Міністерство освіти і науки України
Житомирський державний університет імені Івана Франка
Кафедра комп'ютерних наук та інформаційних технологій

Мосіюк О.О., Федорчук А.Л.

Операційні системи та системне програмування

Навчально-методичний посібник

Житомир
Вид-во ЖДУ ім. І. Франка
2022

УДК 004.75 : 004.9
М 81

Затверджено на засіданні вченої ради Житомирського державного університету імені Івана Франка, протокол № 3 від 04.02.2022р.

Рецензенти:

Катерина МОЛОДЕЦЬКА – керівник навчально-наукового центру інформаційних технологій Поліського національного університету, доктор технічних наук, професор.

Віталій ГУК – кандидат технічних наук, старший викладач кафедри програмного забезпечення автоматизованих систем Черкаського національного університету ім. Б. Хмельницького

Олена УСАТА – кандидат педагогічних наук, доцент, доцент кафедри комп'ютерних наук та інформаційних технологій Житомирського державного університету імені Івана Франка

М 81 **Мосіюк О. О., Федорчук А. Л.** Операційні системи та системне програмування: навчально-методичний посібник. Житомир: Вид-во ЖДУ ім. Івана Франка, 2022. 76 с.

Навчально-методичний посібник, що пропонується, може бути рекомендованим здобувачав вищого рівня освіти, а саме: студентам фізико-математичних факультетів спеціальності 122 Комп'ютерні науки, а також – 014.09 Середня освіта (Інформатика), 014.08 Середня освіта (Фізика), 015.39 Професійна освіта (Цифрові технології), 112 Статистика та інші, а також вчителям інформатики для підготовки до уроків та факультативних занять.

УДК 004.75 : 004.9
©Вид-во ЖДУ ім. Івана Франка, 2022
©Мосіюк О. О., Федорчук А. Л., 2022

ЗМІСТ

Вступ...4
ЧАСТИНА 1. ОПЕРАЦІЙНІ СИСТЕМИ...5
Поняття операційної системи...6
Історія розвитку операційних систем...6
Структура операційної системи:...7
Складові операційної системи:...8
Функції операційної системи:...8
Класифікація операційних систем...9
Основні операційні системи...9
Лабораторна робота 1...12
Лабораторна робота 2...22
Лабораторна робота 3...25
Лабораторна робота 4...31
Лабораторна робота 5...34
ЧАСТИНА 2. НИЗЬКОРІВНЕВА МОВА ПРОГРАМУВАННЯ ASSEMBLER....37
Низькорівнева мова програмування Assembler...38
Двійкова та шістнадцяткова системи числення...38
Адресація даних у пам’яті комп’ютера...39
Встановлення FASM на Debian-подібні дистрибутиви Linux...40
Синтаксис мови Assembler...40
Опис даних у Assembler...41
Позначення констант та міток...42
Базові інструкції мови Assembler...42
Рядкові операції...44
Інструкції керування процесом компіляції...45
Налаштування середовища для програмування на Assembler в операційній системі Linux...45
Перша програма на мові Assembler...47
Універсальна процедура для виведення тексту в термінал Linux...52
Виведення числа у консоль...54
Реалізація функцій конвертації натурального числа у рядок і навпаки, із рядка у натуральне число...56
Процедура для считування текстового рядка із терміналу...64
Створення системних бібліотек...65
Лабораторна робота 6...68
Лабораторна робота 7...69
Додаток 1...70
Додаток 2...71
Додаток 3...73

Вступ

Навчально-методичний посібник створено на базі освітньої компоненти «Операційні системи та системне програмування» освітньої-професійної програми «Сучасні інформаційні технології та програмування» фізико-математичного факультету Житомирського державного університету імені Івана Франка. Важливою частиною сучасної підготовки фахівців, які здобувають професію за спеціальністю 122 Комп'ютерні науки є вивчення основ роботи системного програмного забезпечення, зокрема операційних систем, а також розуміння тих процесів, що відбуваються під час виконання програм центральним процесором.

Основною метою представленого навчально-методичного посібника є розкриття основних понять, пов'язаних із роботою операційних систем, їх видів; навчання студентів працювати із ОС Linux і використовувати засоби віртуалізації, а також розкриття особливостей створення програм за допомогою низькорівневої мови програмування Assembler.

Загалом робота складається із двох частин, у яких розміщуються теоретичні матеріали та завдання до лабораторних занять та додатків.

Перша частина розкриває сутність поняття «операційна система», наводяться сучасні приклади найбільш популярних операційних систем, описується робота із такою програмою як Virtual Box та приділяється значна увага Debian-подібної системи Kubuntu.

Друга частина зосереджена на низькорівневій мови програмування Assembler, яка використовується для програмування драйверів та створення програм роботи мікроконтролерів, які мають обмежені обчислювальні можливості.

У додатках міститься важлива інформація про найактуальніші джерела інформації по тематиці, пов'язаній із операційними системами та системним програмуванням.

Окрім цього посібник може бути рекомендованим до вивчення студентам, які здобувають освіту за спеціальностями 014.09 Середня освіта (Інформатика), 014.08 Середня освіта (Фізика), 015.39 Професійна освіта (Цифрові технології), 112 Статистика та інші, а також вчителям інформатики для підготовки до уроків та факультативних занять..

ЧАСТИНА 1. ОПЕРАЦІЙНІ СИСТЕМИ

Поняття операційної системи

Операційна система – сукупність базових програм, що забезпечує керування апаратною складовою комп’ютера або віртуальної машини, організовує взаємодію з користувачем.

Два погляди на операційну систему (ОС):

- *ОС як розширена машина* (ОС приховує незручний інтерфейс апаратного забезпечення, заміняючи його інтерфейсом прикладного програмування; замість деталей доступу – абстракції);
- *ОС як менеджер ресурсів* (розподіл системних ресурсів на керування ними).

Види керування ресурсами:

- у часі (наприклад, ЦП, принтер)
- у просторі (наприклад, оперативна пам'ять, жорсткий диск)

Історія розвитку операційних систем

Розвиток операційних систем нерозривно пов'язаний з розвитком обчислювальної техніки в цілому. Апаратні та програмні складові змінювалися спільно та здійснювали взаємний вплив один на одного. Прорив у нових технічних можливостях апаратної складової обчислювальної техніки спричиняв розвиток ефективних шляхів вирішення у програмному забезпечені.

Передісторія. Чарльз Бебідж намагався створити «аналітичну машину» («машина Бебіджа»). Хоч машина так і не запрацювала, Бебідж усвідомлював, що під таку машину знадобляться певні програми. Для написання цих програм він найняв Аду Лавлейс, дочку Джорджа ***Байрона***. Згодом на її честь було названо мову програмування Ada.

Операційні системи. Зрозуміло, що на цьому етапі про жодні ОС ще не йдеться. Ідеться хіба що про зародження ідеї програмного забезпечення взагалі.

З появою ***I покоління (1945-1655 рр.)*** комп’ютерної техніки елементною базою були електронні лампи і комунікаційні панелі, які керували основними функціями машини. Програмування здійснювалось лише тільки машинною мовою. Операційних систем не було. Комп’ютер використовували переважно для обчислень.

II покоління (1955-1965 рр.) характеризується появою транзисторів та систем пакетної обробки (для автоматизація та підвищення ефективності роботи дані подавалися групою). З’явилася перша операційна система GMOS, яка була створена General Motors в 1950-х рр. для машини IBM 701. Такі операційні системи називались однопотоковими системами пакетної обробки.

Поява інтегральних схем характеризує ***III покоління (1965-1980 рр.).*** З’являються перші багатозадачні операційні системи. Введення багатозадачності було важливою частиною розробки операційних систем, оскільки дозволило повністю завантажити роботою центральний процесор. Ще одним досягненням даного періоду є ріст мікрокомп’ютерів, що стало стартом створення персональних комп’ютерів.

В кінці 1960-х рр. була розроблена перша версія операційної системи Unix. Вона легко та швидко адаптувалась до нових систем, що дало змогу її отримати широке визнання. Багато сучасних операційних систем, наприклад Apple OS та Linux, похотять та основуються на Unix. Саме Unix можна вважати першою

операційною системою, в якій закладені основні принципи побудови операційних систем.

Нововведення доби комп'ютерів III покоління

Багатозадачність. Кожному завданню відводився свій розділ пам'яті (апаратними засобами). Поки одне завдання чекало на завершення роботи пристрою введення-виведення, друге поки могло виконуватися.

Підкачування (спулинг, spooling – Simultaneous Peripheral Operation On Line). У міру виконання завдань автоматично у звільнені розділи пам'яті завантажуються (підкачуються) наступні завдання. Приклад ОС: OS/360 (для комп'ютерів IBM 360).

Режим поділу часу. Різновид багатозадачності, який передбачає наявність у кожного користувача свого терміналу, з якого він може взаємодіяти з ЕОМ. Центральний процесор надається користувачам по черзі.

Приклади ОС: CTSS (Compatible Time Sharing System), MULTICS (MULTIplexed Information And Computing Service).

І багатозадачність, і підкачування успішно функціонують і в сучасних ОС – щоправда, суть цих технологій зазнала певних змін. Ідеї, що лежали в основі режиму поділу часу, отримали нове життя в технологіях клієнт-сервер, веб-сервісів, хмарних обчислень та ін.

Наступний ***IV період (з 1980р. і дотепер)*** в еволюції операційних систем пов’язаний з появою великих інтегральних схем. Поява мікропроцесорів дало змогу створити новіші та потужніші обчислювальні системи. З’явилася операційна система MS DOS з інтерфейсом командного рядка компанії Microsoft. В 1985 р. почали використовувати назву нової операційної системи Windows з графічним інтерфейсом, який був пов'язаний або з’єднаний з MS DOS.

Цей період характеризується зменшенням собівартості комп’ютерної техніки, і як наслідок появою персонального комп’ютера. З зростанням кількості користувачів, з’являється необхідність створити зручний та зрозумілий інтерфейс для непідготовленого користувача. Саме в той час набула поширення концепція віртуальних машин. Тепер користувач співпрацює зі створеною для нього операційною системою, яку представляють віртуальні машини. Йому не потрібно більше думати про фізичні деталі побудови електронної обчислювальної машини. З’явилося поняття віртуальних ресурсів комп’ютерної техніки, які були еквівалентні реальним. З того часу набуло поширення концепція віртуальних машин.

Структура операційної системи:

Режим ядра (англ. kernel mode) – повний доступ у захищеному режимі роботи процесора до апаратних ресурсів і всієї пам’яті. Функціональна можливість даного режиму полягає в прямому доступі до пристроїв і всім видам пам’яті, більш високому пріоритеті виконання, ніж режим користувача. У режимі ядра працює операційна система.

Режим користувача (англ. user mode) – програма має обмежений доступ до інструкцій даного комп’ютера. В даному режимі працюють всі інші програми, включаючи програми інтерфейсу. В режимі користувача недопустимі команди, що є критичним для роботи системи.

Програмне забезпечення ПК поділяють на такі основні класи:

1. Системне програмне забезпечення (операційна система та сервісні програми);

Системне ПЗ – сукупність програмних засобів, основними функціями яких є керування апаратними ресурсами обчислювальної системи та забезпечення діалогу між користувачем та комп'ютером.

Системне ПЗ:

1. Базове ПЗ: ОС, операційні оболонки.
2. Службове ПЗ: драйвери, утиліти (програми обслуговування мережі, антивірусні програми, програми архівації даних)
3. Системні бібліотеки та системи програмування.

Системне програмне забезпечення може:

- входити до складу ОС;
- не входити до складу ОС, але постачатися розробниками ОС;
- постачатися окремо і бути створеним іншими розробниками.

2. Прикладне програмне забезпечення.

Сукупність програмних засобів для розв'язання завдань у різних предметних галузях.

Складові операційної системи:

Ядро – це програма, яка забезпечує взаємодію між прикладними програмами та апаратними засобами. Це центральна частина операційної системи, яка координує процесами виконання програм та доступом до ресурсів комп'ютера.

Драйвери – програмний модуль, що забезпечує коректну взаємодію з пристроями. Будь-яка модель пристрою для різних операційних систем має свій набір драйверів, які мають бути встановлені або інстальовані у системі перед використанням пристрою.

Інтерфейс – це програма-посередник, яка організовує взаємодію користувача з прикладним та системним програмним забезпеченням. Вона забезпечує зручність у роботі.

Утиліти – це допоміжні програми, призначення яких обслуговувати диски, здійснювати перевірку комп'ютера, налаштування параметрів роботи тощо.

Функції операційної системи:

- автоматичне завантаження ядра операційної системи в оперативну пам'ять з програмного коду в системну область на диску;
- файлова система збереження даних на диску та здійснення доступу до них з можливістю обробки;
- завантаження програм в оперативну пам'ять та керування їх виконанням.

Ядром ОС називається сукупність базових компонентів ОС, які виконують її найважливіші функції, працюють у привілейованому режимі і зазвичай постійно перебувають у пам'яті.

Програма, яка працює у режимі користувача, але потребує виконати дію, реалізовану у ядрі, виконує системний виклик (system call).

Системний виклик нагадує виклик функції у програмуванні. Основна відмінність полягає у тому, що системний виклик передбачає переключення ЦП у режим ядра, а потім знову у режим користувача.

Класифікація операційних систем

За цільовим призначенням

- для мейнфреймів (високошвидкісний багатофункціональний комп’ютер з великою кількістю дисків та об’ємами даних);
- для персональних комп’ютерів (настільні операційні системи з акцентом на підтримку мультимедіа та графічного інтерфейсу);
- для мобільних пристроїв (використання сенсорного інтерфейсу з акцентом роботи з невеликим обсягом ресурсів).

За кількістю одночасно працюючих програм:

- однозадачні (в один момент часу підтримують виконання лише однієї програми);
- багатозадачні (підтримують виконання паралельно декількох задач).

За типом інтерфейсу:

- з текстовим інтерфейсом (командні);
- з графічним інтерфейсом (об’єктно-орієнтовані);

За способом побудови:

- монолітні (всі базові функції ОС реалізовані в ядрі);
- мікроядра (в ядро вміщено мінімальні функції, все інше реалізовується у режимі користувача);
- багаторівневі (існує декілька рівнів, де кожен наступний має менший пріоритет, ніж попередній; найнижчій рівень працює з апаратною частиною, а доступ до найвищого здійснюється через системні виклики);
- віртуальні машини (створення віртуалізації обчислювальних ресурсів).

За апаратною частиною:

- Однопроцесорні (система містить один процесор);
- багатопроцесорні (система містить декілька процесорів);
- мережні (включають можливість доступу інших комп’ютерів локальної мережі до роботи з файлами та іншими серверами);
- розподілені (використовують ресурси локальної мережі як єдину систему).

За кількістю користувачів:

- однокористувацькі (відсутній механізм обмеження доступу до даних);
- багатокористувацькі (вводиться механізм обмеженого доступу до даних);

За типом доступу користувача:

- режим пакетної обробки (здійснюють виконання набору (пакету) завдання з по черговим виконанням з врахуванням пріоритетності);
- режим розділення часу (здійснюють імітацію одночасного виконання всіх завдань, проте кожна задача виконується по черзі в певні часові проміжки);
- режим реального часу (здійснюють імітацію обробки завдань одного користувача в реально заданому часовому проміжку).

Основні операційні системи

Операційна система MS DOS

Операційна система MS DOS із сімейства DOS була розроблена фірмою Microsoft для IMB-сумісних комп’ютерів починаючи з 1981 р. Було розроблено

шість версій та два десятка проміжних варіантів MS DOS. Розробку даного продукту Microsoft припинили у 2000 р. З погляду сучасного користувача, хоч дана система мала досить примітивний вигляд, проте достатньо довгий час вона була популярною завдяки своїй простоті та малій собівартості. Базова система введення-виведення та файлова система використовується і до тепер в сімействі операційних систем Windows.

Операційна система OS/2

Перша версія OS/2 з'явилася у 1987 р. як продукт спільної праці фірми Microsoft та IMB паралельно з розробкою Windows. Це багатозадачна операційна система з графічним багатовіконним інтерфейсом, в якій може одночасно виконуватись 12 програм, проте дана ОС запускається як одна задача.

У 90-х роках XX ст. шляхи двох компаній розійшлись. Microsoft продовжило існування OS/2 у Windows NT.

В наступній версії OS/2 War у 1995 р. фірми IMB були враховані проблеми попередньої, проте кардинальних змін не відбулось. Зокрема, всі властивості ОС були оптимізовані в компактному ядрі. Дана версія використовувалась як клієнтській та і серверний продукт.

Підтримка даного продукту здійснювалась до 2006 р. включно. Подальшого розвитку OS/2 знайшло в рамках проекту eComStation, зокрема на цій системі базується комерційний продукт ArcaOS.

Операційна система UNIX

Перша система UNIX була розроблена в 1969 р. в дослідному центрі Bell Labs компанії AT&T, назва якої походить від оригіналу AT&T UNIX. Операційну систему UNIX характеризує модульний дизайн, уніфікована файлова система та міжпроцесорний механізм зв'язку. Дана система є багатозадачною, має багатокористувацький режим роботи та забезпечує достатню надійність та захист даних. Користувачі можуть об'єднуватись в групи, де один з них може стати адміністратором.

Основні механізми роботи, які були закладені в операційну систему UNIX мали значний вплив на історичний розвиток комп'ютерних операційних систем.

Операційна система Linux

Система Linux є однією з найпоширеніших систем сімейства UNIX. Це багатозадачна та багатокористувацька операційна система, яка орієнтована працювати у мережі. На відміну від інших систем, Linux не має якоїсь «офіційної» комплектації, а натомість ядро Linux входить у велику кількість дистрибутивів. Це приклад вільного та відкритого програмного забезпечення, який розповсюджується безкоштовно у вигляді різних дистрибутивів, готових до використання та мають зручний супровід.

Операційна система Windows

Найпоширеніша операційна система фірми Microsoft для персональних комп'ютері є сімейство Windows. Дану операційну систему характеризує багатовіконний графічний інтерфейс, багатозадачність з можливістю витіснення (завершити роботу з застосунком, у разі його зависання). Однією з переваг даного сімейства є підтримка технології Plug&Play, що спрощує підключення

різних зовнішніх пристроїв. Також, дана система підтримує технологію OLE (англ. Object Linking and Embedding), що дає змогу зв'язувати та впроваджувати об'єкти в інші документи та об'єкти, створені іншими додатками.

Операційна система MacOS

Перша версія операційна система Mac OS з'явилася в 1984 р. з появою першого комп'ютера Macintosh компанії Apple. Перші назви даної системи мали просту назву як System 1, System 2 і так аж до восьмої версії. Проте з 2016 р. офіційна назва Mac OS X була змінена на macOS, а перша назва отримає ім'я Sierra. Найголовнішою заслугою розробників даної системи є впровадження графічного інтерфейсу, що стало еталоном для інших операційних систем, які на той час користувались складними командами командного рядка. Розробники MacOS максимально намагалися робити інтуїтивний інтерфейс, який буде зрозумілим для будь-якого пересічного користувача. Цей підхід запозичили і інші розробники.

Операційна система OpenVMS

В другій половині 1970-х рр. компанією Digital Equiplent Corporation була розроблена серверна операційна система OpenVMS для комп'ютерів VAX. Назва VMS походить від системи віртуальної пам'яті, що є її однією з архітектурних особливостей. Пріоритетом роботи даної системи є багатокористувацький інтерфейс, багатопроцесорна операційна система на основі віртуальної пам'яті, пакетна обробка та обробка транзакцій.

Операційна система Android

Розповсюдження операційна система для мобільних пристроїв з сенсорним екраном Android була створена на базі ядра Linux компанією Google. Дана система розроблена спеціально для інтуїтивного керування жестами. Під дану систему розроблено безліч застосунків, які можна завантажити з Google Play або з інших ресурсів.

Операційна система iOS

Операційна система iOS є мобільною версією від Apple, спочатку для iPhone, згодом для iPad. iOS є похідною версією від MacOS та є UNIX-подібною операційною системою. На відміну від Android, дана система випускається тільки для пристроїв, які створюються компанією Apple.

Лабораторна робота 1

Тема. Встановлення ОС GNU/Linux на персональний комп’ютер (за допомогою VirtualBox).

Мета: На прикладі ОС Linux вивчити процедуру інсталяції ОС сімейства UNIX на персональні комп’ютери. Ознайомитися з задачами адміністрування файлових систем, а також з сучасними файловими системами в ОС Linux.

Теоретичні відомості

Дистрибутиви Linux

Насправді, дистрибутивів Linux існує дуже багато, більше 800. Вони завжди актуальні, доступні для безкоштовного скачування. Звичайно, виділити якісь найкращі серед них досить важко, оскільки універсальних дистрибутивів, підходящих для реалізації геть усіх цілей — просто немає. Все ж, давайте спробуємо систематизувати і виділити найбільш вживані й прогресивні Linux-дистрибутиви.

