ОСНОВИ РОБОТИ ТА ПРОГРАМУВАННЯ В ОПЕРАЦІЙНІЙ СИСТЕМІ LINUX

Book · August 2018

CITATIONS
0

READS
23

2 authors, including:

Nataliya Olex Матвєєва
Dnepropetrovsk National University
3 PUBLICATIONS **0** CITATIONS

SEE PROFILE

Н.О. Матвєєва, В.С. Хандецький, О.В. Спірінцева

ОСНОВИ РОБОТИ ТА ПРОГРАМУВАННЯ В ОПЕРАЦІЙНІЙ СИСТЕМІ LINUX

Міністерство освіти і науки України
Дніпровський національний університет
імені Олеся Гончара

Н.О. Матвєєва, В.С. Хандецький, О.В. Спірінцева

ОСНОВИ РОБОТИ ТА ПРОГРАМУВАННЯ В ОПЕРАЦІЙНІЙ СИСТЕМІ LINUX

Навчальний посібник

Дніпро
ЛІРА
2018

УДК 004.451.9(0.75.8)
М 33

Друкується за рішення вченої ради
Дніпровського національного університету імені Олеся Гончара
(протокол №13 від 31.05.2018)

Рецензенти:
д-р тех. наук, проф., зав. каф. комп'ютерних наук та інформаційних технологій Дніпровського національного університету ім. О. Гончара
В.В. Гнатушенко
д-р тех. наук, проф., зав. каф. інформаційних систем та технологій університету митної справи та фінансів
А.О. Якунін

Матвєєва Н.О.
М 33 Основи роботи та програмування в операційній системі Linux: навчальний посібник /Н.О. Матвєєва, В.С. Хандецький, О.В. Спірінцева – Д.: ЛІРА, 2018, −156 с.

ISBN

У навчальному посібнику розглядаються основні поняття операційної системи Linux та практичні засоби роботи в ній. Посібник складається з теоретичної та практичної частин.Основна увага приділяється основам роботи у командному рядку, вивчається текстовий інтерфейс користувача (команди операційної системи, мова програмування оболонки bash), розглянуті питання системного адміністрування. Посібник містить значну кількість прикладів та практичних завдань.

Навчальний посібник складено на основі узагальнення досвіду викладання дисципліни «Системне програмне забезпечення».

Основною метою посібника є отримання систематизованих знань щодо операційної системи Linux та навчання вирішувати практичні завдання студентами спеціальності "Комп'ютерна інженерія". Може бути рекомендований для студентів інших спеціальностей, пов'язаних з вивченням сучасного програмного забезпечення.

ISBN

ЗМІСТ

ВСТУП 5
РОЗДІЛ 1. Основні поняття операційної системи Linux 10
1.1. Особливості операційної системи Linux. 10
1.2. Файлова система ОС Linux. 14
1.3. Права доступу і атрибути файлу 24
1.4. Облікові записи користувачів 28
1.5. Процеси 35
РОЗДІЛ 2. Програмування засобами командної оболонки 41
2.1. Визначення командної оболонки 41
2.2. Створення простих сценаріїв для командного процесора 43
2.3. Синтаксис командної оболонки 45
2.3.1. Створення умовних переходів 50
2.3.2. Керуючі структури. Оператори розгалуження 54
2.3.3. Оператори циклів 56
2.4. Списки. Складні команди 63
2.5. Функції 66
2.6. Команди 74
2.7. Налагодження сценаріїв 96
РОЗДІЛ 3. Лабораторний практикум 104
Лабораторна робота 1. Основи роботи з ОС LINUX. Створення файлів 104
Лабораторна робота 2. Створення посилань 114
Лабораторна робота 3. Робота з процесами 120
Лабораторна робота 4. Встановлення прав доступу 126
Лабораторна робота 5. Утиліти архівування і стиснення 133
Лабораторна робота 6. Програмування на мові BASH 145
СПИСОК РЕКОМЕНДОВАНОЇ ЛІТЕРАТУРИ 154

Присвячується 100–річчю
Дніпровського національного університету
імені Олеся Гончара

ВСТУП

Останнім часом операційна система (ОС) Linux стає все більш популярною, працюючи на серверах, десктопних комп'ютерах, ноутбуках, різноманітних мобільних пристроях, забезпечуючи системну основу для ОС Android, Tizen, Sailfish. Основні постачальники комп'ютерного обладнання, такі як, наприклад, корпорації IBM і Dell, підтримують ОС Linux, а найбільші розробники програмного забезпечення, наприклад компанія Oracle, забезпечують виконання своїх програм в ОС Linux. Linux стала по-справжньому конкурентоспроможною операційною системою.

ОС Linux зобов'язана своїм існуванням спільним зусиллям багатьох розробників. Linux-співтовариство підтримує ідею вільного програмного забезпечення, тобто вільного від обмежень, і підтримує *відкриту ліцензійну угоду GNU* (GNU General Public License, GPL). GNU означає GNU's Not UNIX. Відповідне програмне забезпечення зазвичай поширюється у вигляді вихідного програмного коду. Це робить систему зручною для навчання, адже кожен може проаналізувати деталі реалізації системних механізмів і інтерфейсів та підвищити свій рівень, досліджуючи конкретні технічні рішення професійних розробників. Важливою є POSIX-сумісність Linux, що дозволяє автоматично поширювати вміння програмувати на інші ОС, наприклад Mac OS X.

Щоб краще зрозуміти сутність сімейства ОС Linux, необхідно повернутися до історії її створення. На початку 80-х років минулого століття талановитий вчений Річард Столлман (Richard Matthew Stallman) вирішив

створити клон популярної у той час в промислових і академічних мережах ОС Unix. На думку Столлмана Unix стала дуже комерціалізованою системою, а її вихідний код став закритим. Він розробив концепцію *Вільного програмного забезпечення* (Free Software), яке свідчило, що користувачі повинні завжди мати можливість створювати, модифікувати і обмінюватися програмами без будь-яких обмежень. Цей принцип лежить в основі так званої *Відкритої ліцензійної угоди GNU*. Свою версію операційної системи Столлман назвав GNU.

На початку дев'яностих років молодий фінський програміст Лінус Торвальдс (Linus Torvalds) зацікавився однією з версій Unix, яка називалася ОС Minix. Встановивши Minix на своєму комп'ютері, Лінус виявив помилки і став їх виправляти, написавши власний емулятор терміналу, котрий виконував перемикання завдань. Додаючи все нові і нові функції, він, фактично, створив нову операційну систему. Після опублікування її вихідного коду в 1991 система відразу ж викликала великий інтерес. Поєднання імені Лінус (Linus) і назви ОС Unix дало назву новому ядру, яке і відомо зараз як *Linux*. Однак найбільш загальним найменуванням є GNU / Linux в силу їх сумісності і відкритості.

В даний час існує багато різних версій Linux, побудованих на основі єдиного ядра. Ці версії прийнято називати *дистрибутивами* (distributions, distros). В дистрибутив входить стандартний, попередньо відкомпільований набір пакетів, котрий включає базову систему Linux, утиліти для інсталяції системи і управління системою, а також готові до інсталяції пакети інструментів.

Одними з найвідоміших дистрибутивів є Ubuntu, Mandriva, Xandros, Red Hat, SUSE. Але ще сотні версій вже існують і з'являються щодня. Така різноманітність можлива тільки завдяки ідеї вільного програмного забезпечення. Створенням своїх версій займаються як окремі спеціалісти, так і великі компанії, які спонсують написання і підтримку нових дистрибутивів.

Дистрибутив Ubuntu поєднує в собі обидва підходи. Його розробка підтримується компанією Canonical Ltd., яка заснована в 2004 році Марком Шаттлвортом (Mark Shuttleworth), і великою спільнотою розробників і користувачів. Сама Ubuntu базується на дистрибутиві Debian (www.debian.org), створеному колективними зусиллями. Кожні шість місяців всі поправки, які були внесені в Debian за останні півроку, вносяться і в Ubuntu.

Операційна система Ubuntu має ряд незаперечних переваг перед аналогічними системами, заснованими на ядрі Linux. Основними з них є:

1. Орієнтованість на кінцевого користувача.

Ubuntu поставляється в декількох версіях, які розраховані на різне застосування: це і серверна, і десктопна версії, версії з різними робочими середовищами і призначеннями. Вона орієнтована на кінцевого користувача з початковим і середнім рівнем технічної підготовленості.

2. Простота установки і використання.

Існує дві редакції установчого диску Ubuntu. Перша являє собою LiveCD з графічним інсталятором на багатьох мовах. Установка за допомогою цього інсталятору є дуже простою, завдяки рідній мові і зрозумілим діалоговим вікнам. Друга редакція диска являє собою Alternate-установник. На даний момент ця редакція диска використовується в тих випадках, коли неможлива установка з LiveCD. У Alternate-версії існує простий і швидкий інсталятор.

Для установки Ubuntu достатнім є наявність лише одного CD, всі інші є в мережі в разі потреби. При виході нової версії Ubuntu не обов'язково встановлювати заново всю систему, а досить оновити її з поточної версії на наступну за допомогою Інтернет. Коли установка завершена, система відразу ж готова до роботи: маємо повний комплект бізнес-додатків, інтернет-додатків, додатків для роботи з графікою та ігор.

3. Філософія і підтримка спільноти Ubuntu.

ОС Linux настільки ж є філософією, наскільки і операційною системою. Можна сказати, що в основі цієї філософії лежить принцип свободи і рівних можливостей. Засновуючи проект Ubuntu, Марк Шаттлворт сформулював декілька основних принципів, які базуються на ідеї вільного програмного забезпечення:

- Кожен користувач повинен мати свободу отримати, використовувати, копіювати, поширювати, вивчати, змінювати і модифікувати програмне забезпечення для будь-яких цілей без оплати ліцензій.
- Кожен користувач повинен мати можливість використовувати програмне забезпечення на тій мові, яка йому зручна.
- Кожен користувач повинен мати необмежену можливість використання програмного забезпечення, навіть якщо це людина з обмеженими можливостями.

Поява і зростання спільноти Ubuntu було повністю обумовлено сформульованими принципами. Однією з найбільших переваг для користувачів є технічна допомога з боку спільноти, яка може бути отримана на форумах Ubuntu (www.ubuntuforums.org).

Встановлення дистрибутива Ubuntu на комп'ютер.

Установка Ubuntu може бути здійснена чотирма різними методами:

1. Спільне завантаження з ОС Windows.
2. Установка Ubuntu з-під Windows (Within Windows - Wubi).
3. Установлення на віртуальний комп'ютер.
4. Альтернативне установлення (Alternate Install).

Кожен з цих способів застосовується для досягнення різних результатів і має свої нюанси.

Якщо бажано повністю перейти на роботу з Ubuntu або використовувати її частіше ніж Windows, то більше підійде перший варіант. Однак при цьому необхідно виконати перерозподіл файлової системи, перед

чим треба виконати деякі підготовчі операції, щоб не втратити дані на комп'ютері.

Другий варіант установки менш ризикований і не вимагає внесення будь-яких змін в файлову систему. При цьому Ubuntu може дещо повільніше працювати, а режим гібернації буде недоступний.

Установка на віртуальний комп'ютер підійде тим користувачам, які бажають працювати в Ubuntu без внесення особливих змін в існуюче робоче середовище. Швидкість роботи операційної системи, встановленої на віртуальному комп'ютері, природно, буде набагато нижче. Візуалізація також буде гірше.

Альтернативна установка проводиться в тих випадках, коли стандартна установка не вдалася в силу будь-яких причин, або комп'ютер є досить застарілим для підтримки візуальних ефектів чи забезпечення необхідної обчислювальної потужності.

Отримання дистрибутиву Ubuntu

Дистрибутив представлений у формі диска LiveCD включає в себе все необхідне для установки. Образ диска у форматі ISO може бути завантажений з сайту Ubuntu за адресою:

http://www.ubuntu.com/getubuntu/download

Використовуючи цей файл образу, необхідно створити інсталяційний диск. Якщо вибрано варіант установки з під Windows, то скачування всього диска є необов'язковим. Достатньо лише завантажити невеликий додаток інсталятора, який потім в процесі установки автоматично отримає усі необхідні компоненти з сайтів Ubuntu.

РОЗДІЛ 1.

Основні поняття операційної системи Linux

1.1. Особливості операційної системи Linux.

ОС Linux базується на таких основних правилах:

 одна задача - одна програма. В Linux не прийнято робити комбайни для виконання «відразу всього». Програма виконує одну просту дію, але виконує її добре.

 є безліч шляхів вирішення задачі. Для розв’язання того чи іншого комплексного завдання кожен має право вибирати свій набір простих компонентів для її розвязання.

 все є файлом. Дійсно, в Linux все представлено у вигляді файлів − програми, настройки, системні дані і навіть пристрої. І з пристроями можна працювати як з простими файлами.

Необхідно бути готовим до вивчення не просто нових програм, а нових методів роботи на комп'ютері.

З технічної точки зору можна виділити наступні особливості Linux:

Реальна багатозадачність − Linux використовує режим розподілу часу центрального процесора (time-sharing system). Працює це так: планувальник виділяє кожному процесу фіксований інтервал для виконання. Після закінчення виділеного часу планувальник призупиняє виконання процесу і передає управління іншому процесу. В інших ОС використовують режим «багатозадачності, яка витісняється», коли процес сам повинен призупинити своє виконання і передати право не виконання іншому процесу.

Багатокористувальницький доступ − Linux не тільки багатозадачна операційна система, але й розрахована на багато користувачів.

Сторінкова організація пам'яті − пам'ять в Linux організована у вигляді сторінок по 4 КБ кожна. Коли фізична оперативна пам'ять

закінчується, включається механізм підкачки і невикористовувані дані скидаються в область підкачки на жорсткий диск.

Завантаження виконуваних модулів "на вимогу" − щоб ядро системи підтримувало певний пристрій або функцію (наприклад, протокол), потрібно додати програмний код до складу ядра. У Linux ця проблема вирішується завантаженням модулів, які додають підтримку певних пристроїв. При цьому в пам'ять завантажуються тільки ті модулі, які необхідні для повноцінної роботи конкретної системи, що дозволяє оптимізувати використання ресурсів комп'ютера.

Спільне використання виконуваних програм − якщо декілька користувачів запустили одну і ту ж програму, то в пам'ять завантажується всього одна копія цієї програми, що дозволяє економити оперативну пам'ять.

Загальні бібліотеки − бібліотеки містять набори процедур, які використовуються різними додатками. Замість того, щоб скомпілювати в один виконуваний файл всі процедури, додаток використовує бібліотеку (яка також може використовуватися іншими додатками), що дозволяє істотно економити місце на диску.

Підтримка різних файлових систем − Linux підтримує багато різноманітних файлових систем, також і файлові системи Windows.

Підтримка різних апаратних платформ − Linux може працювати не тільки на платформах x86 / x64. Підтримуються апаратні платформи ARM, DEC Alpha, SUN Sparc, M68000 (Atari і Amiga), MIPS, PowerPC та інші.

Ядро Linux поширюється на умовах GNU General Public License (GPL), які встановлені організацією Free Software Foundation.

Програміст, який використовує Linux, або створює свої власні системи на базі Linux, не має права перетворювати свій продукт в комерційний (відомчий). Програмне забезпечення, котре розповсюджується на основі GPL, не може **розповсюджуватись тільки у вигляді двійкового коду** (тобто в поставку Linux потрібно включити вихідний код). Це одна з принципових цілей проекту.

Розглянемо операційну систему Linux як єдиний комплекс.

