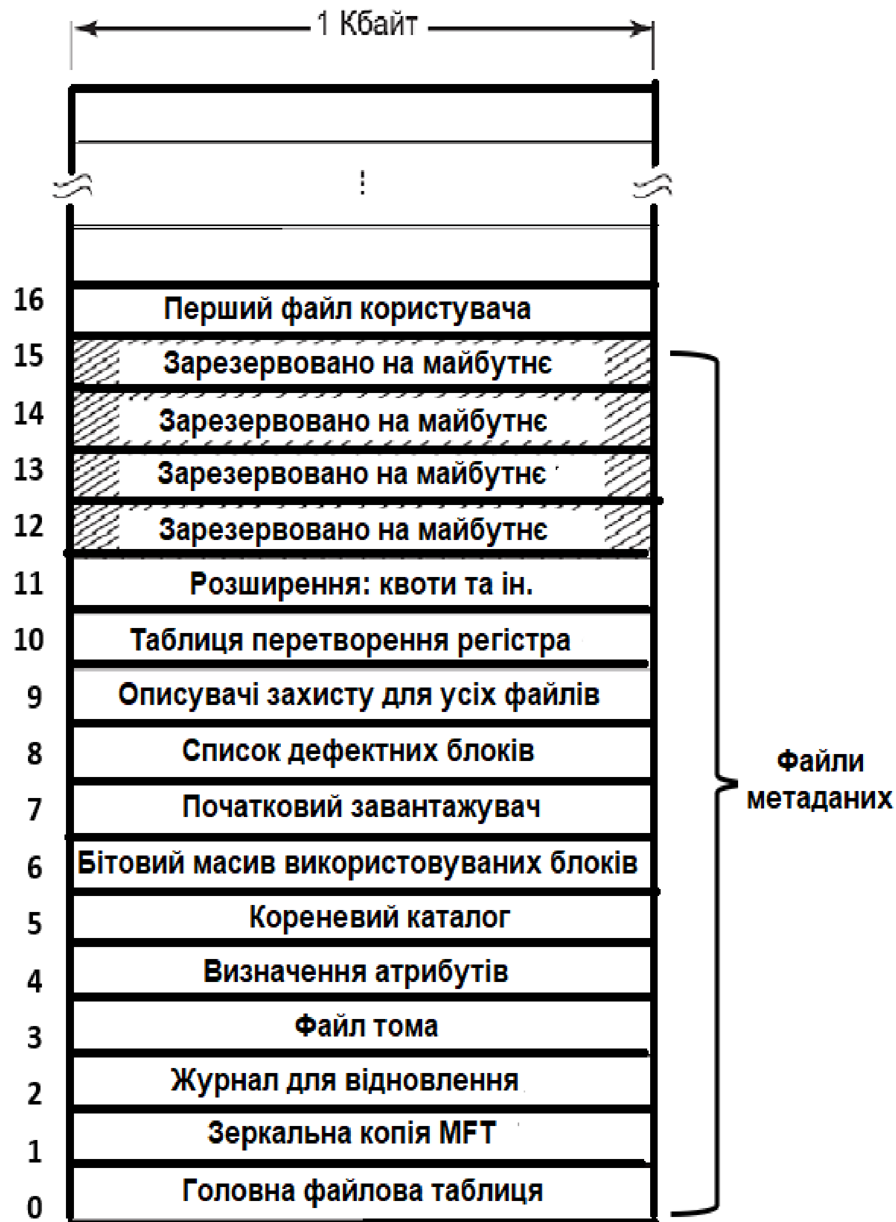


Рис.17.1. Головна таблиця файлів NTFS

В записах MFT зберігається вся інформація про файли, крім власне даних, ім'я файлу, розмір, положення на диску окремих фрагментів і т.д. Якщо одного запису MFT не вистачає, використовується кілька, не обов'язково йдуть підряд. Якщо файл невеликого розміру, то він зберігається в самій MFT, у вільному місці в межах одного запису. Файл в томі ідентифікується файловим посиланням у вигляді 64-разрядного числа. Це номер файлу, який відповідає позиції його запису в MFT і номера послідовності, який збільшується, якщо ця позиція в MFT використовується повторно.

Файл представляється за допомогою потоків. Потоками файлу є не тільки дані, але і його атрибути. Тобто сутність файлу - це його номер в MFT, а все інше, в т.ч. потоки - опційні. Відповідно файлу можна призначити новий потік, записавши в нього будь-які дані. У Windows 2000 так пишеться інформація про автора і зміст файлу. Ці додаткові потоки не проглядаються стандартними засобами, наприклад, розмір файлу - це розмір тільки основного потоку з даними. В результаті можна видалити короткий файл, а звільниться кілька Мб.

Максимальна довжина 1 потоку - 16 Еб. Стандартні атрибути файлів і каталогів тому NTFS мають фіксовані імена і коди типу.

Стандартна інформація про файл:

- Традиційні атрибути *Read Only, Hidden, Archive, System*,
- позначки часу створення і модифікації,
- число каталогів, які посилаються на файл.

Список атрибутів:

- Список атрибутів, з яких складається файл,
- файлове посилання на файловий запис,

-MFT, в якій розташований кожний з атрибутів, якщо файлу необхідно більше одного запису MFT.

Ім'я файлу. Файл в символах *Unicode*. Може мати кілька атрибутів-імен. Наприклад, якщо є зв'язок POSIX з файлом або є ім'я формату 8.3.

Дескриптор захисту.- Структура даних, що оберігає від несанкціонованого доступу. Визначається власником файлу і тими, хто має доступ.

Дані. - Власне дані файлу. У файлі за замовчуванням є один безіменний атрибут даних, він може мати додаткові іменовані атрибути даних. У каталогу немає атрибута даних за замовчуванням, але може мати необов'язкові іменовані атрибути даних.

Корінь індексу, розміщення індексу, бітова карта (тільки для каталогів) - атрибути для індексів імен файлів у великих каталогах.

Кожний запис MFT складається із заголовка запису, за яким слідують пари «заголовок атрибута-значення». Заголовок запису містить системний код, який використовується для перевірки достовірності, послідовний номер (який оновлюється кожний раз, коли запис використовується для нового файлу), лічильник кількості посилань на файл, фактична кількість використаних в запису байтів, ідентифікатор (індекс, порядковий номер) основного запису (використовується тільки для записів розширення), а також деякі інші поля.

NTFS визначає 13 атрибутів, які можуть з'явитися в записах MFT. Атрибути наведені в таблиці 15.

Таблиця 15. Атрибути, які використовуються в записах MFT

Standard information	Біти прапорів, тимчасові мітки і т. д.
File name	Ім'я файла в Unicode, може повторюватися для імені MS-DOS
Security descriptor	Застарів. Інформація безпеки тепер знаходиться в $Extend$Secure
Attribute list	Місцезнаходження додаткових записів MFT (при необхідності)
Object ID	Унікальний для даного тому 64-бітний ідентифікатор файлу
Reparse point	Використовується для монтування і символічних посилань
Volume name	Назва даного тому (використовується тільки в $Volume)
Volume information	Версія томи (використовується тільки в $Volume)
Index root	Використовується для каталогів
Index allocation	Використовується для дуже великих каталогів
Bitmap	Використовується для дуже великих каталогів
Logged utility stream	Управляє журналюванням в $LogFile
Data	Дані потоку, можуть повторюватися

Приклад файлової системи – MS DOS
Запис і читання файлів

У машинах типу IBM PC передбачено два рівні звернення до магнітних дисків. При роботі на нижньому рівні користувач за допомогою переривання BIOS INT 13h звертається безпосередньо до програм управління диском. Типовими операціями цього рівня є запис або читання секторів або форматування доріжки. Файлова система DOS не використовується; необхідна інформація відшукується не по імені файлу, а по номерах поверхні, циліндра і сектора.

Верхній рівень реалізується за допомогою переривання DOS INT 21h, що підтримує, разом з іншими, також і функції обслуговування файлової структури. Програміст працює не з програмами управління фізичним диском, а з файловою системою DOS, маючи можливість оперувати такими поняттями файлової системи, як логічний диск, каталог, файл.

Як відомо, для зручності роботи з великою кількістю різнорідних файлів в DOS використовується деревовидна структура каталогів. Каталогом є файл, зазвичай відносно невеликого розміру, в якому міститься перелік всіх підкаталогів наступного рівня і файлів, що входять в даний каталог. Кожному підкаталогу або файлу в каталозі відводиться один елемент розміром 32 байти, в який DOS заносить інформацію про файл: ім'я, початкову адресу на диску (номер кластера), дату і час створення, довжину, в байтах, а також набір характеристик файлу, так званих його атрибутів. Крім елементів, що відносяться до каталогів, які знаходяться на нижчих рівнях, та файлів, кожний каталог містить ще два елементи: про себе самого і про батьківський каталог. Формат елементу каталога (як кореневого так і будь-якого вкладеного) наведений нижче [17, 18]:

У перших 11 байтах елементу каталогу зберігається ім'я файлу та його розширення. Зазвичай DOS при отриманні від користувача імені файлу автоматично перетворює символи імені та розширення у прописні літери. Тобто в елементі каталогу специфікація файла завжди вказується прописними літерами.

Атрибути файла в окремих бітах . Значення атрибутів наведено нижче (табл.16).

Таблиця 16. Атрибути файлів

Значення	Скорочене позначення	Опис
01h	R	Файл тільки для читання
02h	H	Прихований файл
04h	S	Системний файл
08h	L	Мітка тому
10h	D	Каталог
20h	A	Файл неархівований

Файл може мати декілька атрибутів одночасно. Дата і час створення або останньої модифікації файла записується у вигляді двійкових чисел в окремих полях.