Дистрибутиви, засновані на Debian або використовують формат пакетів Deb

Debian – це супер дистрибутив Linux, якість якого перевірена роками. Навіть сам Лінус Торвальдс відповідав у інтерв’ю, що користувався саме цим дистрибутивом Linux протягом тривалого часу. І нещодавно лише змінив його на Fedora. Debian має великий вибір пакетів (більш 43 -х тисяч) і підтримує велику кількість платформ, тому може бути універсальним у найрізноманітніших сферах. А завдяки тому, що Debian досить простий в інсталяції, його принципі може налаштувати і використовувати практично будь-який користувач. Наприклад, в Німеччині це найпоширеніший дистрибутив для серверних рішень на базі Linux.

Дистрибутив Knoppix

Knoppix – «кнопочка» слово промовляє само за себе, у розумінні — крихітний. Але названий він в честь його розробника Клауса Кноппера. Тобто це дистрибутив Linux, який включає у себе збірку вільних програм, що працюють з компакт-диска (Live-CD, USB та інших сторонніх носіїв). Розроблений Knoppix на основі Debian. Найчастіше цей дистрибутив використовують коли потрібно відновити певні порушені функції різних ОС або в якості операційної системи для презентації програмного забезпечення, а також в ознайомлюванні з Linux без інсталяції та в освітніх цілях.

Дистрибутив Parrot

Parrot – це дистрибутив Linux теж заснований на Debian. Його використання пропонує користувачі велику кількість різних засобів для випробовування захищеності ОС від несанкціонованого доступу до неї. Окрім цього, тут можна знайти інструменти із області криптографії, комп’ютерної криміналістики, а також пакети для забезпечення анонімності. Насправді, Linux Parrot включає у себе дуже багато різних інструментів і буде корисним дистрибутивом не тільки для новачків системного адміністрування, а й для

розробників, оскільки працює стабільно, без збоїв і помилок. Він ідеально підходить для спеціалістів з інформаційної безпеки та мережевих адміністраторів.

Дистрибутиви, засновані на Ubuntu

Ubuntu – це класичний дистрибутив Linux. Таке собі втілення Windows на Linux. У своїй роботі Ubuntu використовує власні репозиторії пакетів, які в свою чергу відрізняються від репозиторіїв Debian. Дистрибутив відзначається хорошою підтримкою мультимедійного контенту. Працює Ubuntu не надто стабільно, та завдяки тому, що цей дистрибутив використовує snap-пакети, установка певного програмного забезпечення через термінал та його оновлення в Snappy Ubuntu Core виконується дуже швидко і просто.

Для новачків особливо корисним в дистрибутиві Ubuntu є те, що через його центр ПЗ (т.зв. званий «Software Center» – магазин) можна завантажувати багато безкоштовних програм, які у більшості випадків можна встановити всього за декілька кліків.

Дистрибутив Linux Mint

Mint – більшість вітчизняних користувачів ніжно його прозивають М’ята, і без М’яти не обходиться жодний рейтинг. Це дистрибутив Linux, розроблений на основі Ubuntu і включає в себе різноманітні оболонки для робочого столу ПК. Від інших дистрибутивів Mint відрізняється, перш за все тим, наскільки він дружній по відношенню до користувачів-новачків, а ще у нього шикарний дизайн, який «радує око». А завдяки тому, що він підтримує пропрієтарне програмне забезпечення, так як наприклад Adobe Flash, цей Linux-дистрибутив добре підходить і для роботи з мультимедійним контентом

Дистрибутив LXLE (Lubuntu Extra Life Extension)

LXLE – це простий дистрибутив Linux, який теж в основі на Ubuntu та комбінує у собі скромні вимоги до системних ресурсів та великі можливості. Займає LXLE мало місця на ПК, але тим не менш, дозволяє користувачеві повноцінно працювати зі своїм комп’ютером. У Linux-дистрибутиві LXLE можна знайти всі необхідні програми, характерні для релізів ОС Linux, що розраховані на настільні ПК. Ця система підходить для комфортного домашнього використання, під її управлінням зможуть працювати далеко не найновіші комп’ютери (не кажучи вже про сучасні ПК). Встановивши дистрибутив LXDE у вас з’явиться як і в Ubuntu багато стандартних програм, типу LibreOffice і Gimp. Єдине, що вам буде необхідно, це самостійно встановити улюблений браузер для роботи в інтернеті.

Дистрибутив Fedora

Fedora – це дистрибутив Linux, на базі Red Hat який дозволяє працювати із безкоштовним серверним програмним забезпеченням на вашому ПК. Fedora працює традиційно на принципах програм із відкритим вихідним кодом. При цьому, розробники цього Linux-дистрибутива мають на меті постійно підтримувати найбільш інноваційних та цікавих концепцій ПЗ. Fedora підходить для тих користувачів, які хочуть бути в курсі всіх новинок у світі безкоштовного ПЗ.

Дистрибутив Kali

Kali – це дистрибутив Linux, що був створений спеціально для тестувальників інформаційної безпеки «хакерів». Але в нагоді стане й кожному **qa engineer, у security testing для проведення penetration testing – тестування безпеки ПЗ на проникнення.** Встановити Kali на ПК можна з носія LiveUSB і liveCD. В даний час в цей Linux-дистрибутив включені інструменти для проведення програмно-технічної експертизи. Також розробники Kali ведуть роботу над підтримкою ієрогліфічних азіатських мов.

Дистрибутив Elementary OS Loki

Elementary OS Loki – стабільний, зручний та зрозумілий у використанні дистрибутив Linux із симпатичним зовнішнім виглядом, що не може не привернути увагу. Візуально він нагадує Mac OS від Apple, але таке порівняння — лише плюс для цієї системи, оскільки це говорить про потенційну зручність у роботі користувача. Одночасно **Elementary OS Loki** дуже легка система, яка пропонує не найбільший, але добре продуманий вибір програмного забезпечення для роботи на ПК.

Дистрибутив Gentoo

Gentoo – це такий дистрибутив Linux із категорії «зроби собі сам» :), який заснований на перетворенні пакетів з вихідних кодів під певне апаратне забезпечення. Для роботи з цим дистрибутивом користувач повинен мати певну кваліфікацію. Після його встановлення юзер має самостійно зібрати все, що йому необхідне для роботи. Це не тільки вимагає від користувача досить високого рівня розуміння ОС Linux, але і багато терпіння та часу. Як результат, Ви отримаєте «ексклюзив» саме те, що треба Вам і нічого зайвого.

Файлові системи Linux

Файлова система визначає порядок організації, збереження та доступу до даних на носіях комп'ютера у вигляді сукупності файлів та папок (каталоги, директорій). В ядро Linux може бути вбудовано декілька файлових систем, кожна яких оптимізована під певний тип завдань. Кожну з них обслуговує спеціальний драйвер в ядрі, але завдяки концепції Virtual File System інтерфейс для користувача завжди однаковий.

На відміну від ОС Windows, в ОС Linux жорсткий диск поділяється не на диски, а на розділи, які позначаються символом / . ОС Linux файлова система представлена за допомогою одного кореневого каталогу, наприклад, /home означає знаходження в каталозі home в кореневому каталозі (/).

Основні файлові системи ОС Linux

1. *Ext2, Ext3, Ext4 Extended Filesystem* – це найстабільніша стандартна файлова система, яка містить більше всього функцій. Завдяки використанню журналювання, тобто ведення журналу транзакцій підвищує стійкість до збоїв та покращено стабільність даної системи. Також збільшено розмір розділу, що дало змогу використовувати дану систему по сьогоднішній день.
2. *JFS* або *Journaled File System* – створена IBM для AIX UNIX як альтернатива ext. На даний час використовується з метою надання стабільності

та мінімального використання ресурсів. Також підтримується використання журнальованої файлової системи.

3. *ReiserFS* – розроблена пізніше з розширеними можливостями та покращеною продуктивністю, також на заміну ext. Особливістю роботи є наявність динамічного розміру блоку, що дозволяє запаковувати декілька файлів в один блок, завдяки чому пришвидшується робота з файлами. Також є можливість зміни розміру розділу під час роботи, проте це може призвести до нестабільності роботи та ризику втрати даних при збіях системи.

4. *XFS* – високопродуктивна файлова система, яка розрахована на роботу з великими файлами. Перевагами даної системи є швидкісна робота з великим файлами, відкладення виділеного місця, збільшення розмірів під час роботи та невеликій розмір службової інформації.

5. *Btrfs або B-Tree File System* – повністю нова файлова система, розрахована на легкості адміністрування та відновлення даних, також зосереджена на стійкості роботи. Така система має високу продуктивність роботи та ряд можливостей (розміщення на декількох розділах, підтримка підтомів, миттєва зміна розміру тощо).

Віртуальна машина

Віртуальні машини представляють собою емуляцію пристроїв на іншому пристрої або, дозволяють запускати віртуальний комп'ютер (як звичайну програму) з потрібною операційною системою на вашому комп'ютері з тієї ж або яка відрізняється. Наприклад, маючи на своєму комп'ютері Windows, ви можете запустити ОС Linux або іншу версію Windows у віртуальній машині і працювати з ними як зі звичайним комп'ютером.

Віртуальні машини – це програмна система, яка забезпечує емуляцію апаратного забезпечення деякої платформи, виділення під її потреби частини власних ресурсів та запуск на такій віртуальній платформі яких завгодно завдань. На відміну від емуляції конкретного пристрою, віртуальна машина виконує повну емуляцію фізичної машини чи середовища для виконання програм.

Уявіть собі таку ситуацію – по роботі вам доводиться працювати в програмі яка працює тільки під операційною системою Windows XP, а у вас встановлена Windows 7 або Windows 8/8.1). Як ви вийдете з цієї ситуації? Хтось змириться і встановить собі на комп'ютер Windows XP (на яку вже не випускаються оновлення безпеки), а хтось, більш кмітливий, встановить на своєму комп'ютері віртуальну машину і встановить в ній Windows XP з можливістю працювати в потрібній програмі.

Тобто, якщо сказати коротко, то віртуальна машина – це повноцінний комп'ютер з процесором, оперативною пам'яттю, вінчестером і навіть BIOS, який працює всередині вашого комп'ютера за допомогою програми емулятора.

Приблизний перелік того, для чого користуються віртуальними машинами:

- тестування додатків;
- тестування мережевих програм в закритих віртуальних мережах;
- тестування роботи додатків з різними параметрами конфігурації ПК та ОС;
- відсутність коштів на додаткові комп'ютери;
- консолідація серверів на одному фізичному комп'ютері;

- здійснення захисту важливих даних та обмеження можливостей певних застосунків;
- безпечний запуск застосунків, які викликають підозру на вміст вірусу або несуть можливу загрозу ОС та ПК;
- навчання різним операційним системам тощо.

Поняття та сама концепція віртуальної машини з'явилася в кінці 1960-рр. XX ст. в Кембріджі в рамках розширення концепції віртуальної пам'яті манчестерської обчислювальної системи Atlas. Ідея віртуальної машини лягла в основу розробки ряду операційних систем, наприклад такої як OpenVMS.

В наш час концепція віртуальних машин стає більш розповсюджуваною та затребуваною опцією. Віртуальна машина – є еквівалентом реальної електронно-обчислювальної машини, що забезпечує користувача всім необхідним функціоналом. Використання віртуальних машин дозволяє запускати декілька ізольованих операційних систем різних сімейств або версій на одній тій й самі апаратній платформі. Це дає змогу знизити витрати на придбання та обслуговування додаткових пристроїв. Оскільки кожна операційна система має свій набір застосунків, які не заважають один одному та у разі збою або якихось неполадок не вплинуть один на одного.

Проте слід пам'ятати, що при використанні такої технології не потрібно забувати про відчуття безпеки, оскільки віртуальні машини схильні до появи різних помилок, що може призвести до втрати або відсутності доступу до критично важливих даних, які зберігаються у віртуальному середовищі.

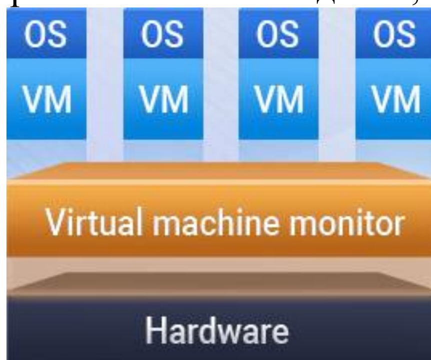


Рис. 1-1. Схема роботи віртуальних машин.

Операційна система, яка встановлена на віртуальну машину називається гостьовою. Кожна така гостьова система запускається в окремому вікні. А сама операційна система, на яку встановлено віртуальну машину є основною або хост-ОС.

Віртуалізація – це процес створення програмної (тобто віртуальної) версії обчислювальної машини з виділеними ресурсами центрального процесора, пам'яті та сховища даних, які запозичуються у фізичного комп'ютера або віддаленого сервера.

Віртуальне обладнання (рис. 1-1), завдяки якому функціонує гостьова ОС, використовує спеціальний механізм – гіпервізор. Це програма або обладнання процесора, яка керує фізичними ресурсами обчислювальної машини та розподіляє їх між декількома операційними системами, що дозволяє працювати їм одночасно. Гіпервізор виконує взаємну ізоляцію операційних систем, які запускаються віртуальними машинами, завдяки розділенню фізичних та логічних пристроїв між собою. Іншими словами, гіпервізор створює фізичні копії апаратних ресурсів комп'ютера, при чому користувач бачить їх як окремий пристрій.

Гіпервізори поділяються на два типи:

1. *Гіпервізори першого типу.* Ще їх називають мікроядром, тонким гіпервізором або автономним, які сприймаються як специфічна компактна операційна система, що встановлюється прямо на «залізо» та

має всі ознаки ОС. До такого типу відносяться такі віртуальні машини як Xen, VMWare, Hyper-V тощо.
2. Гіпервізори другого типу. Називають хостовими (hosted). Представляють собою додатковий програмний шар, який розміщується поверх основної операційної системи. Фактично гіпервізори другого типу працюють як один із процесів основної ОС, частіше за все в ОС Linux. До такого типу відносять: VirtualBox, VMWare Workstation, KVM.

Існують так звані гібридні гіпервізори, які мають властивості першого та другого типу гіпервізорів. Вони керують на пряму процесором і память та через службову ОС гостьові отримують доступ до всіх пристроїв введення-виведення. Сучасні технології не стоять на місці, тому згодом гіпервізори Xen та Hyper-V починають відносити до гібридного типу, який набуває все більшої популярності.

Найпоширеніші віртуальні машини

VirtualBox

Програма для віртуалізації операційних систем VirtualBox була створена німецькою фірмою Innotek в 2007 р. та в подальшому з 2010 р. перейшла до Oracle Corporation. Вона підтримується багатьма ОС, зокрема Linux, Windows, MacOS, OS/2 та інше. Це багатоплатформне програмне забезпечення з відкритим програмним забезпеченням, яке має можливість створювати кілька віртуальних машин та запускати їх одночасно.

VMWare

VMWare платна професійна віртуальна машина компанії EMC Corporation, яка в основному працює з Linux та Windows. Перша версія VMWare Workstation вийшла в 1999 р. Даний продукт пропонує широкий вибір віртуалізацій, що гарно адаптовані під потреби користувача. Більшість платформ є платними, проте також є і безкоштовні версії, зокрема VMWare Player, що має зменшений функціонал.

Parallels Desktop

Це платний продукт компанії Parallels для комп’ютерів Macintosh фірми Apple, що дає можливість користувачам Mac спробувати інші ОС, зокрема Windows та різні дистрибутиви Linux або навіть іншу версію macOS. Віртуальна машина запускається як звичайний застосунок, що не потребує перезавантаження, на відміну від технології Boot Camp фірми Apple та має ряд утиліт, які максимально покращують та оптимізують роботу з віртуальною машиною.

Hyper-V

Hyper-V (кодова назваViridian або відомий як Windows Server Virtualization) є вбудованим гіпервізором, що може створювати віртуальні машини в системах під керівництвом Windows. Якщо на серверному комп’ютері здійснити налаштування як для декількох віртуальних серверів, то це дасть змогу працювати кільком операційним системам разом з їх застосунками. Даний

продукт поширюється в двох варіантах: як компонент Windows або як окремий продукт Hyper-V Server.

Xen

Багатоплатформний гіпервізор, який був створений в 2003 р. в рамках дослідного проекту лабораторії Кембриджського університету компанією XenSource. Однією з особливостей була підтримка паравіртуалізації та апаратної віртуалізації, що відносить цей гіпервізор також і до гібридних. З 2007 р. розробка перейшла у власність компанії Citrix, що дало нову назву продукту – XenServer.

KVM

Гіпервізор KVM був створений в 2006 р. та інтегрований в основне ядро Linux, що були випущені в 2007 р. Пізніше його адаптували як модуль ядра в FreeBSD. KVM забезпечує віртуалізацію в середовищі Linux, яка підтримує апаратну віртуалізацію на базі Intel VT або AMD SVM, що складається з завантажуємого ядра kvn.ko.

Хід роботи

1. Завантаження віртуальної машини

Спочатку потрібно завантажити та встановити програму віртуальної машини. Існує багато віртуальних машин, проте в даній лабораторній роботі будемо розглядати безкоштовну програму VirtualBox (рис. 1-2), яка є однією із найпопулярніших та рекомендованих для новачків.

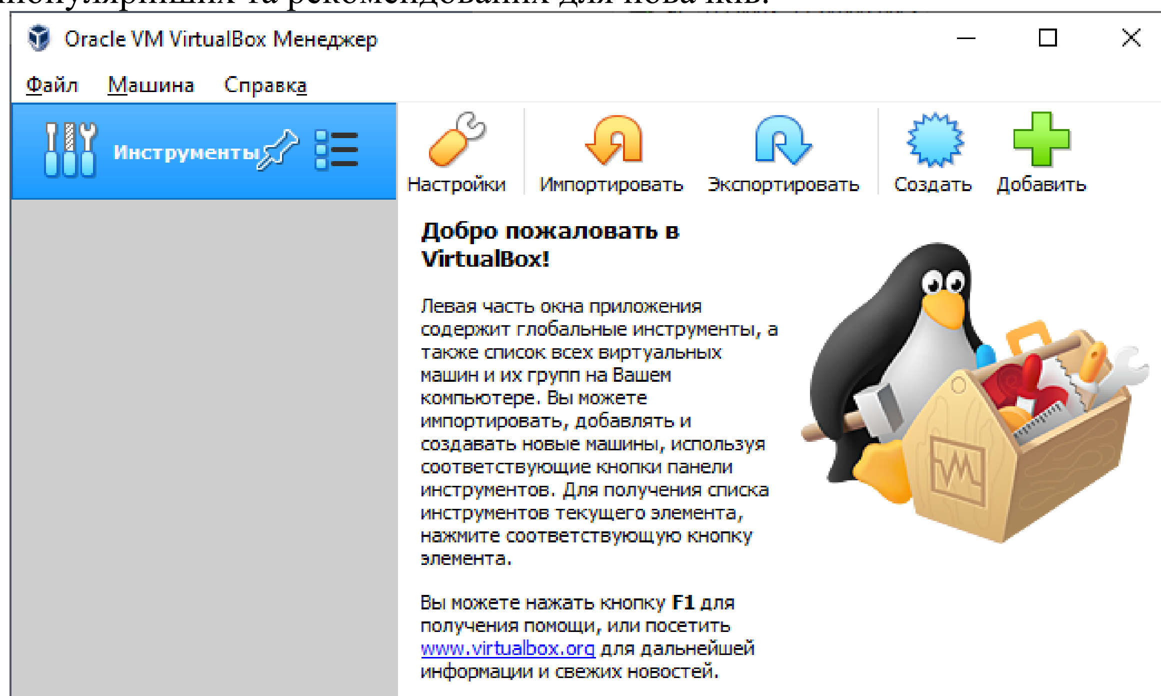


Рис. 1-2. Робоче вікно Virtual Box.

1. Завантажити Virtualbox для Windows hosts скориставшись сторінкою з офіційного сайту https://www.virtualbox.org.

2. Запускаємо завантажений інсталятор і тиснемо кнопку «Далі».

3. У вікні, що з’явиться, залишаємо всі параметри за замовчуванням (при потребі змінити місце встановлення програми) і тиснемо кнопку «Далі».

4. У наступному вікні вибору опцій залишаємо вибрані параметри за замовчуванням.

5. У наступному вікні встановлення буде здійснено попередження про створення нового мережевого підключення, яке може привести до тимчасового відключення вашого активного мережевого підключення. Здійснюємо погодження та натискаємо кнопку «Install» для подальшого встановлення.

6. Потягом декількох хвилин, програма буде здійснювати встановлення. З'явиться повідомлення про закінчення успішного встановлення. Потрібно натиснути кнопку закінчення, тоді при успішному встановленні, завантажиться програма *VirtualBox менеджер*. Менеджер є всього лиш оболонкою, яка здійснює взаємодію між віртуальною машиною та апаратною частиною комп'ютера. Всі налаштування можна залишити за замовчуванням.