Ядро Linux - це основна частина операційної системи. Воно відповідає за розподіл пам'яті, управління процесами і периферійними пристроями. Для підтримки більшого обсягу оперативної пам'яті у порівнянні з фізично встановленою на комп'ютері, ядро дозволяє використовувати область підкачки, розміщуючи сторінки оперативної пам'яті на жорсткому диску.

Ядро Linux підтримує безліч файлових систем, включаючи FAT, FAT32. Власні файлові системи Linux (ext3fs і ext4fs) розроблені для оптимального використання дискового простору.

Компоненти системи Linux зображені на рис.1.1.

Рисунок 1.1. Компоненти ОС Linux

ОС Linux складається з трьох основних груп коду - **ядро, системні бібліотеки та системні утиліти**; найважливішим є відмінність між ядром і всіма іншими компонентами.

Ядро відповідає за підтримку основних концепцій (абстракцій) ОС. Код ядра виконується в привілейованому режимі, і йому повністю доступне все комп'ютерне обладнання. Весь код і структури даних ядра зберігаються і виконуються в єдиному адресному просторі.

Системні бібліотеки визначають стандартний набір функцій, за допомогою яких додатки взаємодіють з ядром, і які реалізують основну

частину функціональності ОС, яка потребує виконання в привілейованому режимі.

Системні утиліти виконують індивідуальні специфічні завдання.

Завантажувані модулі ядра Linux

Одним з найважливіших нововведень в ядрі Linux є завантажувані модулі ядра (loadable kernel modules, LKM). Вони забезпечують ядру гнучкість і функціональність.

Частини (секції) коду ядра можуть компілюватися, завантажуватися і розвантажуватися, незалежно від іншої частини ядра.

Модуль ядра може реалізовувати драйвер пристрою, файлову систему або мережевий протокол.

Модульний інтерфейс дозволяє стороннім розробникам реалізовувати і поширювати на своїх власних умовах драйвери або файлові системи, які не можуть поширюватися на основі GPL.

Модулі ядра дозволяють інсталювати Linux у вигляді стандартного, мінімального ядра, без використання будь-яких вбудованих пристроїв.

Три компоненти модуля Linux підтримують:

управління модулем; реєстрацію драйвера; вирішення конфліктів.

Компонент управління модулем керує завантаженням модуля в пам'ять і його взаємодією з іншою частиною ядра.

Управління модулем розбите на дві частини:

• управління частинами коду модуля в пам'яті ядра;

• управління символами, на які модуль дозволяє посилатися.

Компонент **module requestor** управляє завантаженням запитаних, але ще не завантажених модулів. Він також регулярно опитує ядро, щоб переконатися, що модуль досі використовується, і вивантажує модуль, якщо він довгий час активно не використовувався.

Компонентна реєстрація драйверів надає модулю можливість повідомити ядру, що новий драйвер доступний.

Ядро підтримує динамічну таблицю всіх відомих драйверів і забезпечує набір підпрограм для додавання драйверів в ці таблиці або видалення з них в будь-який час.

Таблиці реєстрації включають такі елементи:

• драйвери пристроїв;

• файлові системи;

• мережеві протоколи;

• двійкові формати.

Модуль вирішення конфліктів надає механізм, який дозволяє різним драйверам пристроїв резервувати апаратні ресурси і захищати ці ресурси від випадкового використання іншими драйверами.

Метою модуля вирішення конфліктів є:

• запобігання конфліктів, які пов'язані з використанням апаратури;

• запобігання **автоперевірки (autoprobes)** від перехрещення з вже існуючими драйверами пристроїв;

• вирішення конфліктів різних драйверів, які намагаються отримати доступ до однієї і тієї ж апаратури.

1.2. Файлова система ОС Linux.

Файлова система ОС Linux, на відміну від операційних систем сімейства Windows, не розділена за томами (дисками, пристроями), а має єдину деревоподібну структуру, в основі якої лежить кореневий каталог. *Кореневий каталог* − це рівень файлової системи, вище якого по дереву каталогів піднятися неможливо. В ОС Linux кореневий каталог позначається, як / (слеш). Система дозволяє встановлювати багато кореневих каталогів. Так, наприклад, для деякого користувача **max/home** буде кореневим каталогом і при зверненні до клієнта **max** на зміну каталогу на кореневій, користувач буде потрапляти в **/home.**

Кожному користувачеві в файловій системі виділяється *домашній каталог* – спеціальний каталог, необхідний для зберігання своїх особистих даних. При вході користувача в систему, він одразу опиняється у своєму домашньому каталозі. Зазвичай права доступу до домашнього каталогу користувача виставлені таким чином, що доступ до каталогу заборонено всім, крім власника та адміністратора *root*.

З точки зору користувача файлова система − це логічна структура каталогів та файлів. Для всіх Linux-подібних систем ця деревоподібна структура росте з одного кореня: вона починається з кореневого каталогу, батьківського по відношенню до всіх інших, а фізичні файлові системи різного типу, що знаходяться на різних розділах і навіть на віддалених машинах, представляються, як гілки цього дерева.

Linux підтримує багато різних файлових систем. У якості кореневої файлової системи доступні: ext2, ext3, ext4, XFS, ReiserFS, JFS.

Всі перераховані файлові системи (крім ext2) ведуть журнали своєї роботи, що дозволяє відновити дані у випадку збою. Перед тим як виконати операцію, журнальована файлова система записує цю операцію в журнал, а після виконання операції видаляє запис з журналу.

У сучасних дистрибутивах за замовчанням установлена файлова система ext4 (Ubuntu - файлова система ext4). При необхідності можна вибрати інші файлові системи. Розглянемо деякі з існуючих систем.

Файлова система XFS була розроблена компанією Silicon Graphics у 2001 році. Її особливістю є висока продуктивність (до 7 Гбайт/с). XFS може працювати з блоками розміром від 512 байт до 64 Кбайт.

Файлова система ReiserFS є найекономнішою, оскільки дозволяє зберігати декілька файлів в одному блоці (інші файлові системи можуть зберігати в одному блоці тільки один файл або одну його частину). Але у цієї файлової системи є два недоліки: вона нестійка до збоїв і її продуктивність сильно знижується при фрагментації.

Файлова система JFS (розробка IBM) спочатку з'явилася в операційній системі AIX, а потім була модифікована під Linux. Основні переваги – надійність і висока продуктивність (вища, ніж у XFS). Але у неї маленький розмір блоку (від 512 байт до 4 Кбайт). Отже, вона припустима на сервері баз даних, але не для роботи з даними мультимедіа, оскільки блоку розміром 4 Кбайт для роботи, наприклад, з відео в реальному часі, буде замало.

Імена файлів в операційній системі Linux

У порівнянні з ОС Windows у Linux інші правила побудови імен файлів. В Linux немає розширення імені файлу. В Windows, наприклад, для файлу Document1.doc іменем файлу є фрагмент Document1, а doc – це расширення. У Linux Document1.doc – це ім'я файлу.

Максимальна довжина імені файлу – 254 символи. Ім'я може містити будь-які символи (в тому числі і кирилицю), окрім, **/ \ ? < > * " |** . Але кирилицю в іменах файлів взагалі не рекомендовано. При обміні файлами по електронній пошті, замість імені можуть бути незрозумілі символи, ім'я файлу краще писати латиницею.

Якщо ім'я файлу починається з крапки, то файл вважається «прихованим»: деякі команди його «не бачать».

Операційна система Linux чутлива до регістру: FILE.txt і File.Txt – це два різних файла.

Імена каталогів будуються за тими ж правилами, що й імена файлів. Повним ім'ям файлу (або шляхом до файлу) називається список вкладених один у одний каталоги, котрі закінчуються власним ім'ям файлу. Список може починатися з будь-якого каталогу, тому що в деревоподібній структурі між будь-якими двома вузлами існує шлях. Якщо цей список починається з кореневого каталогу, то шлях називається *абсолютним*. Якщо з будь-якого іншого — *відносним*.

Кореневий каталог позначається символом «/» (слеш), і цим же символом поділяються імена каталогів в списку. Таким чином, абсолютним

ім'ям файлу *first* в домашньому каталозі користувача **max** буде */home/max/first*.

У кожного каталогу існують два особливих «підкаталога» з іменами «дві крапки» (..) і «крапка» (.). Перший з них служить вказівкою на батьківський каталог, а другий — сам на себе.

Для кореневого каталогу, у якого немає батька, обидва ці «підкаталоги» вказують на кореневий каталог. За допомогою цих імен утворюються відносні імена файлів.

Файли і пристрої

В операційній системі Linux є файли пристроїв, котрі дозволяють працювати з пристроєм, як із звичайним файлом. Файли пристроїв знаходяться в каталозі /dev (скорочення від *devices*).

Найпоширеніші приклади файлів пристроїв:

- **/dev/sdx –** файл пристрою жорсткого диска (SATA/SCSI/ATA), x – це порядок підключення диска до шини;
- **/dev/sdxN –** файл пристрою розділу жорсткого диска, N – номер розділу;
- **/dev/scdN** або **/dev/srN –** привід CD/DVD;
- **/dev/mouse –** файл пристрою миші;
- **/dev/modem –** файл пристрою модема;
- **/dev/ttySn –** файл послідовного порту, **n** – номер порту (ttyS0 відповідає COM1, ttyS1 – COM2 тощо).

Файли пристроїв бувають двох типів: блокові і символьні. Обмін інформацією з блоковими пристроями, наприклад з жорстким диском, здійснюється блоками інформації, а з символьними – окремими символами. Приклад символьного пристрою – послідовний порт. З жорсткими дисками найскладніше, оскільки один і той же пристрій може в різних версіях одного і того ж дистрибутива називатися по різному.

Стандартні каталоги ОС Linux

Файлова система Linux містить наступні каталоги:

- **/ –** кореневий каталог;
- **/bin –** містить стандартні програми Linux, необхідні для роботи в системі: командні оболонки, файлові утиліти тощо;
- **/boot –** каталог завантажувача, містить образи ядра, може містити конфігураційні та допоміжні файли завантажувача;
- **/dev –** містить файли пристроїв;
- **/etc –** містить конфігураційні файли системи;
- **/home –** домашні каталоги користувачів;
- **/lib –** бібліотеки і модулі;
- **/lost+found –** відновлені після некоректного размонтування файлової системи файли та каталоги;
- **/media –** точки монтування автоматично змонтованих змінних носіїв даних;
- **/mnt –** містить точки монтування;
- **/proc –** каталог псевдофайлової системи *proofs*, надає інформацію щодо процесів;
- **/root –** каталог суперкористувача *root*;
- **/sbin –** каталог системних утиліт, які має право виконувати користувач *root*;
- **/srv –** службові каталоги різних мережевих служб, наприклад, FTP і www – серверів;
- **/sys –** каталог псевдофайлової системи *sysfs*, надає інформацію про систему;
- **/tmp –** каталог для тимчасових файлів;
- **/usr –** користувальницькі програми, документацію, вихідні коди програм і ядра;

- **/var** – дані системи, які постійно змінюються, наприклад, спулы системи друку, поштові скриньки, протоколи, замки тощо.

Уявлення щодо файлів з точки зору ОС Linux

Файл − це іменований блок інформації, який зберігається на диску і має такі ознаки: фіксоване імя (назва файлу) та певне логічне уявлення. В операційних системах на базі ядра Linux вся інформація щодо файлу прив'язана не до імені, а до *індексного дескриптора*. У кожного файлу є унікальний індексний дескриптор, який містить відомості про файл: в яких блоках диску зберігається вміст файлу, розмір файлу, час його створення та інше. Пронумеровані індексні дескриптори файлів містяться в спеціальній таблиці. Кожен логічний і фізичний диск має власну таблицю індексних дескрипторів. Саме номер індексного дескриптора є справжним ім'ям файлу в системі.

Оскільки індексні дескриптори є номерами, а файлів у операційній системі зазвичай дуже багато, то шукати файл за номером його дескриптора дуже незручно. Тому будь-якому файлу в системі присвоюється осмислене ім'я (словесне), яке не містить інформації про файл, а лише вказує (посилається) на його дескриптор.

Ім'я файлу, котре посилається на його індексний дескриптор, називається *жорстким посиланням*. Механізм жорстких посилань - це основний спосіб звертатися до файлу за ім'ям в операційних системах, які ґрунтуються на ядрі Linux.

Файл у системі ідентифікує номер індексного дескриптора, а ім'я файлу містить лише вказівник на нього. Таких покажчиків можна створити безліч, всі вони будуть указувати на один об'єкт. Тобто, *файл в операційній системі Linux може мати декілька імен*.

Видалення одного жорсткого посилання на файл не призводить до його видалення із системи, якщо існують інші жорсткі посилання на файл. Усі жорсткі посилання рівноправні між собою, незалежно від часу створення,

місцезнаходження в структурі каталогів та інше. Файл буде доступний системі, поки існує хоч одне жорстке посилання на нього. У разі видалення всіх посилань на файл, він видаляється з системи, оскільки стає недоступним їй.

Жорсткі посилання можна створювати тільки на файли, але не на каталоги. Не можна створити жорстке посилання з одного диска на інший. Останнє означає, що не можна створити жорстке посилання на файл, котрий знаходиться, наприклад, на зміному носії (флеш-пам'ять, CD-RW) або іншому розділі жорсткого диска.

У такій ситуації на допомогу приходять *м'які посилання*. Часто їх також називають *символьними посиланнями*. Вони являють собою файли, які вказують не на індексні дескриптори, а на імена файлів (рис. 1.2).

Жорстке посилання вказує безпосередньо на індексний дескриптор, а м'яке − на жорстке посилання. Якщо видалити усі жорсткі посилання на файл, то ніяке м'яке посилання "не спрацює" .

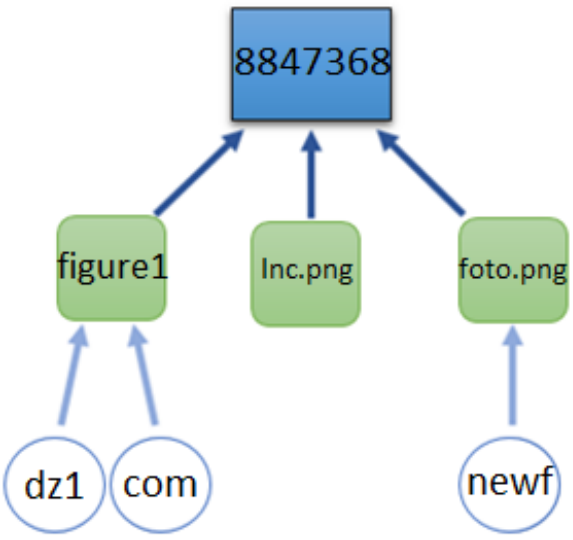


Рисунок 1.2. Зв'язок між посиланнями

На рис. 1.2 зображений односторонній зв'язок між символьними посиланнями та іменами файлів, а також між іменами та індексним дескриптором. Верхній прямокутник відповідає індексному дескриптору

файла, квадрати з округленими кутами – іменам файлів, а кола – символьним посиланням. Індексний дескриптор файлу завжди один, а імен може бути декілька. Також може існувати необмежена кількість символьних посилань на кожне ім'я файлу. При видаленні жорсткого посилання, на яке було м'яке посилання, останне не успадковує зв'язок з дескриптором і втрачає свою "працездатність". Якщо в даному прикладі видалити жорстке посилання з ім'ям *figure1*, то файли *dz1* і *com* стануть непотрібними, відкрити файл *8847368* за їх допомогою вже буде неможливо. Такі посилання часто називають "битими".

Слід розрізняти копіювання файлів та створення посилань. При копіюванні файла створюється новий файл з власним індексним дескриптором, дані якого записуються на вільне місце на диску. У разі ж створення жорсткого посилання, файл залишається в однині з тим же індексним дескриптором, з'являється лише додатковий покажчик на нього.