В байтах 26 – 27 елемента каталогу записується номер кластеру, з якого починається файл на диску.

При створенні нового файлу DOS сама відшукує на диску вільне місце і призначає його новому файлу, створюючи новий елемент в каталозі і заповнюючи його відповідною цьому файлу інформацією. Хоча мінімальною порцією інформації, яку передає контролер диска в процесі запису або читання файлу, є сектор, і програми BIOS працюють якраз з секторами, файлова система призначає місце на диску цілими кластерами. *Розмір кластера на дискеті становить один сектор (512 байт); на жорсткому диску в кластер можуть входити 4. ..8 секторів.* Таким чином, мінімальний фізичний розмір файлу, навіть якщо дані в нім займають лише декілька байтів, складає один кластер. Проте *в елементі каталога вказується не фізична, а логічна довжина файлу, тобто обсяг даних у байтах.*

Методика роботи з файлами істотно визначається тією обставиною, що кожен файл може займати на диску декілька несуміжних областей, тобто бути розривним. Така система виділення дискового простору дозволяє, по-перше, в процесі роботи з файлом багато разів дописувати в нього нові дані, збільшуючи при цьому довжину файлу, і, по-друге, знімає проблеми з фрагментацією диска, оскільки навіть найменші і розрізнені вільні області на диску можуть бути

використані для розміщення нового файлу. Слід мати на увазі, що сильно фрагментований файл вимагає помітно більше часу для читання або запису, що знижує швидкість виконання програми, що працює з ним.

Робота з файлами припускає використання дескрипторів (файлових індексів), які в першому наближенні можна розглядати як номери відкритих файлів.

Процедура читання-запису файлу в загальному випадку розпадається на наступні операції:

- створення файлу із заданим іменем у вказаному каталозі або відкриття файлу, якщо він був створений раніше;
- запис у файл або читання з файлу всього вмісту або будь-якої його частини;
- закриття файлу.

У більшості випадків робота з файлом починається з виконання операції його відкриття, для чого передбачена функція DOS. Відкриваючи файл, DOS призначає йому черговий вільний блок опису файлу в системній таблиці відкритих файлів (*System File Table, SFT*). Обсяг цієї таблиці, що визначає максимальне число файлів, з якими можна працювати одночасно, задається на етапі конфігурації DOS директивою FILES файлу CONFIG.SYS.

Знайшовши в системі каталогів диска елемент, що описує файл, який відкривається, DOS заносить у виділений йому блок SFT *основні характеристики файлу, такі, як ім'я, довжина, атрибути, дата і час створення, стартовий кластер, фізична адреса на диску елементу каталога, який містить інформацію про файл, і ряд інших.* Частина інформації переписується в блок SFT з елементу каталога, частина (наприклад, покажчик на блок параметрів диска, де зберігається інформація про фізичні характеристики диска) DOS поставляє сама. Важливим елементом блоку опису файлу є комірка, що складається з двох слів, в якій зберігається покажчик, - номер байта відносно початку файлу, з якого почнеться чергова операція запису або читання. Наявність покажчика дозволяє організувати прямий доступ до файлу, тобто читання або запис, починаючи з будь-якого місця файлу. Посилання на номер виділеного файлу блоку в SFT DOS повертає у програму у вигляді дескриптора.

Звернення до відкритого файлу (запис, читання, зміна характеристик файлу і т. д.) здійснюється за дескриптором, який йому наданий. Невідкритий файл дескриптора не має, і система працювати з ним не може. У міру виконання операцій з відкритим файлом DOS модифікує інформацію в блоці SFT; вміст SFT завжди відображає поточний стан файлу.

По закінченню роботи з файлом його треба закрити відповідною функцією DOS. В процесі закриття здійснюється скидання на диск буферів DOS, модифікація елементу каталога і звільнення блоку опису файлу в SFT разом із закріпленим за ним дескриптором. І те і інше можна тепер використовувати для роботи з іншим файлом. Таким чином, система може послідовно працювати з необмеженою кількістю файлів, але число одночасно відкритих файлів визначається обсягом системної таблиці файлів.

По завершенні програми (для цього передбачена функція DOS 4Ch) виконується автоматичне закриття всіх відкритих в програмі файлів. Тому в

простих і не дуже відповідальних програмах файли можна явним чином не закривати - вони все одно будуть закриті системою.

1. Створення і відкриття файлів

Функція DOS ***3Ch*** Створити файл

Вхід:

AX = 3Ch
CX = атрибут файлу
біт 7: файл можна відкривати різним процесам в Novell Netware
біт 6: не використовується
біт 5: архівний біт (1, якщо файл не зберігався)
біт 4: каталог (повинен бути 0 для функції 3Ch)
біт 3: мітка тому (ігнорується функцією 3Ch)
біт 2: системний файл
біт 1: прихований файл
біт 0: файл тільки для читання
DS:DX = адреса ASCIZ-рядка з повним іменем файлу (ASCIZ-рядок ASCII-символів, що закінчується нулем)

Вихід:

CF = 0 і AX = ідентифікатор файлу, якщо не відбулася помилка
CF = 1 і AX = 03h, якщо шлях не знайдений
CF = 1 і AX = 04h, якщо дуже багато відкриті файли
CF = 1 і AX = 05h, якщо доступ заборонений

Якщо файл вже існує, функція 3Ch все одно відкриває його, привласнюючи йому нульову довжину. Щоб цього не відбулося, слід користуватися функцією 5Bh.

Наприклад:

```
fname db 'lira.txt', 0
…………………………….
new_file proc
        mov ah, 3Ch
        mov cx, 0
        mov dx, offset fname          ; або lea dx, fname
        int 21h
        jc AM                         ; перевірка cf=0. Якщо cf=1, файл не створений
        ret
    AM:
```

Функція DOS ***3Dh*** *Відкрити існуючий файл*

Вхід:

AX = 3Dh
AL = режим доступу
біти 0 - : права доступу
00: читання
01: запис
10: читання і запис

біт 1: відкрити для запису
біт 2 - 3: зарезервовані (0)
біт 4 - 6: режим доступу для інших процесів
000: режим сумісності (решта процесів також повинна відкривати цей файл в режимі сумісності)
001: всі операції заборонені
010: запис заборонений
011: читання заборонене
100: заборон немає
біт 7: файл не успадковується породжуваними процесами
DS:DX = адреса ASCIZ-рядка з повним іменем файлу
CL = маска атрибутів файлів

Вихід:

CF = 0 і AX = ідентифікатор файлу, якщо не відбулася помилка
CF = 1 і AX = код помилки (02h - файл не знайдений, 03h - шлях не знайдений, 04h - дуже багато відкритих файлів, 05h - доступ заборонений, 0Ch - неправильний режим доступу)

Наприклад:

```
mov ah, 3Dh
mov al, 2; читання і запис
int 21h
```

Функція DOS ***5Bh*** Створити і відкрити новий файл

Вхід:

AX = 5Bh
CX = атрибут файлу
DS:DX = адреса ASCIZ-рядка з шляхом, що закінчується символом «\», і тринадцятьма нульовими байтами в кінці.

Вихід:

CF = 0 і AX = ідентифікатор файлу, відкритого для читання/запису в режимі сумісності, якщо не відбулася помилка (у рядок за адресою DS:DX дописується ім'я файлу)
CF = 1 і AX = код помилки (03h - шлях не знайдений, 04h - дуже багато відкриті файли, 05h - доступ заборонений)

Функція 5Ah створює файл з унікальним іменем, яке не є насправді тимчасовим, його слід спеціально видаляти, для чого його ім'я і записується в рядок в DS:DX.

В усіх випадках рядок з повним іменем файла має вигляд:
filespec db ‘d:\test\filename.txt’, 0

2. Читання і запис у файл

Функція DOS 3Fh Читання з файлу або пристрою

Вхід:

AH = 3Fh
BX = ідентифікатор (дискриптор – число, яке операційна система надає відповідному об’єкту), BX=0

CX = число байт
DS:DX = адреса буферу для прийому даних
Вихід:
CF = 0 и AX = число зчитаних байтів, якщо не виникла помилка
CF = 1 и AX = 05h, якщо доступ заборонений,
06h, якщо невірний ідентифікатор
Наприклад:

```
mov ah, 3Fh
mov bx, 0
mov cx, 80
mov dx, offset buf          ; адреса даних
int 21h
```

Якщо при читанні з файлу число фактично зчитаних байтів в AX менше, ніж замовлене число в CX, при читанні був досягнутий кінець файлу. Кожна наступна операція читання, так само як і запис, починається не з початку файлу, а з того байта, на якому зупинилася попередня операція читання/запису.

Функція DOS 40h - Запис у файл або пристрій
Вхід:
AH = 40h
BX = ідентифікатор
CX = число байтів
DS:DX = адреса буфера з даними
Вихід:
CF = 0 та AX = число записаних байтів, якщо не виникла помилка
CF = 1 та AX = 05h, якщо доступ заборонений,
06h, якщо невірний ідентифікатор

3. Закриття та видалення файлу

*Функція DOS **3Eh** - Закрити файл*
Вхід:
AH = 3Eh
BX = ідентифікатор
Вихід:
CF = 0, якщо не виникла помилка
CF = 1 та AX = 6, якщо невірний ідентифікатор

*Функція DOS **41h** - Видалення* файла
Вхід:
AH = 41h
DS:DX = адреса ASCIZ-рядка з повним іменем файлу
Вихід:
CF = 0, якщо файл видалений
CF = 1 та AH = 02h, якщо файл не знайдено,

03h — якщо шлях не знайдено,
05h — якщо доступ заборонений

Видалити файл можна тільки після того як він буде закритий.