2. Створення віртуальної машини

1. Для створення віртуальної машини натискаємо кнопку «Створити» та вводимо ім'я віртуальної машини (наприклад, Ubuntu_VM, оскільки будемо встановлювати дистрибутив Ubuntu) вибираємо тип ОС та версію Ubuntu 32-bit.\

2. Далі встановлюємо кількість оперативної памяті не більшу за ту, яка у вас встановлена фізично (наприклад, 2048 мегабайт).

3. Далі потрібно вибрати або створити віртуальний жорсткий диск. Так як програма запускається вперше, тому потрібно його створити.

4. Тип встановлюємо за замовчування. Оскільки VDI є стандартним типом даної програми, що повністю буде сумісно з *VirtualBox* та буде прирівнюватись до швидкості роботи реального жорсткого диску.

5. Вибираємо тип віртуального диску: *динамічний* (по мірі заповнення даних розмір буде зростати, диск буде створюватись швидше, але працювати повільніше) та *фіксований* (розмір залишається таким, як при створенні, буде швидко створюватись, проте повільніше працювати). Краще створити динамічний віртуальний диск, оскільки він на початку не буде займати багато місця.

6. Вказати ім'я диска, його розмір та місце, де будуть зберігатися віртуальний диск та всі файли. Краще вибрати диск, на якому не встановлена ваша основна ОС. Розмір можете вказати за допомогою повзунка або в графі праворуч.

7. Підготовчі роботи по створенню будуть завершені, потрібно тільки натиснути кнопку «Створити» і трохи почекати.

Буде створена віртуальна машина без ОС, яку можна встановити, скориставшись образом диску.

3. Завантаження дистрибутиву Linux

1. Зайдіть на сторінку https://kubuntu.org/getkubuntu/ .

2. Скачайте Kubuntu 18.04.5 LTS 32 біт або 20.04 LTS 64 біт.

4. Налаштування віртуальної машини

Для здійснення налаштувань, натискаємо кнопку «Налаштування» та ознайомлюємося з вікном налаштування віртуальної машини.

1. Вкладка «Загальні» – загальні дані про віртуальну машину такі як, назва, тип та версія ОС, папка з даними. Всі параметри залишаємо за замовчуванням.

2. Вкладка «Система» – налаштування материнської плати та процесора (віртуальних). Всі параметри залишаємо за замовчуванням, але можна змінити параметр «Дискети» та на вкладці «Процесори» задати два процесори.

Вибираємо в лівому меню Система вкладку Материнська плата і дивимося на опцію Включити IO APIC. Вона повинна бути включена для кращої продуктивності.

3. Вкладка «Дисплей» – налаштування віртуальної відеокарти. Додаємо відеопам’яті, можна встановити на максимум. Поставити галочку «Включити 3D-прискорення».

4. Вкладка «Носії» – підключаємо образ диску з ОС для встановлення. У списку пристроїв вінчестер і CD привід потрібно підключити образ диску (див малюнок)

Пусто – Образ – Шлях до файлу Kubunta – OK (рис. 1-3)

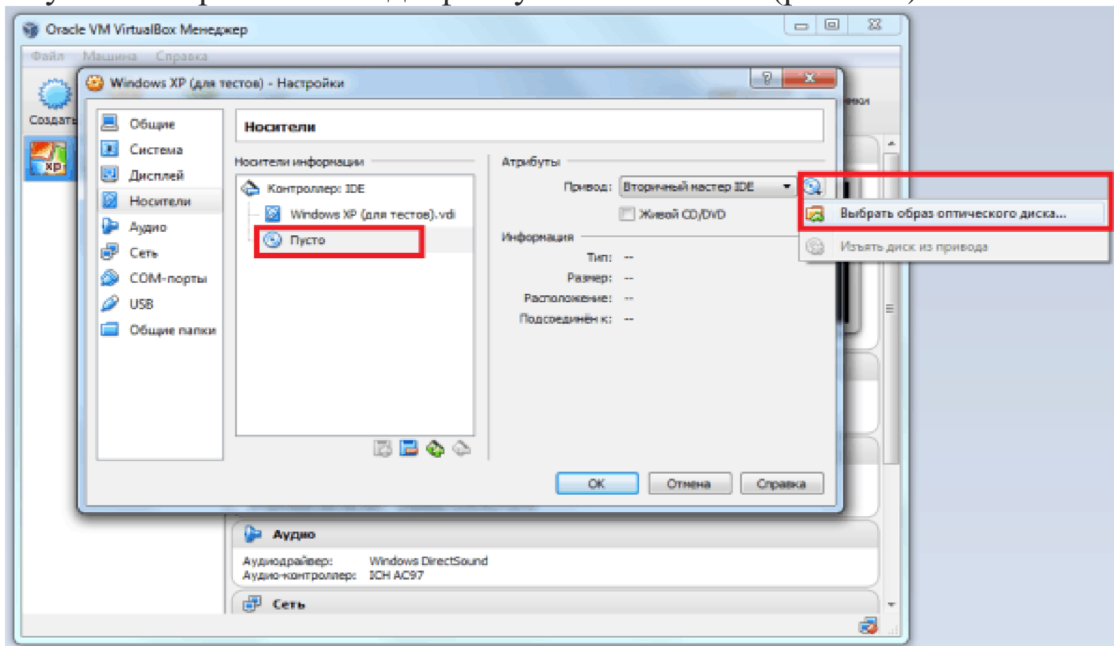


Рис. 1-3. Вибір образу диску.

5. Вкладка «Аудіо» – налаштування аудіо. Всі параметри залишаємо без змін.

6. Вкладка «Мережа» – задає мережні карти для віртуальної машини. За замовчування підключений лише один з типом підключення NAT. Оскільки віртуальна машина в такому випадку буде працювати з вашою фізичною картою безпосередньо, тому залишаємо все без змін.

7. Вкладка «COM порти» – на даний час є неактуальною. Всі параметри залишаємо без змін.

8. Вкладка «USB» – також залишаємо без змін, оскільки можна буде підключати фізично usb.

9. Вкладка «Загальні папки» – можна задати шлях до папки, яка буде спільна для фізичного та віртуального комп’ютера.

Після налаштування віртуальної машини тиснемо «ОК» і тиснемо «Запустити». Якщо ви все зробили правильно, то при запуску машини у вас повинна початися завантаження з образу диска.

Запуск машини – мова англійська – install Kubuntu – англійська розкладка – Normal і зняти галочки Other option – розбиття диску залишаємо 1 – continue – time zone Київ – ввести ім'я User та пароль 1 – встановити – перезавантажити.

Контрольні запитання

1. Яка є класифікація операційних систем?
2. Що таке ОС Linux? Структура файлової системи Linux.
3. Що таке дистрибутив? Опишіть основні дистрибутиви Linux, в чому їх відмінність?
4. Перерахуйте основні етапи встановлення операційної системи.
5. Що таке віртуалізація, віртуальна машина? Наведіть приклади.
6. Які основні переваги та недоліки використання віртуальних машин?
7. Опишіть основні етапи встановлення віртуальної машини, особливості роботи та налаштування.

Лабораторна робота 2

Тема. Команди терміналу ОС Linux.

Мета: На прикладі ОС Linux вивчити основні команди терміналу.

Теоретичні відомості

Термінал – це емулятор консолі. До появи графічного інтерфейсу всі команди вводились в командний рядок або термінал. В ньому можна здійснювати різні команди та більш гнучке налаштування системи.

Команди в Linux відрізняються від команд DOS та Windows тим, що зазвичай вони коротші. В терміналі миготливий курсор вказує на позицію введення команди, яка починається з поточного шляху та імені комп'ютера за якими слідує символ $, % або #. Він означає, що команди будуть виконуватись від суперкористувача root. Символ ~ означає шлях до поточної домашньої директорії користувача.

У терміналі Linux при успішному виконанні команди на екран може не виводитись ніякої інформації. Виводиться тільки попередження при порушенні виконанні команди або в разі помилки.

Історію використаних команд можна побачити, гортаючи стрілки вгору та вниз на клавіатурі. Якщо команда або ім'я набрані частково, то його можна ввести автоматично за допомогою клавіші Tab.

Команди можуть бути введені з параметрами і без них. Синтаксис введення більшості стандартних команд:

`$команда опції параметр1 параметр 2…`

Команди термінала Ubuntu

Завантажити термінал: через пуск або пошук **Konsole**. Можна змінювати розміри та масштаб вікна терміналу.

Команда `ls` (list – показує каталоги або директорії). Синім кольором підсвічуються папки.

Ключі команди: `ls -l` (англ. law – права) – більш детальний список каталогу та `ls -a` – всі файли та директорії (приховані також). Можна комбінувати ключі: `ls -a -l`.

Команда `pwd` (present working directory – поточний робочий каталог або print working directory – вивести робочий каталог) дає можливість побачити, де ми знаходимось.

Команда `cd` (change directory – змінити директорію) – для переходу в інший каталог:

`cd Des` (Tab програма розуміє і сама підтягує необхідну назву папки).

`cd ..` – піднімає на каталог вище.

`cd -` – повертає до попереднього каталогу.

Команда `mkdir` (make directory – створити директорію) – створює директорію. Наприклад, `mkdir papka`.

Створення файлу:

1. назва файлу.розширення. Відкриємо папку papka (cd papka) та створимо текстовий файл (> file1.txt).Перевірити можна через команду ls -l або через графічний інтерфейс.

2. назва папки/назва файлу. Вийти з папки papka (cd ..). > papka/file2.txt. Перевірити можна через команду ls -l (зайти в папку cd papka) або через графічний інтерфейс.

Команда видалення файлу rm (remove). Наприклад, rm file2.txt.

Команда echo – виводить повідомлення: echo hi. Можна записати текст у будь який файл: echo ″повідомлення″ > назва файлу (echo hi > file1.txt).

Команда cat – для перевірки вмісту файла: cat file1.txt. Для перевірки також можна відкрити файл в графічному редакторі.

Конанда file – показує тип файлу: file file.txt.

Команда cp – скопіювати файли. наприклад, cp file.txt filecopy.txt.

Команда rmdir – видаляє директорію (порожню).

Команда find – шукає файл. Наприклад, в директорії find file.txt або find file.txt papka/.

Команда mv – переміщення або перейменування файлів та директорій.

Хід роботи

1. У домашній директорії створюємо теку з іменем ud_PIB за допомогою команди mkdir, де PIB – це перші літери прізвища, імені, по-батькові студента. Потім переходимо до створеної теки в якій створюємо наступні теки.

```
/home/ud_PIB/ud_PIB_1/ud_PIB_12
/home/ud_PIB/ud_PIB_1/ud_PIB_13
/home/ud_PIB/ud_PIB_2 md
/home/ud_PIB/ud_PIB_2/ud_PIB_21
/home/ud_PIB/ud_PIB_2/ud_PIB_22 md
/home/ud_PIB/ud_PIB_2/ud_PIB_22/ud_PIB_221
/home/ud_PIB/ud_PIB_2/ud_PIB_22/ud_PIB_222
/home/ud_PIB/ud_PIB_2/ud_PIB_22/ud_PIB_223
/home/ud_PIB/ud_PIB_2/ud_PIB_23
/home/ud_PIB/ud_PIB_3
/home/ud_PIB/ud_PIB_3/ud_PIB_31
/home/ud_PIB/ud_PIB_3/ud_PIB_32
/home/ud_PIB/ud_PIB_3/ud_PIB_33
/home/ud_PIB/ud_PIB_3/ud_PIB_33/ud_PIB_331
/home/ud_PIB/ud_PIB_3/ud_PIB_33/ud_PIB_332
/home/ud_PIB/ud_PIB_3/ud_PIB_33/ud_PIB_333
```

2. Виконати зміну поточного каталогу на домашній каталог.

3. Вивести на екран дерево щойно створених каталогів. (для цього виконайте оновлення системи – sudo apt-get update, а потім встановіть програму sudo apt-get install tree). Використовуючи цю комаду (tree) можна вивести деревоподібну структуру каталога.

4. За допомогою команди `rm` видаліть із теки директорії `ud_PIB_331`, `ud_PIB_332` і `ud_PIB_333` та знову виведіть дерево модифікованих каталогів.

5. Виконайте рекурсивне видалення каталогу `ud_PIB_2`.

6. У каталозі `ud_PIB_33` створіть такі файли як: `First.txt`, `Second.gif`, `Third.py`. Відредагуйте їх за допомогою відповідних програм.

7. Виконайте копіювання цих файлів у теку `ud_PIB_1`.

8. Перейменуйте скопійовані файли на `1.txt`, `2.gif`, `3.py`.

9. Створіть звіт про виконану роботу. У звіті мають бути подані скріншоти екранів віртуальної машини та опис до них.

10. Звіт бажано виконувати за допомогою Google Docs.

Контрольні запитання

1. Що таке термінала Ubuntu? Як його завантажити?
2. Які основні команди для роботи з терміналом? Перерахуйте та опишіть особливості роботи з ними.

Корисні посилання

1. https://www.bodhilinux.com/download/
2. https://app.prntscr.com/uk/ – програма для створення скріншотів

Лабораторна робота 3

Тема. Конфігурацію ядра ОС Linux.

Мета: Виконати конфігурацію ядра ОС Linux.

Теоретичні відомості

Компіляція ядра в Linux

Головною частиною операційної системи є ядро. Ядро є посередником між пристроями комп'ютера та його програмним забезпеченням. Це спеціальна програма, яка виконує функції розподілу апаратних ресурсів та організовує роботу багатьох програм.

Ядро Linux почав розробляти в 1991 р. Лінукс Торвальдс. Далі до розвитку даного проекту долучилось багато людей, де Linux почав випускатись під ліцензією GNU GPL (General Public License). Це стало наймасштабнішим прикладом відкритого програмного забезпечення.

Ядро Linux підтримує багатозадачність, віртуальну пам'ять, динамічні бібліотеки, відкладене завантаження, продуктивну систему керування пам'яттю та є монолітним ядром з підтримкою завантажувальних модулів.

Більшість ядер ОС поділяються на тири типи:

1. *мікроядра* – складаються з декількох незалежних модулів, що завантажуються в пам'ять по мірі потреби та працюють в окремих ядерних просторах. В такому ядрі здійснюється керування тільки тим, що потрібно. Перевагами мікроядра є надійність та модульність, а недоліком – його можлива повільність.

2. *монолітні* – протилежні мікроядрам, оскільки в пам'яті комп'ютера майже весь час знаходиться весь код ядра, тому швидкість його роботи більша. Однак можливі збої в системи, якщо виникнуть якісь проблеми в некоректній поведінці, оскільки всі речі виконуються в режимі суперкористувача.

3. *гибридні* – сполучає в собі підходи роботи своїх попередників, тобто ядро може вибирати з чим потрібно працювати в користувацькому режимі, а що в просторі ядра. Але такий підхід має деякі проблеми, які успадковані від типу мікроядра.

Ядра ОС Linux створюються та випускаються стабільними версіями. Зазвичай, в таких версіях усунені помилки та добавлені необхідні драйвери приладів. До 2011 р. вважали, що парна нумерація ядра є стабільною і навпаки. Проте з часом від такої теорії відмовилися.

Досвідчені користувачі дистрибутивів Linux часто завантажують собі нове ядро, розпаковують його та потім вносять зміни в конфігурації вихідного коду. Далі здійснюється компіляція ядра та розміщення його в завантажувальній директорії зі зміною налаштувань завантаження. Таким чином, здійснюється включення або відключення потрібних (або не потрібних) драйверів пристроїв та створення конфігурації ядра під свої потреби для оптимальної роботи ОС.

Перегляньте поточну версію ядра

Команда `Df` (`Df -h`) – переглянути чи достатньо місця для встановлення нової версії

Дивимось на рядок: `/dev/sda1 size(21 G), Used (6.9 G), Avail (13 G)`

В даному випадку – 13 ГБайт (цього достатньо).

Перегляд версії ядра: `Uname-a` – ця команда виводить інформацію про всю систему, в тому числі і про ядро.

Наприклад, `5.4.0-52 generic`

Пояснення: архітектура і версія ядра – 5.4.0-52. Перша цифра – це мажорний номер версії, на даний момент – це 5, 4 – мінорна версія, ядро вже трохи застаріло, зараз вже актуальна версія 5.9, цифра 0 – це номер ревізії, а 52 – це вже відноситься до номеру збірки від розробників дистрибутива , кожен раз, коли до ядра потрібно додати нові патчі або виправлення воно збирається заново, а до номера додається це число.

Або можна переглянути командою:

`cat /proc/cmdline`

відповідно зайти у директорію `root – proc` (знаходиться в корені диску) і відкрити файл `version` (рис. 1-4).

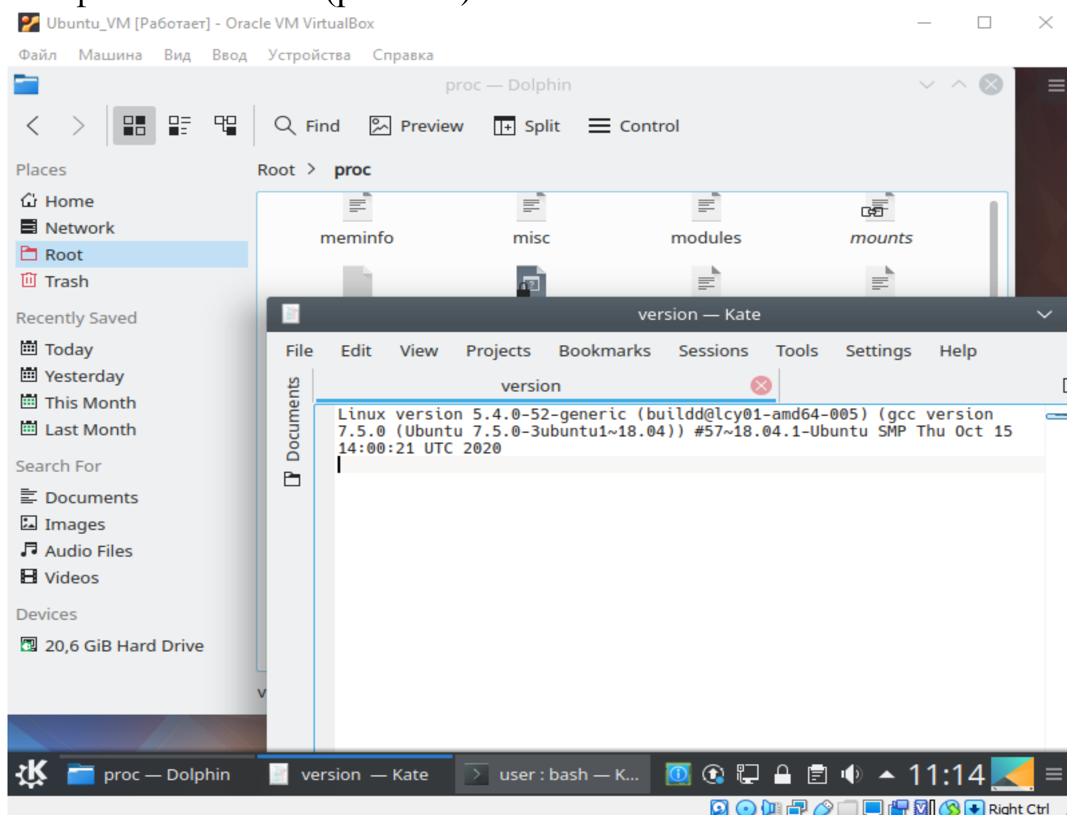


Рис. 1-4. Перегляд версії ядра Linux.

Завантаження нової версії ядра Linux

Завантажити нову версію ядра Linux потрібно із сайта https://www.kernel.org/ (рис. 1-5).

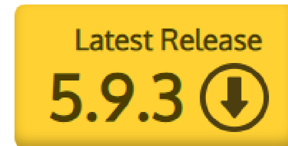
Спочатку створюємо директорію для збереження `mkdir ~/kernel; cd ~/kernel`. Наприклад, створюємо директорію `kernel_sources`.

Заходимо на цей сайт у браузері Linux (віртуальна машина) і копіюємо посилання на нову версію ядра у форматі `tar.xz`.