При внесенні змін у файл, до якого звертались під одним ім’ям, ці зміни виявляться і при зверненні до файлу під іншим ім'ям. При створенні копії файла і подальшій зміні даних у цієї копії, дані первинного файлу не змінюються.

У разі м'яких посилань, хоч і створюється новий файл, але він не містить даних файла-оригіналу, а лише посилається на жорстке посилання.

Типи файлів

З точки зору Linux-подібних ОС, файл являє собою потік або послідовність байтів. Такий підхід дозволяє поширити поняття файлу на безліч ресурсів не тільки локального комп'ютера, але і віддаленого, пов'язаного локальною мережею будь-якого роду. Доступ до такого ресурсу здійснюється через універсальний інтерфейс, завдяки чому запис даних у файл, відправка їх на фізичний пристрій або обмін з іншою працюючою програмою відбувається аналогічно. Це дуже спрощує організацію і обмін даними.

В ОС Linux можна виділити наступні типи файлів:

• ***звичайні файли*** − послідовність байтів (текстові документи, виконуючи програми, бібліотеки тощо);

• ***каталоги*** − іменовані набори посилань на інші файли;

• ***файли фізичних пристроїв***, котрі поділяються на:

– *файли блокових пристроїв*, драйвери яких буферизують введення-виведення за допомогою ядра;

– *файли байт-орієнтованих* або *символьних пристроїв*, які дозволяють пов'язаним з ними драйверам виконувати буферизацію власними засобами;

• ***символьні посилання*** (symlink, symbolic link);

• ***іменовані канали*** (named pipes);

• ***гнізда*** (sockets).

Звичайні файли і каталоги

Властивості (атрибути) файлів і каталогів можна вивести на термінал за допомогою команди **ls** з ключем -1:

```
#ls -l  /home/max/first1
-rwxr-xr-- 1 den users 0 Feb 14 19:08 /home/max/first1
```

Розглянемо ці властивості.

Перший символ у рядку означає тип файлу. В даному випадку дефіс означає простий файл, d - каталог, b - блоковий пристрій, c - символьний пристрій, l – символьне посилання, p - іменований канал та s - гніздо.

Наступні 9 символів означають права доступу до файлу. Вони розподіляються за трьома трійками, які демонструють права: власника, членів його групи і всіх інших. Далі вказані ім'я власника файлу і ім'я його групи; розмір файлу в байтах; дата і час останньої модифікації і ім'я файлу.

Для каталогу виведення команди **ls** виглядає так само, але значення деяких властивостей відрізняються.

```
#ls -l -a /home/max
drwx  ------ 3 max users 4096Feb 14 19:02 .
drwxr-xr-x 4 root root 4096Feb 02 11:32 ..
[. . .]
```

Ключ -a використовують, щоб побачити псевдоподкаталоги «.» і «..» (їх імена починаються з точки, це «приховані» файли). Біт читання в правах доступу означає право переглядати зміст каталогу, запису − право створювати і видаляти файли з каталога, виконання − право переходити в цей каталог (робити його своїм поточним каталогом).

Файли фізичних пристроїв

Усі підключені до комп'ютера пристрої сприймаються операційною системою як файли: виведення інформації на термінал, друк на принтері, відправка пошти − все це, з точки зору ОС, є запис до файлу. Технічно файл пристрою − це комунікаційний інтерфейс драйвера, який видає взаємодію з цим пристроєм. Більшість таких файлів розташовується в каталозі /dev.

Можна розглянути цей каталог, ввівши команду ls -l /dev. Листинг цієї команди займає декілька екранів, це створює привід для знайомства з командою-фільтром, яка виводить отримані дані по одному екрану за раз:

#ls –l /dev | more

Для перегляду наступного екрану натискають пробіл; щоб перервати роботу команди - Ctrl + C.

Жорсткі та символічні посилання

Жорсткі посилання є просто іншим ім'ям для вихідного файлу. Після створення такого посилання його неможливо відрізнити від вихідного імені файлу. “Справжнього” імені у файла немає, точніше, всі такі імена будуть справжніми. Команда **ls** показує кількість саме таких жорстких посилань. Видалення файлу за допомогою одного з його імен зменшує на одиницю кількість посилань, і остаточно файл буде видалений тільки тоді, коли ця кількість дорівнюватиме нулю. Тому зручно використовувати жорсткі посилання для того, щоб запобігти випадковому видаленню важливого файла.

Іменовані канали

Цей тип файлу ще називається буфером FIFO (First In - First Out). Через файли такого типу два незалежних процеси (дві працюючі програми) можуть

обмінюватися даними: все, що записано в файл одним процесом, може бути прочитано звідти іншим. Іменований канал створюється командою **mkfifo.**

Гнізда

Механізм гнізд (сокетів, sockets) вперше з'явився у версії 4.3 BSD UNIX. Пізніше він перетворився в одну з найпопулярніших систем мережевого обміну повідомленнями, реалізовану в багатьох, не тільки Linux-подібних, операційних системах.

Власне гніздо − це абстрактна кінцева точка підключення до мережі. Процес відправляє дані в мережу, записуючи їх в файл гнізда. При цьому процеси, які встановили зв'язок через пару гнізд, можуть бути запущені як на різних комп'ютерах, так і на одному.

Міжпроцесний обмін через гнізда використовується такими стандартними компонентами Linux, як служба обліку *syslog* і віконна система *X Window*.

1.3. Права доступу і атрибути файлу

Для кожного об'єкта файлової системи Linux існує набір прав доступу, який визначає взаємодію користувача з цим об'єктом. Об'єктами можуть бути файли, каталоги, процеси, а також спеціальні файли (наприклад, пристрої). Оскільки, у кожного об'єкта в ОС Linux є власник, то права доступу застосовуються щодо власника файлу. Права складаються з набору трьох груп, по три атрибути:

- читання (r), запис (w), виконання (x) для власника;
- читання, запис, виконання для групи власника;
- читання, запис, виконання для усіх інших.

Такі права можна представити коротким записом:

rwxrwxrwx − дозволено читання, запис і виконання усім;

rwxr-xr-x − запис дозволено тільки для власника файлу, а читання і виконання усім.

rw-rw-r-- − запис дозволено для власника файлу і групи власника файлу, а читання − усім.

Такий розподіл прав дозволяє гнучко управляти ресурсами, доступними користувачам.

Для каталогів використовуються ті ж прапорці повноважень, що й для звичайних файлів, однак інтерпретуються вони інакше. Наявність у користувача повноважень читання каталогу дозволяє йому переглядати його (наприклад, команда ***ls***). Користувач, який має повноваження запису, може створювати і видаляти файли з цього каталогу. Причому видалити файл з каталогу користувач може навіть якщо у нього немає прав на запис до файлу. Повноваження виконання дозволяють користувачу входити до каталогу та переглядати всі підкаталоги (команда ***cd***). Без повноважень виконання об’єкти файлової системи, які знаходяться в цьому каталозі, недоступні. Без повноважень читання об’єкти файлової системи, які знаходяться в каталозі, не можна переглядати, однак доступ до них можна отримати, якщо відомо повний шлях до цього об’єкта на диску.

Таким чином, виникає можливість створення так званих "прихованих" каталогів, коли неможливо отримати список файлів, але користувач точно знає ім'я файлу та може скопіювати його з "прихованого" каталогу.

Для визначення прав доступу в операційній системі Linux існує багато команд. Основні з них − це **chmod, chown** і **chgrp.**

Команда ***chmod*** *(Change MODe - змінити режим)* - змінює права доступу до файлу. Для використання цієї команди необхідно мати права власника файлу або права *root*. Синтаксис команди:

chmod mode filename

де *filename* - ім'я файлу, у якого змінюються права доступу; *mode* - права доступу, котрі встановлюються на файл.

Права доступу можна записати в двох варіантах - символьному і абсолютному.

Для символьного вигляду використання команди chmod буде виглядати наступним чином:

```
                    |r|
          |u|       |w|
          |g|  |+|  |x|
chmod     |o|  |-|  |X|   Filename
          |a|  |=|  |u|
                    |g|
                    |o|
```

де:

u, g − встановлення прав для користувача та групи, відповідно.

o − встановлення прав усім користувачам, котрі не входять до групи, якій належить даний файл;

a − встановлення прав усіх користувачів системи, тобто власник, група та усі інші;

+, -, = − додати, видалити, встановити дозвіл, відповідно.

r, w, x, X, u, g, o − право читання, запису, виконання, виконання якщо є таке право ще у когось з груп доступу, такі ж як у власника, такі ж як у групи, такі ж як у інших користувачів, відповідно.

filename − файл, у якого змінюються права.

Для використання абсолютного режиму необхідно представити права доступу до файлу у вигляді 3-х двійкових груп.Наприклад:

rwx r-x r - - буде виглядати:

111 101 100

Тепер кожну двійкову групу перевести у 8-річне число:

111 – 7, 101 – 5, 100 – 4. Отримаємо 754.

Для завдання таких прав до файлу необхідно виконати команду:

```
#ls –l first.txt
-rw-rw-r--- 1 max max 39 Nov 25 15:20 first.txt
# chmod 754 first.txt
#ls –l first.txt
-rwxr-xr-- 1 max max 39 Nov 25 15:20 first.txt
```

Для переведення з символьного представлення у абсолютне можна скористатися табл. 1.1.

Наибільш популярні права доступу:

□ 644 – власнику можна читати та змінювати файл, іншим користувачам – тільки читати;

□ 666 – читати та змінювати файл можна всім користувачам;

□ 777 – усім можна читати, змінювати та виконувати файл.

Таблиця 1.1

Відповідність чисел двійкової та вісімкової систем числення

Двійкова система	Вісімкова система
000	0
001	1
010	2
011	3
100	4
101	5
110	6
111	7

Нагадаємо, що для каталогу право виконання – це право перегляду його вмісту.

Іноді символьний метод простіше. Наприклад, маємо файл *script*, котрий треба зробити виконуваним, для цього маємо команду:

```
chmod +x script
```

Для заборони виконання файлу вказуємо параметр – x :

```
chmod -x script
```

Команда ***chown*** (*CHange OWNer* – змінити власника) – дозволяє змінити власника файлу, тобто «подарувати» файл. Для використання цієї команди необхідно або мати права власника поточного файлу, або права *root*. Синтаксис команди:

chown username:groupname filename,

де *username* – ім'я користувача – нового власника файлу;

groupname – ім'я групи – нового власника файлу;

filename – ім'я файлу, у котрого змінюється власник.

Ім’я групи в синтаксисі команди можна не вказувати, тоді буде зміна тільки власника файлу.

Команда ***chgrp*** використовується для зміни власника групи файлу. Синтаксис її:

chgrp groupname filename,

де: *groupname* – ім’я групи, якій буде належати файл;

filename – ім’я файлу, який буде змінюватися.

Майте на увазі, що використовувати команди *chown* та *chmod* може тільки користувач-власник файлу й користувач *root*, а команду *chgrp* – дозволяє рядовим користувачам змінювати групи, але тільки ті, членами яких вони є і користувачу *root*.

1.4. Облікові записи користувачів

Операційна система Linux є не тільки багатозадачною, але і багатокористувальницькою, тобто ця операційна система дозволяє одночасно декільком користувачам працювати з нею. Але система повинна дізнаватися, який або які з користувачів працюють в даний час. Саме для цих цілей в Linux існує два поняття - ***облікові записи*** та ***автентифікація***, які є частинами одного механізму.

Обліковий запис користувача − це необхідна для системи інформація щодо користувача, яка зберігається в спеціальних файлах. Інформація використовується Linux для автентифікації користувача і призначення йому прав доступу.

Автентифікація − системна процедура, котра дозволяє Linux визначити, який саме користувач здійснює вхід.

Вся інформація щодо користувачів зазвичай зберігатися в файлах **/etc/passwd** і **/etc/grpoup**.

/etc/passwd − файл містить інформацію про користувачів. Запис для кожного користувача займає один рядок:

ім'я користувача:пароль:UID:GID:повне им'я:домашній каталог:оболонка

Приклади:

root:x:0:0:root:/root:/bin/bash

max:x:500:500:Maxsim:/max:/bin/bash

ім'я користувача − ім'я, яке вводиться користувачем при запрошенні *login* при автентифікації в системі.

пароль - зазвичай хешований пароль користувача, який зашифрований за незворотним алгоритмом MD5.

UID - числовий ідентифікатор користувача. Система використовує його для розподілу прав файлів і процесів.

GID - числовий ідентифікатор групи. Імена груп розташовані в файлі */etc/group*. Система використовує його для розподілу прав файлів і процесів.

Справжнє ім'я користувача − використовується в адміністративних цілях, а також командами типу *finger* (отримання інформації щодо користувача через мережу).

Домашній каталог − повний шлях до домашнього каталогу користувача.

Оболонка − командна оболонка, яку використовує користувач в сеансі. Для нормальної роботи вона повинна бути вказана в файлі реєстрації оболонок */etc/shells.*

Тепер розглянемо файл ***/etc/group***. Він містить інформацію щодо груп, до яких належать користувачі:

ім'я групи:пароль:GID:користувачі, включені в кілька груп

Приклад:

student:x:0:root, bin,daemon

Ім'я групи − ім'я, що використовується для зручності використання таких програм, як *newgrp.*

Шифрований пароль − використовується при зміні групи командою *newgrp*. Пароль для груп може бути відсутнім.

GID − числовий ідентифікатор групи. Система використовує його для розподілу прав файлів і процесів.

Користувачі, включені до декількох груп. У цьому полі через кому відображаються ті користувачі, у яких за замовчуванням (в файлі */etc/passwd*) призначена інша група.

На сьогоднішній день зберігання паролів в файлах *passwd* і *group* вважається ненадійним. У нових версіях ОС Linux застосовуються так звані тіньові файли паролів − **shadow** і **gshadow**. Права на них призначені таким чином, що навіть читання цих фалів без прав суперкористувача неможливе. Необхідно врахувати, що нормальне функціонування системи при використанні тіньових файлів має на увазі і одночасну наявність файлів *passwd* і *group*.

Додавання облікових записів користувачів

Детально розберемося, що відбувається при створенні нового облікового запису користувача.

По-перше, створюється запис у файлі */etc/passwd*. Формат запису наступний:

ім'я_користувача: пароль: UID: GID: *повне_ім'я: домашній_каталог: оболонка*

Розглянемо фрагмент цього файлу для суперкористувача *root* та звичайного користувача *max*:

root: x:0:0:root:/root:/bin/bash

max: x:500:500:Maxsim:/home/max:/bin/bash

Перше поле − це логін користувача, який він вводить для реєстрації в системі. Пароль в сучасних системах в цьому файлі не вказується, а друге поле залишилося просто для сумісності зі старими системами.

Третє і четверте поле − це UID (UserID) і GID (GroupID) − ідентифікатори користувача і групи, відповідно. Ідентифікатор користувача *root* завжди дорівнює 0, як і ідентифікатор групи *root*. Список груп можна знайти у файлі */etc/groups*.

П'яте поле − це справжнє ім'я користувача. Може бути не заповнене, а може містити прізвище, ім'я та по батькові користувача − все залежить від педантичності адміністратора системи.

Шосте поле містить ім'я домашнього каталогу. Зазвичай це каталог */home / <ім'я користувача>*.

Останнє поле − ім'я командного інтерпретатора, який буде обробляти введені вами команди після реєстрації у системі.

По-друге, при створенні користувача формується каталог */home /<ім'я користувача>*, в який копіюється вміст каталогу */etc/skel*.