Розглянемо декілька прикладів операцій запису і читання файлів на диску.

Приклад: Створення файлу і запис в нього даних

```
;У сегменті команд
; Створимо файл
        mov   ah,3ch                 ; Функція створення файлу
        mov   CX,0                   ; Без атрибутів
        mov   DX, offset fname       ; Адреса імені файлу
        int   21h
        mov   handle, AX             ; Зберігаємо дескриптор файлу
; Запишемо у файл дані (у даному прикладу - текстовий рядок)
        mov   AH, 40h                ; Функція запису у файл
        mov   BX, handle             ; Дескриптор
        mov   CX, buflen             ; Число записуваних байтів
        mov   DX, offset bufout      ; Адреса даних
        int   21h
 ; Закриємо файл (немає необхідності)
        mov AH, 3Eh                  ; Функція закриття файлу
        mov BX, handle               ; Дескриптор
        int 21h
;В сегменті даних
bufout  db 'Файл номер 1'            ; Дані для запису у файл
buflen=$-bufout                      ; Її довжина 112 байт
handle  dw 0                   ; Комірка для дескриптора
fname      db   ‘MYFILE.001’, 0      ; Ім'я файлу у форматі ASCIZ
```

Функція 3Ch дозволяє створити на диску файл із заданим ім'ям. Специфікація файлу, тобто шлях до нього разом з іменем файлу і його розширенням, вказується у вигляді символьного рядка, що завершується двійковим нулем ("рядок ASCIZ"). Для специфікації файлу в програмі діють звичайні правила DOS, Так, якщо в специфікації відсутній шлях, файл створюється в поточному каталозі поточного диску (як у вище приведеному прикладі); якщо вказані шлях і ім'я файлу, він створюється у відповідному каталозі поточного диска.

Якщо функція 3Ch виявляє, що на диску вже є файл з вказаним ім'ям, вона фактично знищує його і створює новий з тим же ім'ям. Тому вище наведений приклад можна виконувати багато разів; при кожному прогоні програми файл MYFILE.001 створюватиметься наново.

Функція 40h дозволяє записувати дані на будь-який пристрій, у тому числі і у файл на диску. Конкретний приймач даних задається його дескриптором. Слід відмітити, що при запису у файл, як, втім, і при виведенні на будь-який пристрій, в приймач даних надходять лише ті дані, які вказані в програмі. Ніякі додаткові коди, які, наприклад, позначають кінець файлу, не записуються. Таким чином, створений

у прикладі файл матиме довжину точно 12 байтів (хоча фактично займе на диску цілий кластер, в якому після наших даних буде "сміття")

У якості даних, записуваних у файл, в прикладі вибраний текстовий рядок. В цьому випадку легко прочитати файл за допомогою будь-якого текстового редактора і проконтролювати правильність виконання програми. Насправді у файл можна виводити будь-які дані. Функція запису 40h (як і функція читання 3Fh, яка буде використана у прикладі) розглядає дані, що пересилаються, просто як послідовність байтів, ніяк не аналізуючи їх значення.

Приклад. **Читання файлу**

```
;У сегменті  команд
;Відкриємо файл
        mov     AH, 3Dh             ; Функція відкриття файлу
        mov     AL, 2               ; Доступ для читання-запису
        mov     DX, offset  fname   ; Адреса імені файлу
        int     2lh
        mov     handle, AX          ; Збережемо дескриптор
;Поставимо запит на читання   80 байт
        mov     AH, 3Fh             ; Функція читання
        mov     BX, handle          ; Дескриптор
        mov     CX, 80              ; Скільки читати
        mov     DX, offset bufin    ; Сюди
        int     21h
        mov     CX, AX              ; Скільки реально прочитали
; Виведемо прочитане на екран
        mov AH,40h                  ; Функція запису
        mov BX, 1                   ; Дескриптор стандартного виводу
        mov DX, offset bufin        ; Виводиться CX байт
        int 21h
;У сегменті даних
        bufin  db 80 dup (' ')      ; Буфер введення
        handle dw 0                 ; Комірка для дескриптора
        fname  db 'MYFILE.001',0   ;Ім’я файлу у форматі ASCIZ
```

Функція 3Dh дозволяє відкрити вже наявний файл. У регістрах DS:DX задається адреса специфікації файлу у вигляді рядка ASCIZ; у регістрі AL - режим доступу (0 - читання, 1 - запис, 2 - читання і запис). Функція повертає дескриптор відкритого файлу в регістрі AX.

Читання файлу здійснюється викликом функції 3Fh, яка вимагає зазначення як вхідних параметрів дескриптора джерела даних (у регістрі BX), адреси приймального буфера (у регістрах DS:BX) і кількості байтів, що передаються (у регістрі CX). Якщо ми хочемо прочитати весь вміст файлу, але не знаємо точно його довжину, можна в запиті на читання вказати свідомо більше число байтів (не більше 65 535). Функція 3Fh сама визначить довжину файлу і прочитає весь його вміст до кінця. Після повернення з DOS в регістрі CX міститиметься число

фактично прочитаних байтів. У вище наведеному прикладі вміст AX переноситься в CX і використовується потім як параметр для функції 40h, за допомогою якої прочитані дані виводяться на екран для контролю.

DOS надає можливість звернення до будь-якого байта файлу за його номером. З цією метою для кожного відкритого файлу DOS створює і підтримує покажчик (що зберігається в SFT), який є відносним номером байта у файлі, починаючи від якого виконуватимуться запис або читання даних. Покажчик тільки що відкритого або створеного файлу позиціонується системою на початок файлу, а функції читання або запису зміщують його на число прочитаних або записаних байтів. Таким чином, повторне використання функцій читання або запису реалізує послідовний доступ до файлу. Для організації прямого доступу до довільного місця файлу передбачена функція 42h, що дозволяє задати положення покажчика відносно початку файлу (для цього треба задати AL=0), кінця файлу (AL=2) або поточного положення покажчика (AL=1). Саме значення зсуву покажчика (із знаком) заноситься в регістри **CX (**старша половина) і DX (молодша). У прикладі нижче проілюструвавши методика прямого доступу до файлу.

Приклад **Прямий доступ до файлу**

```
;  У сегменті команд
; Відкриємо файл
        mov   AH, 3Dh             ; Функція відкриття файлу
        mov   AL, 2                ; Доступ для читання-запису
        mov   DX, offset  fname   ; Адреса імені файлу
        int      2lh
        mov   handle, AX          ; Збережемо дескриптор
;Встановимо покажчик на байт номер 11
        mov       AH, 42h         ; Функція встановлення покажчика
        mov       AL, 0            ; Режим - від початку файлу
        mov        BX, handle     ; Дескриптор
        mov        CX, 0          ; Старша половина покажчика
        mov        DX, 11         ; Молодша половина покажчика
        int        2lh
;Модифікуємо файл
        mov       AH, 40h         ; Функція запису
        mov       BX, handle    ; Дескриптор
        mov       CX, 2         ; Число записуваних байтів
        mov       DX, offset mod ; Звідси виводити
        int       21h
;У сегменті даних
mod    db  '2a'                      ; Буфер виводу
handle dw 0                         ; Комірка для дескриптора
fname   db  'MYFILE.001', 0      ; Ім'я файлу у форматі ASCIZ
```

У приведеному прикладі передбачається, що модифікації підлягає створений раніше файл MYFILE.001, що містить рядок Файл номер 1, у якій слід замінити 1 на 2а (що вимагає, до речі, як заміни останнього байта файлу, так і збільшення довжини файлу на 1 байт).

Відкривши файл і отримавши його дескриптор, можна викликати функцію 42h для установки покажчика у файлі. У параметрах цієї функції ми указуємо режим установки покажчика (від початку файлу) і 32-бітове значення зсуву у файлі (у нашому випадку 11), Після цього викликом функції 40h виконується запис у файл 2 байт з програмного буфера mod.

У наведеному прикладі, де нове виведення у файл, потрібно виконати від його останнього байта, було б простіше задати положення покажчика від кінця файлу. Для цього в параметрах функції 42h слід вказати режим 02, а в регістрову пару CX:DX помістити число «-1» (оскільки ми хочемо зміститися на 1 байт ліворуч від кінця файлу); 32-розрядне число «-1» має машинне представлення FFFFFFFFh. Тому рядки заповнення регістрів CX і DX виглядатимуть таким чином:

```
mov   CX, 0FFFFh
mov   DX, 0FFFFh
```

Якщо програмістові важко у визначенні машинного представлення негативного числа, можна передбачити в полях даних комірку з двох слів для покажчика, вказавши її вміст у десятковій формі;

```
pointer dd -1
```

В процесі трансляції для числа «-1» буде знайдене і записане в комірку pointer його машинне представлення. В цьому випадку для заповнення регістрів CX і DX буде потрібно наступні програмні рядки:

```
mov    CX, word ptr pointer+2     ; Старша частина числа
mov   DX, word ptr pointer          ; Молодша частина числа
```

Контрольні питання:

1. Як організована файлова система?
2. Що таке абсолютне ім’я файлу?
3. Що таке відносне ім’я файлу?
4. Що таке поточний каталог?
5. Яка структура файлової системи NTFS?
6. Що таке голова таблиця файлів?
7. Які вам відомі атрибути файлів?
8. Що таке дескриптор захисту?
9. Який формат каталогу в MS DOS?
10. Що таке файловий дескриптор?
11. Як працює функція 3Ch відкриття та створення файлів?
12. Як працює функція 3Dh відкриття існуючого файлу?
13. Як працює функція 3Fh читання і запису у файл?
14. Як працює функція 40h запису у файл або пристрій?
15. Як організувати прямий доступ до файлу?