The Linux Kernel Archives

About | Contact us | FAQ | Releases | Signatures | Site news

Protocol	Location
HTTP	https://www.kernel.org/pub/
GIT	https://git.kernel.org/
RSYNC	rsync://rsync.kernel.org/pub/



mainline:	5.10-rc2	2020-11-01	[tarball]		[patch]	[inc. patch]	[view diff]	[browse]	
stable:	5.9.3	2020-11-01	[tarball]	[pgp]	[patch]	[inc. patch]	[view diff]	[browse]	[changelog]
stable:	5.8.18 [EOL]	2020-11-01	[tarball]	[pgp]	[patch]	[inc. patch]	[view diff]	[browse]	[changelog]
longterm:	5.4.74	2020-11-01	[tarball]	[pgp]	[patch]	[inc. patch]	[view diff]	[browse]	[changelog]
longterm:	4.19.154	2020-10-30	[tarball]	[pgp]	[patch]	[inc. patch]	[view diff]	[browse]	[changelog]
longterm:	4.14.203	2020-10-29	[tarball]	[pgp]	[patch]	[inc. patch]	[view diff]	[browse]	[changelog]
longterm:	4.9.241	2020-10-29	[tarball]	[pgp]	[patch]	[inc. patch]	[view diff]	[browse]	[changelog]
longterm:	4.4.241	2020-10-29	[tarball]	[pgp]	[patch]	[inc. patch]	[view diff]	[browse]	[changelog]
linux-next:	next-20201102	2020-11-02						[browse]	

Other resources

Cgit
Bugzilla
Mirrors
Documentation
Patchwork
Linux.com
Wikis
Kernel Mailing Lists
Linux Foundation

Social

Site Atom feed
Releases Atom Feed
Kernel Planet

Рис. 1-5. Центральна сторінка сайту https://www.kernel.org/.

В терміналі команда wget (для завантажування):

```
wget https://cdn.kernel.org/pub/linux/kernel/v5.x/linux-5.9.2.tar.xz
```

Тобто: Команда + посилання (скопійоване).

Завантажуємо `Linux - 5.9.2.tar.xz` – це архів zip.

В середині скачаного ядра (складається із драйверів, які називаються модулі) є програми написані мовою С. Функції ядра – керувати «залізом».

В кожній папці є файл `Makefile` – який пояснює компілятору як скомпілювати конкретну папку. З якими параметрами потрібно запустити `GCC Compiler`. Він перетворює команди в машинний код конкретного комп'ютера. Простіше кажучи він бере команди мовою С і їх перетворює в машинний код Instruction Set конкретного виду комп'ютера.

При встановленні можна вибирати потрібні параметри.

Таким чином можна створити свій Kernel з тими драйверами, які потрібні конкретно для нас. Бо їх є багато.

Ознайомлення з деревом каталогу вихідних текстів ядра ОС Linux.

Для перегляду списку дисків та розділів є команда `lsblk`. Використовується, щоб дізнатись інформацію про всі блокові пристрої, які є розділеними жорсткими дисками та іншими пристроями зберігання даних (оптичні пристрої, флеш-накопичувачі).

Щоб переглянути список каталогов в папці, використовуємо команду `ls`.

Команда для розархівування

Розархівовуємо заданий файл командою `tar -Jxvf name_file`.

Розархівовуємо архів за допомогою утиліти tar.

Встановлення додаткових пакетів

```
sudo apt-get install build-essential libncurses-dev bison flex libssl-dev libelf-dev
```

Цей крок необхідно виконати, якщо ядро на комп'ютері збирається вперше.

Набираємо рядок: `sudo apt-get install build-essential libncurses-dev bison flex libssl-dev libelf-dev`.

Можна виконати команди окремо, наприклад:

`sudo apt-get install build-essential` (набір інструментів для встановлення будь-чого)

`sudo apt-get install libncurses-dev` (бібліотека призначення для введення-виведення на термінал)

`sudo apt-get install bison` (синтаксичний аналізатор)

`sudo apt-get install flex` (лексичний аналізатор)

`sudo apt-get install libssl-dev` (бібліотеки)

`sudo apt-get install libelf-dev` (бібліотеки)

У разі, якщо ви хочете використовувати `config`, `oldconfig`, `defconfig`, `localmodconfig` або `localyesconfig`, вам більше не потрібні ніякі додаткові пакети. У випадку ж з рештою варіантами необхідно встановити також додаткові пакети.

Запуск конфігуратор `make nconfig`

Далі все залежить від того, яким способом ви хочете зробити конфігурацію ядра. Це можна зробити декількома способами.

`config` – традиційний спосіб, при якому створюються запити для встановлення всіх значень параметрів конфігурації. Даний спосіб не рекомендують для початківців.

`oldconfig` – основуючись на поточній конфігурації ядра, файл створюється автоматично. Даний спосіб рекомендований для недосвідчених користувачів.

`defconfig` – створюється файл конфігурації автоматично, основуючись на параметрах за замовчування.

`nconfig` – даний спосіб не потребує послідовного введення всіх значень параметрів та має псевдографічний інтерфейс ручної конфігурації.

`xconfig` – має графічний інтерфейс ручної конфігурації, який не потребує послідовного введення значень усіх параметрів.

`gconfig` – даний спосіб конфігурації рекомендований в середовищі GNOME. Також має графічний інтерфейс ручної конфігурації, який не потребує послідовного введення значень усіх параметрів

`localmodconfig` – даний спосіб характеризується створенням автоматичного файлу конфігурації, що буде включати тільки потреби даного пристрою. В такому випадку велика частина ядра буде замодульована.

`localyesconfig` – даний спосіб схожий на *localmodconfig*, проте тут велика частина буде включена в ядро. Такий спосіб рекомендовано для недосвідчений користувачів.

Заходимо в розархівовану папку.

`cd linux-5.9.2`

Визначаємось з методом конфігурації ядра.

Наприклад, `make oldconfig`

Якщо не спрацьовує, потрібно довстановлювати утиліту `make`: `sudo apt install make` (примітка, запросить пароль).

Хід роботи

1. Завантажте та встановіть на віртуальну машину один із дистрибутивів Ubuntu (Xubuntu, Kubuntu тощо).
2. Перегляньте поточну версію ядра.
3. Завантажте нову версію ядра Linux із сайта https://www.kernel.org/.
4. Ознайомитися з деревом каталогу вихідних текстів ядра ОС Linux.
5. Розархівовуємо заданий файл командою `tar -Jxvf name_file`.
6. Встановлюємо додаткові пакети

```
sudo apt-get install build-essential libncurses-dev bison flex libssl-dev libelf-dev
```

7. Запускаємо конфігуратор make nconfig
8. Ознайомитися з поточною конфігурацію ядра.
   a. Загальні налаштування.
   b. Підтримка завантажуваних модулів.
   c. Блоки.
   d. Мережеві налаштування.
   e. Драйвери пристроїв.
   f. Файлові системи.
   g. Параметри безпеки.
   h. Криптографічні параметри.
   i. Бібліотечні процедури.
9. Відключити або включити певний параметр

Номер по списку	***Вивід інформації при роботі з***
1	USB
2	Bluetooth

Номер по списку	*Вивід інформації при роботі з*
3	WiFi
4	Ethernet
5	Ext4
6	Ramfs
7	Kernel support for MISC binaries
8	Cross-endian support for vhost
9	TOMOYO Linux Support
10	PS/2 driver library

10. Запускаємо конфігуратор `make -jn`, де `n` – кількість потоків.

11. Результат створення образу ядра `bzImage` буде розміщений у папці `/arch/x86[amd64]/boot/`.

12. Створіть звіт про виконану роботу. У звіті мають бути подані скріншоти екранів віртуальної машини та опис до них.

13. Звіт бажано виконувати за допомогою Google Docs.

Контрольні запитання

1. Що таке компіляція ядра Linux? Як її здійснити.

2. Якою командою можна переглянути поточну версію ядра ОС Linux?

3. За допомогою якої команди і з якою метою встановлюються додаткові пакети ядра Linux?

4. Які є способи встановлення конфігурація ядра Linux? Їх особливості.

Лабораторна робота 4

Тема. Адміністрування користувачів ОС Linux.

Мета: навчитися здійснювати управління користувачами в ОС Linux та змінювати права доступу до даних.

Теоретичні відомості

В ОС Linux є три типи користувачів: користувач root (суперкористувач, який має необмежені права доступу), звичайні та системні (процес, що виконується на комп'ютері) користувачі. Інформація про облікові записи користувачів зберігається в звичайному текстовому файлі `/etc/passwd`. А зашифровані паролі у `/etc/shadow`. Для перегляду активних користувачів достатньо скористатися `w`.

Всі записи представлені містять інформацію про користувача та розмішені в окремому рядку в семи полях розділених двокрапкою, де містить я така інформація: ім'я користувача, зашифрований пароль, числовий ідентифікатор користувача, числовий ідентифікатор групи, поле коментаря, робочий каталог користувача та командний інтерпретатор. Наприклад, `user1:X:573:574:Linux: /home/user:/bin/bash`.

Для створення нового користувача існує спеціальна утиліта `useradd`**,** скориставшись таким записом: `useradd опції ім'я-користувача`. Також можна скористатися різноманітними опціями, наприклад:

`-b` – базова директорія за замовчуванням `/home`;

`-d` – домашня директорія користувача, співпадає з ім'ям користувача;

`-m` – створює відразу домашню директорію користувача;

`-g` – задає основну групу для нового користувача тощо.

Додавання нового користувача також можна здійснювати іншою утилітою – `adduser`, проте вона може не підтримуватись деякими дистрибутивами. Хоч ця утиліта дуже схожа на `useradd`, але вона більш універсальна та має ряд переваг. Утиліта `adduser` при створені нового користувача автоматизує ряд дій для коректної його роботи. Наприклад, створює домашню директорію користувача та його головну групу, дає можливість відразу задати паролі тощо.

Можна також створювати групу користувачів утилітою `groupadd`, скориставшись командою `groupadd -g ім'я-групи ідентифікатор-користувача`. Так як один користувач може входити до складу декількох груп, тому потрібно спочатку створити групу, а вже потім туди додавати користувача. Слід зазначити, що облікові записи групи користувачів зберігаються у файлі `/etc/group`.

Для вилучення облікового запису користувача достатньо скористатися опцією `userdel` або `adduser`. Достатньо вказати в терміналі команду `userdel (або adduser) імя користувача`. Для того, щоб видалити робочу директорію користувача, потрібно скористатись ключем `-r`.

У кожного користувача можуть бути свої права на здійснення дій з файлами та директоріями, яких розрізняють три типи:

1. Право на читання (r або 4 – read) – дає можливість читати файли або переглядати директорії.

2. Право на запис (w або 2 – write) – дає право робити записи та редагувати або створювати файли.

3. Право на виконання (x або 1 – execute) – дає можливість запускати файл або заходити в директорію. Але якщо немає права на читання, тоді користувач бачить порожню директорію.

Якщо потрібно не надавати права, тоді замість букви ставиться «-» або в числовому значені – «0». Зазначимо, що крім символьного представлення, можна задавати числове представлення права доступу (вісімкова система). При такому використані перша цифра відноситься до власника, друга – до групи, третя- -до всіх інших. Наприклад,

rwx (4-2-1)=4+2+1=7;
r-x (4-0-1)=4+0+1=5;
--x (0-0-1)=0+0+1=1.

Як результат, символьне представлення -rwxr-x--x отримаємо в числовому як 751.

Всі ці правила застосовують відразу для всіх користувачів, тоді виділяють три категорії: власник (u), група (g) та всі інші (o). Таким чином, в ОС Linux доступ до файлів отримує власник або якщо користувачу надали доступ. Тільки користувач root має всі права доступу до файлів.

Щоб дізнатися, які права доступу надалі, можна скористатися командою ls з ключем -l. Тоді будуть показані всі файли з директорії у вигляді списку з атрибутами. Спочатку вказується тип файлу, далі дев'ять позначок – права спочатку власника, потім для групи і далі для всіх інших. Тип файлу може позначатись як, - (простий файл), d (директорія), l (символічне посилання), c (файл символьного пристрою), b (файл блочного пристрою), s (файл локального сокета), p (канал).

Наприклад, команда ls -l file.txt відтворить -rwxr-x--x. Це означає, що це простий файл, де власник має всі права, група тільки читання та виконання без права запису, а всі інші – тільки виконання.

Режим доступу до файлу можна змінити за допомогою утиліти chmod, де синтаксит має такий вигляд: $chmod опції категорія дія файл. Можна застосовувати одну із опцій -r (застосувати зміни до всіх файлів та директорій рекурсивно). Дія може бути або знак «+» (добавити), або «–» (прибрати). Наприклад, команда $ chmod ugo+rwx file – додає всім користувачам повний доступ до файлу; $ chmod g+rx file – додає право групі на читання та виконання. Також можна змінювати доступ використовуючи числове представлення: $ chmod 765 file – у власника всі права, у групи – право на читання та запис, у всіх інших – заборона на запис.

Хід роботи

1. Завантажте та встановіть на віртуальну машину один із дистрибутивів Ubuntu (Xubuntu, Kubuntu тощо).
2. Переглянути всіх користувачів у системі. Для цього слід вивести на екран терміналу дані із файла passwd. (команда cat/etc/passwd).

3. Переглянути активних користувачів (для ОС UBUNTU команда `w`).

4. Створити нового користувача використовуючи команду `useradd` із параметрами `-d`, `-m`, які дозволяють створити домашній каталог користувача. Користувач повинен мати ім'я **Прізвище_NNg,** де NN – це номер групи. Прізвище має бути записане англійською транслітерацією.

5. Задати для користувача **Прізвище_NNg** пароль з 4 символів.

6. Змінити користувача, увійшовши під його іменем та паролем.

7. Продемонструвати домашній каталог користувача у терміналі.

8. Створити нового користувача **Прізвище_NNg_new**, користуючись командою adduser.

9. Увійти у систему за допомогою нового користувача **Прізвище_NNg.** Продемонструйте його домашній каталог у терміналі

10. Вивести на екран терміналу всіх активних користувачів.

11. Переглянути історію входу у систему користувачів (команда last -a)

12. Для кожного користувача створіть у його домашньому каталозі файл **Прізвище_NNg.txt.**

13. Задайте права доступу до цього файлу для всіх інших користувачів системи у відповідності до варіанту.

Варіант	***Завдання***
1	---
2	--x
3	-w-
4	-wx
5	-wx
6	r-x
7	rw-
8	rwx
9	r--
10	--x

14. Продемонструйте права на файл на екрані терміналу.

15. Створіть звіт про виконану роботу. У звіті мають бути подані скріншоти екранів віртуальної машини та опис до них.

16. Звіт бажано виконувати за допомогою Google Docs.

Контрольні запитання

1. Як здійснюється створення та керування користувачами та групами в ОС Linux?

2. Яким чином переглянути всіх користувачів у системі?

3. Як змінювати права доступу до даних для інших користувачів ОС Linux?

4. Як здійснити зміну пароля для користувача?

Лабораторна робота 5

Тема. Процеси. Перегляд log файлів.

Мета: навчитися керувати процесами в ОС Linux та здійснювати перегляд log файлів.

Теоретичні відомості

Процеси в ОС Linux

Процесом в ОС Linux називається програма або команда, яка виконується. Так як ОС Linux є багатозадачною ОС, то в ній одночасно може виконуватись декілька процесів. Кожному з них присвоюється ім'я – PID (Process IDentificator –персональний ідентифікатор). За цим ідентифікатором можна звертатися до процесу.

Атрибути процесу

Процес в ядрі представляється просто як структура з безліччю полів.

- Ідентифікатор процесу (`pid`).
- Відкриті файлові дескриптори (`fd`).
- Обробники сигналів (`signal handler`).
- Поточний робочий каталог (`cwd`).
- Змінні оточення (`environ`).
- Код повернення.

Кожен процес може знаходитись у чотирьох станах: активний (запущений), фоновий, відкладений та зупинений процес. В кожний момент часу може бити активним лише один процес. Активним є також процес, з яким взаємодіє користувач (процес отримує інформацію з клавіатури та посилає результати на екран). Фонові процеси не мають потреби в діалозі з користувачем та не отримують інформації з терміналу. Якщо процес в даний момент часу не виконується або тимчасово його призупинили, то такий процес називається відкладеним. Зупинений процес вже є знищеним назавжди. Його потрібно запускати спочатку. На відміну від зупиненого процесу, роботу призупиненого можна відновити з того місця, де його призупинили.

Процеси можна створювати, знищувати, відновлювати, змінювати пріоритет процесу, блокувати, пробуджувати та запускати.

Головним батьківським процесом `PPID` є найперший процес, який запущений системою та має назву `init`. Кожен процес має мати обов'язково батьківський, таким чином утворюється ієрархія процесів, яка бере початок від початкового процесу `init`.

Командою `ps` можна здійснити перегляд та вивести на екран список всіх процесів, що виконуються в даний момент часу. Також з командою `ps` можливе використання опцій для відображення потрібних процесів за певним параметром. Наприклад, `-A`, `-e` – вибрати всі процеси; `-d` – вивід всіх процесів

(з фоновими), крім лідерів процесів сесії (лідер – це коли один процес запускає інший).

Список запущений процесів можна переглянути за допомогою утиліти `top`. Вона не завжди є постачається з ОС, наприклад в дистрибутиві Ubuntu потрібно її встановлювати командою $ `sudo apt install top`.

Процес можна створити за допомогою команди `fork`. Під час запуску відбувається перевірка вільної пам'яті та вразі наявності створюється новий процес, що є нащадком поточного та точною копією процесу, що був викликаний. Цьому процесу створюється новий унікальний ідентифікатор.

Для здійснення завершення процесу використовується команда `kill`. Також активний процес можна завершити використовуючи сполучення клавіш ***Ctrl+C*** або клавішу ***Del***.

В Linux існує поняття пріоритету процесів. Це означає на скільки більше часу буде віддано процесу в порівняні з іншими. Пріоритет можна змінити за допомогою команди `nice`:

`nice -n`, де `n` – величина, на яку змінюється початкове значення тобто почнеться новий процес з вибраним пріоритетом.

`renice -число PID` – змінює значення пріоритету для активного процесу.

`log файли`

Для усунення помилок в роботі системи часто виникає потреба в їх перегляді та усунені неполадок. Операційна система та працюючі застосунки постійно створюють різні типи повідомлень, які реєструються в різних файлах журналів.

Розрізняють такі категорії журналів, як: додатки (`Application Logs`), події (`Event Logs`), служби (`Service Logs`) та системні (`System Logs`). Більшість log файлів знаходяться в директорії `var/log`.

Системні `log` файли містять великий об'єм інформації про функціональність системи. Наприклад:

`/var/log/syslog` – глобальний системний журнал, в якому фіксуються всі повідомлення з моменту запуску системи від ядра ОС, різних служб, знайдених пристроїв тощо.

`/var/log/auth.log` – містить інформацію про авторизацію користувача в системі як успішну, так і невдалу спроби входу в систему.

`/var/log/boot.log` – зберігається інформація, що стосується завантаження ОС.

Для перегляду `log` файлів можна скористатись декількома утилітами ОС Linux, наприклад найбільш вживані це `cat`, `less`, `more` тощо.

Хід роботи

1. Створити нового користувача системи із іменем **ПІБ_Group_NN,** де NN – це номер групи.
2. Задайте для нього папку із аналогічним іменем.
3. Вивести на екран всі працюючі процеси для цього користувача (команда ps).

4. Запустіть окремим процесом текстовий редактор, програму для перегляду зображень і ще одну на Ваш вибір програму.

5. Продемонструйте перелік запущених процесів за допомогою команди `top`.

6. Припиніть роботу запущених Вами процесів за допомогою команди `kill`.

7. Продемонструйте, що процеси були знищені, знову запустивши команду top.

8. Продемонструйте вміст такого лог-файлу як `/var/log/syslog`. Відкрийте його у терміналі та за допомогою звичайного текстового редактору.

9. Відкрийте `/var/log/auth.log` у терміналі.

10. Покажіть час входження у систему користувача **ПІБ_Group_ NN**.

11. Продемонструйте інформацію у `/var/log/boot.log`.

12. Створіть звіт про виконану роботу. У звіті мають бути подані скріншоти екранів віртуальної машини та опис до них.

13. Звіт бажано виконувати за допомогою Google Docs.

Контрольні запитання

1. Як здійснити перегляд всіх діючих процесів для користувача?
2. Як припинити роботу діючого процесу?
3. Як здійснити перегляд log файлів?

ЧАСТИНА 2. НИЗЬКОРІВНЕВА МОВА ПРОГРАМУВАННЯ ASSEMBLER.

Низькорівнева мова програмування Assembler

Більшість сучасних програм пишуться із використанням мов високого рівня C++, C#, Java, Python, Kotlin. Проте існує ряд задач, які неможливо вирішити навіть за допомогою звичного C. До таких проблемних ситуацій можна віднести програми для мікроконтролерів, що мають надзвичайно малий обсяг пам’яті (наприклад 256 kB), або ж створення спеціалізованих драйверів та критичних для роботи операційних систем бібліотек, у яких швидкість виконання є надзвичайно важливою. Саме тому використання низькорівневої мови програмування Assembler є не тільки виправданою та рекомендованою.

Assembler – це програма, яка транслює зрозумілий для людини текст із умовних позначень команд процесора, у послідовність відповідних машинних інструкцій, зрозумілих процесору. Умовні текстові позначення машинних інструкцій називають *мнемоніками*.

Особливістю програмування мовою Assembler є те, що програміст однозначно вказує із яких машинних команд буде складатися сама програма і які комірки пам’яті будуть використовуватися для збереження даних. Перевагою такого підходу є висока швидкість виконання програми, недоліком – складність написання та розуміння таких програм.