Каталог */etc/skel* містить «джентльменський набір» − файли конфігурації за замовчуванням, які повинні бути в будь-якому призначеному для користувача каталозі. Назва каталогу *skel* (від англ. *Skeleton*) повністю виправдовує себе − він дійсно містить «скелет» домашнього каталогу користувача.

У операційній системі Windows всі користувачі мають однакові повноваження. У операційній системі Linux, крім звичайних користувачів, існує один (і тільки один) користувач з необмеженими правами. Ідентифікатори UID і GID такого користувача завжди 0. Його ім'я, як правило, **root.** Для користувача **root** права доступу до файлів і процесів не перевіряються системою. При роботі з обліковим записом **root** необхідно бути дуже обережним, тому що завжди існує можливість знищити систему. Система повністю підвладна цьому користувачеві.

Якщо виконати небезпечну команду, зареєструвавшись під ім'ям звичайного користувача, система повідомить, що у вас немає повноважень. Якщо ж працювати від імені користувача **root,** то навіть команда на видалення кореневої файлової системи буде системою виконана.

Тому в дистрибутиві Ubuntu звичайний обліковий запис **root** відключено - ви не зможете увійти в систему, використовуючи обліковий запис **root**. Зроблено це з міркувань безпеки, таким чином розробники намагаються захистити систему від некоректних дій.

У ОС Linux використовується розвинена система розподілу прав користувачів. Але з точки зору безпеки для точного впізнання одного імені користувача недостатньо. Саме тому використовується пароль − довільний набір символів довільної довжини, які зазвичай обмежуються лише використовуваними методами шифрування.

На сьогоднішній день в більшості версій Linux паролі шифруються за алгоритмами 3DES і MD5. Алгоритм 3DES є оборотним, тобто такий пароль можна розшифрувати, MD5 - це необоротне перетворення.

При автентифікації введений пароль шифрується тим же методом, що і вихідний, а потім порівнюються вже зашифровані копії. Якщо вони однакові, то автентифікація вважається успішною.

Враховуючи щоденне збільшення вимог до безпеки, в Linux є можливість використовувати приховані паролі. Файли */etc/passwd* і */etc/group* доступні для читання всім користувачам, що є досить великою проблемою для безпеки системи. Саме тому в сучасних версіях Linux краще використовувати приховані паролі. Такі паролі розміщуються в файлах */etc/shadow* і */etc/gshadow*, для паролів користувачів і груп, відповідно. Вони доступні для читання тільки користувачеві *root*.

Команда **login** використовується при вході в систему. Вона перевіряє правильність введення імені і пароля користувача, змінює каталог на домашній, вибудовує оточення і запускає командний інтерпретатор. Команду **login** не можна запускати з командного рядка - це викличе помилку і може привести до завершення сеансу.

Тимчасове отримання права користувача ***root***

Деякі операції, такі, наприклад, як встановлення програмного забезпечення або зміна конфігураційних файлів, вимагають повноважень *root*. Для їх отримання потрібно використовувати команду *sudo*:

sudo <*команда, _яку_потрібно_виконатиь_з_правами_ root*>

Наприклад, потрібно змінити файл */etc/apt/sources.list* - застосуєте для цього команду:

#sudo gedit /etc/apt/sources.list

Програма *gedit* - текстовий редактор, йому передається один параметр - ім'я файлу, який потрібно відкрити. Якщо ввести цю ж команду, але без *sudo* (просто так: *gedit /etc/apt/sources.list*), текстовий редактор все одно запуститься і відкриє файл, але зберегти зміни в ньому не зможемо, оскільки у нас не вистачає повноважень.

Команда *sudo* перед виконанням запросить пароль:

#sudo gedit /etc/apt/sources.list

*Password***:**

Необхідно ввести *свій пароль користувача* − той, що використовується для входу до системи, але не пароль користувача *root* (до речі, ми його і не знаємо). Використовувати команду *sudo* мають право не всі користувачі, а тільки, котрі занесені до файлу /*etc/sudoers.*

Пам'ятайте, що введений пароль зберігається 15 хвилин, тому через 15 хвилин програма може знову запросити у пароль (якщо за ці 15 хвилин не буде завершено роботу програми).

Якщо потрібно виконати серію команд з правами *root*, можна використати команду: *sudo -i*. Тоді не потрібно кожний раз на початку строки набирати *sudo*.

Існує ще одна команда, яка дозволяє змінити ідентифікатор користувача в процесі сеансу, це **su.** Її синтаксис:

su username

де *username* - ім'я користувача, яке буде використовуватися. Після цього програма запросить пароль. При правильному введені паролю, *su* запустить новий командний інтерпретатор з правами користувача, зазначеного у команді *su* і присвоїть сеансу його ідентифікатори. Якщо ім'я користувача не вказане, то команда *su* використовує ім'я *root*.

#su root
Password:
#_

При використанні користувачем *root* команди *su* ніколи не запитується пароль. Для закриття сесії *su* потрібно або ввести команду *exit*, або закрити термінал.

Команда ***newgrp*** аналогічна за своїми можливостями з ***su*** з тією різницею, що відбувається зміна групи. Користувач повинен бути включений до групи, яка вказується в командному рядку *newgrp*. При використанні команди *newgrp* користувачем *root* вона ніколи не запитує пароль. Синтаксис команди:

```
newgrp groupname,
```

де *groupname* - ім'я групи, на яку користувач змінює поточну.

Команда ***passwd*** є інструментом для зміни пароля в Linux. Для зміни свого пароля досить набрати в командному рядку *passwd*:

```
#passwd
Changing password for student
(current) UNIX password:
New password:
Retype new password:
passwd: all authentication tokens updated successfully
#_
```

Для зміни пароля групи і управління групою використовується команда ***gpasswd***. Для зміни пароля досить набрати в командному рядку ***gpasswd groupname***. Змінити пароль можна тільки адміністратору групи. Якщо пароль не порожній, то для членів групи виклик *newgrp* пароля не вимагає, а не члени групи повинні ввести пароль. Адміністратор групи може додавати і видаляти користувачів за допомогою параметрів -a і –d, відповідно. Адміністратори можуть використовувати параметр -r для видалення пароля групи.

1.5. Процеси

Головна частина ОС Linux, котра постійно знаходиться в оперативній пам'яті, називається *ядром* (Kernel). Ядро ОС Linux обробляє переривання від пристроїв, виконує запити системних процесів і додатків, призначених для користувача, розподіляє віртуальну пам'ять, створює і знищує процеси, забезпечує багатозадачність за допомогою перемикання між ними, містить драйвери пристроїв, обслуговує файлову систему (Рис. 1.3).

Рисунок 1.3. Структура ОС Linux

Призначені для користувача процеси не можуть безпосередньо породжувати інші процеси, зчитувати або записувати на диск інформацію, виводити дані на екран або створювати гніздо (socket) для обміну з мережею. Для виконання цих дій процеси повинні скористатися сервісами ядра. Звернення за такими послугами називаються *системними викликами*. Початкове завантаження системи полягає в тому, що файл з образом ядра зчитується в оперативну пам'ять, починаючи з нульової адреси. Цей файл знаходиться в каталозі */boot*.

В Linux-подібних системах на відміну від інших ОС ядро мінімізовано і не виконує жодної функції, яка використовується безпосередньо користувачем. Для цього застосовуються численні утиліти, які виступають в якості посередників між користувачем і ядром. Тільки в комплекті з ними ядро утворює повноцінну операційну систему. Ці компоненти ОС надійшли з проекту GNU, учасники якого з 1984 року працюють над створенням повноцінної Linux-подібної ОС, що цілком складається з вільно поширюваного програмного забезпечення. Це поняття є базовим в Linux-подібних системах. Процес можна уявити собі як віртуальну машину, котра знаходиться у розпорядженні одного завдання. Кожен процес вважає, що він на машині один і може розпоряджатися усіма її ресурсами. Насправді ж процеси надійно ізольовані один від одного, так що крах одного не може пошкодити усій системи. Кожен процес виконується у власній віртуальній пам'яті (рис. 1.4), до якої інший процес втрутитися не може. Цим і забезпечується стійкість усієї системи.

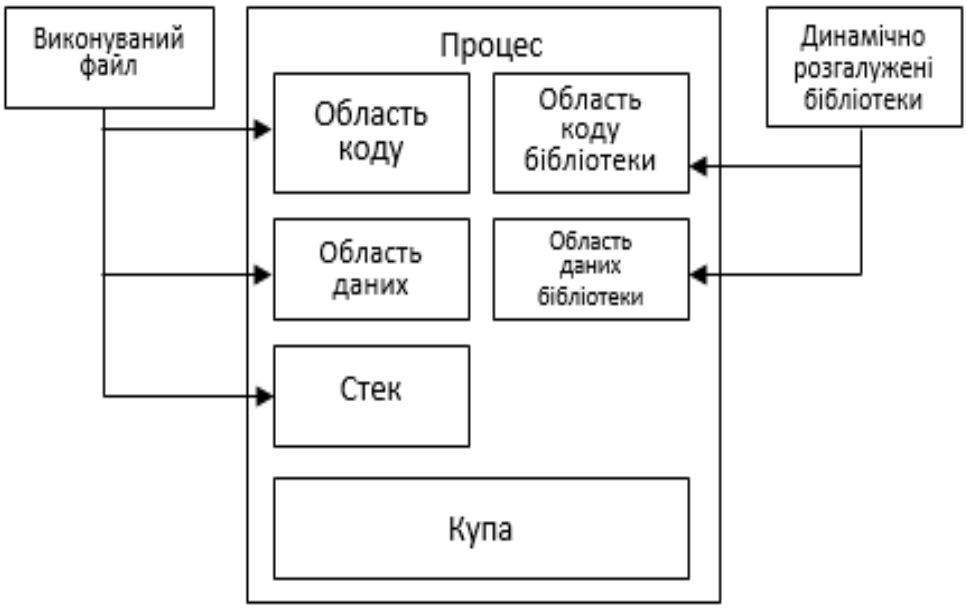


Рисунок 1.4. Віртуальна пам’ять процесу

Кожному процесу дозволено вважати, що його адреси починаються з нульової і від неї нарощуються. Таким чином, в 32-розрядній ОС процес

може адресувати 4 гігабайти оперативної пам'яті. Механізм віртуальної пам'яті дозволяє процесу думати, що саме стільки йому і виділено, хоча фізично обсяг ОЗП комп'ютера може бути набагато менше. Відсутню пам'ять заміняє жорсткий диск шляхом запису сторінок пам'яті, які тимчасово не використовуються, в розділ підкачки (swopping). Розділимість бібліотек між процесами забезпечується тим, що їх код і статичні дані відображаються на одну і ту ж ділянку фізичної оперативної пам'яті.

Всі процеси в ОС Linux утворюють ієрархічну структуру з єдиним коренем - процесом **init** – батьком всіх процесів, всі інші процеси є дочірніми. Кожен процес має номер (свій **PID**, **Process ID**), а також номер свого батьківського процесу (**PPID**, **Parent Process ID**). Для процесу **init PID** дорівнює **1**, і також **PPID=1** для **init,** тобто він сам собі батько. Для того, щоб отримати інформацію щодо процесів у системі Linux використовуються команди **ps** і **pstree**. Ці дві команди виводять список процесів, запущених в даний момент в системі, згідно з встановленими критеріями.

Через велику кількість процесів, вивід цієї команди буде займати кілька сторінок, тому для відображення доцільно використовувати конструкцію:

```
#pstree [options] | more
```

яка організовує посторінковий вивід інформації. Для зміни сторінок необхідно натиснути пробіл.

Символ «|» - це конвеєр. Його можна представити як канал, в який один процес тільки записує, а інший - тільки читає з нього. Вибірка і розміщення інформації в такий канал відбувається за принципом FIFO (*First In / First Out*). За допомогою конвеєра вивід однієї команди подається на вхід іншої. Це один з найпотужніших засобів Linux, котрий дозволяє комбінувати з простих команд довгі і витончені ланцюжки обробки даних.

Наприклад, потрібно зареєструватися та знайти довідку по конкретній команді, не перериваючи процесів, котрі протікають на інших консолях. Віртуальну консоль обслуговує програма *mingetty*, яка після реєстрації на цій консолі заміщає свій код на код командної оболонки. Тобто, потрібно

підрахувати кількість процесів *getty*. Існує команда **wc** (word count), яка вміє підраховувати кількість рядків, слів або байтів у файлі. Існує команда **grep,** котра вміє вибирати з файлу рядки, що містять вказаний фрагмент тексту. З'єднаємо їх конвеєром: $ps -e | grep getty | wc -l і отримаємо результат.

Знаючи **ID** процесу, можливо керувати процесом, тобто завершати процес або змінювати його пріоритет. Примусове завершення процесу необхідне якщо він «завис», а зміна пріоритету може знадобиться, якщо треба щоб скоріше процес закінчився.

Для знищення процесу (а також всіх його нащадків) використовуються команди **kill** та **killall**. Команда **kill** вимагає як аргумент номер процесу, а **killall** − ім'я процесу. Обидві ці команди опціонально можуть приймати номер сигналу, який повинен бути посланий процесу в якості аргументу. За замовчуванням, вони посилають сигнал номер **15(TERM)** для відповідного процесу (або кількох процесів). У загальному випадку, коли необхідно завершити виконання відразу декількох процесів потрібно просто вбити «батька».

Якщо користувач не працюєте під правами *root*, то будуть знищені тільки ті процеси, які йому належать. Запуск команди **killall** з правами *root* може призвести до перезавантаження всієї системи.

#killall [ім’я процесу]

#kill [-номер сигналу] PID

де *PID* - ідентифікатор процесу, який можна дізнатися за допомогою команди *ps*.

Для більш гнучкого управління процесами використовується програма **top**, яка виконує вивід списку процесів з сортування за процесорним часом. Дана програма надає інформацію щодо завантаження процесора, пам'яті, часу виконання Для виходу з програми застосовують клавішу <Q>.

Для гнучкого управління використовується ряд додаткових опцій:

- **k**: використовується для знищення процесу з зазначеним PID;
- **M**: сортування процесів за кількістю використовуваної ними пам'яті ;

- **P**: сортування процесів за використовуваним ними часом CPU (поле %CPU; метод сортування за замовчуванням);
- **u**: відображає процеси зазначеного користувача. Якщо користувача не буде вказано, то будуть відображені всі процеси;
- **i**: відображає всі процеси; ця команда надасть інформацію тільки щодо працюючих в даний момент процесів (процеси, у яких поле **STAT** має значення **R**, **Running**). Повторне використання цієї команди, знову поверне у попередній стан відображення.

Якщо програма запущена, вона дозволяє вводити ключі в діалоговому режимі. Однак, якщо опція вказується при введенні команди, то потрібно відразу ввести аргумент ключа.

Наприклад, команда:

```
#top –k 1234,
```

знищує процес з *PID 1234*.

Вивести список процесів, які протікають в даний час можна командою *ps*, наприклад:

```
# ps
PID TT STAT TIME COMMAND
24  3   S   0:03 (bash)
161 3   R   0:00  ps
#
```

Номери процесів (*process ID*, або *PID*), зазначені у першій колонці, є унікальними номерами, які система присвоює кожному працюючому процесу. Остання колонка, *COMMAND*, вказує ім'я працюючої команди. Повний список можна отримати командою *ps -aux*.

Використовуючи властивість керування завданнями, можна запустити відразу кілька завдань і перемикатися між ними. Управління завданнями може бути корисно, якщо, наприклад, потрібно редагувати великий текстовий файл і потрібно тимчасово перервати редагування, щоб зробити яку-небудь іншу операцію. За допомогою функцій управління завданнями можна тимчасово залишити редактор, повернутися до запрошення командної

оболонки і виконати які-небудь інші дії. Коли вони будуть виконані, можна повернутися назад до роботи з редактором в тому ж стані, в якому він був покинутий.