Тема 11. Мережні операційні системи
Лекція 18. Мережеві операційні системи

Призначення мережевих операційних систем. Функціональні компоненти мережевої операційної системи. Поняття: сервер, клієнт (робоча станція). Однорангові, дворангові мережі. Мережеві служби та мережеві сервіси. Розподілені операційні системи. Реплікації. Розподілені файлові системи. Операційна система Windows Server 2019.

Структура мережевої операційної системи

Призначення мережевих операційних систем. Комп'ютерна мережа дозволяє користувачеві працювати зі своїм комп'ютером як з автономним і дає йому можливість доступу до інформаційних та апаратних ресурсів інших комп'ютерів. Тому мережева операційна система (***Network Operating System – NOS***) виконує як усі функції локальної операційної системи, так і має засоби взаємодії по мережі з операційними системами інших комп’ютерів.

Мережева ОС - це комплекс програм, який забезпечує обробку, зберігання і передачу даних в мережі [8, 12]. Мережева операційна система виконує функції прикладної платформи, надає різноманітні види мережевих служб і підтримує роботу прикладних процесів, які виконуються в абонентських системах. Мережеві операційні системи використовують клієнт-серверну або однорангову архітектуру. Компоненти NOS розташовуються на всіх робочих станціях, включених в мережу. NOS визначає взаємозв'язану групу протоколів верхніх рівнів, які забезпечують виконання основних функцій мережі. До них, в першу чергу, відносяться:

- адресація об'єктів мережі;
- функціонування мережевих служб;
- забезпечення безпеки даних;
- управління мережею.

Мережева ОС грає роль інтерфейсу, що приховує від користувача усі деталі низькорівневих програмно-апаратних засобів мережі. Залежно від того, який віртуальний образ створює ОС для того, щоб підмінити їм реальну апаратуру комп'ютерної мережі, розрізняють: *мережеві ОС та розподілені ОС.*

Користувач мережевої ОС завжди пам'ятає, що він має справу з мережевими ресурсами. Він завжди знає, де зберігаються його файли, знає, на якій машині виконується його завдання. В ідеальному випадку мережева ОС повинна представити користувачеві мережеві ресурси у вигляді ресурсів єдиної централізованої віртуальної машини. Такі ОС називають *розподіленими* ОС.

Розподілена ОС, розподіляючи роботи по різних машинах системи, примушує набір мережевих машин працювати як віртуальний уніфікований процесор. Користувач такої ОС взагалі знає на якій машині виконується його завдання. Розподілена ОС є єдиною ОС в масштабах обчислювальної системи. Кожний комп'ютер мережі, що працює під управлінням розподіленої ОС, виконує частину функцій цієї ОС. В даний час практично всі мережеві ОС ще дуже далекі від дійсної

розподіленості. В літературі є ще інші визначення мережевої операційної системи у вузькому та у широкому сенсі.

Під *мережевою операційною системою в широкому сенсі* розуміється сукупність операційних систем окремих комп'ютерів, що взаємодіють з метою обміну повідомленнями і розділення ресурсів по єдиних правилах - протоколах.

У вузькому сенсі мережева ОС - це операційна система окремого комп'ютера, що забезпечує можливість працювати в йому мережі, тобто мережева ОС може розглядається як набір ОС окремих комп'ютерів, які входять до складу мережі, причому на різних комп'ютерах мережі можуть виконуватися однакові або різні ОС. Кожна з цих ОС приймає незалежні рішення про створення і завершення своїх власних процесів і управління локальними ресурсами. Також кожна ОС включає взаємно узгоджений набір комунікаційних протоколів для організації взаємодії процесів, що виконуються на різних комп'ютерах мережі, і розділення ресурсів комп'ютерів між користувачами мережі. У мережевій операційній системі окремої машини можна виділити такі функціональні компоненти (рис. 18.1) [8, 12]:

Рис. 18.1. Структурна схема мережевої операційної системи

- *Засоби управління локальними ресурсами комп'ютера*: функції розподілу оперативної пам'яті між процесами, планування і диспетчеризації процесів, управління процесорами в мультипроцесорних машинах, управління периферійними пристроями і інші функції управління ресурсами локальних ОС. Ці засоби реалізують функції ОС автономного комп’ютера.
- *Мережеві засоби*:

- *Засоби надання власних ресурсів і послуг в загальне користування* - серверна частина ОС (*сервер*). Ці засоби забезпечують, наприклад, блокування файлів і записів, що необхідне для їх сумісного використання; ведення довідників імен мережевих ресурсів; обробку запитів віддаленого доступу до власної файлової системи і бази даних; управління чергами запитів віддалених користувачів до своїх периферійних пристроїв.

- *Засоби запиту доступу до віддалених ресурсів і послуг і їх використання - клієнтська частина ОС* (*редиректор*). Ця частина виконує розпізнавання і перенаправлення в мережу запитів до віддалених ресурсів від застосунків і користувачів, при цьому запит поступає від застосунку в локальній формі, а передається в мережу в іншій формі, відповідній вимогам сервера. Клієнтська частина також здійснює прийом відповідей від серверів і перетворення їх в локальний формат, так що для застосунку виконання локальних і віддалених запитів не розрізняється.

- *Комунікаційні (транспортні) засоби ОС*, за допомогою яких відбувається транспортування повідомлень по мережі та обмін повідомленнями у мережі. Ця частина забезпечує:

- ♦ формування повідомлення;
- ♦ розбиття повідомлення на частини (пакети, кадри);
- ♦ перетворення імені комп'ютера у числову адресу;
- ♦ передачу повідомлення по мережі;
- ♦ вибір маршруту у складній мережі;
- ♦ надійність передачі і тому подібне.

Залежно від функцій, що покладаються на конкретний комп'ютер, в його операційній системі може бути відсутнім або клієнтська, або серверна частина (рис.18.2) [8, 12].

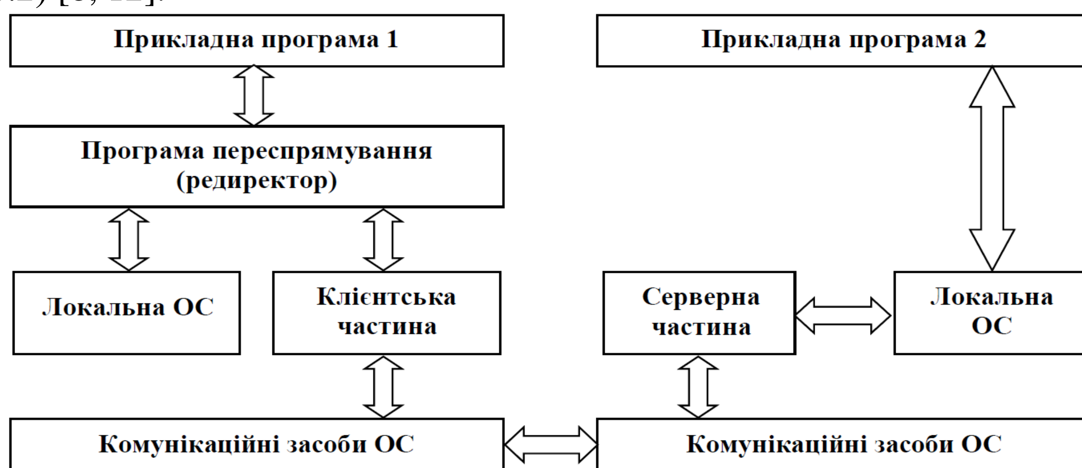


Рис.18.2. Структурна схема взаємодії сервера та клієнта в мережі

Клієнтське програмне забезпечення. Для роботи з мережею на клієнтських робочих станціях має бути встановлено клієнтське програмне забезпечення. Це програмне забезпечення забезпечує доступ до ресурсів, розташованих на мережевому сервері. Трьома найбільш важливими компонентами клієнтського програмного забезпечення є *редиректори (redirector), розподільники (designator) і імена UNC (UNC pathnames).*

Редиректори. Редиректор - мережеве програмне забезпечення, яке приймає запити введення/виведення для віддалених файлів, іменованих каналів або поштових слотів і далі перепризначає їх мережевим сервісам іншого комп'ютера. Редиректор перехоплює всі запити, що надходять від застосунків, та їх аналізує.

Тобто в операційній системі клієнта є програма переспрямування (*redirector*), яка знаходиться в оперативній пам’яті комп’ютера резидентно. Коли прикладна програма звертається із запитом до операційної системи комп’ютера, редиректор перехоплює запит, аналізує, хто його може виконати, і спрямовує або в ОС того ж комп’ютера, або в мережу, до сервера, якому адресовано цей запит. При цьому користувач не бачить, до яких ресурсів (свого комп’ютера чи мережі) він звертається, і клієнтська частина перетворює запит з локальної форми у мережевий формат (\\server\dir1\file) та передає його транспортній підсистемі, яка відповідає за доставку повідомлень вказаному серверу.

У свою чергу, серверна частина операційної системи сервера приймає цей запит, перетворює його в локальний формат (наприклад, c:, o:, z:, lpt1:) і передає для виконання своїй локальній ОС. Після того, як результат одержаний, сервер звертається до транспортної підсистеми і направляє відповідь клієнту, що видав запит. Клієнтська частина перетворює результат у відповідний формат та адресує його тому застосунку, який видав запит.

Розподільники. Розподільник (*designator*) являє собою частину програмного забезпечення, що управляє присвоєнням букв накопичувача (*drive letter*) як локальним, так і віддаленим мережевим ресурсам або розділюваним дисководам, що допомагає у взаємодії з мережевими ресурсами.