Володіння мовою Assembler дозволяє зрозуміти як програми взаємодіють із операційною системою та процесором; як представляються дані у пам’яті; яким чином процесор виконує інструкції, а програма отримує доступ до зовнішніх пристроїв тощо.

На сьогодні існує велика кількість варіантів мови Assembler, які використовуються для вирішення різних задач. До найбільш відомих варто віднести такі: MASM – Microsoft Macro Assembler (поширюються на основі ліцензії Microsoft EULA); FASM або Flat Assembler, NASM або Netwide Assembler (обидва поширюються на основі ліцензії BSD) і TASM – Turbo Assembler, розроблений компанією Borland і на даний момент його підтримку припинено. У подальшому більше уваги буде приділятися саме FASM. Це пояснюється тим, що транслятор характеризується швидкістю компіляції, оптимізацією скомпільованого коду та існуванням варіантів для різних операційних систем.

Двійкова та шістнадцяткова системи числення

Як всім відомо, основою представлення інформації у сучасних комп’ютерах виступає двійкова система, яка містить два числа 1 – наявність сигналу, 0 – його відсутність. По суті центральний процесор поступово опрацьовує кожен біт інформації (один двійковий розряд). Упорядкована сукупність бітів формує команди, завдяки яких, власне, і відбувається керування роботою комп’ютера або ж іншого цифрового пристрою. Звичайно такий запис числа є досить незручним, оскільки містить велику кількість позицій для запису навіть невеликих десяткових чисел. Щоб зменшити такі двійкові представлення часто застосовують шістнадцяткову систему числення. Вона містить 16 цифр. Символи 0 – 9 позначають аналогічні цифри десяткової системи, а символи A, B,

C, D, E, F є записами десяткових чисел 10, 11, 12,13, 14, 15 у шістнадцятковому представленні.

За допомогою наступної таблиці 2-1 досить швидко можна здійснити перехід між двіковою та шістнадцятковою системами.

Таблиця 2-1.

Десяткова	**Двійкова**	**Шістнадцяткова**
0	0000	0
1	0001	1
2	0010	2
3	0011	3
4	0100	4
5	0101	5
6	0110	6
7	0111	7
8	1000	8
9	1001	9
10	1010	A
11	1011	B
12	1100	C
13	1101	D
14	1110	E
15	1111	F

Для конвертації бінарного числа у шістнадцяткову систему варто розбити двійковий запис на групи бітів по 4 починаючи із права і записати групам відповідники із шістнадцяткової системи числення.

Адресація даних у пам'яті комп'ютера

Перш ніж перейти до вивчення самої мови Assembler необхідно розуміти яким чином відбувається виконання команд власне самим процесором. Так процес, завдяки якому процесор керує виконанням інструкцій, називають *циклом виконання*. Він складається із трьох основних кроків.

Крок 1. Отримання інструкції із пам'яті.
Крок 2. Ідентифікація інструкції.
Крок 3. Виконання інструкції.

Важливо розуміти, що процесор зберігає дані у пам'яті в зворотній послідовності. Тобто менший розряд зберігається у нижньому регістрі (по суті ліворуч), а старший – у верхньому (праворуч). Тут необхідно зауважити, що компоненти процесора, які містять дані та їх адреси у пам'яті, називають регістрами.

Розрізняють два основних виду адресації пам'яті.

Абсолютна адреса – пряме посилання на конкретне місце у пам'яті.

Сегментована адреса – адреса сегменту пам’яті із зазначеним зміщенням.

Для загального використання доступні такі регістри як rax, rbx, rcx, rdx, у яких дозволяється розміщувати інформацію загальним обсягом до 64 біт. При цьому вони включають у себе і регістри нижчих рівнів. Структуру кожного із них ілюструють таблиці 2-2 – 2-5.

Таблиця 2-2.

<table>
<tr><td colspan="4">rax (64 bit)</td></tr>
<tr><td rowspan="3">32 bit</td><td colspan="3">eax (32 bit)</td></tr>
<tr><td rowspan="2">16 bit</td><td colspan="2">ax (16 bit)</td></tr>
<tr><td>ah (8 bit)</td><td>al (8 bit)</td></tr>
</table>

Таблиця 2-3.

<table>
<tr><td colspan="4">rbx (64 bit)</td></tr>
<tr><td rowspan="3">32 bit</td><td colspan="3">ebx (32 bit)</td></tr>
<tr><td rowspan="2">16 bit</td><td colspan="2">bx (16 bit)</td></tr>
<tr><td>bh (8 bit)</td><td>bl (8 bit)</td></tr>
</table>

Таблиця 2-2.

<table>
<tr><td colspan="4">rcx (64 bit)</td></tr>
<tr><td rowspan="3">32 bit</td><td colspan="3">ecx (32 bit)</td></tr>
<tr><td rowspan="2">16 bit</td><td colspan="2">cx (16 bit)</td></tr>
<tr><td>ch (8 bit)</td><td>cl (8 bit)</td></tr>
</table>

Таблиця 2-2.

<table>
<tr><td colspan="4">rdx (64 bit)</td></tr>
<tr><td rowspan="3">32 bit</td><td colspan="3">edx (32 bit)</td></tr>
<tr><td rowspan="2">16 bit</td><td colspan="2">dx (16 bit)</td></tr>
<tr><td>dh (8 bit)</td><td>dl (8 bit)</td></tr>
</table>

Встановлення FASM на Debian-подібні дистрибутиви Linux

Для того, щоб встановити та почати використовувати FASM версію мови Assembler на Debian-подібну операційну систему Linux достатньо скористатися такими командами.

1. `sudo apt update` – для оновлення бази даних зв’язків із офіційними репозиторіями.

2. `sudo apt install fasm` – власна команда, яка і запускає процес встановлення вибраної версії транслятора.

Звичайно, у процесі розгортання програми на комп’ютері користувачу необхідно буде надати відповідні дозволи щоб пройти автентифікацію, але загалом цей процес не є довготривалим.

Синтаксис мови Assembler

Першг ніж перейти до розгляду команд, варто зробити ряд важливих зауважень. FASM використовує синтаксис від Intel для позначення інструкцій процесора. Також важливо вказати на те, що всі записи чутливі до регістру символів.

Розглянемо ключові правила запису інструкцій процесора.

1. Інструкції у асемблері розділяються розривами рядків, а також важливо щоб тільки одна інструкція розміщувалася у одному рядку.

2. Якщо рядок містить знак крапки із комою, що не поміщений у лапки, то решта рядка вважається коментарем до рядка.

3. Якщо рядок завершується символом «\», то наступний приєднується до заданого. Це дозволяє записати довгі вирази.

4. У асемблері допускається використання такі знаки і конструкції: знаки символів, що є спеціально зарезервовані (+, –, *, /,=,<, > …); послідовності, що поміщені у одинарні чи подвійні лапки, об'єднуються у рядок; послідовності інших символів, які від'єднанні від решти або пропусками (пробілами) сприймаються також як символ.

5. Кожна інструкція складається із мнемонік та різної кількості операндів, що відділяються комами Операнди можуть бути регістрами, певними значеннями або ж адресами комірок пам'яті.

6. Якщо операндом є дані у пам'яті комп'ютера, то у такому випадку він поміщається у квадратні дужки.

При цьому важливо розуміти, що самі операнди також займають місце у пам'яті і щоб коректно їх використовувати варто розуміти скільки вони можуть займати місця і як позначаються (таблиця 2-6).

Таблиця 2-6.

Розміри операндів		
Операнд	**Біти**	**Байти**
`byte`	8	1
`word`	16	2
`dword`	32	4
`fword, pword`	48	6
`qword`	64	8
`tbyte`	80	10
`dqword`	128	16

Опис даних у Assembler

Для того щоб описати дані або ж зарезервувати для них місце використовується ряд позначень.

Всі вони представлені у таблиці 2-7.

Після того як буде вказана інструкція для опису даних вставляється знак пропуску і через кому має розміщуватися одне або декілька числових значень.

Розглянемо призначення інструкцій більш детальніше. `db` – визначає найпростіші дані; `du` і `dw` – можуть містити рядки, що поміщаються у лапки; інструкції `dp` і `df` – надають можливість записати дані, які складаються із двох числових значень, які розділені двокрапкою. Для `dt` є можливість зберігати слово із чотирьох частин, які розділені двокрапкою. Якщо інструкція `dt` записана для одного параметра, то допускається значення із плаваючою точкою.

`file` – містить послідовність байтів із файлу. Важливо, що як параметр має вказуватися у лапках шлях до файлу.

Таблиця 2-7.

Інструкції для опису даних		
Розмір у байтах	**Інструкції для визначення даних**	**Інструкції для резервування даних**
1	`db`	`rb`
	`file`	
2	`dw`	`rw`
	`du`	
4	`dd`	`rd`
6	`dp`	`rp`
	`df`	`rf`
8	`dq`	`dq`
10	`dt`	`df`

Якщо використовується інструкція резервування даних, то обов’язково має міститися ціле числове значення, яке вказуватиме скільки комірок визначеного розміру має резервуватися.

Позначення констант та міток

Для того, щоб визначити константу в асемблері необхідно вказати ім’я константи, поставити знак дорівнює та записати числове значення. З мітками дещо складніше. Найпростіший спосіб визначити мітку це поставити двокрапку після назви мітки. Важливо розуміти, що після цієї інструкції може міститися інша команда і навіть перехід до іншої частини програми.

Мітки можна створювати і за допомогою інструкції `label`, після чого вказуються ім’я мітки, розмір оператора і інші параметри.

Базові інструкції мови Assembler

Інструкції переміщення даних

`mov` – переносить байт, слово або ж подвійне слово із одного операнда в інший. Це можуть бути перенесення між регістрами загального призначення, між регістрами і пам’яттю та навпаки, але не можна передавати дані із однієї комірки в іншу.

`xchg` – міняє місцями значення двох операндів. Операндами можуть бути регістри загального призначення, регістр і комірка пам’яті. Порядок запису не важливий.

`push` – поміщає операнд на вершину стека.

`pop` – заберає операнд із вершини стека, якщо він там знаходиться.

Арифметичні операції із двома операндами

`add` – заміняє перший операнд-адресат сумою, вказаних операндів

`inc` – додає до вказаного операнда 1

`sub` – знаходить різницю між першим і другим операндом та замінює на результат перший.

`dec` – віднімає від операнда 1 і замінює його.

`mul` – виконує бінарне множення операндів.

`imul` – виконує знакове множення операндів.

`div` – виконує без знакове ділення першого оперндa на другий.

`idiv` – виконує знакове ділення.

`cmp` – порівнює два операнда.

При цьому варто враховувати, що у результаті виконання операції може виникнути ситуація, коли результат виходить за межі розміру пам’яті, який виділяється для збереження операнда.

Логічні інструкції

`not` – інвертує біти у заданому операнді

`and` – виконує «логічне множення» бітів для вказаних операндів

`or` – виконає «логічне додавання» для вказаних операндів

`xor` – логічне виключне «або»

Всі операції виконуються побітово і обов’язково має враховуватися ситуація, коли результат виходить за межі описаного операнда.

Інструкції для передачі керування

`jmp` – передає керування у задане місце. Адреса призначення може бути визначена безпосередньо у самій інструкції за допомогою регістра або ж комірки пам’яті. Розмір залежить наскільки близький чи далекий перхід, а отже і яку інструкцію застосовувати 16 або 32 бітну.

`call` – дозволяє передати керування програмою процедурі, при чому у стеку зберігається сама інструкція, яка знаходиться за командою `call`.

`ret`, `retn` або `retf` – зупиняє виконання процедури і повертає керування назад, самій програмі.

Інструкції умови переходу виконують передачу керування процедурі у залежності від флагів центрального процесора під час виконання самої прогрclient. Мнемоніки умови можна отримати додаванням символа «`j`» та додаткових позначок, які представлені у таблиці 2-8.

Таблиця 2-8.

Мнемоніка	Тестова умова	Опис
o	OF = 1	переповнення
no	OF = 0	відсутнє переповнення
c b nae	CF = 1	перенесення меньше не більше і не рівно
nc ae nb	CF = 0	відсутній перенос вище або рівно не нижче
e z	ZF = 1	рівність нуль
ne nz	ZF = 0	не рівно не нуль

Таблиця 2-8.

Мнемоніка	Тестова умова	Опис
be na	CF or ZF = 1	нижче або рівно не вище
a nbe	CF or ZF = 0	вище не нижче і не рівно
s	SF = 1	знакове
ns	SF = 0	беззнакове
p pe	PF = 1	парне
np po	PF = 0	не парне
l nge	SF xor OF = 1	меньше не більше і не рівно
ge nl	SF xor OF = 0	більше або рівне не менше
le ng	(SF xor OF) or ZF = 1	менше або рівно не більше
g nle	(SF xor OF) or ZF = 0	більше не менше і не рівно

`loop` – це умовні переходи, які використовують значення із регістра (зазвичай із CX або ECX) для визначення кількості повторів циклу, зменшуючи на 1 розміщене у них значення. Цикл автоматично завершується після виконання всіх ітерацій. Регістри CX та ECX використовуються у залежності від бітності.

`jcxz` – дозволяє перейти до наступної мітки, якщо у регістрі CX (ECX) знаходиться 0.

Рядкові операції

Рядкові операції працюють із одним елементом рядка. Цим елементом може бути байт, слово або ж подвійне слово. Зазвичай рядкові елементи пов'язуються із регістрами SI і DI та похідними від них. Після кожної рядкової операції вони автоматично оновлюються до вказівника на наступний елемент рядка.

Опишемо призначення основних команд.

`movs` – переводить рядковий елемент, на який указує регістр SI, у місце, на яке вказує регістр DI. Розмір операнда може бути байтом, словом або ж подвійним словом. Операндом-адресатом зазвичай є пам'ять.

`scas` – виокремлює рядковий елемент-адресат із регістрів AL, AX та оновлює флаги AF, SF, PF, CF і OF.

`stos` – поміщає значення AL, AX у рядковий елемент-адресат.

`lods` – розміщає рядкові елементи у регістрах AL, AX.

`ins` – переводить байт, слово або ж подвійне слово із порту введення (який регіструється регістром DX) у рядковий елемент.

`outs` – переводить вказаний елемент на порт виводу, що адресується регістром DX.

Також варто зазанчити, що із рядковими інструкціями використовують префікси повторення rep, repe/repz для повторного виконання рядкових операцій щодо кожного нового елементу рядка.

Інструкції керування процесом компіляції

За допомогою інструкції `if` транслятору вказують на блок команд, які виконуються на процесорі за умови, що вираз, який записаний після ключового слова дорівнює True. `elseif` вказує на додаткові умови, за яких виконується певна частина коду. Після `else` записуються та частина програми, яка обчислюватиметься лише у тому разі, якщо жодна із умов не виконається.

Приклад 2-1.

```
If count > 0
    Mov cx, count
    Rep movsb
Endif
```

Зверніть увагу, що інструкція завершується ключовим словом `endif`.

Для того щоб повторити певну кількість разів блок коду використовують такі команди.

`times` – дозволяє виконати команду певну кількість разів.

`repeat` – надає змоги повврторити задану кількість разів цілий блок інструкцій. Блок коду має обов’язково завершеним командою endrpeat

`break` – дострокове припинення виконання циклу.

`while` – блок команд виконується доти, доки є правильною умова, записана після інструкції `while`. Знову ж блок команд завершується ключовим словом `endwhile`.

Це основні функції, які найчастіше використовуються для написання макросів мовою Assembler та керувати власне самим процесом компіляції. Сюди ж варто додати інструкцію `include`, яка дозволяє додавати сторонні бібліотеки у програму.

Особливості застосування всіх перелічених інструкцій більш детально розглянемо на практичних прикладах.

Налаштування середовища для програмування на Assembler в операційній системі Linux

Перш ніж перейти до написання програм варто налаштовувати робоче середовище. Всі налаштування стосуватимуться ОС Kubuntu 20.04 LTS. Абривіатура LTS означає довготривалу підтримку, а саме 5 років від початку релізу. Таким чином основні кроки будуть актуальними ще довгий час.

На першому етепі встановимо FASM, як один із найактуальніший на момент написання посібника. Щоб його встановити у терміналі набрати команду.

```
sudo apt install fasm
```

Після натискання кнопки введення, користувачу буде запропоновано виконати автентифікацію, увівши пароль. Варто зауважити, що у терміналі Linux

введення пароля не відображається. Саме встановлення програми відбудеться тільки після підтвердження процесу користувачем. Загалом розгортання FASM, за умов якісного підключення до мережі Internet, завершиться за декілька хвилин.

Наступним кроком буде розгортання інтегрованого середовища розробки. Розглядатимемо вільнопоширюване IDE Visual Studio Code, яке доступне для таких операційних систем як Windows, Linux та OS X. Але перш ніж його встановити на розглядувану операційну систему kubuntu 20.04 LTS необхідно розгорнути .NET Framework. Детальну інформацію про його розгортання для різних дистрибутивів Linux та версій OS X можна отримати на офіційному сайті Microsoft.

Отже розкриємо основні етапи встановлення VS Code для ОС kubuntu 20.04 LTS.

Для початку треба завантажити спеціалізований пакетний менеджер Microsoft. Щоб це зробити варто скористатися утилітою wget, після написання якої у терміналі через пропуск необхідно вказати посилання на файл, що треба завантажити. Загалом інструкція виглядає наступним чином.

```
wget    https://packages.microsoft.com/config/ubuntu/20.10/packages-microsoft-prod.deb -O packages-microsoft-prod.deb
```

Щоб інсталювати власне сам завантажений пакет треба вказати наступну команду терміналу.

```
sudo dpkg -i packages-microsoft-prod.deb
```

Після цього оновлюємо систему.

```
sudo apt-get update
```

Далі розгортаємо пакет, який дозволяє отримати доступ до `apt` репозиторіїв із використанням `https` протоколу. Він встановлюється за допомогою наступної команди у терміналі.

```
sudo apt-get install apt-transport-https
```

Знову оновлюємо систему і лише після цих налаштувань можна починати розгортати .NET Runtime або ж .NET SDK. У першому варіанті корисною буде наступна інструкція терміналу.

```
sudo apt-get install dotnet-runtime-5.0
```

Для встановленя повної версії .NET SDK у терміналі записують наступну команду.

```
sudo apt-get install dotnet-sdk-5.0
```

Після виконання цих кроків можна приступати до встановлення власне інтегрованого середовища розробки Visual Studio Code. Найпростіше це зробити за допомогою пакетного менеджера `snap`. Щоб його встановити необхідно у терміналі набрати наступну команду.

```
sudo apt install snapd
```

І на останок, саме за допомогою цього пакету розгортаємо недохідне IDE на вказаній операційній системі. Інструкція виглядає наступним чином.

```
sudo snap install code --classic
```

Результатом цих кроків буде розгортання якісного, зручного і відкритого середовища, яке можна швидко адаптувати для написання програм будь-якої складності. У цьому конкретному випадку необхідно налаштувати текстовий редактор до набору програм мовою Assembler. Щоб це зробити, після відкриття VS Code переходимо у розширення та у полі пошуку вводимо назву Retro Assembler. Після чого необхідно вибрати доповнення та встановити. Retro Assembler являє собою набір налаштувань та бібліотек, які дозволяють спростити набір та розуміння коду мовою Assembler.

У результаті маєте отримати схожий до рисунку 2-1 результат.

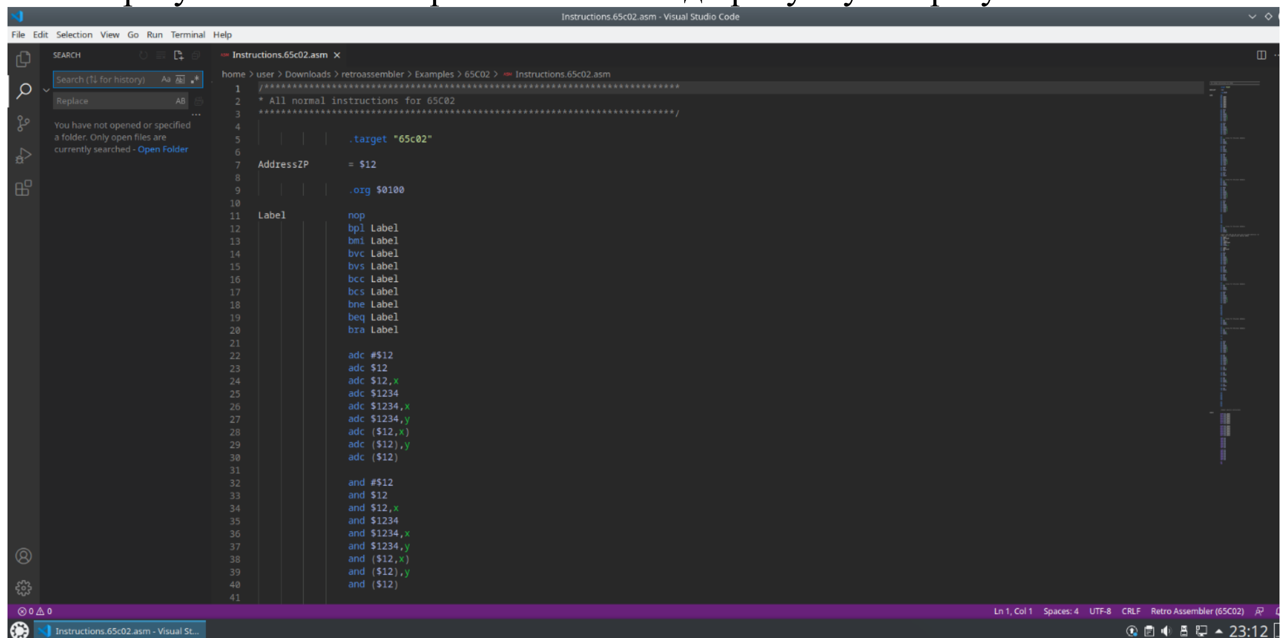


Рис. 2-1. Файл із кодом на мові Assembler відкритий у IDE VS Code із встановленим розширенням Retro Assembler.