Сигнали

Механізм сигналів − це засіб, який дозволяє повідомляти процесам щодо деяких подій у системі, а процесу-одержувачу − належним чином на ці повідомлення реагувати. Послати сигнал може сам процес (наприклад, при спробі ділення на нуль), ядро (при збої обладнання), користувач або інший процес. Всього в Linux існує 63 сигнали, що позначаються своїми номерами або символічними іменами. Імена усіх сигналів починаються з SIG, і цю приставку часто опускають: так, сигнал, що вимагає припинити виконання процесу, називається SIGKILL, або KILL, або сигнал під номером 9. Отримавши сигнал, процес може: ігнорувати його; викликати для обробки встановлену за замовчуванням функцію; викликати власний обробник. Деякі сигнали перехопити або ігнорувати неможливо. Користувач може послати сигнал процесу з ідентифікатором *PID* командою

```
$ kill [-s <сигнал>] <PID>
```

де <сигнал> - це номер або символічне ім'я. Деякі сигнали посилаються після натискання комбінації клавіш. Так, Ctrl + C посилає сигнал INT, а Ctrl + \ (зворотний слеш) - сигнал QUIT. Отримує ці сигнали той процес, який працює на активній консолі − наприклад, очікує вашого введення. Команда *kill* носить таку назву тому, що частіше за все використовується для примусового завершення процесів, котрі вийшли з-під контролю та забирають багато ресурсів або просто «зависли». За умовчанням вона посилає сигнал TERM. Він відрізняється від сигналу KILL тим, що наказує процесу завершитися акуратно, закривши відкриті їм файли, видаливши тимчасові та інше.

РОЗДІЛ 2

Програмування засобами командної оболонки

2.1. Визначення командної оболонки

Linux-програми представлені файлами двох типів: виконуваними (executable) і сценаріями або пакетними файлами (script). Виконувані файли − це програми, які можуть безпосередньо виконуватися на комп'ютері; вони відповідають файлам ОС Windows з розширенням exe. Сценарії або пакетні файли − це набори команд для виконання іншою програмою, інтерпретатором. Вони відповідають в ОС Windows файлам з розширенням bat чи cmd або інтерпретується програмами на мові Basic.

ОС Linux не вимагає для виконуваних або пакетних файлів певні імена або будь-які розширення. Для позначення файлу як програми, яка здатна виконуватися, застосовуються атрибути файлової системи. В ОС Linux можна заміняти пакетні файли програмами, які відкомпілювалися, (і навпаки), не впливаючи на інші програми або користувачів, які звертаються до них. На рівні користувача, по суті, між ними немає різниці.

Оболонка командного рядка операційної системи Linux потужний інструмент, за допомогою якого можна виконувати найсерйозніші завдання найбільш легким способом.

Одна з причин застосування командної оболонки − можливість швидкого і простого програмування. Більш того, командна оболонка завжди присутня навіть коли встановлюють спрощену ОС Linux. Оболонка дуже зручна і для невеликих утиліт, котрі виконують відносно просте завдання, для якого продуктивність менш важлива, ніж надбудова, супровід і переносимість. Застосування оболонки для управління процесами забезпечує виконання команд в заданому порядку, котрий залежить від успішного завершення кожного етапу виконання.

Зовні командна оболонка дуже схожа на режим командного рядка в ОС Windows, але вона набагато могутніша і здатна виконувати самостійно

дуже складні програми. Наприклад, введення і виведення можна перенаправити за допомогою символів ‘<’ і ‘>’, передавати дані між двома одночасно працюючими програмами за допомогою символу ‘|’, а перехоплювати виведення підпроцесу за допомогою конструкції $(...). Можливо не тільки виконувати команди і викликати утиліти ОС Linux, але і розробляти їх.

Оболонка може запускатися на виконання в двох режимах – *інтерактивному* і *неінтерактивному*. Коли оболонка пропонує користувачеві запрошення, вона працює в *інтерактивному* режимі. Це означає, що вона приймає введення від користувача і виконує команди, які користувач вказує. Завершення роботи з оболонкою у такому разі відбувається за командою користувача.

В *неінтерактивному* режимі оболонка не взаємодіє з користувачем. Замість цього вона читає команди з деякого файла і виконує їх. При досягненні кінця файлу робота оболонки завершується. Такі файли називають *сценаріями* або *скриптами*. Такий підхід полегшує налагодження, тому що легко можна виконувати програму по рядках і не витрачати час на перекомпіляцію. При цьому результати роботи команд можуть становити або самостійну цінність, або слугувати вхідними даними для інших команд. Сценарії − це потужний спосіб автоматизації часто виконуваних дій. Але для задач, яким важливий час виконання або необхідно інтенсивне використання процесора, командна оболонка виявляється незручним середовищем. Командну оболонку можна представити як інтерфейс між користувачем і ОС Linux, котрий дозволяє вводити команди для виконання операційною системою.

В ОС Linux може використовуватися декілька встановлених командних оболонок (sh, bash, csh, ksh, zsh тощо) і різні користувачі вибирають ту, яка їм більше до вподоби. Усі командні оболонки, встановлені в системі, прописані у файлі */etc/shells*.

З точки зору користувача зазначені оболонки мало чим відрізняються. Всі вони дозволяють виконувати введені користувачем команди. Але зазначені оболонки відрізняються синтаксисом мови опису сценаріїв.

Тому основну увагу приділимо створенню bash-сценаріїв, оскільки оболонка *bash* найпопулярніша. У разі запуску оболонка *bash* виконує сценарій *.bashrc*, який знаходиться в домашньому каталозі користувача. У цьому файлі можна вказати команди, які потрібно виконати відразу після входу користувача до системи. Даний файл не обов'язковий і може бути відсутнім. В файлі *.bash_history* (теж знаходиться в домашньому каталозі) зберігається історія команд, які вводяться користувачем.

У Linux стандартна командна оболонка, завжди встановлюється як */bin/sh* і входить до комплекту засобів проекту GNU, котрий називається *bash* (GNU Bourne-Again SHell). В більшості дистрибутивів Linux програма */bin/sh*, командна оболонка за замовчуванням, − це посилання на програму */bin/bash*.

Командний інтерпретатор *bash* − це найбільш часто використовувана командна оболонка (командний інтерпретатор) операційної системи Linux. Основне призначення *bash* − виконання команд, введених користувачем. Користувач вводить команду, *bash* шукає програму, яка відповідає команді, в каталогах, зазначених у змінній оточення PATH. Якщо така програма знайдена, то *bash* запускає її і передає параметри, які вводяться користувачем. В іншому випадку виводиться повідомлення щодо неможливості виконання команди.

2.2. Створення простих сценаріїв для командного процесора

Сценарії командного процесора зручно використовувати для автоматизації завдань, що повторюються. До оболонки *bash* та інших командних оболонок включені базові компоненти, засновані на різних мовах програмування, наприклад: цикли, умови, оператори варіантів та інше.

Сценарії командного процесора є простими текстовими файлами, їх можна створювати за допомогою будь-якого текстового редактора (наприклад, *vi* або *nano*).

Перший рядок повинен вказувати, яку саме оболонку збираємося використовувати. Нас цікавить *bash*, тому перший рядок файлу буде таким:

```
#!/bin/bash
```

Зверніть увагу − якщо між # та ! виявиться пробіл, то дана директива не спрацює, оскільки буде сприйнята як звичайний коментар. В інших рядках цього файлу символ решітки # використовується для позначення коментарів, які оболонка не обробляє. Однак, перший рядок - це особливий випадок, тут решітка, за якою слідує знак оклику #! і шлях до *bash*, вказують системі на те, що сценарій створено саме для *bash*.

```
#!/bin/bash
# This is a comment
pwd
whoami
exit 0
```

Сценарій працює так. Спочатку команда *pwd* виводить на екран відомості щодо поточної робочої директорії, потім команда *whoami* показує ім'я користувача, який увійшов до системи.

Оскільки сценарій по суті обробляється як стандартне введення командної оболонки, він може містити будь-які команди ОС Linux, на які посилається змінна оточення *PATH*. Команда *exit* гарантує, що сценарій поверне осмислений код завершення. Він рідко перевіряється при інтерактивному виконанні програм, але якщо необхідно запускати даний сценарій з іншого сценарію і перевіряти, чи успішно він завершився, повернення відповідного коду завершення дуже важливо. У програмуванні засобами командної оболонки «0» означає успіх. Оскільки представлений варіант сценарію не може виявити помилки, він завжди повертає код успішного завершення.

Для запуску файл сценарію командного процесора повинен бути виконуваним. Наприклад, створеному сценарію дамо ім'я файлу *first.sh*, зробимо його виконуваним:

```
#chmod u + x first.sh
```

і він повинен при запуску бути частиною змінної *PATH* або визначатися за його повним або відносним шляхом. Іншими словами, при спробі запустити сценарій, отримаємо наступний результат:

 # first.sh bash: first.sh: command not found Невідома команда

У прикладі папка, котра містить файл *first.sh*, не включена до змінної *PATH*. Для вирішення цієї проблеми досить відредагувати шлях, скопіювати сценарій в папку змінної *PATH*.

Не слід ставити крапку «.» у змінній оточення *PATH* з метою позначення, що команда буде виконуватися з поточної папки. Оскільки її ім'я може збігтися з ім'ям важливої і широко використовуваної команди (наприклад, *1s* або *cat*), що викличе перезапис старих команд новими, якщо у них будуть однакові імена, і вони опиняться в одній папці, а це може поставити під загрозу безпеку всієї системи.

Можна ввести *./first* в каталог, який містить сценарій, щоб задати командній оболонці повний відносний шлях до файлу. Вказівка шляху, котрий починається з символів . /, надає ще одну перевагу: в цьому випадку випадково не можна виконати іншу команду з тим же ім’ям, що і у сценарію.

```
#./first
```

В іменах сценаріїв не використовуються ніякі розширення і суфікси. ОС Linux, як правило, рідко застосовує при іменуванні файлів розширення для вказівки типу файлу. Кращий спосіб перевірити, сценарій це чи ні - застосувати команду *file*, наприклад, *file first* або *file /bin/bash*.

2.3. Синтаксис командної оболонки

Існує два типа змінних, котрі можна використовувати в bash-скриптах:

- користувальницькі змінні;
- змінні оточення.

Користувальницькі змінні

Оболонка дозволяє створювати і вилучати власні змінні користувачу, а також присвоювати їм початкові значення:

ім'я_змінної=значення

За замовчуванням усі змінні вважаються рядками і зберігаються як рядки, навіть коли мають числові значення. Командна оболонка і деякі утиліти перетворюють рядки, котрі містять числа, в числові значення, коли потрібно їх обробити належним чином.

Linux – регістрозалежна система, тому командна оболонка вважає *fog* і *Fog* двома різними змінними, які відрізняються від третьої змінної *foG*.

У командній оболонці можна отримати доступ до значення змінної, якщо перед її ім'ям ввести знак $. При привласненні змінній будь-якого значення просто використовується ім'я змінної, яка при необхідності буде створена динамічно. Легко перевірити значення змінної, вивівши її на термінал за допомогою команди **echo** і вказавши перед її ім'ям знак $.

Коли змінна стає не потрібною її можна вилучити операцією *unset ім'я_змінної* .

Наведемо приклади:

```
#sable=white
#echo $sable
white
#sable="snow white"
#echo $sable
 snow white
#sable=7+5
#echo $sable
7+5
```

Треба звернути увагу, що при наявності пробілів у вмісті змінної її поміщають в лапки. Крім того, не може бути пробілів праворуч і ліворуч від знака рівності.

Значення змінної можна прочитати за допомогою команди *read***.** Вона приймає один параметр − ім'я змінної, в яку зчитуються дані, і потім чекає, поки користувач введе який-небудь текст. Команда *read* завершується після натискання клавіші <Enter>. При читанні значення змінної з терміналу поміщати її в лапки не потрібно:

```
#read sable
snow white
#echo $sable
snow white
```

Використання лапок

Зазвичай параметри в сценаріях відокремлюються символами, які не відображаються, або знаками форматування (наприклад, пробіл, знак табуляції або перехід на новий рядок). При необхідності щоб параметр містив один або декілька символів, які не відображаються, його слід взяти у лапки. Поведінка змінних, таких як $*foo*, укладених в лапки, залежить від виду використовуваних лапок. Якщо розмістити в подвійні лапки $ - представлення змінної, воно під час виконання замінюється значенням змінної. Якщо розташувати його в одинарних лапках або апострофах ніякої заміни не відбувається. Можна скасувати спеціальне призначення символу $, вставивши перед ним символ \ (зворотний слеш).

Наведемо приклад, в якому покажемо як лапки впливають на виведення змінної:

```
#!/bin/sh
var2="Good day"
echo $var2
echo "$var2"
echo ‘$var2'
echo \$var2
echo Enter some text
read var2
echo ‘$var2’ now equals $var2
exit 0
```

Цей сценарій на екран виводить наступне:

 # ./variable
Good day
Good day
$var2
$var2
Enter some text
Hello World
$var2 now equals Hello World

Створюється зміна *var2*, їй привласнюється рядок *Good day*. Вміст змінної виводиться на екран за допомогою команди *echo*, яка демонструє як символ $ розкриває вміст змінної. Бачимо, що використання подвійних лапок не впливає на розкриття вмісту змінної, а одинарні лапки й зворотний слеш впливають. Потім застосовуємо команду *read* для отримання введення від користувача.

Змінні оточення

Коли стартує сценарій командної оболонки, деяким змінним привласнюються початкові значення з оточення або робочого середовища. Зазвичай такі змінні позначають прописними літерами, щоб відрізняти їх в сценаріях від змінних користувача, які прийнято позначати малими літерами. Такі змінні залежать від персональних налаштувань користувача. Багато з них перераховані на сторінках довідкових посібників, а основні наведені в табл. 2.2.

Таблиця 2.2

Змінні оточення операційної системи Linux

Змінні оточення	Опис
$HOME	Домашній каталог поточного користувача
$PATH	Список каталогів для пошуку команд, який розділяється двокрапками
$PS1	Підказка чи запрошення командної строки
$IFS	Роздільник полів введення
$0	Ім'я сценарію командної оболонки
HOSTNAME	Ім’я комп’ютера
PWD	Поточний каталог
USER	Ім’я користувача
$#	Кількість параметрів, які передаються
$$	ID (ідентифікатор) процесу сценарію оболонки

Передача параметрів сценарію

Сценарій можна викликати з параметрами, при цьому створюється декілька додаткових змінних. Якщо параметри не передаються, змінна оточення $# все одно існує, але дорівнює 0. Сценарію на мові *bash* можуть передаватися параметри командного рядка, їх значення можна отримати за допомогою змінних $0, $1, $2, ..., $n-1, $n. При цьому $0 - ім'я файлу сценарію, $1 - перший параметр, $2 - другий, ... $n - останній.

Приклад. Демонстрування простих маніпуляцій зі змінними. Сценарій записується у файл *try_var,* для перетворення його в виконуваний необхідно виконати команду

#chmod + x try_var.

Тіло сценарію:

```
#!/bin/sh
salutation="Hello"
echo $salutation
echo "The program $0 is now running"
echo "The second parameter was $2"
echo "The first parameter was $1"
echo "The parameter list was $*"
echo "The user's home directory is $HOME"
echo "Please enter a new greeting"
read salutation
echo $salutation
echo "The script is now complete"
exit 0
```

При виконанні цього сценарію отримаємо наступний вивід:

```
# . /try_var foo bar baz
Hello
The program  ./try_var is now running
The second parameter was bar
The first parameter was foo
The parameter list was foo bar baz
The user's home directory is /home/rick
Please enter a new greeting
Sire
Sire
The script is now complete
```

#

Сценарій створює змінну *salutation*, виводить на екран її вміст і потім показує, що вже сформовані і мають відповідні значення різні змінні-параметри і змінна оточення $*HOME*.