Коли між мережевим ресурсом і буквою локального накопичувача створена асоціація, відома також як відображення дисковода (mapping a drive), розподільник відстежує присвоєння такої літери дисководу мережевому ресурсу. Потім, коли користувач або застосунок отримають доступ до диску, розподільник замінить букву дисковода на мережеву адресу ресурсу, перш ніж запит буде посланий редиректору.

Імена UNC (*Universal Naming Convention - Універсальна угода по найменуванню)*. UNC - це стандартний засіб іменування мережевих ресурсів. Ці імена мають форму *\\Ім'я_сервера\ім'я_ресурсу*. З UNC здатні працювати застосунки та утиліти командного рядка, які використовують імена UNC замість відображення мережевих дисків.

Підходи до побудови мережевих операційних систем

Перші мережеві ОС були сукупністю існуючої локальної ОС і надбудованої над нею мережевої оболонки. При цьому до локальної ОС вбудовувався мінімум мережевих функцій, необхідних для роботи мережевої оболонки, яка виконувала основні мережеві функції. Прикладом такого підходу є використання на кожній машині мережі операційної системи MS DOS (у якої, починаючи з її третьої версії, з'явилися такі вбудовані функції, як блокування файлів і записів (блокування потрібне як для колективного доступу багатьох машин до файлів на сервері, так і для доступу багатьох одночасно виконуваних завдань до локальних файлів). *Принцип побудови мережевих ОС у вигляді мережевої оболонки над локальною ОС* використовується і в сучасних ОС, таких, наприклад, як *LANtastic* (над ОС OS/2), *Personal Ware* (над DOS 7), NetWare (клієнти).

Проте ефективнішим є шлях розробки операційних систем, які з самого початку призначені для роботи в мережі. Мережеві функції у ОС такого типу глибоко вбудовані в основні модулі системи, що забезпечує їх логічну побудову, простоту експлуатації і модифікації, а також високу продуктивність.

Прикладом такої ОС є система Windows NT фірми Microsoft (*Windows NT Workstation та Windows NT Server)*, яка за рахунок вбудованості мережевих засобів забезпечує вищі показники продуктивності і захищеності інформації у порівнянні з мережевою ОС LAN Manager тієї ж фірми (сумісна розробка з IBM), надбудовою, що є, над локальною операційною системою OS/2. Також до систем такого типу належать *NetWare (Server) for UNIX, Solans, HP-UX.*

Таким чином, розрізняють ОС із *вбудованими мережевими функціями і оболонки над локальними ОС*. За іншою ознакою класифікації розрізняють мережеві ОС *однорангові і функціонально несиметричні (для систем "клієнт / сервер").*

Однорангова мережева ОС і ОС з виділеними серверами

Залежно від того, як розподілені функції між комп'ютерами мережі, мережеві операційні системи, а отже, і мережі поділяються на два класи: *однорангові і дворангові,* а також багаторангові мережі.

Дворангові мережі частіше називають мережами з виділеними серверами.

Якщо комп'ютер надає свої ресурси іншим користувачам мережі, то він виступає як сервер. При цьому комп'ютер, що вдається до послуг іншої машини, є клієнтом. Як вже було сказано вище, комп'ютер, що працює в мережі, може виконувати функції або клієнта, або сервера, або суміщати обидві ці функції.

В однорангових мережах всі комп'ютери рівні в правах доступу до ресурсів один одного. Кожний користувач може за своїм бажанням оголосити будь-який ресурс свого комп'ютера таким, що розділяється, після чого інші користувачі можуть його експлуатувати. Тобто в одноранговій мережі користувач одночасно є клієнтом і сервером. В таких мережах на всіх комп'ютерах встановлюється одна і та ж ОС, яка надає всім комп'ютерам в мережі потенційно рівні можливості. Однорангові мережі можуть бути побудовані, наприклад, на базі ОС LANtastic, Personal Ware, Windows for Workgroup, Windows NT Workstation.

В однорангових мережах відсутня спеціалізація ОС залежно від переважаючої функціональної спрямованості - клієнта або сервера. Всі варіації реалізуються засобами конфігурації одного і того ж варіанту ОС.

Однорангові мережі зазвичай простіше в організації і експлуатації, проте вони застосовуються в основному для об'єднання невеликих груп користувачів (до 10 комп’ютерів), що не пред'являють великих вимог до обсягів інформації, що зберігається, її захищеності від несанкціонованого доступу і до швидкості доступу. При підвищених вимогах до цих характеристик більш відповідними є дворангові мережі, де сервер краще вирішує задачу обслуговування користувачів своїми ресурсами, оскільки його апаратура і мережева операційна система спеціально спроектовані для цієї мети.

У дворангових системах, окрім клієнтської та серверної частини робочих станцій, окремо виділяються сервери (називаються *виділені сервери*) для виконання

серверних функцій, які надають у загальне користування іншим комп’ютерам, а саме:

- *файл-сервер* (надання файлів у загальне користування решті усім користувачам мережі);
- *сервер бази даних* (СУБД MS SQL Server, Oracle, MySQL), засіб для зберігання, доступу, обробки інформації;
- *факс-сервер* (організація спільного використання факсу);
- *принт-сервер* або сервер друку (організація спільного використання принтера декількома комп’ютерами);
- *сервер робочої групи* (поєднує в собі можливості файлового сервера, сервера застосунків, бази даних, принт/факс серверів, поштового та інших) що забезпечує багатофункціональне рішення для групи комп’ютерів (не менше 20);
- *сервер віддаленого доступу* (забезпечує віддалену роботу клієнтів, які підключені до серверу так, як наче вони працюють з сервером у локальній мережі);
- *контролер домену* (*Domain Controller server*) - головний комп'ютер в локальній мережі, який має ієрархічну структуру - домен. Через контролер домену здійснюється централізоване управління ресурсами домену - обліковими записами комп'ютерів і користувачів. За допомогою служби директорій *Active Directory* він зберігає дані про користувачів і здійснює їх аутентифікацію для доступу до ресурсів локальної мережі. Працює під управлінням серверних ОС від MS Windows, починаючи з Windows 2000 Server. Контролер домену може виконувати роль файлового сервера і сервера друку;
- *поштовий сервер (mail server),* або сервер електронної пошти розпізнає адресу вхідних повідомлень електронної кореспонденції та розподіляє її по скринькам у локальній мережі, а також здійснює відправку вихідної інформації;
- *сервери FTP* - невід'ємна частина технічного забезпечення Всесвітньої Павутини. Їх завдання - переміщати файли за запитом простих файлових менеджерів за допомогою стандартного протоколу *File Transfer Protocol*;
- *проксі-сервер* - посередник між користувачами локальної мережі та Інтернетом. Забезпечує безпечний вихід в інтернет, захищаючи від небажаного доступу ззовні і за необхідності обмежуючи вихід на певні ресурси користувачам локальної мережі. Крім того, виконує ряд інших функцій: облік і економія трафіку шляхом стиснення даних, кешування, анонімізація доступу;
- *Web-сервер (сервер web- застосунків*) - спеціально виділений комп'ютер, який відповідає за доступ до сайту кампанії користувачів Інтернету, коректне і швидке відображення статичних або динамічних сторінок.

Прикладом ОС, орієнтованої на побудову мережі з виділеним сервером, є операційна система Windows NT. На відміну від NetWare, обидва варіанти даної мережевої ОС - Windows NT Server (для виділеного сервера) і Windows NT Workstation (для робочої станції) - можуть підтримувати функції і клієнта, і сервера. Але серверний варіант Windows NT має більше можливостей для надання ресурсів свого комп'ютера іншим користувачам мережі, оскільки може виконувати ширший набір функцій, підтримує більшу кількість одночасних з'єднань з клієнтами, реалізує централізоване управління мережею, має розвиненіші засоби захисту.

В мережі з виділеним сервером усі комп'ютери в загальному випадку можуть виконувати одночасно ролі і сервера, і клієнта, ця мережа функціонально не симетрична: апаратно і програмно в ній реалізовано два типи комп'ютерів:

- одні, більшою мірою орієнтовані на виконання серверних функцій і такі, що працюють під управлінням спеціалізованих серверних ОС,

- а інші, в основному, виконують клієнтські функції і працюють під управлінням відповідного цьому призначенню варіанту ОС.

Функціональна несиметричність, як правило, викликає і несиметричність апаратури - для виділених серверів використовуються потужніші комп'ютери з великими обсягами оперативної і зовнішньої пам'яті, особливо для виділених серверів віддаленого доступу. Таким чином, функціональна несиметричність в мережах з виділеним сервером супроводжується несиметричністю операційних систем (спеціалізація ОС) і апаратною несиметричністю (спеціалізація комп'ютерів).

Основні функції мережевої ОС:

- управління каталогами та файлами;
- управління ресурсами;
- комунікаційні функції;
- захист від несанкціонованого доступу;
- забезпечення відмовостійкості;
- управління мережею [8, 12].

Управління каталогами та файлами є однією з першочергових функцій мережевої операційної системи, яка обслуговується спеціальною мережевою файловою підсистемою. Користувач отримує від цієї підсистеми можливість звертатися до файлів, фізично розташованих на сервері або в іншій станції даних, застосовуючи звичні для локальної роботи мовні засоби. При обміні файлами повинен бути забезпечений необхідний рівень конфіденційності обміну (секретності даних).

Управління ресурсами включає запити і надання ресурсів.

Комунікаційні функції забезпечують адресацію, буферизацію, маршрутизацію.

Захист від несанкціонованого доступу можливий на будь-якому з наступних рівнів: обмеження доступу в певний час, і (або) для певних станцій, і (або) певне число раз; обмеження сукупності доступних конкретному користувачеві директорій; обмеження для конкретного користувача списку можливих дій (наприклад, тільки читання файлів); позначка файлів символами типу "тільки читання", "прихованість при перегляді списку файлів".