Перша програма на мові Assembler

Процес створення програми за допомогою низькорівневої мови програмування Assembler має включати ряд важливих етапів. По-перше – необхідно вказати тип та розрядність операційної системи. У випадку Linux інструкція, яка призначено для цього, виглядатиме наступним чином.

```
format ELF64
```

Тут format є ключовим словом мови, ELF – вказує на тип операційної системи (ELF – означає, що програма створюється для операційної системи Linux, а PE – для Windows), 64 позначає її розрядність.

Якщо ж програма створюватиметься для 32-бітної архітектури, то інструкція матиме наступний вид запису.

```
format ELF
```

Важливо зауважити, що від того, яку бітність вкаже у своїй програмі фахівець, залежить доступність регістрів для роботи. Так для 64-бітної архітектури є можливість використовувати регістри *rax, rbx, rdx, rcx*; 32-бітна дозволяє працювати із регістрами *eax, ebx, ecx, edx*; 16-бітні оперують із *ax, bx, cx, dx* регістрами; 8-бітні опрацьовують – *ah/al, bh/bl, ch/cl, dh/dl*.

Після оголошення типу та роздільної здатності архітектури необхідно описати процедуру, яка дозволить вказати на початок виконання програми. Для цього скористаємося ключовим словом public, а саму процедуру назвемо із використанням символа нижнього підкреслення.

```
public _start
```

Інструкція public вказує на те, що ця процедура буде доступна для всієї програми і запускатиметься першою.

Далі опишемо стилістичне оформлення коду. Для зручності сприймання кожну значиму частину коду відділятимемо від решти порожнім рядком. Текст, який розміщується після крапки із комою вважається коментарем і не сприймається компілятором. Перед написанням кожної процедури у коментарях описуються регістри, у які дані будуть передаватися для опрацювання їх у тілі процедури, та регістри, у яких поміщатиметься результат виконання коду, представленого у тілі процедури.

Тепер опишемо, власне, саму коротку процедуру _start. Задаємо назву, після якої ставимо двокрапку та переходимо на новий рядок і виконуємо відступ у чотири пропуски (знову ж для зручності сприймання коду) та записуємо інструкцію call exit. Як уже зазначалося у теоретичному блоці, команда call вказує на виклик певної процедури (у цьому прикладі такою є процедура exit, яка відповідатиме за завершення роботи програми). Вона не відноситься до зарезервованих слів та команд і її роботу необхідно ще описати.

Загалом вся суть створюваної першої програми на мові Assembler полягає у завершенні роботи цієї ж програми, але вона достатньо коротка і за її допомогою досить легко проілюструвати всі кроки, починаючи від написання коду на мові програмування Asstmbler до її компіляції та запуску у терміналі Linux.

Отже опишемо код цієї процедури exit, текст якої представлено у прикладі 2-2.

Приклад 2-2.

```
exit:
    mov rax, 1
    mov rbx, 0
    int 0x80
```

Три рядки тіла процедури `exit` дозволяють вказати операційній системі, які сигнали надсилати процесору для завершення роботи представленої програми. Щоб це було виконано у спеціальні регістри необхідно перемістити певні значення. Для цього скористаємося командою `mov`, яка дозволяє передавати значення у визначені регістри. По суті відбувається цілеспрямована вказівка ОС, що має виконати процесор у конкретний момент часу. Так у першому рядку коду представленої процедури вказується, що значення 1 переноситься у регістр *rax* і це відповідає системному виклику завершення роботи програми. Наступним рядком `mov rbx, 0` вказуємо, яке значення має бути повернуте операційній системі у разі успішного завершення програми (0 – коректне завершення програми). Ну і останній рядок виконує системний виклик для операційної системи Linux. По суті відбувається передача керування від програми до ядра операційної системи.

Загалом код першої програми виконаної у асемблері має виглядати так, як це представлено у прикладі 2-3.

Приклад 2-3.

```
1. format ELF64
2. public _start

3. _start:
4.     call exit

5. exit:
6.     mov rax, 1 ; exit
7.     mov rbx, 0 ; return
8.     int 0x80
```

Тепер зберігаємо файл із розширенням *asm*, який вказуватиме на те, що використовуємо мову програмування Assembler. Цей текст програми було збережено у файл *Assem#01.asm*.

Далі розглянемо процес запуску щойно створеної програми. Оскільки використовується як основа компілятор FASM, то скористаємося ним для того, щоб створити об'єктний файл (Compiled Object File). Такий файл, по суті, містить двійковий запис створеної програми і може бути перетворений за допомогою програми-компонувальника у виконуваний файл. Для цього у терміналі необхідно набрати команду `fasm`, а далі, через пропуск, назву файлу із кодом (у терміналі ця команда виглядатиме так – `fasm Assem#01.asm`).

Якщо код був написано коректно і без помилок, а процес компіляції завершився успішно, то у терміналі має відобразитися повідомлення наступного зразка (рис. 2-2).

При цьому у папці буде створений новий файл із розширенням o (для цього прикладу таким файлом буде Assem#01.o). Далі виконуємо компонування програми із використанням утиліти ld. Команда у терміналі, яка дозволить виконати цю операцію виглядатиме наступним чином[1]: ld Assem#01.o –o Assem#01 (рис. 2-2). Результатом цієї операції буде створений виконуючий файл Assem#01 (пам'ятаємо, що у операційній системі Linux не обов'язково має розширення, як це можна спостерігати у ОС Windows).

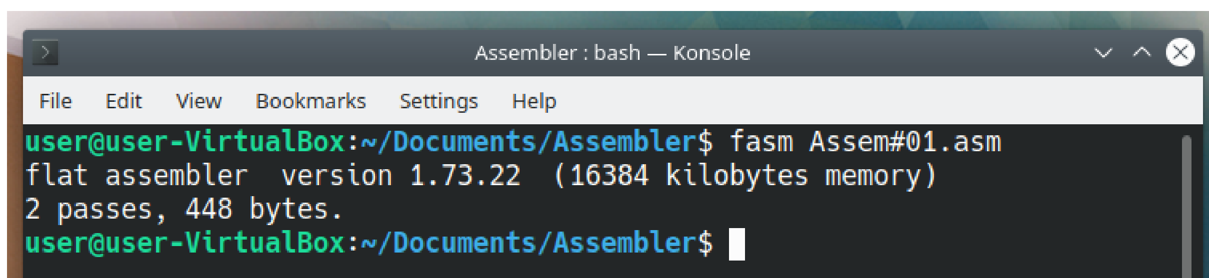


Рис. 2-2. Повідомлення fasm у разі успішного виконання компіляції файлу Assem#01.asm

Щоб запустити щойно створену програму достатньо у терміналі набрати наступну команду `./Assem#01`. Після виконання цього простого коду не буде виведено жодного повідомлення, а у терміналі з'явиться нове запрошення до введення нової команди, що свідчитиме про успішне виконання програми. Загалом всі описані дії у терміналі виглядатимуть наступним чином (рис. 2-3).

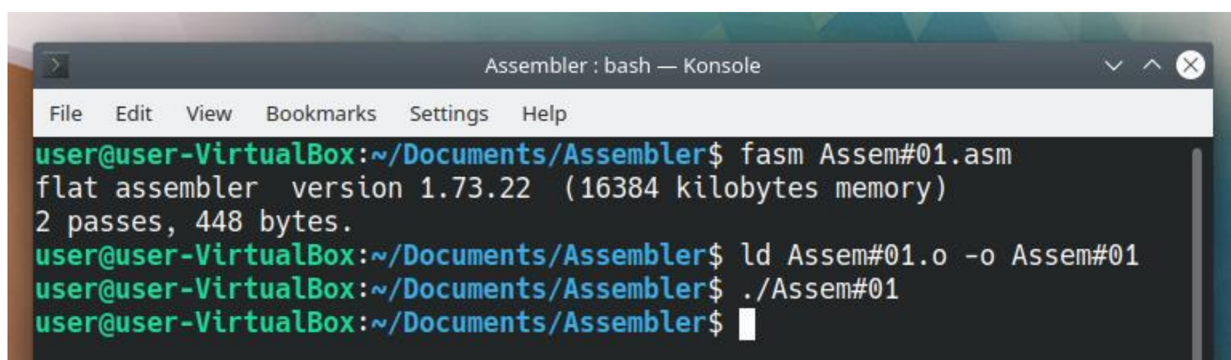


Рис. 2-3 Етапи процесу компонування програми, написаної на мові Assembler, у терміналі Linux, починаючи від компіляції та завершуючи запуском.

Тепер спробуємо вивести на екран текстове повідомлення, зокрема рядок із наступним змістом: «Hello students!!!». Для початку резервуємо область пам'яті. Ця дія виконується за допомогою наступної інструкції.

```
text db "Hello students!!!", 0
```

Розберемо більш детально цей запис. `text` – назва мітки, `db` – опис типу даних, які будуть зберігатися у комірках пам'яті. Сам запис `db` є скороченням

[1] У випадку, коли не відбувається компонування виконуючого файлу, то це означає, що у системі не встановлено необхідної для цього утиліти binutils. Для її встановлення в Debian подібних дистрибутивах необхідно у терміналі ввести команди sudo apt-get update і sudo apt-get install binutils-gold та погодитися на встановлення.

від `data bytes`. Далі, через пропуск у лапках розміщується власне сам вираз «Hello students!!!», а потім через кому і пропуск задається число 0 і таким чином вказуємо системі, що рядок завершився, а всі інші дані вже не будуть стосуватися цього текстового рядка.

Тепер явно вказуємо довжину рядка у символах.

```
len equ 18
```

Щойно описані інструкції мають бути розміщені перед запуском основної процедури `_start`. Тепер модифікуємо саму процедуру `_start` так, щоб вивести рядок у термінал Linux. Для цього необхідно у спеціальні регістри передати відповідні значення, які будуть вказівками для операційної системи до виконання дій друку. За ці операції відповідає команда `mov`, а загалом ця частина коду виглядатиме наступним чином.

```
mov rax, 4
mov rbx, 1
```

Далі необхідно перемістити дані (власне сам рядок та інформацію про кількість символів) про повідомлення у регістри, з який ОС буде забирати дані для виводу на екран.

```
mov rcx, text
mov rdx, len
```

Зважайте на те, що при виконанні цієї прогарами після її запуску в терміналі буде виведено саме повідомлення і разом із ним, у продовженні, запрошення операційної системи до введення наступної команди. Така ситуація виникає тільки через те, що не було явно вказано спеціалізованого символу. Нажаль Assembler не розуміє символ «\n», який у мовах програмування високого рівня вказує про переведення курсора на новий рядок. Щоб виконати таку дію у мітці, яка позначає зарезервовану пам'ять для рядка, додають код ASCII для символу «\n». У десятковій системі це число 10, а у шістнадсятковій 0xA.

```
text db "Hello students!!!", 10, 0
```

або

```
text db "Hello students!!!", 0xA, 0
```

Звичайно ж довжина рядка має становити вже не 18, а 19 символів.

Виконавши збірку модифікованої програми і запустивши її маємо отримати результат, аналогічний до того, який представлений на рисунку 2-4.

Загалом весь код програми буде виглядати наступним чином (приклад 2-4).

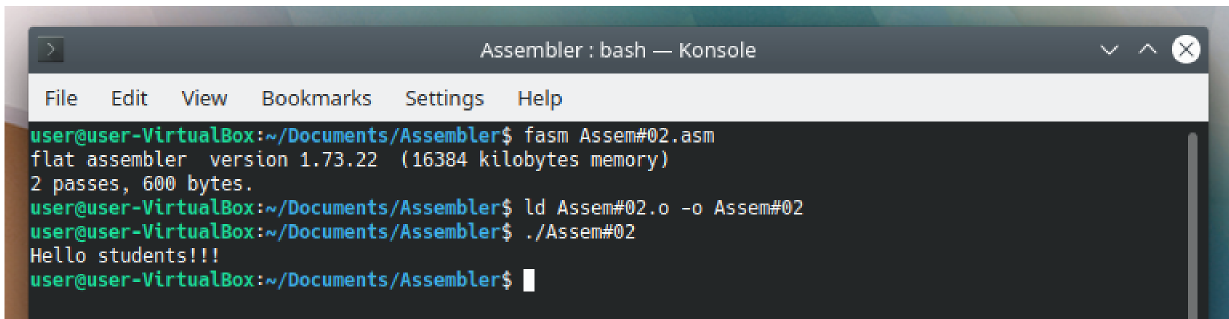


Рис. 2-4. Приклад виведення на екран повідомлення програмою, яка написана мовою Assembler

Приклад 2-4.

```
1.    format ELF64
2.    public _start
3.
4.    text db "Hello students!!!", 10, 0
5.    len equ 19
6.
7.    _start:
8.        mov rax, 4
9.        mov rbx, 1
10.       mov rcx, text
11.       mov rdx, len
12.       int 0x80
13.       call exit
14.
15.   exit:
16.       mov rax, 1
17.       mov rbx, 0
18.       int 0x80
```

Варто звернути увагу, що у основній процедурі викликається exit – процедура, яка дозволяє завершити роботу програми коректно та передати керування операційній системі.

Універсальна процедура для виведення тексту в термінал Linux

Якщо проаналізувати код останньої програми, то зразу ж буде виділятися ключовий її недолік: при зміні текстового рядка власноруч необхідно постійно вказувати його довжину. Для зручності сприймання код, який виконує друк повідомлення у терміналі, виокремити у окрему процедуру і викликати її лише тоді, коли це буде необхідно, а не переписувати йогог повністю.

Реалізаціє такої процедури має наступний вигляд (приклад 2-5)

Приклад 2-5

```
1.    format ELF64
2.    public _start
3.
4.    new_line equ 0xA
5.
6.    text db "hello", 10, 0
7.
8.    _start:
9.        mov rax, text
10.       call new_print
11.       call exit
12.
13.   new_print:
14.       push rax
15.       push rbx
16.       push rcx
```

```
17.         push rdx
18.         mov rcx, rax
19.         call length_str
20.         mov rdx, rax
21.         mov rax, 4
22.         mov rbx, 1
23.         int 0x80
24.         pop rdx
25.         pop rcx
26.         pop rbx
27.         pop rax
28.         ret
29.
30.     length_str:
31.         push rdx
32.         mov rdx, 0
33.         .iter:
34.             cmp [rax+rdx], byte 0
35.             je .close
36.             inc rdx
37.             jmp .iter
38.         .close:
39.             mov rax, rdx
40.             pop rdx
41.             ret
42.
43.     exit:
44.         mov rax, 1 ; exit
45.         mov rbx, 0 ; return
46.         int 0x80
```

Розберемо більш детально представлений вище код. Перша мітка (`new_line`) відповідатиме за перенесення курсора на новий рядок, а її програмне оформлення буде подано так як у рядку 6 цього коду.

Блок коду для основної процедури `_start` буде складатися із таких трьох рядків: у першому рядку процедури переміщаємо текстове повідомлення із пам'яті у регістр *rax*, далі викликаємо процедуру друку, роботу якої опишемо докладніше далі по тексту, а останній рядок здійснює вихід із програми та передає керування операційній системі.

Повноцінна робота процедури друку не можлива без автоматичного розрахунку довжини рядка. Вона описана починаючи із 30 по 41 рядок описаного вище коду. Розрахунок довжин відбувається наступним чином: за допомогою спеціалізованих циклічних інструкцій перебиратимемо всі символи до тих пір, поки не зустрінеться символ завершення рідка. Для мови низькорівневого програмування Assembler таким є 0. Розкриємо як цей описаний алгоритм поданий у коді прикладу 2-5.

Першим кроком убезпечуємо від випадкових змін регістр *rdx*, перемістивши значення у стек, а далі обнуляємо занчення в цьому регістрі.

Далі створюємо внутрішню процедуру (`.iter`), яка буде своєрідною міткою, до якої звертатимемося, щоб повторно виконати ряд дій. Важливо звернути увагу на те, що назва такої конструкції починається із крапки. Далі за допомогою команди `cmp` перевіряємо чи визначений байт рядка відповідає символу 0. У випадку рівності викликається процедура `.close`, яка у регістр *rax* поміщає довжину рядка, що зберігається у регістрі *rdx*, переміщає дані із стеку в регістр rdx та завершує роботу процедури `length_str`.

У випадку, коли умова, записана після ключового слова cmp не виконується, тоді збільшуємо значення у регістрі rdx (лічильник символів) на 1 і вказуємо про перехід програми на мітку `.iter`. Також важливо зауважити, що адресація комірки пам'яті виконується за допомогою квадратних дужок.

Тепер перейдемо до опису процедури, яка виводитиме текст у термінал Linux. У представленому вищі коді вона має назву `new_print`. На першому етапі дублюємо всі значення, які були розміщені у регістрах *rax*, *rbx*, *rcx*, *rdx* у стек і таким чином убезпечуємо дані, що там розміщувалися, від випадкової зміни. Далі дублюємо із регістра *rax* текст у регістр *rcx*. Далі викликаємо

процедуру розрахунку довжини рядка. Результат передаємо у регістр *rdx*. І на кінець повідомляємо операційній системі про друк повідомлення у терміналі.

Виведення числа у консоль

У попередньому прикладі було розглянуто програму для виведення на екран рядка тексту, тепер ж, опишемо процедуру для друку на екрані терміналу Linux натурального числа та числа 0. Проблемною є ситуація у тому, що мова Assembler не має вбудованих функцій для конвертації числа у рядок і навпаки. Тому потрібно створити процедуру, яка робитиме, по суті, це перетворення у процесі виведення числа на екран.

Для цього реалізуємо дві частини програми: для початку друкуватимемо на екрані один символ, а потім реалізуємо виведення всього числа. По суті число друкуватиметься по цифрам. Сам код програми, у якій реалізовано основні процедури подано у прикладі 2-6.

Приклад 2-6

```
1.    format ELF64
2.    public _start
3.
4.    buffer:
5.        buffer_char rb 1
6.
7.    _start:
8.        mov rax, "A"
9.        call print_char
10.       call new_line
11.       mov rax, 0
12.       call print_num
13.       call new_line
14.       call exit
15.
16.   print_char:
17.       push rax
18.       push rbx
19.       push rcx
20.       push rdx
21.       mov [buffer_char], al
22.       mov rax, 4
23.       mov rbx, 1
24.       mov rcx, buffer_char
25.       mov rdx, 1
26.       int 0x80
27.       pop rdx
28.       pop rcx
29.       pop rbx
30.       pop rax
31.       ret
32.
33.   new_line:
34.       push rax
35.       mov rax, 0xA
36.       call print_char
37.       pop rax
38.       ret
38.   print_num:
39.       push rax
40.       push rbx
41.       push rcx
42.       push rdx
43.       mov rcx, 0
44.       .digit_to_stack:
45.           mov rdx, 0
46.           mov rbx, 10
47.           div rbx
48.           add rdx, "0"
49.           push rdx
50.           inc rcx
51.           cmp rax, 0
52.           je .print_digit
53.           jmp .digit_to_stack
54.       .print_digit:
55.           cmp rcx, 0
56.           je .close
57.           pop rax
58.           call print_char
59.           dec rcx
60.           jmp .print_digit
61.       .close:
62.           pop rdx
63.           pop rcx
64.           pop rbx
65.           pop rax
66.           ret
67.
68.   exit:
69.       mov rax, 1 ; exit
70.       mov rbx, 0 ; return
71.       int 0x80
```

Якщо поглянути на представлений приклад, то у ньому для виведення числа реалізовано три процедури: одна для збереження даних та дві `print_char` та `print_num` для виведення символу і числа на екран. Окремо було створено процедуру для переведення курсора на новий рядок. Опишемо їх докладніше.

Отже першою важливою процедурою є `buffer`. Вона не виконує жодних дій, лише містить опис мітки, яка позначає область пам'яті у якій зберігатиметься лише один байт інформації, чого цілком достатньо для розміщення одного символа у кодування ASCII.