2.3.1 Створення умовних переходів

Основою для всіх мов програмування є засоби перевірки умов і виконання різних дій з урахуванням результатів цієї перевірки. Сценарій командної оболонки може перевіряти код завершення деякої команди, котра викликається з командного рядка. Тому важливо завжди включати в створювані сценарії команду **exit** з певним значенням.

Команда test або [

На практиці в більшості сценаріїв використовується команда [або test - логічна перевірка командної оболонки. У деяких системах команди [та test - синоніми, за винятком того, що при використанні команди [для зручності читання застосовується і завершальна частина]. Команда [в програмному коді спрощує синтаксис і робить його більш схожим на інші мови програмування. Команда **test** не часто застосовується за межами сценаріїв командної оболонки. Багато користувачів ОС Linux, які ніколи раніше не писали сценаріїв, намагаються створювати прості програми і називають їх *test*. Якщо така програма не працює, ймовірно, вона конфліктує з командою оболонки *test*. Для з'ясування чи є у вашій системі зовнішня команда з такою назвою, спробуйте набрати щось таке, наприклад, *which test* і перевірити, яка саме команда *test* виконується в даний момент, або використовуйте форму *./test*, щоб бути впевненим, що виконується сценарій з поточного каталогу.

Наприклад, розглянемо команду *test* для перевірки наявності файлу. Для цього використовуємо наступну команду: *test -f <ім'я файлу>,* тому в сценарії можна написати:

```
if test -f fred.c
then
…
```

```
fi
```

Це можна написати наступним чином:

```
if [ -f fred.c ]
then
...
fi
```

Код завершення команди *test* (виконана чи умова) визначає чи буде виконуватися умовний програмний код.

Команда *test вираз* перевіряє указаний *вираз* і закінчує свою роботу з кодом завершення 0 (істина) або 1 (неправда).

Треба вставляти пробіли між квадратною дужкою [і умовою, котра перевіряється. Це легко засвоїти, якщо пам'ятати, що символ [являється синонімом команди *test*, а після імені команди завжди ставиться пробіл.

Варіанти умов, які можна застосовувати в команді *test*, розділяються на три типи: ***строкові порівняння***, ***числові порівняння*** і ***перевірка файлових прапорів*** (file conditionals). Почнемо зі строкових порівнянь (табл. 2.3).

Таблиця 2.3

Порівняння рядків

Варіанти умов	Результат
рядок1 = рядок2	True (істина), якщо рядки однакові
рядок1 != рядок2	True (істина), якщо рядки різні
рядок1 <. рядок2	True (істина), якщо рядок1 менше, ніж рядок2
рядок1 > рядок2	True (істина), якщо рядок1 більше, ніж рядок2.
-n рядок	True (істина), якщо рядок має ненульову довжину
-z рядок	True (істина), якщо рядок має нульову довжину

```
#!/bin/bash
user ="nataliya"
if [$user = $USER]
then
echo "The user $user is the current logged in user"
fi
```

В результаті виконання скрипта отримаємо наступне.

The user nataliya is the current logged in user

Існує така особливість порівняння строк. Оператори «>» і «<» необхідно екранувати за допомогою зворотної косої риски, інакше скрипт буде працювати неправильно, хоча повідомлення щодо помилок не з'являться. Скрипт інтерпретує знак «>» як команду перенаправлення виводу. Розглянемо, як треба працювати з такими операторами в коді:

```
#!/bin/bash
val1=text
val2="another text"
if [ $val1 \> $val2 ]
then
echo "$val1 is greater than $val2"
else
echo "$val1 is less than $val2"
fi
```

Результати роботи скрипта наведено на рис. 2.1.

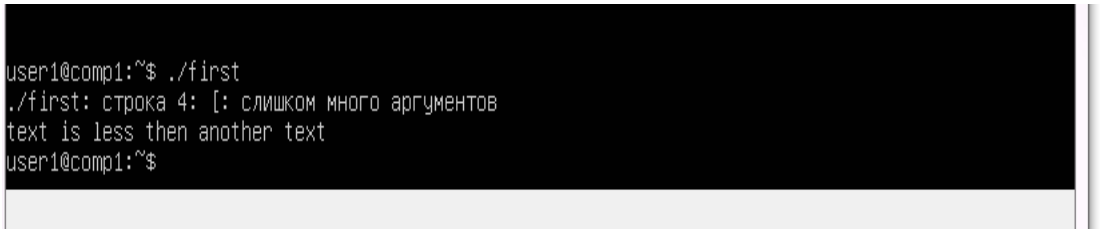


Рисунок 2.1 Приклад порівняння рядків, виводиться попередження

Зверніть увагу, що скрипт хоча і виконується, але виводить попередження : *./first: рядок 4: [: too many arguments*

Для позбавлення від попередження візьмемо `$val2` у подвійні лапки:

```
#!/bin/bash
val1=text
val2="another text"
if [ $val1 \> "$val2" ]
then
echo "$val1 is greater than $val2"
else
echo "$val1 is less than $val2"
fi
```

Тепер все працює як треба (рис. 2.2).

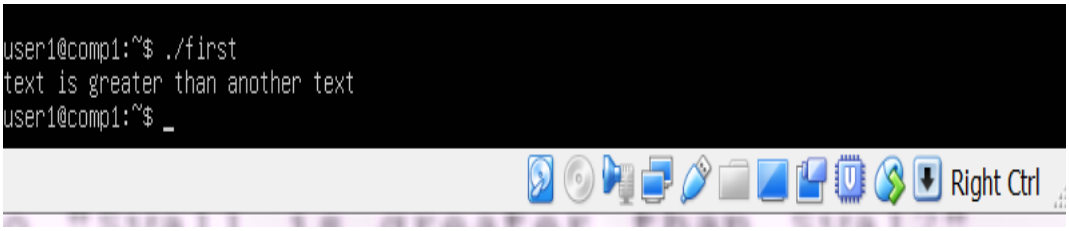


Рисунок 2.2 Порівняння рядків

Таблиця 2.4

Порівняння числових виразів

вираз1 *–eq* вираз2	true(), якщо числа
вираз1 *–ne* вираз2	true(), якщо числа не рівні
вираз1 *–lt* вираз2	true(), якщо менше
вираз1*–le* вираз2	true(), якщо менше чи дорівнює
вираз1 *–gt* вираз2	true(), якщо більше
вираз1 *–ge* вираз2	true(), якщо більше чи дорівнює

Наведемо **приклад**:

```
#!/bin/bash
val1=14
if [ $val1 -gt 5 ]
then
echo "The test value $val1 is greater than 5"
else
echo "The test value $val1 is not greater than 5"
fi
```

Результат виконання скрипта:

The test value is greater than 5

Основний синтаксис перевірки характеристик файлів наступний:

test *опція і'мя_файла*

В наведеній нижче таблиці зведені можливі опції команди *test* для перевірки характеристик файлів (табл. 2.5).

Таблиця 2.5

Перевірка прапорців файлів

Опція	опис
–b	true(), якщо файл існує і є блочним файлом
–c	true(), якщо файл існує і є символьним файлом
–d	true(), якщо файл існує і є каталогом
–e	true(), якщо файл існує
–f	true(), якщо файл існує і є звичайним файлом
–l	true(), якщо файл існує і є символічним посиланням
–r	true(), якщо файл існує і доступний для читання
–s	true(), якщо файл існує і має ненульовий розмір
-S	true(), якщо файл існує і є сокетом
–w	true(), якщо файл існує і доступний для запису
–x	true(), якщо файл існує і доступний для виконання

Приклад тестування стану файлу */bin/bash*.

```
#!/bin/ bash
if [ -f /bin/bash ]
then
echo "file /bin/bash exists"
fi
if [ -d /bin/bash ]
then
echo "/bin/bash is a directory"
else
```

2.3.2 Керуючі структури. Оператори розгалуження

В командній оболонці є низка керуючих структур або конструкцій, схожих на аналогічні структури в інших мовах програмування.

Керуючий оператор **if:** перевіряє результат виконання команди і в залежності від *умови* виконує ту чи іншу групу операторів

```
if умова_1 then
оператори_1
```

elif умова_2 **then**
оператори_2
…
elif умова_N **then**
оператори _N
else
оператори_N+1
fi

В наведеному запису як *elif*, так і *else* можуть бути відсутніми. Якщо тіло оператора *if* невелике, то звичайно використовують запис в один рядок:

if умова_1 ; **then** оператори_1 ; **else** оператори_2 ; **fi** .

Оператор **case**

Якщо необхідно зробити вибір з багатьох варіантів, то більш зручніше використовувати оператор **case**, котрий має такий синтаксичний запис:

case змінна **in**
зразок [| зразок]. . .) оператори; ;
зразок [| зразок]. . .) оператори ;;
esac

Конструкція оператора *case* дозволяє досить витонченим способом порівнювати вміст змінної зі зразками і потім виконувати різні оператори в залежності від того, з яким зразком знайдена відповідність. Це набагато простіше, ніж перевіряти декілька умов, використовуючи множину операторів *if, elif* і *else*.

Два символи ';;' після списку операторів мають той же зміст, що і оператор *break* в програмах на мові Сі. Як тільки рядок збігається з введеною інформацією, оператор *case* виконує код, наступний за), і завершується. Якщо ніяких співпадінь не знайдено, то оператор *case* завершується без виконання яких-небудь дій.

Застосовуючи конструкцію *case* з метасимволами в зразках, такими як *, треба бути особливо уважними. Проблема полягає в тому, що береться до уваги перша знайдена відповідність із зразком, незважаючи на те, що в наступних гілках можуть бути зразки з більш точною відповідністю.

Застосування єдиного символу * буде відповідати збігом з будь-яким введеним рядком, тому рекомендується помістити цей варіант після всіх інших зразків рядків для того, щоб задати деяку стандартну поведінку оператора *case*, якщо не будуть знайдені збіги з іншими рядками-зразками. Це можливо, тому що оператор *case* порівнює з кожним рядком-зразком по черзі. Він не шукає найкращу відповідність, а лише першу яка зустрілась. Умова, прийнята за замовчуванням, часто виявляється нездійсненою, тому застосування метасимвола '*' може допомогти в налагодженні сценаріїв.

2.3.3 Оператори циклів

Конструкція **for** застосовується для обробки в циклі ряду значень, які можуть являти собою будь-яку множину рядків. Рядки можуть бути просто перераховані в програмі або, що буває частіше, являти собою результат виконуваної командною оболонкою підстановки імен файлів. Синтаксис цього оператора:

for *var* **in** *list*
do
оператори
done

Для кожної ітерації циклу в змінну *var* буде записуватися наступне значення зі списку *list.* В першому проході циклу, таким чином, буде задіяне перше значення зі списку. В другому — друге, і так далі — до тих пір, поки цикл не дійде до останнього елемента.

Приклад. Застосування циклу *for* до фіксованих рядків. У командній оболонці значення зазвичай представлені у вигляді рядків, тому можна написати такий сценарій:

```
#!/bin/bash
for food in meat fish 123
do
echo $food
done
exit 0
```

В результаті отримаємо наступне:

meat

fish

123

В даному прикладі створюється змінна *food* і їй в кожному проході циклу *for* присвоюються різні значення. Оскільки оболонка вважає за замовчуванням всі змінні строковими, застосовувати рядок *123* так само допустимо, як і рядок *fish*.

Ще одним засобом ініціалізації циклу *for* є передавання йому списку, котрий є результатом роботи деякої команди. Тут використовується підстановка команд для їх виконання та отримання результатів їх роботи.

```
#!/bin/bash
file="myfile"
for var in $(cat $file)
do
echo " $var"
done
```

В прикладі використовується команда *cat*, яка читає вміст файлу. Отриманий список значень передається в цикл і виводиться на екран. Зверніть увагу на те, що файл, до якого звертаємось, містить список слів, котрі розділяються знаками переводу рядка, пробіли при цьому не використовуються (рис. 2.3).

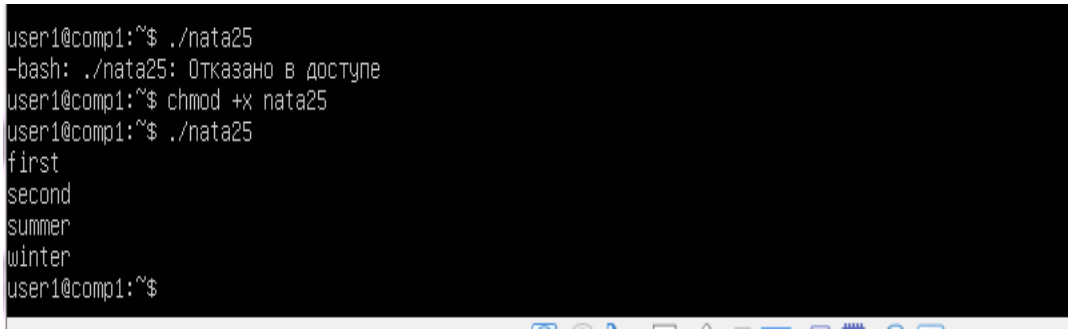


Рисунок 2.3 Вміст файлу перебирається за допомогою циклу

Треба враховувати, що подібний підхід для обробки даних по рядках, не спрацює для файлів більш складної структури, в рядках яких може

міститися по декілька слів розділених пробілами. Цикл буде обробляти окремі слова, а не рядки.

Цикл *for* можна використовувати в командній оболонці разом з метасимволами або знаками підстановки для імен файлів. Це означає застосування метасимвола для строкових значень і надання оболонці можливості підставляти всі значення на етапі виконання.

Приклад. Вивести на екран всі імена файлів сценаріїв в поточному каталозі, котрі починаються з літери *f* і закінчуються символами *.sh*. Це можна зробити наступним чином:

```
#! /bin /bash
for file in $(ls f*.sh);
do lpr  $file
done
exit 0
```

У наведеному прикладі показано застосування синтаксичної конструкції *$(команда)*. Зазвичай список параметрів для циклу *for* задається виводом команди, включеної в конструкцію $(). Командна оболонка розкриває *f*.sh*, підставляючи імена всіх файлів, які відповідають цьому шаблону.

Треба пам'ятати, що всі підстановки змінних в сценаріях командної оболонки здійснюються під час виконання сценарію, а не в процесі написання, тому всі синтаксичні помилки в оголошеннях змінних виявляються тільки на етапі виконання.