Відмовостійкість визначається наявністю в мережі автономного джерела живлення, відображенням або дублюванням інформації в дискових накопичувачах. Відображення полягає в зберіганні двох копій даних на двох дисках, підключених до одного контролера, а дублювання означає підключення кожного з цих двох дисків до різних контролерів. Мережева ОС, яка реалізує дублювання дисків, забезпечує більш високий рівень відмовостійкості.

Мережеві служби і мережеві сервіси

Сукупність серверної і клієнтської частин ОС, що надають доступ до конкретного типу ресурсу комп'ютера через мережу, називається *мережевою службою*. Наприклад, клієнтська і серверна частини ОС, які спільно забезпечують доступ через мережу до файлової системи комп'ютера, утворюють файлову службу. Тобто *служба є мережевим компонентом, який реалізує певний набір послуг.*

Послуги, які мережева служба надає користувачам мережі, називається *мережевим сервісом.* Тобто сервіс є інтерфейсом між споживачем послуг і самою службою.

Кожна служба пов'язана з певним типом мережевих ресурсів і певним способом доступу до цих ресурсів [8, 12].

Наприклад:

- служба друку забезпечує доступ до принтерів, що розділяються, і надає сервіс друку;
- поштова служба надає доступ до інформаційного ресурсу мережі - електронним листам.

Способом доступу відрізняється, наприклад, служба віддаленого доступу – вона надає користувачам доступ до всіх ресурсів мережі через комутовані телефонні канали.

Серед мережевих служб виділяються:

- служби, орієнтовані на користувача;
- служби, орієнтовані на адміністратора.

Служби, орієнтовані на адміністратора, використовуються для організації роботи мережі.

Наприклад:

служба каталогів (централізована довідкова служба), призначена для ведення бази даних про всіх користувачів мережі, в деяких системах – і для бази програмних і апаратних компонентів мережі (наприклад, NDS компанії Novell; StreetTalk компанії Banyan);

служба моніторингу мережі дозволяє захоплювати і аналізувати мережевий трафік;

служба безпеки (частиною її є логічний вхід з перевіркою пароля);

служба резервного копіювання і архівації.

Мережева служба може бути представлена в ОС або обома (клієнтською і серверною) частинами, або тільки однією з них.

Розподілені операційні системи

Розподілена ОС являє собою єдину операційну систему в масштабах обчислювальної системи. Кожний комп'ютер мережі, що працює під управлінням розподіленою ОС, виконує частину функцій цієї глобальної ОС. Розподілена ОС об'єднує усі комп'ютери мережі в тому сенсі, що вони працюють в тісній кооперації один з одним для ефективного використання всіх ресурсів комп'ютерної мережі.

Розподілена організація операційної системи дозволяє спростити роботу користувачів і програмістів в мережевих середовищах. У розподіленій ОС

реалізовані механізми, які дають можливість користувачеві представляти і сприймати мережу у вигляді традиційного однопроцесорного комп'ютера.

Характерними ознаками розподіленої організації ОС є:

- наявність єдиної довідкової служби ресурсів, що розділяються;
- єдиної служби часу;
- використання механізму виклику віддалених процедур (RPC) для прозорого розподілу програмних процедур по машинах;
- багатонитяної обробки, що дозволяє розпаралелювати обчислення в рамках одного завдання і виконувати це завдання відразу на декількох комп'ютерах мережі;
- наявність механізмів реплікації, транзакції;
- наявність інших розподілених служб.

Класичним прикладом розподіленої системи є система перетворення символьних імен в мережеві IP-адреса DNS (*Domain Name System* - система доменних імен – комп'ютерна розподілена система для отримання інформації про домен). Система імен – організована ієрархічно розподілена система, з дублюванням всіх функцій між двома і більш серверами.

Запит користувача на перетворення імені (наприклад, w3c.org) в мережеву адресу передається серверу розпізнавання імен постачальника послуг інтернету. Сервер розпізнавання імен по черзі опитує сервери з ієрархії служби імен. Опитування починається з кореневих серверів, який повертає адреси серверів, відповідальних за зону домена. Потім опитується сервер, що відповідальний за зону (в даному випадку – *.org*), повертає адреси серверів, відповідальних за домен другого рівня, і так далі. Сервери імен кеширують інформацію про відповідність імен і адрес для зменшення навантаження на систему. Програмне забезпечення на комп'ютері користувача зазвичай має можливість з'єднатися з як мінімум двома різними серверами розпізнавання імен.

В системі розпізнавання імен відсутні механізми забезпечення безпеки. Це призводить до регулярних атак на сервери імен в надії вивести їх з ладу, наприклад, великою кількістю запитів.

Реплікація

Розподілені системи часто забезпечують реплікацію (тиражування) файлів як однієї з послуг, що надаються клієнтам. *Реплікація - це асинхронне перенесення змін даних* початкової файлової системи у файлові системи, що належать різним вузлам розподіленої файлової системи. Іншими словами, система оперує декількома копіями одного і того файлу (репліки), причому кожна копія знаходиться на окремому файловому сервері і при цьому забезпечується автоматичне узгодження даних в копіях файлу. Про існування реплік відомо всім комп'ютерам мережі. Є декілька причин для надання цього сервісу, головними з яких є:

1. Збільшення надійності (при відмові одного серверу файл залишається доступним на іншому сервері за рахунок наявності незалежних копій кожного файлу на різних файл-серверах).

2. Розподіл навантаження між декількома серверами (знижується навантаження на файлові сервери за рахунок розподілу навантаження між декількома серверами, оскільки клієнти можуть звертатися до даних реплікованого файлу на найближчий файловий сервер).

Ключовим питанням, пов'язаним з реплікацією, є прозорість.

В одних системах користувачі повністю обізнані про те, що їх файли реплікуються і залучені в управління реплікацією. В інших системах реплікація виконується повністю автоматично. Система при цьому називається реплікаційно прозорою. Прозорість реплікації залежить від двох чинників:

- використовуваної схеми іменування реплік;
- ступені залученості користувача в управління реплікацією.

Іменування реплік

Система іменування, яка відображає ім'я файлу на його мережевий ідентифікатор, що однозначно визначає місце зберігання файлів, дозволяє реалізувати прозорість доступу до файлу, що реплікується. Реалізується така схема таким чином.

Використовується централізована довідкова служба, яка дозволяє зберігати відображення імен файлів на їх мережеві ідентифікатори (наприклад, IP-адреса серверів). Файлу привласнюється ім'я, що не містить старшої частини, відповідного імені комп'ютера. У довідковій службі цьому імені відповідає декілька ідентифікаторів, які вказують на сервери, що зберігають репліки файлу. При зверненні до файлу застосунок використовує ім'я, а довідкова система повертає йому один з ідентифікаторів, який вказує на сервер, що зберігає репліку.

Схема реалізується просто для незмінних файлів (репліки завжди ідентичні).

Для реалізації повністю прозорого доступу до змінних реплікованих файлів потрібне ведення бази, що зберігає відомості про те, які з реплік містять останню версію даних, а які ще не оновлені.

В сучасних мережевих файлових системах реалізована схема іменування, при якій потрібна явне зазначення імені сервера при зверненні до файлу, тобто непрозора система реплікації.

Управління реплікацією

Під управлінням реплікації мається на увазі визначення кількості реплік і вибір серверів для зберігання кожної репліки.

У *прозорій системі* реплікації такі рішення ухвалюються автоматично при створенні файлу на основі правил стратегії реплікації, визначених заздалегідь адміністратором системи, – неявна реплікація. Застосунок не вказує місце розміщення файлу. Файлова система самостійно вибирає сервер для розміщення першої репліки файлу. Потім у фоновому режимі система створює ще декілька реплік файлу.

У *непрозорій системі* рішення ухвалюються за участю користувача, який створює файли, або розробника застосунку, якщо застосунок створює файли, - явна реплікація. Для кожної репліки явно указується сервер. Згодом за бажанням можна видалити репліку.

Узгодження реплік

Це одне з найбільш важливих питань при розробці системи реплікації.

Коли дані в одній з реплік модифікуються, необхідно розповсюдити модифікацію на інші репліки.

Існує декілька способів забезпечення узгодженості реплік:

- читання будь-якої – запис в усі (UNIX). Читання виконується з будь-якої копії. При запису у файл всі репліки блокуються, виконується запис у кожну копію, блокування знімається. Недолік – не можна провести запис у файли на непрацездатних серверах;
- запис у доступні – читання виконується з будь-якої копії. Будь-який сервер, що зберігає репліку файлу, після перезавантаження повинен з'єднатися з іншим сервером і отримати оновлену репліку;
- первинна репліка – запис дозволяється тільки в одну репліку, яка називається первинною. Усі інші називаються вторинними. З вторинних можна тільки читати. Після модифікації первинної, всі сервери з вторинними репліками повинні зв'язатися з сервером, що зберігає первинну, і отримати оновлення. Недолік – низька надійність (при відмові первинного сервера неможливі модифікації файлу);
- кворум – узагальнення попередніх підходів.

Сучасні розподілені файлові системи

Розподілені файлові системи забезпечують здатність спільно використовувати диски, каталоги, і файли по мережі – це одне з найбільш значних досягнень сучасних інформаційних технологій. Ця здатність може істотно скоротити вимоги до дискового простору комп'ютерів і полегшити спільну роботу користувачів. Комп'ютери з *Microsoft Windows і MacOS Apple/MacOS X* використовують для цього механізм сумісного використання дисків і директорій.