Процедура `print_char` є спрощеним варіантом процедури, яка виводить у термінал звичний рядок тексту. Послідовність операцій така ж, єдина відмінність полягає в тому, що для передачі на друк символа, спочатку поміщаємо у завчасно створену область пам'яті (`buffer_char`), а довжина рядка складатиме всього один символ.

На основі цієї процедури у прикладі продемонстровано приклад підпрограми, яка дозволяє переводити курсор на новий рядок у терміналі (`new_line`).

Тепер розглянемо процедуру друку натуральних чисел (`print_num`). Вона містить три важливі частини: розклад числа на цифри, друк цифр та завершення процедури. Вони реалізовані за допомогою підпроцедур: `.digit_to_stack, .print_digit, .close.`

На початку, щоб не втратити дані у регістрах загального користування, переміщуємо інформацію у стек. Далі у регістр *rcx* поміщаємо нуль. Фактично він відповідатиме за обрахунок кількості цифр у числі.

Суть першої підпроцедури виокремити у представленому числі цифри і помістити їх у стек послідовно, починаючи із найменшого розряду. Робота алгоритму полягає у тому, що число ділиться на кожному кроці на 10. Остачею, у такому випадку будуть одиниці, десятки, сотні, тисячі тощо. Це буде відбуватися доти, доки частка не стане дорівнюватиме нулю, що і буде зупинкою повторного виконання визначеного блоку команд.

Коротко охарактеризуємо команди, які цей цикл утворюють (рядки 44 – 53 прикладу 2-6). Першим кроком (команда cmp) задаємо умови, при яких відбувається перехід до нової підпроцедури. Якщо така умова виконається, то наступною буде виконуватися процедура `.print_digit.` Зауважимо, що при діленні частка поміщатиметься у регістр *rax*, а остача від ділення – у регістр *rdx*. Далі відбувається конвертація числа у символ (`add rdx, "0"`) і цей символ поміщається у стек, а лічильник у регістрі *rcx* збільшується на 1. Зауважимо, що до числа, додається символ нуль, а не число 0. Це можливо тільки через те, що Assembler автоматично замінює символ 0 на його код у таблиці ASCII (код «0» у таблиці має значення 48). Таким чином додаючи до коду 48 цифру отримуватимемо ASCII код відповідної цифри: «0» – 48, «1» – 49, «2» – 50, «3» – 51, …

Як тільки у регістрі *rax* отримаємо значення 0 програма перейде до виконання підпроцедури `.print_digit.` Вона працює у зворотному порядку. Умовою виходу із постійних викликів підпроцедури `.print_digit` є ситуація, коли регістрі *rcx* буде розміщуватися значення 0. Це означитиме, що програма

повернула із стеку всі символи цифр та надрукувала їх у терміналі. Якщо вона буде виконана, то керування буде передано процедурі `.close`, яка зупинить роботу процедури `print_num`. Загалом підпроцедура `.close` є досить простою. У ній забераємо із стека всі попередні значення регістрів та повертаємо керування програмою основній процедурі `_start`.

Реалізація функцій конвертації натурального числа у рядок і навпаки, із рядка у натуральне число

Загалом процедура конвертації числа у рядок повторює алгоритм попередньої програми, яка стосувалася друку числа у терміналі. Єдиною відмінністю від попереднього прикладу є те, що у цьому випадку конвертоване числове значення має зберігатися у пам'яті комп'ютера. Код самої програми представлено прикладі 2-7.

Приклад 2-7.

```
1.   format ELF64
2.   public _start
3.
4.   count_digits equ 256
5.
6.   buffer:
7.       buffer_char rb 1
8.       buffer_num rb count_digits
9.
10.
11.  _start:
12.      mov rax, 6842
13.      mov rbx, buffer_num
14.      mov rcx, count_digits
15.      call num_to_str
16.      mov rax, buffer_num
17.      call new_print
18.      call new_line
19.      call exit
20.
21.  print_char:
22.      push rax
23.      push rbx
24.      push rcx
25.      push rdx
26.      mov  [buffer_char], al
27.      mov rax, 4
28.      mov rbx, 1
29.      mov rcx, buffer_char
30.      mov rdx, 1
31.      int 0x80
32.      pop rdx
33.      pop rcx
34.      pop rbx
35.      pop rax
36.      ret
37.
38.  new_line:
39.      push rax
40.      mov rax, 0xA
41.      call print_char
42.      pop rax
43.      ret
44.
45.  print_num:
46.      push rax
47.      push rbx
48.      push rcx
49.      push rdx
50.      mov rcx, 0
51.      .digit_to_stack:
52.          mov rdx, 0
53.          mov rbx, 10
54.          div rbx
55.          add rdx, "0"
56.          push rdx
57.          inc rcx
58.          cmp rax, 0
59.          je .convert_to_string
60.          jmp .digit_to_stack
61.      .convert_to_string:
62.          cmp rcx, 0
63.          je .close
64.          pop rax
65.          call print_char
66.          dec rcx
67.          jmp .convert_to_string
68.      .close:
69.          pop rdx
70.          pop rcx
71.          pop rbx
72.          pop rax
73.          ret
74.
75.  num_to_str:
76.      push rax
```

```
77.      push rbx
78.      push rcx
79.      push rdx
80.      push rsi
81.      mov rsi, rcx
82.      dec rsi
83.      mov rcx, 0
84.      .digit_to_stack:
85.          push rbx
86.          mov rdx, 0
87.          mov rbx, 10
88.          div rbx
89.          pop rbx
90.          add rdx, "0"
91.          push rdx
92.          inc rcx
93.          cmp rax, 0
94.          je .next_step
95.          jmp .digit_to_stack
96.      .next_step:
97.          mov rdx, rcx
98.          mov rcx, 0
99.      .convert_to_string:
100.         cmp rcx, rsi
101.         je .pop_iter
102.         cmp rcx, rdx
103.         je .null_char
104.         pop rax
105.         mov [rbx+rcx], rax
106.         inc rcx
107.         jmp .convert_to_string
108.     .pop_iter:
109.         cmp rcx, rdx
110.         je .close
111.         pop rax
112.         inc rcx
113.         jmp .pop_iter
114.     .null_char:
115.         mov rsi, rdx
116.     .close:
117.         mov [rbx+rsi], byte 0
118.             pop rsi
119.             pop rdx
120.             pop rcx
121.             pop rbx
122.             pop rax
123.             ret
124.
125. new_print:
126.     push rax
127.     push rbx
128.     push rcx
129.     push rdx
130.     mov rcx, rax
131.     call length_str
132.     mov rdx, rax
133.     mov rax, 4
134.     mov rbx, 1
135.     int 0x80
136.     pop rdx
137.     pop rcx
138.     pop rbx
139.     pop rax
140.     ret
141.
142. length_str:
143.     push rdx
144.     xor rdx, rdx
145.     .iter:
146.         cmp [rax+rdx], byte 0
147.         je .close
148.         inc rdx
149.         jmp .iter
150.     .close:
151.         mov rax, rdx
152.         pop rdx
153.         ret
154. exit:
155.     mov rax, 1 ; exit
156.     mov rbx, 0 ; return
157.     int 0x80
158.
```

Представлений код містить і попередні процедури, були описані у попередніх прикладах, так що їх можна використовувати для перевірки коректності роботи процедури конвертації.

Коротко опишемо важливі зміни у процедурі `buffer`. Перед самою процедурою оголосимо мітку на пам'ять, у якій зберігатиметься максимальна довжина рядка. У даному випадку це 256 символів. Далі у процедурі `buffer` створимо мітку на пам'ять для збереження конвертованого у рядок числа (`buffer_num rb count_digits`).

Тепер розкриємо принцип роботи процедури для конвертації числа у рядок. У представленому вище прикладі за це відповідає код починаючи із рядка з номером 75 до рядка – 123.

У основній процедурі (12 – 14 рядки) записані команди, після яких число переносять у регістер *rax*, регістр *rbx* пов'язують із зарезервованою частиною пам'яті, а у *rcx* вносять максимальну можливу довжину нового рядка. Стан стеку і регістрів, які будуть задіяні у роботі процедури представлені у таблиці 2-9.

Таблиця 2-9.

buffer_num	rax	rbx	rcx	rdx	rsi	Стек
	6842	~	256	Дані попередніх програм	Дані попередніх програм	Дані попередніх програм

Після виклику процедури num_to_str, за допомогою команди push, всі дані передаються у стек і тепер розподіл даних виглядатиме так як представляє таблиця 2-10.

Таблиця 2-10.

buffer_num	rax	rbx	rcx	rdx	rsi	Стек
	6842	~	256	Дані попередніх програм	Дані попередніх програм	Дані попередніх програм із стека **rsi** Дані попередніх програм із стека **rdx** 256 ~ 6842 Дані попередніх програм

Переносимо інформацію про максимальну довжину рядка із регістра *rcx* у регістр *rsi* (таблиця 2-11).

Таблиця 2-11.

buffer_num	rax	rbx	rcx	rdx	rsi	Стек
	6842	~	256	Дані попередніх програм	256	Дані попередніх програм із стека **rsi** Дані попередніх програм із стека **rdx** 256 ~ 6842 Дані попередніх програм

Зменшуємо значення у регістрі *rsi* на 1, для того щоб записати останній символ 0 – символ завершення рядка, а також обнуляємо регістр *rcx* (таблиці 2-12).

Таблиця 2-12.

buffer_num	rax	rbx	rcx	rdx	rsi	Стек
	6842	~	0	Дані попередніх програм	255	Дані попередніх програм із стека **rsi** Дані попередніх програм із стека **rdx** 256 ~ 6842 Дані попередніх програм

Далі у регістр *rdx* вносимо значення 0, а у регістр *rbx* – 10. Підпроцедура `.digit_to_stack` дозволить розділити число на цифри. Команда `div` виконує ділення значення, яке знаходиться у регістрі *rax* на значення у регістрі *rbx* (тобто на 10). При цьому на початку підпроцедури посилання на пам'ять `buffer_num`, переміщаємо у стек (таблиця 2-13).

Таблиця 2-13.

buffer_num	rax	rbx	rcx	rdx	rsi	Стек
	684	10	0	2	255	~ Дані попередніх програм із стека **rsi** Дані попередніх програм із стека **rdx** 256 ~ 6842 Дані попередніх програм

Наступним кроком відбувається переведення, власне, числа у символ, яким цю цифру позначають. Конвертація аналогічна до того підходу, який був продемонстрований при створенні процедури друку чисел (за допомогою команди `add`). Отже, у результаті виконання цих операцій маємо 50 – ASCII код цифри 2. Після чого поміщаємо код символу стек, а лічильник цифр (значення регістру *rcx*) збільшуємо на 1. Після всіх операцій стан системи виглядатиме наступним чином (таблиця 2-14).

Таблиця 2-14.

buffer_num	rax	rbx	rcx	rdx	rsi	Стек
	684	10	1	2	255	50 Дані попередніх програм із стека **rsi** Дані попередніх програм із стека **rdx** 256 ~ 6842 Дані попередніх програм

Підпроцедура `.digit_to_stack` викликається доти, доки значення у регістрі *rax* не буде рівним 0. Наступні таблиці (від 2-15 до 2-17) ілюструють проміжні етапи виконання алгоритму, що дозволяє конвертувати кожну цифру числа у символ та перенести його у стек.

Якщо ж значення у регістрі буде рівним 0, то програма переходить до виконання процедури `.next_step`. Вона дозволяє перемістити значення про кількість цифр числа із регістра *rcx* у регістр *rdx*. Також буде відновлено посилання до області пам'яті, яке зарезервоване за міткою `buffer_num`, що буде відбуватися циклічно на кожні ітерації.

Таблиця 2-15.

buffer_num	rax	rbx	rcx	rdx	rsi	Стек
	68	10	2	4	255	52 50 Дані попередніх програм із стека **rsi** Дані попередніх програм із стека **rdx** 256 ~ 6842 Дані попередніх програм

Таблиця 2-16.

buffer_num	rax	rbx	rcx	rdx	rsi	Стек
	6	10	3	8	255	56 52 50 Дані попередніх програм із стека **rsi** Дані попередніх програм із стека **rdx** 256 ~ 6842 Дані попередніх програм

Таблиця 2-17.

buffer_num	rax	rbx	rcx	rdx	rsi	Стек
	0	10	4	6	255	54 56 52 50 Дані попередніх програм із стека **rsi** Дані попередніх програм із стека **rdx** 256 ~ 6842 Дані попередніх програм

У підпроцедурі `.convert_to_string` відбувається зворотній процес – формування рядка із стеку. Повторні виклики процедури будуть відбуватися доти, доки не будуть вилучені всі символи цифр із стека та поміщені у зарезервовано область пам’яті. Наступні таблиці (від 2-18 до 2-21) ілюструють стан стеку, регістрів та зарезервованої області пам’яті.

Таблиця 2-18.

buffer_num	rax	rbx	rcx	rdx	rsi	Стек
54,	0	54,	1	4	255	56 52 50 Дані попередніх програм із стека **rsi** Дані попередніх програм із стека **rdx** 256 ~ 6842 Дані попередніх програм

Таблиця 2-19.

buffer_num	rax	rbx	rcx	rdx	rsi	Стек
54, 56,	0	54, 56,	2	4	255	52 50 Дані попередніх програм із стека **rsi** Дані попередніх програм із стека **rdx** 256 ~ 6842 Дані попередніх програм

Таблиця 2-20.

buffer_num	rax	rbx	rcx	rdx	rsi	Стек
54, 56, 52,	0	54, 56, 52,	3	4	255	50 Дані попередніх програм із стека **rsi** Дані попередніх програм із стека **rdx** 256 ~ 6842 Дані попередніх програм

Перехід `je .pop_iter` дозволяє вивільнити стек повністю, якщо конвертоване число має більше цифр ніж 255, а `je .null_char` надає змогу завершити рядок, коли цифр у числі менше 255. По суті після виконання цієї процедури регістр rsi матиме значення 4, що, у свою чергу, дозволить точно вказати (4 + 1) на розміщення символу завершення рядка. Інструкція, яка записана першою у завершальній підпорцедурі `.close`, саме дозволяє це зробити, а решта команд забезпечує вивільнення решти інформації із стеку та передається керування основній процедурі. У основній процедурі отриманий рядок буде передано у регістр *rax*, що дозволить його вивести у терміналі за допомогою вже створених процедур.

Таблиця 2-21.

buffer_num	rax	rbx	rcx	rdx	rsi	Стек
54, 56, 52, 50	0	54, 56, 52, 50	4	4	255	Дані попередніх програм із стека **rsi** Дані попередніх програм із стека **rdx** 256 ~ 6842 Дані попередніх програм

Тепер розглянемо іншу ситуацію, коли натуральне число, яке записане у рядку, необхідно представити у звичному числовому вигляді. Алгоритм схожий до того, який дозволяє друкувати текстовий рядок. Розглянемо його більш докладно (приклад 2-8).

Приклад 2-8.

```
1.   format ELF64
2.   public _start
3.
4.   count_digits equ 256
5.   num_str db "2564", 0
6.
7.   buffer:
8.       buffer_char rb 1
9.       buffer_num rb count_digits
10.
11.  _start:
12.      mov rax, num_str
13.      call str_to_num
14.      call print_num
15.      call new_line
16.      call exit
17.
18.  str_to_num:
19.      push rbx
20.      push rcx
21.      push rdx
22.      mov rbx, 0
23.      mov rcx, 0
24.      .char_to_stack:
25.          cmp [rax+rbx], byte 0
26.          je .next_step
27.          mov cl, [rax+rbx]
28.          sub cl, "0"
29.          push rcx
30.          inc rbx
31.          jmp .char_to_stack
32.      .next_step:
33.          mov rcx, 1
34.          mov rax, 0
35.      .convert_to_num:
36.          cmp rbx, 0
37.          je .close
38.          pop rdx
39.          imul rdx, rcx
40.          imul rcx, 10
41.          add rax, rdx
42.          dec rbx
43.          jmp .convert_to_num
44.      .close:
45.          pop rdx
46.          pop rcx
47.          pop rbx
48.          ret
49.
50.  print_char:
51.      push rax
52.      push rbx
53.      push rcx
54.      push rdx
55.      mov  [buffer_char], al
56.      mov rax, 4
57.      mov rbx, 1
58.      mov rcx, buffer_char
59.      mov rdx, 1
60.      int 0x80
61.      pop rdx
62.      pop rcx
63.      pop rbx
64.      pop rax
65.      ret
66.
67.  new_line:
68.      push rax
69.      mov rax, 0xA
70.      call print_char
71.      pop rax
72.      ret
73.
74.  print_num:
75.      push rax
76.      push rbx
77.      push rcx
78.      push rdx
79.      mov rcx, 0
80.      .digit_to_stack:
81.          mov rdx, 0
82.          mov rbx, 10
83.          div rbx
84.          add rdx, "0"
```

```
85.         push rdx
86.         inc rcx
87.         cmp rax, 0
88.         je .convert_to_string
89.         jmp .digit_to_stack
90.     .convert_to_string:
91.         cmp rcx, 0
92.         je .close
93.         pop rax
94.         call print_char
95.         dec rcx
96.         jmp .convert_to_string
97.     .close:
98.         pop rdx
99.         pop rcx
100.        pop rbx
101.        pop rax
102.        ret
103.
104. exit:
105.     mov rax, 1 ; exit
106.     mov rbx, 0 ; return
107.     int 0x80
108.
```

Дані у процедуру будуть передаватися через регістр *rax* і у цей же регістр повертатимемо числове значення.

Для початку, як уже часто робилося, збережемо поточні дані регістрів у стеку (зокрема інформацію із регістрі *rbx*, *rcx*, *rdx*). Обнуляємо значення у цих же регістрах. У *rbx* буде зберігатися кількість цифр числа, яке записане у вигляді рядка, а *rcx* використовуватимемо для збереження розряду.

Далі розглянемо цикл, що дозволяє перемістити всі цифри у стек для подальших операцій із ними. За цю дію відповідає підпроцедура `.char_to_stack`. Програма посимвольно переглядає рядок і переміщає у стек всі символи, які позначають цифри, починаючи із найбільшого розряду. Це робиться для того, щоб у стеку останнім було поміщено одиниці. Тоді для формування числа необхідно із стеку вийняти одиниці, потім цифру для десятків, сотень, тисяч тощо. Тобто ця дія буде аналогічною до такої операції.

$$4*1 + 6*10 + 5*100 +2*1000 = 2564$$

Умова виходу із такого циклу, це досягнення кінця рядка – тобто символу 0.

Далі поміщаємо вибраний символ у найнижчий регістр *cl* (інструкція `mov cl [rax+rbx])` і виконаємо конвертацію у цифру. Ця операція вимагає додаткового пояснення.

У регістрі cl знаходиться лише ASCII код символу, який відповідає певній цифрі, а у той же час необхідно отримати саму цифру. Щоб таку операцію виконати треба від ASCII коду цифри відняти значення символу нуль. Оскільки Assembler замість символу «0» підставить значення його коду, то у результаті отримаємо необхідний результат. Після чого цифру поміщаємо у стек.

Коли цикл завершиться програма преходить до підпрограми `.next_step`, у якій у регістр *rcx* переміщаємо 1, а у регістр *rax* поміщаємо 0. Це необхідно зробити для того щоб *rax* поміщати результат конвертації, а у *rcx* міститиметься еквівалент розряду (1 – одиниці; 10 – десятки; 100 – сотні; 1000 – тисячі …).

Тепер розглянемо процедуру, яка братиме цифру із стека множитиме на розряд та додаватиме до значення, поміщеного у регістрі та оновлюючи там значення. За ці дії відповідатиме `.convert_to_num`. Умовою виходу із повторного визову цієї процедури буде ситуація, коли мінімальна цифра, яка зберігається у регістрі *rbx* буде рівна 0.

Далі забираємо із стека цифру, яка відповідає одиницям, виконуємо множення на розряд, яке розміщено у регістрі *rcx*; збільшуємо розряд, виконуємо сумування результату і збереження його у регістрі rax та зменшуємо лічильник на 1. І так повторюємо тіло процедури `.convert_to_num` до тих пір поки не вилучимо всі цифри із стеку.

Результат виконання процедури `str_to_num` буде число, яке розміщуватиметься у регістрі *rax*, а отже його можна надрукувати у терміналі вже створеними раніше підпрограмами, наприклад `print_num`.

Процедура для считування текстового рядка із терміналу

Тепер, коли описано процедури для роботи із рядком і числами, а також представлені команди для друку їх у терміналі Linux, важливо розкрити яким чином можна вводити дані – зокрема текстовий рядок. Текст програми представлено у прикладі 2-9.