Зверніть увагу, як ініціалізується цикл, а саме – на підстановочний знак «*» у кінці адреси папки. Цей символ можна сприймати як шаблон, котрий означає «усі файли з усякими іменами». Він дозволяє організувати автоматичну підстановку імен файлів, які відповідають шаблону.

while

Оскільки за замовчуванням командна оболонка вважає всі значення рядками, оператор *for* використовується для циклічної обробки наборів рядків. Якщо заздалегідь не відомо скільки разів слід виконати послідовність команд, то, краще застосовувати цикл *while* з наступним синтаксичним записом:

```
while команда перевірки умови
do оператори
done
```

Приклад програми досить слабкої перевірки паролів.

```
#! /bin/bash
echo "Enter password"
read trythis
while [ "$trythis"! = "Secret"];
do echo "Sorry, try again"
read trythis
done
exit 0
```

Наступні рядки можуть служити прикладом виведення даного сценарію:

Enter password

password

Sorry, try again secret $

Це небезпечний спосіб з'ясування пароля, але він цілком підходить для демонстрування застосування циклу *while*. Оператори, які знаходяться між операторами *do* і *done*, виконуються нескінченне число разів до тих пір, поки умова залишається істинною (true). В даному випадку ви перевіряєте чи рівне значення змінної *trythis* рядку *secret*. Цикл буде виконуватися, поки $ *trythis* не дорівнюватиме *secret*, потім виконання сценарію продовжиться з оператора, наступного відразу за оператором *done*.

```
#!/bin/bash
var1=5
while [ $var1 -gt 0 ]
```

```
do
echo $var1
var1=$[ $var1 - 1 ]
done
exit 0
```

При вході до циклу перевіряється змінна *$var1*, чи більша вона за 0. Якщо це так, то виконується тіло циклу, в якому із значення змінної віднімається одиниця. Так відбувається в кожній ітерації, при цьому виводимо на консоль значення змінної до його модифікації. Як тільки *$var1* набуде значення 0, цикл закінчується.

until

У циклу until наступна синтаксична запис:

until умова
do оператори
done

Вона дуже схожа на синтаксичний запис циклу *while*, але зі зворотною перевіркою умови. Іншими словами, цикл залишиться активним, поки умова не стане істинною (true).

Зауваження. Якщо потрібно виконати цикл хоча б один раз, застосовують цикл *while*; якщо такої необхідності немає, використовують цикл *until*.

Мова програмування оболонки містить оператори, котрі порушують нормальне виконання циклу. Ці оператори мають назву – *break* і *continue*.

Керування циклами

Можливо, після входу до циклу необхідно буде зупинити його при досягненні змінною циклу певного значення, яке не відповідає спочатку заданій умові закінчення циклу. І в подібних випадках знадобляться наступні дві команди:

- break
- continue

Команда break дозволяє перервати виконання циклу. Її можна використовувати і для циклів *for*, і для циклів *while*:

```
#!/bin/bash
```

```
for var1 in 1 2 3 4 5 6 7 8 9 10
do
if [ $var1 -eq 5 ]
then
break
fi
echo "Number: $var1"
done
exit 0
```

Такий цикл у звичайних умовах пройде по всьому списку значень зі списку. Однак, в нашому випадку, його виконання буде припинено, коли змінна *$var1* буде дорівнювати 5.

Той же приклад, але для циклу *while*:

```
#!/bin/bash
var1=1
while [ $var1 -lt 10 ]
do
if [ $var1 -eq 5 ]
then
break
fi
echo "Iteration: $var1"
var1=$(( $var1 + 1 ))
done
exit 0
```

Команда *break* перериває цикл, коли змінна *$var1* стане дорівнювати 5.

Коли у тілі циклу зустрічається команда *continue*, то поточна ітерація закінчується достроково і починається наступна, при цьому виходу з циклу не відбувається. Розглянемо команду *continue* в циклі *for*:

```
#!/bin/bash
for (( var1 = 1; var1 < 15; var1++ ))
do
if [ $var1 -gt 5 ] && [ $var1 -lt 10 ]
then
continue
fi
echo "Iteration number: $var1"
done
```

Коли умова всередині циклу виконується, тобто коли *$var1* більше 5 та менше 10, оболонка виконує команду *continue*. Це призводить до пропуску команд, що залишилися у тілі циклу, і переходу до наступної ітерації.

Обробка виведення, котре виконується у циклі

Дані, які виводяться у циклі, можна обробляти передаючи їх на конвеєр або перенаправляти виведення. Виконується це за допомогою додавання команд обробки виведення після інструкції *done*.

Наприклад, замість того, щоб показувати на екрані те, що виводиться у циклі, можна записати все до файлу або передати ще кудись:

```
#!/bin/bash
for (( a = 1; a < 10; a++ ))
do
echo "Number is $a"
done > myfile.txt
echo "finished."
```

Оболонка створює файл *myfile.txt* та перенаправляє у нього всі значення від 1 до 9 .

Цикл ***select*** дозволяє створювати зручне меню. Його користь проявляється у випадку, коли користувачеві необхідно обрати один елемент з запропонованого списку. Оператор *select* має такий же вигляд, як і оператор *for*, за виключенням ключового слова.

При виконанні цього оператора циклу усі елементи зі списку висвічуються на екрані разом з їх порядковими номерами, після чого з'являється спеціальне запрошення для введення. Зазвичай воно має вигляд ‘#’. Введений користувачем номер пункту меню записується в змінну *REPLY*. Якщо $*REPLY* містить номер пункту меню, то в змінну ім'я заноситься значення відповідного елементу зі списку. Інакше список буде виданий наново. Після того, як користувачем буде зроблений допустимий вибір, виконаються команди зі списку, після чого виконання циклу повторюється із самого початку (висвітиться запрошення для введення і тощо). Для повторного відображення меню у відповідь на запрошення введення слід

натиснути клавішу <Enter>. Вихід з циклу здійснюється тими ж засобами, що і для *while* і *for*. Нижче наводиться приклад використання оператора *select*. У користувача запрошується тип пристрою "миша", і залежно від зробленого вибору виконуються певні дії. У даному випадку це просто видача повідомлення, що підтверджує зроблений вибір. У наведеному сценарії звернемо увагу на другий рядок. У більшості випадків вид запрошення, котре діє за умовчанням, користувача не влаштовує. Визначаючи змінну *PS3*, можна задати необхідний текст запрошення.

```
#!/bin/sh
#«конфігурування пристрою «миша»
PS3="Оберіть тип пристрою «миша»: "
select ITEM in Microsoft Logitech ps2 none
do
case $ITEM in
Microsoft) echo "дії по «миши» Microsoft " ;;
Logitech) echo "дії по «миши» Logitech" ;;
ps2) echo "дії по «миши» ps2" ;;
none) echo "обрано – немає «миши»" ;;
esac
break
done
```

2.4. Списки. Складні команди

У командній оболонці існує пара спеціальних конструкцій для роботи зі списками команд: І-список (AND list) і АБО-список (OR list). Вони часто застосовуються разом.

І-список. Ця конструкція дозволяє виконувати послідовність команд за умови успішного завершення попередньої команди. Синтаксис її:

оператор1 && оператор2 && оператор3 && . . .

Виконання операторів починається зліва направо, якщо *оператор1* повертає значення true (істина), виконується оператор, розташований праворуч від першого оператора. Виконання триває до тих пір, поки черговий оператор не поверне значення false (неправда), після чого вже

оператори списку не виконуються. Операція && перевіряє умову попередньої команди.

Кожен оператор виконується незалежно, дозволяючи поєднувати в одному списку множину різних команд. І-список успішно обробляється, якщо всі команди виконані успішно, в іншому випадку його обробка закінчується невдало.

Приклад. Спочатку звертаємось до файлу *file_one* (для перевірки його наявності, і якщо файл не існує створюємо його) і потім видаляєте файл *file_two*. Далі І-список перевіряє наявність кожного файлу і насамперед виводить на екран певний текст.

```
#!/bin/sh
touch file_one
rm -f file_two
if [ -f file_one ] && echo "hello" &&
&& [ -f file_two ] && echo " there"
then
echo "in if"
else
echo "in else"
fi
exit 0
```

При виконанні сценарію, отримаємо наступний вивід:

hello

in else

Команди *touch* та *rm* гарантують, що файли в поточному каталозі знаходяться у відомому стані. Далі І-список виконує команду [-f file_one], яка повертає значення true, тому що файл існує. Оскільки попередній оператор завершився успішно, тепер виконується команда *echo*. Вона теж завершується успішно (*echo* завжди повертає true). Потім виконується третя перевірка [-f file_two]. Вона повертає значення false, тому що файл не існує. Оскільки остання команда повернула false, заключна команда *echo* не виконується. В результаті І-список повертає значення false, тому в операторі *if* виконується варіант *else*.

АБО-список. Ця конструкція дозволяє виконувати послідовність команд доти, поки одна з команд не поверне значення true, і далі не виконується більш нічого. Маємо наступний синтаксичний запис:

оператор 1 | | оператор2 | | оператор3 | | ...

Оператори виконуються зліва направо. Якщо черговий оператор повертає значення false, виконується наступний за ним оператор. Це продовжується до того, поки черговий оператор не поверне значення true, після цього ніякі оператори вже не виконуються.

АБО-список дуже схожий на І-список, за винятком того, що правило для виконання наступного оператора - виконання попереднього оператора зі значенням false.

Приклад. Внесемо у попередній сценарій деякі зміни.

```
#!/bin/sh
rm -f file_one
if [ -f file_one ] II echo "hello" I| echo " there"
then
echo "in if"
else
echo "in else"
fi
exit 0
```

У результаті виконання сценарію буде отриманий наступний вивід:

hello

in if

Перші два рядки просто задають файли для іншої частини сценарію. Перша команда списку [-f file_one] повертає значення false, тому що файлу в каталозі немає. Далі виконується команда *echo*. Вона повертає значення true, і більше в АБО-списку не виконуються ніякі команди. Оператор *if* отримує зі списку значення true, оскільки одна з команд АБО-списку (команда *echo*) повернула це значення.

Результат, що повертається обома цими списками, − це результат останньої виконаної команди списку.

Описані конструкції списків виконуються так само, як аналогічні конструкції на мові С при перевірці множинних умов. Для визначення результату виконується мінімальна кількість операторів. Оператори, які не впливають на кінцевий результат, не виконуються. Зазвичай цей підхід називають оптимізацією обчислень (short circuit evaluation).

Послідовність команд, яка формується за допомогою команд списків, можна замінити конструкціями *if- else*, але отримаємо громіздкий результат.

Якщо треба застосовувати декілька операторів у тому місці програмного коду, де дозволено тільки один, наприклад в АБО-списку або І-списку, то можна помістити оператори в фігурні дужки {}.

2.5. Функції

У сценаріях оболонки можливо визначення і використання функцій. Під функцією розуміється пойменована група команд. У разі присутності коду, котрий повторюється з невеликими варіаціями, то виникає необхідність застосування функції. Завершену групу команд зручніше представити окремою функцією, що дозволяє спростити читання та сприйняття сценарію.

Для визначення функції застосовується такий синтаксис:

```
ім’я_ функції ()
{
оператори
}
```

Визначення функції обов’язково повинне передувати її першому застосуванню. Виклик функції відбувається шляхом вказівки її імені у якості команди. Функцію можна визивати необхідну кількість разів.

Наприклад. Створення простої функції

```
#!/bin/sh
fox () {
echo "Function foo is executing"
```

```
}
echo "script starting"
fox
echo "script ended"
exit 0
```

Після виконання на екрані отримаємо наступний текст:

script starting

Function fox is executing

script ended

Сценарій починає виконуватися з першого рядка. Далі зустрічається конструкція *‘fox() {‘* і визначається функція з ім’ям *fox*. Запам'ятовується посилання на функцію *fox* і продовжується виконання сценарію після дужки ‘}’. Коли зустрічається рядок з єдиним ім'ям *fox*, командна оболонка знає, що потрібно виконати заздалегідь визначену функцію. Коли функція завершиться, виконання сценарію продовжиться з рядка, наступного за викликом функції *fox*.

Використання команди return

Команда **return** дозволяє задавати цілочисельний код завершення, який повертається функцією. Існує два способи роботи з результатом виклика функції. Розглянемо перший:

```
#!/bin/bash
function func {
read -p "Enter a value: " value
echo "adding value"
return $(( $value + 10 ))
}
func
echo "The new value is $?"
```

Функція *func* додає число 10 до числа, котре міститься в змінній *$value*, значення якого задає користувач під час роботи сценарію. Потім вона повертає результат, використовуючи команду *return*. Результат того, що повертає функція, виводиться командою *echo* з використанням змінної $? (рис. 2.4).

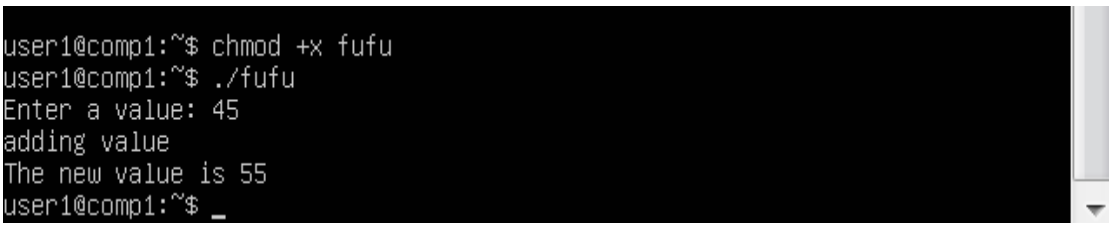


Рисунок 2.4 Виведення результатів скрипта

Якщо виконувати яку-небудь іншу команду до витягу із змінної $? значення, котре повертається функцією, це значення буде загублене. Річ у тім, що дана змінна зберігає код повернення останньої виконаної команди. Треба мати на увазі, що максимальне число, яке може повернути команда *return* — 255. Якщо функція повинна повернути більше число або строку, то необхідний другий підхід.

Запис виведення функції в зміну

Ще одним способом повернення результатів роботи функції є запис даних, які виводяться функцією, в змінну. Такий підхід дозволяє обійти обмеження команди *return* і повернути із функції усякі дані. Розглянемо приклад:

```
#!/bin/bash
function func {
echo -p "Enter a value: "
read value
echo $(( $value + 10 ))
}
result=$(func)
echo "The value is $result"
```

Після виконання даного скрипта отримуємо (рис. 2.5):

Рисунок 2.5 Запис результатів роботи функції у змінну.

При виклику функції їй можна передавати аргументи і при своєму закінченні функція може повернути результат виконання.

Функції можуть використовувати стандартні позиційні параметри, в які записується те, що передається їм при виклику. Наприклад, ім'я функції зберігається у параметрі $0, перший аргумент, який їй передається, — в $1, другий — в $2 і тощо. Кількість аргументів, які передаються функції, можна узнати, якщо звернутися до змінної $#.

Аргументи передаються функції при запису їх після її імені:

```
func $val1 10 20
```

Приклад, в якому функція викликається з аргументами і займається їх обробкою:

```
#!/bin/bash
function addnum {
if [ $# -eq 0 ] || [ $# -gt 2 ]
then
echo -1
elif [ $# -eq 1 ]
then
echo $(( $1 + $1 ))
else
echo $(( $1 + $2 ))
fi
}
echo -n "Adding 10 and 15: "
value=$(addnum 10 15)
echo $value
echo -n "Adding one number: "
value=$(addnum 10)
echo $value
echo -n "Adding no numbers: "
value=$(addnum)
echo $value
echo -n "Adding three numbers: "
value=$(addnum 10 15 20)
echo $value
```

Функція *addnum* перевіряє число переданих їй при виклику з скрипта аргументів. Якщо їх немає або їх більше двох, функція повертає значення -1. Якщо параметр всього один, вона додає його до нього самого і повертає результат. Якщо параметрів два, функція складає їх. Зверніть увагу на те, що функція не може безпосередньо працювати з параметрами, які передані скрипту при його запуску з командного рядка. Наприклад, напишемо такий сценарій:

```
#!/bin/bash
function myfunc {
echo $(( $1 + $2 ))
}
if [ $# -eq 2 ]
then
value=$( myfunc)
echo "The result is $value"
else
echo "Usage: myfunc  a b"
fi
```

Якщо викликати цей скрипт, а точніше викликати функцію, яка в ньому оголошується, то отримаємо повідомлення щодо помилки.

Замість цього, якщо в функції планується використовувати параметри, передані скрипту при виклику з командного рядка, треба передати їх їй при виклику:

```
#!/bin/bash
function myfunc {
echo $(( $1 + $2 ))
}
if [ $# -eq 2 ]
then
value=$(myfunc $1 $2)
echo "The result is $value"
else
echo "Usage: myfunc a b"
fi
```

Тепер скрипт працює без помилок (рис. 2.6).