В системах *Linux/Unix* для тих же самих завдань традиційно використовується NFS – мережева файлова система. NFS – це найвідоміший механізм сумісного доступу до файлів для Linux та інших Unix-систем, тому що він присутній в багатьох Unix-подібних системах і дуже простий в налаштуванні. NFS підтримується ядром Linux, і утиліти, пов'язані з NFS, присутні в кожному дистрибутиві. Але в світі Linux існують і сучасніші механізми для сумісного використання файлів і каталогів. Кожен з них має певні переваги в настройці або у використанні.

Розподілена файлова система *OpenAFS – це Open Source*-аналог відомої комерційної розподіленої файлової системи AFS. Підтримка для розподілених файлових систем *InterMezzo* і *Coda* вже присутній в нових ядрах Linux з серії 2.4. Нові механізми сумісного використання файлів, засновані на Web (наприклад WebDAV), теж можуть використовуватися як файлові системи.

Розподілені файлові системи OpenAFS і Coda мають власні механізми управління розділами, які спрощують можливості зберігання загальнодоступної інформації. Вони так само підтримують дублювання – здатність робити копії

розділів і зберігати їх на інших файлових серверах. Якщо один файловий сервер стає недоступним, то все одно до даних, що зберігаються на його розділах, можна дістати доступ за допомогою наявних резервних копій цих розділів.

Найголовніша відмінність між підходом Windows/MacOS (сумісне використання каталогів і дисків) і підходом Linux, MacOS X та інших Unix-подібних багатокористувацьких операційних систем – в тому, як ці операційні системи використовують і організовують розділи. Windows/MacOS експортують розділи як окремі каталоги або диски, і віддалені системи, які хочуть звернутися до загальнодоступних пристроїв, повинні обов'язково підключити їх до себе.

Коли найвищий рівень організації у файловій системі – це розділ диска (наприклад, як у файлових системах Windows), робочі станції клієнтів для отримання доступу до цих даних повинні обов'язково підключитися до розділу і призначити йому окрему букву в своїй локальній розкладці (наприклад, диск E, F, G, і т.д). Букви можуть бути призначені мережевим розділам в призначених для користувача і групових профілях Windows (для стандартизації). Але, на жаль, не на всіх комп'ютерах розташування букв може бути однаковим. Наприклад, на комп'ютері з великою кількістю жорстких дисків і розділів потрібні букви можуть бути зайняті, і тому доведеться давати мережевим розділам інші позначення.

Навпаки, файлова система Unix – це ієрархічна файлова система, до якої додаткові розділи додаються за допомогою монтування їх до існуючої директорії. Це дозволяє ефективно додати будь-яке джерело даних в будь-яку існуючу файлову систему. Якщо ви вмонтовуєте нове джерело інформації до каталога, що є частиною розподіленої файлової системи, він відразу ж стає доступним всім клієнтам цієї розподіленої системи.

Сучасні розподілені файлові системи типу OpenAFS або Coda включають спеціальні сервіси для управління розділами. Це дозволяє вам змонтувати розділи різних файлових серверів в центральну ієрархію директорій, підтримувану файловими системами. OpenAFS використовує центральний каталог, так званий «/afs», а Coda використовує «/coda». Ці ієрархії директорій доступні всім клієнтам розподіленої файлової системи, і виглядають однаково на будь-якій з клієнтських робочих станцій. Це дає можливість користувачам працювати зі своїми файлами однаково на будь-якому комп'ютері. Якщо ваш настільний комп'ютер не працює, ви абсолютно спокійно можете використовувати будь-який інший – всі ваші файли знаходяться в безпеці на сервері.

Розподілені файлові системи, що надають одні і ті ж дані багатьом різним комп'ютерним системам, дають користувачам можливість використовувати будь-яку операційну систему, краще всього відповідну для їх завдань. Користувачі Macintosh можуть користуватися всіма перевагами графічних інструментальних засобів, доступних в Mac OS, і одночасно зберігати свої дані на централізованих файлових серверах. Користувачі Windows так само можуть мати доступ до стійкої глобальної файлової системи. Розподілені файлові системи особливо привабливі при спробі координації роботи між групами, розташованими в різних містах, державах, або навіть в різних країнах. Перевага – загальні дані завжди доступні по мережі, незалежно від вашого місцезнаходження.

Розширення файлових систем за допомогою Web

До створення розподілених файлових систем сумісне використання файлів через мережу обмежувалося простими передачами файлів за допомогою використання протоколу передачі файлів – FTP (*File Transfer Protocol*). Поява Всесвітньої павутини в значній мірі спростила процес роботи з FTP – тепер не потрібно знати команди, тому що протокол FTP інтегрований в більшість браузерів. Здатність легко передавати файли через Web також призвела до розширення Павутини й істотного поліпшення основного протоколу передачі гіпертексту – HTTP (*HyperText Transfer Protocol*), який зараз є підставою для багатьох систем розподіленого використання файлів.

Найвідоміша з них – це *WebDAV*, яка розшифровується як «Web-система розподіленої авторизації і контролю версій» (*Web-enabled Distributed Authoring and Versioning).* WebDAV– це набір розширень до протоколу HTTP, що забезпечує сумісне середовище для користувачів, яке дозволяє їм викачувати, упорядковувати і редагувати файли, що зберігаються на Web-серверах.

Підтримка WebDAV вбудована в багато популярних Web-серверів, наприклад, *Apache*, що ґрунтуються на пізнавальних механізмах сервера. (Від простих файлів .htaccess до інтегрованих NIS, LDAP, або навіть механізму аутентифікації Windows). Використання WebDAV для доступу і модифікації файлів через Web вбудоване в операційні системи MacOS X, в нові версії *Microsoft Internet Explorer*, а так само доступно і в *Linux* при використанні таких застосувань, як менеджер файлів *Nautilus*. Хоча це і не файлова система в традиційному сенсі, але ви можете навіть змонтувати WebDAV в Linux, використовуючи завантажуваний модуль ядра під назвою *davfs*.

WebDAV забезпечує такі стандартні для розподілених систем можливості, як блокування файлів, створення, перейменування, копіювання, видалення файлів, а так само підтримує такі просунуті можливості, як метадані файлу (докладніша інформація про файл – заголовок, тема, ким створений, і т.д). В найближчому майбутньому WebDAV включатиме інтегровану підтримку управління версіями, яка спростить роботу багатьох користувачів над загальними файлами, відстежуючи зміни, авторів цих змін, і інші аспекти загального використання документа. Ці можливості контролю над версіями забезпечуються відповідно до протоколу DELTAV, який активно розробляється Робочою групою DeltaV, – підрозділу Проектувальної групи Інтернету (IETF – *Internet Engineering Task Force).* Деякі проекти, наприклад, Subversion (WEBDAV і DeltaV-заснована на заміні стандарту CVS), вже доступні в альфа-версії. Subversion забезпечує систему контролю над версіями і збереження архіву файлу на основі бази даних, API мови C, і моделює версійну файлову систему, легко доступну через Web.

Можливості операційних систем Windows Server та Windows Server 2019

Системні вимоги, які висуваються до серверу при його експлуатації, як правило, враховують наступне:

- визначитися як саме функції має виконувати сервер: це сервер мережевої інфраструктури, сервер служби каталогів, файл-сервер, принт-сервер, сервер

віддаленого доступу, поштовий сервер, сервер баз даних, сервер застосунків, web-сервер і т.д.);

- спрогнозувати навантаження на сервер;
- визначитися з конфігурацією сервера (тип та кількість процесорів, обсяг оперативної пам'яті, параметри дискової підсистеми, мережеві адаптери й ін.);
- визначитися з типом операційної системи ((Standard, Enterprise, Datacenter, Web) та процедурою установки (подальшою модернізацією) та налаштування системи.

Microsoft Windows Server – найпотужніша операційна для ПК, в якій реалізовані засоби управління системою й адміністрування:

Active Directory – розширювана і масштабована служба каталогів, в якій використовується простір імен, заснований на стандартній Інтернет-службі іменування доменів (*Domain Name System*, *DNS*);

IntelliMirror – засоби конфігурації, які підтримують дзеркальне відображення даних користувача, параметри середовища, адміністрування, установку та обслуговування програмного забезпечення;

Terminal Services – служби терміналів, які забезпечують віддалений вхід в систему й управління іншими системами Windows Server 2003;

Windows Script Host – сервер сценаріїв Windows для автоматизації таких поширених завдань адміністрування, як створення облікових записів користувачів і звітів по журналах подій.

Операційна система Windows Server 2019

У жовтні 2018 року була випущена ОС Windows Server 2019. Починаючи з Windows Server 2016 був прийнятий новий цикл виходу релізів. На даний час є два канали поширення:

- LTSC (*Long-term servicing channel*) - реліз, який виходить через 2-3 роки, з 5-річною основною та 5-річною розширеною підтримкою,

- **Semi-Annual Channel** - релізи, які виходять кожні півроку, мають основний цикл підтримки протягом 6 місяців і розширену підтримку протягом 18 місяців.

Для чого потрібні ці два канали? Microsoft активно впроваджує нововведення в свою хмарну платформу *Azure*. Це підтримка віртуальних машин *Linux*, контейнери з Linux і Windows, і багато інших технології.

Замовники, які використовують ці технології в хмарі, також хочуть їх використовувати і в своїх датацентрах (центрах обробки даних). **Semi-Annual Channel** скорочує розрив у можливостях між Azure і локальними датацентрами.

Піврічні релізи призначені для динамічних у розвитку компаній, які перейшли до гнучкої сервісної моделі надання IT-послуг бізнесу.