Приклад 2-9

```
1.  format ELF64
2.  public _start
3.
4.  count_digits equ 256
5.  count_chars equ 256
6.
7.  buffer:
8.      buffer_char rb 1
9.      buffer_num rb count_digits
10.     buffer_str rb count_chars
11.
12.
13. _start:
14.     mov rax, buffer_str
15.     mov rbx, count_chars
16.     call input_str
17.     call new_print
18.     call new_line
19.     call exit
20.
21. input_str:
22.     push rax
23.     push rbx
24.     push rcx
25.     push rdx
26.     push rax
27.     mov rcx, buffer_str
28.     mov rdx, rbx
29.     mov rax, 3
30.     mov rbx, 2
31.     int 0x80
32.     pop rbx
33.     mov [rbx+rax-1], byte 0
34.     pop rdx
35.     pop rcx
36.     pop rbx
37.     pop rax
38.     ret
39.
40. print_char:
41.     push rax
42.     push rbx
43.     push rcx
44.     push rdx
45.     mov  [buffer_char], al
46.     mov rax, 4
47.     mov rbx, 1
48.     mov rcx, buffer_char
49.     mov rdx, 1
50.     int 0x80
51.     pop rdx
52.     pop rcx
53.     pop rbx
54.     pop rax
55.     ret
56.
57. new_line:
58.     push rax
59.     mov rax, 0xA
60.     call print_char
61.     pop rax
62.     ret
63.
64. new_print:
65.     push rax
66.     push rbx
67.     push rcx
68.     push rdx
69.     mov rcx, rax
70.     call length_str
71.     mov rdx, rax
72.     mov rax, 4
73.     mov rbx, 1
74.     int 0x80
75.     pop rdx
76.     pop rcx
77.     pop rbx
78.     pop rax
79.     ret
80.     push rax
```

```
81.  length_str:                          89.       .close:
82.      push rdx                         90.           mov rax, rdx
83.      xor rdx, rdx                     91.           pop rdx
84.      .iter:                           92.           ret
85.          cmp [rax+rdx], byte 0        93.   exit:
86.          je .close                    94.       mov rax, 1 ; exit
87.          inc rdx                      95.       mov rbx, 0 ; return
88.          jmp .iter                    96.       int 0x80
```

Загалом процедура `input_str` складається із таких частин. По-перше записуємо інструкції, які дозволяють розмістити дані із регістрів у стеку. Далі у регістрах *rcx* та *rdx* розміщуємо посилання на зарезервовану область пам'яті та вказуємо, яку максимальну кількість символів може бути розміщено у рядку.

Наступним кроком у регістр *rax* поміщаємо значення 3 (фактично даємо вказівку операційній системі почати зчитувати дані із терміналу), а операція `mov rbx, 2` вказує на стандартний пристрій введення – клавіатуру.

Задаємо системний виклик `int 0x80` та додаємо у кінці символ 0, позначаючи таким чином кінець рядка.

Для того, щоб процедура працювала коректно спочатку у регістр *rax* розмістимо посилання на область пам'яті, а у *rbx* – кількість комірок, які будуть резервуватися. Результат зчитування даних, після їх введення, буде поміщено у регістр *rax*. Такий підхід дозволить використовувати дані із регістра для подальших операцій над ними, зокрема виведенням на екран.

Створення системних бібліотек

Запропоновані приклади програм, створених мовою Assembler, варто використовувати комплексно – наприклад створити бібліотеку і на її основі вже розробляти програми для власних потреб. Тому розглянемо як такий підхід можна реалізувати для компілятора FASM.

На основі вже створених процедур продемонструємо яким чином такий підхід реалізувати.

Спочатку створимо каталог, який міститиме основний файл проекту та каталог для створюваної бібліотеки (наприклад *lib* або *source*). Назва самого каталогу проекту може бути довільною, але повинна відображати його сутність. Далі у каталозі *lib* створимо наступні файли: *sys.asm* у якій розмістимо процедури завершення програми та зчитування рядка із консолі; *print.asm* міститиме підпрограми для друку символів, чисел та рядків, а також переведення курсора на новий рядок; *convert.asm* – засоби для конвертації натурального числа у рядок та навпаки. Текст цих файлів представлений у прикладах …

Більш детально розберемо приклад файлу *print.asm* (приклад 2-10). На початку вказується тип та розрядність операційної системи. Далі помічаємо, що всі процедури, які міститимуться у файлі вважатимуться публічними або ж відкритими і лише після цих ключових інструкцій розміщуємо код самих процедур.

Приклад 2-10

```
1.   format ELF64
2.
3.   public new_print
4.   public print_num
5.   public new_line
6.   public length_str
7.
8.   buffer:
9.       buffer_char rb 1
10.
11.  new_print:
12.      push rax
13.      push rbx
14.      push rcx
15.      push rdx
16.      mov rcx, rax
17.      call length_str
18.      mov rdx, rax
19.      mov rax, 4
20.      mov rbx, 1
21.      int 0x80
22.      pop rdx
23.      pop rcx
24.      pop rbx
25.      pop rax
26.      ret
27.
28.  length_str:
29.      push rdx
30.      xor rdx, rdx
31.      .iter:
32.          cmp [rax+rdx], byte 0
33.          je .close
34.          inc rdx
35.          jmp .iter
36.      .close:
37.          mov rax, rdx
38.          pop rdx
39.          ret
40.
41.  print_char:
42.      push rax
43.      push rbx
44.      push rcx
45.      push rdx
46.      mov  [buffer_char], al
47.      mov rax, 4
48.      mov rbx, 1
49.      mov rcx, buffer_char
50.      mov rdx, 1
51.      int 0x80
52.      pop rdx
53.      pop rcx
54.      pop rbx
55.      pop rax
56.      ret
57.
58.  new_line:
59.      push rax
60.      mov rax, 0xA
61.      call print_char
62.      pop rax
63.      ret
64.
65.  print_num:
66.      push rax
67.      push rbx
68.      push rcx
69.      push rdx
70.      mov rcx, 0
71.      .digit_to_stack:
72.          mov rdx, 0
73.          mov rbx, 10
74.          div rbx
75.          add rdx, "0"
76.          push rdx
77.          inc rcx
78.          cmp rax, 0
79.          je .print_digit
80.          jmp .digit_to_stack
81.      .print_digit:
82.          cmp rcx, 0
83.          je .close
84.          pop rax
85.          call print_char
86.          dec rcx
87.          jmp .print_digit
88.      .close:
89.          pop rdx
90.          pop rcx
91.          pop rbx
92.          pop rax
93.          ret
94.
```

Далі створимо файл main.asm, який буде розміщено у основному каталозі проекту. Її текст представлено нижче.

Розберемо її код більш детально. Інструкції `format` та `public _start` є стандартними, які вказують на розрядність операційної системи і її тип та основну процедуру програми. Потім наводимо перелік процедур, які будемо імпортувати та виконувати у програмі. Для цього користуємося інструкцією extrn. І лише після цього описуємо основну програму в процедурі `_start`.

Наступним етапом є компонування програми на основі створеної бібліотеки та файлу *main.asm*. Так спершу потрібно створити двійкові об’єктні

файли із тих, що розміщені у каталозі *lib*. Для цього виконуємо послідовно команди `fasm sys.asm`, `fasm print.asm`, `convert.asm`. А також таку дію треба виконати і для файлу main.asm.

Приклад 2-10.

```
1.   format ELF64

2.   public _start

3.   extrn str_to_num
4.   extrn print_num
5.   extrn new_line
6.   extrn exit

7.   num_str db "2564", 0

8.   _start:
9.   mov rax, num_str
10.  call str_to_num
11.  call print_num
12.  call new_line
13.  call exit
```

І останній крок це компонування всіх об'єктний файлів у один виконуваний. Це робиться за допомогою команди `ld` у терміналі. Для представленого прикладу вона виглядатиме наступним чином: `ld main.o lib/convert.o lib/print.o lib/sys.o -o main`. Звичайно, команда має виконуватися у тому ж каталозі, де розміщений основний файл.

Лабораторна робота 6

Тема. Базові команди програмування мовою Assembler

Мета: навчитися встановлювати компілятор FASM та розгортати для нього інтегроване середовище розробки в операційній системі Linux

Хід виконання

1. У віртуальній машині всатновити Debian-подібний дистрибутив Linux.
2. Встановіть компілятор FASM та утиліту ld для компонування виконуючих файлів.
3. Підготовте операційну систему до встановлення вільнопоширюваного інтегрованого середовища розробки Visual Studio Code та встановіть його.
4. Розгорніть у ньому доповнення Retro Assembler.
5. Наберіть першу програму мовою Assembler, скомпілюйте її у бінарний об’єктний файл, виконайте компонування та здійсніть запуск цієї програми.
6. Проаналізуйте код та поясніть призначення усіх його команд.
7. Розберіть кожен із представлених прикладів та виконайте їх набір та запуск.
8. Дослідіть їх роботу покроково та поясніть призначення кожної інструкції.

Контрольні запитання

1. Поняття мови програмування Assembler, види його компілятора та їх призначення.
2. Чим відрізняються діалекти мови Assembler (FASM, MASM та NASM)?
3. Налаштування робочого середовища для створення програм мовою Assembler в операційній системі Linux.
4. Особливості створення бінарних об’єктних файлів та компонування програми і її запуск у терміналі Linux.

Лабораторна робота 7

Тема. Програмування мовою Assembler

Мета: навчитися створювати програми мовою Assembler

Хід виконання

1. Напишіть програму за допомогою низькорівневої мови програмування Assembler для задачі,яка вказана у Вашому варіанті[2].
2. Використайте вже створені на поперредній лабораторній роботі процедури та інтегруйте їх у єдину бібліотеку для використання у новоствореній програмі.
3. Виконайте компіляцію та компонування програми.
4. Здійсність запуск програми у терміналі та продемонструйте результати викладачу.

Варіант	Завдання
1	Встановити рівень навчальних досягнень учня (початковий, середній, достатній, високий) відповідно до заданої оцінки (від 1 до 12). Вивести Initial для початкового рівня (оцінка від 1 до 3), Average для середнього (оцінка від 4 до 6), Sufficient для достатнього (оцінка від 7 до 9) і High для високого (оцінка від 10 до 12).
2	Задано трицифрове число. Визначити, яка цифра в ньому є більшою – перша чи остання. Вивести більшу з вказаних цифр. У випадку їх рівності вивести знак "=" (без лапок).
3	Визначити тип трикутника (рівносторонній, рівнобедрений, різносторонній) за заданими довжинами його сторін.
4	Задано трицифрове число. Визначити добуток його цифр.
5	Визначити назву пори року за заданим номером місяця, використовуючи складені умови. Для весняних місяців вивести Spring, для літніх - Summer, для осінніх - Autumn і для зимових - Winter.
6	На вході програми маємо два натуральних числа a та n, кожне в окремому рядку. На вихід потрібно подати an.
7	Програма на вході приймає натуральне число m – вага в грамах. Потрібно вивести в одному рядку через проміжок повні кілограми та грами цієї ваги.
8	Програма на вході приймає натуральне число m – довжина в сантиметрах. Потрібно вивести в одному рядку через проміжок повні метри та сантиметри цієї довжини.
9	Від початку доби пройшло m хвилин. Скільки годин і хвилин показує годинник на цей момент?
10	На вхід програми подається деякий текст, наприклад True. Потрібно продублювати цей текст в першому рядку, двічі повторити в другому і тричі в третьому. В окремих рядках між копіями тексту має бути проміжок.

Контрольні запитання

1. Яким чином можна перетворити звичайний файл asm у підключаємий модуль?
2. Яка команда дозволяє виконати підключення певної бібліотеки?
3. Як відбувається збірка програми, що створена на основі бібліотеки?.

[2] Всі завдання взято із сайту https://www.eolymp.com/uk/.

Додаток 1

Список рекомендованої літератури

Основна:

1. Авраменко В. С., Авраменко А. С. Основи операційних систем. Навчальний посібник. Черкаси: ЧНУ імені Богдана Хмельницького, 2018. 524 с.
2. Головня О.С. Технології віртуалізації у навчанні операційних систем бакалаврів інформатики. Методичні рекомендації для викладачів вищ.навч.закл. Житомир: Рута, 2017. 54 с.
3. Головня О. С. Операційні системи та системне програмування: методичний посібник для студ. вищих. навч. закл. Житомир: Рута, 2018. 338 с.:
4. Лав Р. Ядро Linux. Описание процесса разработки. Вильямс, 2013. 496 с.
5. Федотова-Півень М.І., Миронець І.В., Півень О.Б., Сисоєнко С.В., Миронюк Т.В. Операційні системи: навч. посіб. Черкаси : ЧДТУ, 2019. 225 с.
6. Харченко В. П., Знаковська Є. А., Бородін В. А. Операційні системи та системне програмування. Київ, 2012. 360 с.

Додаткова:

1. Carswell R., Shen Jiang, Hardee M. E., Mahajan A., Touchette T. Guide to Parallel Operating Systems with Windows 10 and Linux. Cengage Learning, 2016. 640 p.
2. Irvine K. R. Assembly Language for x86 Processors / K. R. Irvine. Pearson, 2014. 720 p.
3. McFedries P. Windows 10 Simplified. Visua, 2015. 288 p.
4. Nemeth E., Snyder G., Hein T., Whaley B., Mackin's D. UNIX and Linux System Administration Handbook. Addison-Wesley Professional, 2017. 1232 p.
5. Tanenbaum A., Bos H. Modern Operating Systems, 4 th ed., Bos H. Pearson, 2014. 1136 p
6. Wright B., Plesniarski L. Microsoft Specialist Guide to Microsoft Windows 10. Cengage Learning, 2016. 756 p.

Інтернет- ресурси:

1. Saunders M. How to write a simple operating system. URL: http://mikeos.sourceforge.net/write-your-own-os.html
2. Wienand I. Computer Science from the Bottom Up. URL: http://www.bottomupcs.com/
3. Офіційна сторінка Codecademy. URL: https://www.codecademy.com
4. Офіційна сторінка Microsoft Windows 10. URL: https://www.microsoft.com/uk-ua/software-download/windows10
5. Офіційна сторінка Linux URL: https://www.linux.org/
6. Офіційна сторінка Canonical UBUNTU URL: https://ubuntu.com/
7. Офіційна сторінка FreeBSD URL: https://www.freebsd.org/
8. Офіційна сторінка Fedora URL: https://getfedora.org/uk/
9. Офіційна сторінка OpenSUSE URL: https://www.opensuse.org/
10. Офіційна сторінка Debian URL https://www.debian.org/index.uk.html

Додаток 2

Найпопулярніші дистрибутиви Linux

Ubuntu

Дистрибутив Ubuntu, який розроблений Canonical, базується на основі Debian та існує з 2004 року. Він має версію для настільних комп'ютерів, а також для серверів. В мережі Internet існує безліч навчальних матеріалів, інструкцій та рішень. Ubuntu вважається одним із найкращих дистрибутивів для домашнього використання та робочих станцій.

https://ubuntu.com/

Debian

Дистрибутив розвивається лише спільнотою користувачів та існує, починаючи з 1995 року. Debian вважається ще більш стабільним ніж Ubuntu, через що нові версії виходять приблизно раз на два роки, а підтримка старих версій триватє приблизно п'ять років. Він знаходить застосування там, де потрібна стабільність, а не новизна програмного забезпечення, але на домашніх комп'ютерах його також можна використовувати.

https://www.debian.org/index.uk.html

CentOS

Дистрибутив CentOS все ще залишається на третьому місці за популярністю. На відміну від Debian, який ще досить часто використовується як операційна система для персональних комп’ютерів, то CentOS призначений для серверів.

https://www.centos.org/

Linux Mint

Існує з 2006 року та базується на LTS версіях Ubuntu. Дистрибутив має фіксований графік релізу і кожен наступний реліз виходить майже згодом після випуску LTS версії Ubuntu. Загалом Linux Mint більше орієнтований на початківців у світі Linux і має можливості виконувати налаштування системи максимально просто.

https://linuxmint.com/

Fedora

Цей дистрибутив розробляється компанією Red Hat насамперед як система для робочих станцій та персональних комп’ютерів. Всі нові можливості, які у майбутньому будуть розміщені у Red Hat Enterprice Linux, спочатку з'являються в Fedora і проходять тетування.

https://getfedora.org/uk/

Manjaro

Це один із найпопулярніших дистрибутивів, заснованих на основі Arch Linux. Manjaro використовує стабільні версії програмного забезпечення, надаючи окремі репозиторії пакетів, в яких воно вже протестоване, стабільне і, швидше за все, не викличе нестандартних ситуацій при оновленні системи. У дистрибутиві, як і у Arch Linux використовується система Rolling релізів. Це означає, що користувач завжди матимете найсвіжішу версію операційної системи на своєму комп'ютері.

https://manjaro.org/

OpenSUSE

Достатньо популярна операційна система від копанії SUSE. Вона створена на основі напрацювань SUSE Linux Enterprice та використовує систему керування пакетами RPM. Дистрибутив має дві редакції. Редакція із звичайними релізами під назвою Leap та редакція з Rolling релізами під назвою Tumbleweed. Такий підхід надаї змоги користувачу вибрати, що йому більше до вподоби – стабільна система або ж найновіше програмне забезпечення. Нові версії редакції Leap виходять приблизно раз на рік.

https://www.opensuse.org/

Arch Linux

Arch Linux це дистрибутив можна налаштувати всі необхідні компоненти. Він використовує систему rolling релізів для того, щоб у користувачів завжди було найсвіжіше програмне забезпечення, а також власний формат пакетів і пакетний менеджер pacman. Для цієї операційної системи користувач можете вибрати драйвери, графічне оточення, набір програм і повністю налаштувати усі необхідін параметри.

https://archlinux.org/

Додаток 3

Найпопулярніші графічні оточення Linux

KDE Plasma

https://kde.org/uk/plasma-desktop/

KDE – це одне з найбільш популярних графічних середовищ організації робочого столу для операційної системи Linux. У порівнянні з іншими графічними оточеннями вона має найсучасніший зовнішній вигляд. Вона також включає набір інструментів та утиліт, які забезпечують гнучкість та адаптивність KDE до кожного користувача.

Серед:

- сучасний і якісно організований інтерфейс користувача;
- повністю настроюється;
- якісний набір базових програм.

GNOME

https://www.gnome.org/

GNOME скорочення від виразу GNU Network Object Model Environment. Фактично друге за популярністю графічне середовище робочого столу і складається тільки з безкоштовного програмного забезпечення з відкритим вихідним кодом (FOSS). Воно розроблене для забезпечення простоти, легкості доступу та надійності роботи користувачів.

Це середовище робочого столу використовує сервер відображення графіки X Window System і підтримує сучасний протокол відображення Wayland.

Особливості:

- наявність сенсорногой графічного інтерфейсу для мобільних пристроїв або планшетів;
- підтримка розширень оболонки GNOME;
- вбудована підтримка програм на основі GTK.

Cinnamon

Графічне оточення Cinnamon беззастережно вважається одним з найкращих середовищ робочого столу Linux на рівні з GNOME та KDE. Це відбулося завдяки спільноті Linux Mint, яка адаптувала GNOME 3 і налаштувала її для потреб операційної системи Mint. З часом проект переріс у окрему версію графічного оточення Cinnamon, яке перетворилося на повноцінне середовище робочого столу.

Серед особливостей, які відрізняють Cinnamon від інших робочих столів Linux, варто назвати:

- швидки, чистий та схожий на Windows інтерфейс;
- легка тематика та дизайн.

MATE

Середовища робочого столу MATE і Cinnamon мають багато спільного з погляду історії та походження. Як і Cinnamon, MATE також є відгалуженням від проекту GNOME, а саме версії GNOME 2. MATE – це дуже легке середовище робочого столу з інтуїтивно зрозумілим та привабливим інтерфейсом. Його важливою особливістю є те, що користувач може запустити ОС із цим графічним середовищенм на ПК із низькими системними вимогами, наприклад Raspberry Pi.

Особливості:

- простий, легкий та зручний у використанні;
- звичайні програми.

XFCE

https://www.xfce.org/

Xfce – це ще одне середовище робочого столу, яке розраховане на машини з обмеженими апаратними ресурсами. Воно створене відповідно до стандартів, запропонованих freedesktop.org та не містить анімації та спецефектів. У той же час містить всі необхідні компоненти та функції, які можна очікувати від популярного полегшеного середовища робочого столу. Ще однією важливою особливістю Xfce є підтримка кількох платформ UNIX (NetBSD, FreeBSD, OpenBSD, Solaris, Cygwin та macOS X).

Особливості:

- простий та зручний у використанні;
- найкраще підходить для ПК із низькими системними вимогами;
- підтримка кількох UNIX-подібних платформ.

Навчальне видання

Мосіюк Олександр Олександрович
Федорчук Анна Леонідівна

Операційні системи та системне програмування

Навчально-методичний посібник

Дизайн обкладинки О. О. Мосіюк
Редактор: А. Л. Федорчук
Комп'ютерне верстання А. Л. Федорчук

Видавництво Житомирського державного університету імені Івана Франка
10008, м. Житомир, вул. Велика Бердичівська, 40
Свідоцтво суб'єкта видавничої справи:
ЖТ № 10 від 07.12.2004 р.
електронна пошта (E-mail): zu@zu.edu.ua