Релізи **LTSC** призначені для компаній, які використовують сталі застосунки з тривалим циклом підтримки, наприклад, *Exchange Server, SharePoint Server, SQL Server*, а також інфраструктурні ролі, програмно визначені датацентри та гіперконвергентна інфраструктура (гіперконвергентні рішення поєднують в собі обчислювальні потужності, високошвидкісні системи зберігання, мережеву інфраструктуру і керуються за допомогою одного програмного продукту).

Windows Server 2019 - це саме реліз в каналі LTSC, який містить в собі усі оновлення функціоналу з Windows Server 2016 і наступних піврічних релізів.

Основні зусилля розробників Windows Server 2019 були спрямовані на чотири ключові області:

Гібридна хмара - Windows Server 2019 та новий центр адміністрування *Windows Admin Center* дозволяють легко використовувати спільно з серверною операційною системою хмарні служби *Azure: Azure Backup, Azure Site Recovery*, управління оновленнями *Azure, Azure AD Authentication* й іншими.

Безпека - є одним з найважливіших пріоритетів для замовників. Windows Server 2019 має вбудовані можливості для скрути проникнути зловмисникам і закріпитися в системі. Це відомі по Windows 10 технології *Defender ATP (служба безпеки - Advanced Threat Protection в захиснику* Windows) та *Defender Exploit Guard (захист від експлойтів – захист від зараження вашої системи шкідливими програмами).* Тобто розширений захист від загроз Azure або Azure ATP, а саме, хмарне рішення безпеки, яке використовує сигнали локальної служби Active Directory для виявлення і аналізу складних загроз.

Платформа застосунків - контейнери стають сучасним трендом для упаковки і доставки застосунків в різні системи. При цьому Windows Server може виконувати не тільки розроблені для Windows програми, а й застосунки в Linux. Для цього в Windows Server 2019 є контейнери Linux, *підсистема Windows для Linux (WSL),* а також значно знижені обсяги образів контейнерів.

Гіперконвергентна інфраструктура – дозволяє в рамках одного сервера стандартної архітектури поєднати обчислення і сховище. Цей підхід значно знижує вартість інфраструктури, при цьому забезпечуючи відмінну продуктивність і масштабованість.

Windows Admin Center (WAC) – це новий засіб адміністрування серверів, який встановлюється локально в інфраструктурі і дозволяє адмініструвати локальні і хмарні екземпляри Windows Server, комп'ютери Windows 10, кластери і гіперконвергентну інфраструктуру.

WAC доповнює, а не замінює існуючі засоби адміністрування, такі як консолі *mmc, Server Manager.* Консоль управління (mmc) використовується для створення, збереження і відкриття засобів адміністрування, які називаються консолями, що керують апаратними, програмними і мережевими компонентами операційної системи Microsoft Windows. MMC працює у всіх клієнтських операційних системах, які в даний час підтримуються. Підключення до WAC здійснюється з браузера.

Для виконання завдань використовується технології віддаленого управління WinRM (*Windows Remote Management* - служба віддаленого управління для операційних систем Windows), WMI (*Windows Management Instrumentation* - технологія, яка за допомогою єдиного інтерфейсу дозволяє управляти компонентами як локальної, так і віддаленої операційної системи) і скрипти PowerShell (оболонка.

Системна аналітика. Windows Server 2019 став інтелектуальнішим. За допомогою нової функції *System Insights* реалізується прогнозна аналітика, що дозволяє перейти від реактивного до проактивного управління серверами. Модель

машинного навчання враховує лічильники продуктивності та події для точного прогнозування проблем з вільним місцем на дисковій підсистемі, визначення трендів для процесорних обчислень, мережевій взаємодії та продуктивності сховища.

Новинки в підсистемі зберігання. ***Storage Migration Service*** **–** це нова технологія для переміщення даних зі старих серверів на нові.

Міграція даних відбувається в декілька етапів:

- інвентаризація даних на різних серверах;
- швидке перенесення файлів, мережевих папок і конфігурацій безпеки з вихідних серверів;
- захоплення управління і підміна ідентифікатора сервера і налаштувань мережі зі старого сервера на новий.

Сервіс *Azure File Sync* трансформує традиційні файлові сервери і розширює обсяг зберігання до практично недосяжних у реальному житті обсягів. Дані розподіляються по декількох рівнях: гарячий кеш - це дані, що зберігаються на дисках файлового сервера і доступні з максимальною швидкістю для користувачів. У міру зменшення пріоритетності дані непомітно переміщуються в Azure. *Azure File Sync* можна використовувати спільно з будь-якими протоколами для доступу до файлів: SMB, NFS і FTPS.

Storage Replica (**реплікація локального сховища – дискового тома.)** Це технологія захисту від катастроф, яка вперше з'явилася в Windows Server 2016. Тобто дані одного тома автоматично синхронізуються по мережі на вторинний сервер, на якому буде доступна ідентична копія тома.

***Storage Spaces Direct*.** Локальні дискові простори – це необхідний компонент для побудови гіперконвергентної інфраструктури і масштабування файл-сервера.

Зміни у відмовостійкій кластеризації. У Windows Server 2019 впроваджені набори кластерів (*Cluster sets*), що збільшують масштабованість до сотень вузлів. Обчислювальний кластер - це набір комп'ютерів (обчислювальних вузлів), об'єднаних деякою комунікаційною мережею. Кожний обчислювальний вузол має свою оперативну пам'ять, працює під управлінням своєї операційної системи. Найбільш поширеним є використання однорідних кластерів, тобто таких, де всі вузли абсолютно однакові за своєю архітектурою та продуктивністю.

Платформа застосунків – підтримка контейнерів. З’явилася можливість запускати контейнери на основі Windows і Linux на одному і тому ж вузлі контейнера за допомогою єдиної керуючої програми *Docker*. Це дозволяє працювати у різнорідному середовищі вузлів контейнерів і надавати розробникам гнучкість у створенні застосунків. Тепер можна запускати робочі столи, виконувані бінарні файли і bash-скрипти Linux, не вдаючись до використання віртуалізованого середовища. Це стало можливо завдяки спеціального прошарку *Windows Subsystem for Linux* (WSL) на рівні ядра операційної системи. WSL здійснює трансляцію системних викликів Linux у виклики Windows. Вперше така можливість з'явилася в ОС Windows 10, а тепер розробники запровадили її і у серверній операційній системі.

Контрольні питання:

1. Що таке мережева операційна система?
2. Що таке розподілена операційна система?
3. Яка структура мережевої операційної системи?
4. Які вам відомі підходи до побудови мережевих операційних систем?
5. Що таке однорангова мережева операційна система та її особливості?
6. Що таке дворангова операційна система?
7. Наведіть виділені сервери у дворангових операційних системах та їх призначення.
8. Які основні функції мережевої операційної системи?
9. Що таке мережева служба?
10. Що таке мережевий сервіс?
11. Наведіть відомі вам мережеві служби.
12. Наведіть характерні ознаки розподіленої операційної системи.
13. Що таке Domein Name System?
14. Що таке реплікація даних?
15. Як здійснюється управління репліками?
16. Назвіть сучасні розподілені файлові системи.

СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ

1. Tanenbaum Andrew s., Bos Herbert. Modern Operating Systems. Vrije Universiteit Amsterdam, The Netherlands. Publisher: Pearson India; 4th edition. 2016.P.1137.

2. Yosifovich P., Ionescu A., Russinovich R., Solomon D. Windows Internals Seventh Edition Part 1: System architecture, processes, threads, memory management, and more, Seventh Edition. Publisher(s): Microsoft Press. 2017. 1120 P.

3. Allievi A., Russinovich M.,Ionescu A., Solomon D. Windows Internals, Part 2 (Developer Reference) 7th Edition. Publisher : Microsoft Press. 2021. 912 P.

4. Alekseev V., Matveev M. Windows 10 na primerakh. Praktika, praktika i tolko praktika. Publisher : Nauka i tehnika. 2018. 272 P.

5. Deitel H., Deitel P. Operating Systems Publisher : Prentice Hall. 2022. 1260 P.

6. Wilson A. Windows 10: New 2020 Complete User Guide to Learn Microsoft Windows 10 with 580 Tips & Tricks. Kindle Edition. 2019. 105 P.

7. Willensburty A. WINDOWS 10 : 2021 User Learning Guide to Master the Operating System Of Windows 10 with Shortcuts and Tips & Tricks. Kindle Edition. 2021. 120 P.

8. Olifer N.A. Olifer V.G. Network Operating System. Russian Edition. 2009. 528 P.

9. Douglas Comer. The Cloud Computing Book: The Future of Computing Explained. Publisher : Chapman and Hall/CRC. 2021. 288 P.

10. Dan C. Marinescu. Cloud Computing: Theory and Practice. Publisher : Morgan Kaufmann. 2022. 672 P.

11. Abraham Silberschatz, Greg Gagne, Peter B. Galvin. Operating System Concepts. Publisher : Wiley. 1040 P.

12. Gerardus B. Network Operating System A Complete Guide – 2020. Edition Kindle Edition. 2019. 255 P.

13. Stallings W. Operating Systems: Internals and Design Principles. 9th Edition. Publisher : Pearson. 2017. 800 P.

14. Arpaci-Dusseau R., Arpaci-Dusseau A.Operating Systems: Three Easy Pieces. Publisher : Arpaci-Dusseau Books. 2015. 714 P.

15. Bott Ed., Stinson C. Windows 10 Inside Out 4th Edition. Publisher : Microsoft Press. 2020. 848 P.

16. Rathbone A. Windows 10 For Dummies. 4th Edition. Publisher : For Dummies; 4th edition. 2020. 464 P.

17. Abel P.IBM PC assembly language and programming. 5th ed. Prentice Hall. 2004. 545 P.

18. Yurov V.Assembler. Publisher : Book on Demand Ltd .2012. 642 P.