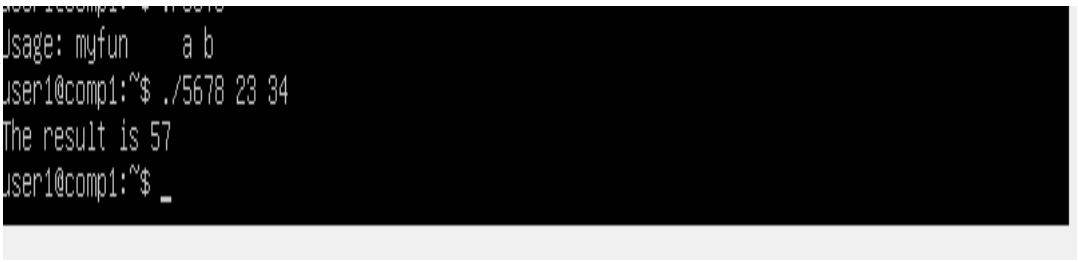


Рисунок 2.6 Передавання функції параметрів, з якими скрипт запускається

Коли функція викликається, позиційні параметри сценарію $ *, $ @, $ #, $1, $2 та інші замінюються параметрами функції. Саме так зчитуються параметри, які передаються функції. При завершені функції параметри відновлять свої колишні значення.

Можна змусити функцію повертати числові значення за допомогою команди *return*. Звичайний спосіб повернення функцією строкових значень − збереження рядка в змінній, яку можна використовувати після завершення функції.

Робота зі змінними в функціях

Змінні, які використовуються в сценаріях, характеризуються областю видимості. Це − ті місця коду, де можна працювати з цими змінними. Існують два види змінних:

• Глобальні змінні.

• Локальні змінні.

Глобальні змінні − це змінні, які видно з будь-якого місця *bash*-скрипта. Якщо оголосити глобальну змінну в основному коді скрипта, до такої змінної можна звернутися з функції. Це ж справедливо і для глобальних змінних, оголошених у функціях. Звертатися до них можна і в основному коді скрипта після виклику функцій.

За замовчуванням всі оголошені в скриптах змінні глобальні. Так, до змінних, оголошених за межами функцій, можна без проблем звертатися з функцій:

```
#!/bin/bash
```

```
function myfunc {
value=$(( $value + 10 ))
}
echo -p "Enter a value: " value
read value
myfunc
echo "The new value is: $value"
```

Коли змінній присвоюється нове значення в функції, це нове значення не губиться, коли скрипт звертається до неї після завершення роботи функції. Саме це можна бачити в попередньому прикладі.

Змінні, які оголошують і використовують всередині функції, називаються *локальними*. Для цього використовується ключове слово *local* перед ім'ям змінної:

local temp = $ (($ value + 5))

Якщо за межами функції є змінна з таким же ім'ям, це на неї не вплине. Ключове слово *local* дозволяє відокремити змінні, що використовуються всередині функції, від інших змінних. Розглянемо приклад:

```
#!/bin/bash
function myfunc {
local temp=$[ $value + 5 ]
echo "The Temp from inside function is $temp"
}
temp=4
myfunc
echo "The temp from outside is $temp"
```

Коли працюємо зі змінною $*temp* всередині функції, це не впливає на значення, призначене змінній з таким же ім'ям за її межами.

Можна оголошувати локальні змінні у функціях за допомогою ключового слова *local*. У цьому випадку змінна дійсна тільки в межах функції. В інших випадках функція може звертатися до змінних командної оболонки, які мають глобальну область дії. Якщо ім'я локальної змінної співпадає з ім'ям глобальної, в межах функції локальна змінна перекриває

глобальну. Для того щоб переконатися в цьому на практиці, можна змінити попередній сценарій наступним чином.

```
#!/bin/sh
sample_text="global variable"
food () {
local sample_text="local variable"
echo "Function foo is executing"
echo $sample_text
}
echo "script starting"
echo $sample_text
food
echo "script ended"
echo $sample_text
exit 0
```

За умови відсутності команди *return*, яка задає значення, що повертається, функція повертає статус завершення останньої виконаної команди.

Приклад. У сценарії *myname* показано, як у функцію передаються параметри і як функції можуть повернути логічний результат true або false. Можна викликати цей сценарій з параметром, котрий задає ім'я, яке будемо використовувати в запиті.

1. Після заголовку командної оболонки визначаємо функцію *yes_or_no*.

```
#!/bin/sh
yes_or_no() {
echo "Is your name $* ? "
while true
do
echo -n "Enter yes or no: "
read x
case "$x" in
y | yes ) return 0;;
n | no ) return 1;;
* ) echo "Answer yes or no"
esac
done
}
```

2. Основна частина програми.

```
echo "Original parameters are $*"
if yes_or_no "$1"
then
echo "Hi $1, nice name"
else
echo "Never mind"
fi
exit 0
```

Типовий вивід сценарію виглядатиме наступним чином:

$. /my_name Bill Grey

Original parameters are Bill Grey l

Is your name Bill ?

Enter yes or no: yes

Hi Bill, nice name

Коли сценарій починає виконуватися, функція визначена, але ще не виконується. В операторі *if* сценарій викликає функцію *yes_or_no*, передаючи їй частину рядка як параметри після заміни $1 першим параметром вихідного сценарію рядком *Bill*. Функція використовує ці параметри, в даний момент зберігаються в позиційних параметрах $ 1, $ 2 і тощо. І повертає значення в програму. Залежно від значення, яке повертається функцією, конструкція *if* виконує один з операторів.

2.6. Команди

У сценаріях командної оболонки існує два види команд: "звичайні", які можуть виконуватися і у командному рядку (їх називають зовнішніми командами), і вбудовані команди (їх називають внутрішніми командами). Внутрішні команди реалізовані всередині оболонки і не можуть викликатися як зовнішні програми. Але більшість внутрішніх команд представлено і у вигляді автономних програм, ця умова - частина вимог стандарту *POSIX*. Зазвичай не важливо, команда зовнішня чи внутрішня, за винятком того, що внутрішні команди діють ефективніше.

***Команда* «:»** — фіктивна команда. Вона корисна для спрощення логіки в створені умов, будучи псевдонімом команди true. Оскільки команда : вбудована, вона виконується швидше, ніж true, хоча її вивід набагато менш читабельний. Можна знайти цю команду в створені умови для циклів *while*. Конструкція *while*: виконує нескінченний цикл замість більш загального *while true*. Конструкція: також корисна для умовного завдання змінних. наприклад,

```
: ${var:=value}
```

Без ‘:’ командна оболонка спробує інтерпретувати *$var* як команду.

***Команда* «.».** Команда "крапка" (.) виконує команду в поточній оболонці:. *./shell_script.* Зазвичай, коли сценарій виконує зовнішню команду або сценарій, створюється нове оточення (підоболонка), команда виконується в новому оточенні і потім оточення видаляється, за винятком коду завершення, який повертається в батьківську оболонку. Зовнішня команда *source* і команда "крапка" (ще два синоніма) виконують команди, наведені в сценарії, в тій же командній оболонці, яка виконує сценарій.

Оскільки за замовчуванням під час роботи сценарію створюється нове оточення, будь-які зміни змінних оточення, зроблені в сценарії, губляться. З іншої сторони, команда "крапка" дозволяє сценарієм змінювати поточне оточення. Це часто буває корисно, коли сценарій застосовується як оболонка для налаштування оточення, призначеного для подальшого виконання будь-якої іншої команди. Наприклад, ви працюєте над декількома різними проектами одночасно, може виявитися, що необхідно виконувати команди з різними параметрами, наприклад, запускати старішу версію компілятора для підтримки старої програми.

У сценаріях командної оболонки команда "крапка" грає роль, трохи схожу на роль директиви #include в мовах програмування C і C ++. І хоча вона не підключає сценарій в буквальному сенсі слова, вона дійсно виконує команду в поточному контексті, тому можна застосовувати її для включення змінних і визначення функцій у сценарію.

Приклад. Команда "крапка" застосовується в командному рядку, але не можна використовувати її і в сценарії.

1. Припустимо, маємо два файли, які містять параметри оточення для двох різних середовищ розробки. Для установки оточення, призначеного для старих класичних команд, *classic_set*, можна застосувати наступний програмний код.

```
#!/bin/sh
version=classic
PATH=/usr/local/old_bin:/usr/bin:/bin:
PSl="classic> "
```

2. Для нових команд приймається *latestset*.

```
#!/bin/sh
version=latest
PATH=/usr/local/new_bin:/usr/bin:/bin:
PS1=" latest version> "
```

Можна встановити оточення, застосовуючи ці сценарії в поєднанні з командою "крапка", як показано в наступній порції прикладу далі.

```
$ . ./classic_set
classic> echo $version
classic
class ic> . /latest_set
latest version> echo $version
latest
latest version>
```

Сценарії виконуються, використовується команда "крапка", тому кожен з них виконується в поточній командній оболонці. Це дозволяє сценаріям змінювати параметри оточення в поточній оболонці, яка зберігає зміни навіть після того, як сценарій завершився.

echo

Команда **echo** використовується для виведення рядка з подальшим переходом на новий рядок. При цьому виникає загальна проблема: видалення символу переходу на новий рядок.

На жаль, в різних версіях ОС UNIX реалізовані різні рішення. В ОС Linux загальноприйнятий метод

```
echo -n "string to output"
```

Але можна стикатися і з варіантом

```
echo -e "string to output \ c"
```

Другий варіант *echo -e* розрахований на те, що задіяна інтерпретація символів escape-послідовності, котрі починаються зі зворотного слеша, таких як ‘\c’ для пригнічення нового рядка, ‘\t’ для виведення табуляції, ‘\n’ для виведення символів повернення каретки.

eval

Команда *eval* дозволяє обчислювати аргументи. Вона вбудована в командну оболонку і зазвичай не представлена як окрема команда. Найкраще її дію демонструє короткий приклад:

```
foo=10
x=foo
y='$'$x
echo $y
```

Буде виведено *$foo*. Але, код

```
foo=10
x=foo
eval y='$f$x
echo $y
```

виведе на екран 10. Таким чином, *eval* схожа на додатковий знак $: вона повертає значення змінної.

Команда *eval* дуже корисна, вона дозволяє генерувати і виконувати код на льоту. Застосування цієї команди ускладнює налагодження сценарію, але дозволяє робити те, що в іншому випадку виконати складно або навіть неможливо.

exec

Команди *exec* застосовується у двох варіантах. Зазвичай її використовують для зміни поточної командної оболонки іншою програмою.

Наприклад, рядок *exec wall "Thanks for all the fish"* в сценарії замінить поточну оболонку командою *wall*. Рядки, які йдуть за командою *exec*, не

обробляються, тому що командна оболонка, яка виконувала сценарій, більше не існує.

Другий варіант застосування *exec* - модифікація поточних дескрипторів файлів *exec 3 <afile*

Ця команда відкриває файловий дескриптор 3 для читання з файлу *afile*. Цей варіант рідко використовується.

exit n

Команда *exit* викликає закінчення сценарію з кодом завершення n. Якщо застосовувати її в рядку підказки або запрошення будь-якої інтерактивної командної оболонки, вона призведе до виходу з системи. Якщо дозволити сценарію завершитися без вказівки коду завершення, статус останньої виконаної в сценарії команди використовується як значення, котре повертається. Завдання коду завершення вважається хорошим стилем програмування.

При програмуванні сценаріїв у командній оболонці код завершення 0 означає успішне завершення сценарію, коди від 1 до 125 включно − коди помилок, які можна використовувати для налагодження в сценаріях. Решта значень зарезервовані відповідно до табл.2.6

Таблиця 2.6

Код завершення	**Опис**
126	файл не є виконуваним
127	невідома команда
128 і вище	сигнал, що з’явився

Багатьом програмістам на мовах C і C ++ використання нуля як ознаки успішного завершення може здатися дещо незвичним. Велика перевага сценаріїв − можливість застосування 125 кодів помилок, визначених користувачем, і відсутність необхідності в глобальній змінній для збереження коду помилки.

Приклад. Успішне завершення, якщо в поточному каталозі існує файл з ім'ям *.profile*.

```
#!/bin/sh
if [ -f .profile ]; then
exit 0
fi
exit 1
```

Цей сценарій можна переписати у вигляді одного рядка, використовуючи комбінацію І-списку і АБО-списку, описаних раніше:

```
[ -f .profile ] && exit 0 || exit 1
```

export

Команда *export* робить змінну, яка називається її параметром, доступною в підоболонці. За замовчуванням змінні, які створені в командній оболонці, не доступні в нових дочірніх підоболонках, котрі запускаються з поточної. Команда *export* створює зі свого параметра змінну оточення, яку видно іншими сценаріями і програмами, котрі запускаються з поточної програми. Висловлюючись професійною мовою, змінні, які експортуються, формують змінні оточення в будь-яких дочірніх процесах, породжених командною оболонкою. Найкраще проілюструвати це прикладом з двох сценаріїв: *export1* і *export2*.

Приклад. *Експорт змінних*

1. Першим представимо сценарій *export2*.

```
#!/bin/sh
echo "$foo"
echo "$bar"
```

2. Далі *export1*. У кінці сценарію запускається *export2*.

```
#!/bin/sh
foo="The first meta-syntactic variable"
export bar="The second meta-syntactic variable"
export2
```

Якщо запустити їх, то отримаєте наступний результат.

$ ./exportl

The second meta-syntactic variable

Сценарій *export2* виводить значення двох змінних. У сценарії *exportl* задаються значення обох змінних, але тільки змінна *bar* позначається як продукція, що експортується, тому, коли згодом запускається сценарій *export2*, значення змінної *foo* втрачено, а значення змінної *bar* експортовано в другий сценарій. На екрані з'являється порожній рядок, оскільки $*foo* нічого не містить і виведення змінної зі значенням null призводить до відображення нового рядка.

Після того як змінна була експортована з командної оболонки, вона експортується в будь-які сценарії, котрі запускаються з цієї оболонки, і в будь-які командні оболонки, які в свою чергу запускають ці сценарії, тощо. Якщо б сценарій *export2* викликав інший сценарій, в ньому змінна *bar* також була б доступна.

Примітка

Команди *set -a* або *set -allexport* експортують всі змінні,− відповідно.

expr

Команда *expr* обчислює вираз, складений з її аргументів. Найчастіше вона застосовується для підрахунку простих арифметичних виразів у наступному вигляді:

x=’expr $x + 1’

Символи ‘‘ (зворотні лапки або апостроф) змушують змінну x прийняти результат виконання команди *expr $ x + 1*. Її можна також записати за допомогою синтаксичної конструкції $() замість зворотної лапки, наприклад, наступним чином*: x = $ (expr $ x + 1)*.

Команда *expr* володіє великими можливостями, за її допомогою можна обчислювати різні вирази. Основні види обчислень перераховані в табл.2.7.

Таблиця 2.7

Види обчислення

Обчислення виразу	**Опис**
вираз 1 | вираз 2	вираз 1, якщо вираз 1 не дорівнює нулю, в іншому випадку вираз 2
вираз 1 & вираз 2	нуль, якщо обидва вирази дорівнюють нулю, в іншому випадку вираз 1
вираз 1 = вираз 2	рівність
вираз 1 > вираз 2	більш ніж
вираз 1 >= вираз 2	більше або дорівнює
вираз 1 < вираз 2	менше
вираз 1 <= вираз 2	менше або дорівнює
вираз 1 != вираз 2	нерівність
вираз 1 + вираз 2	додавання
вираз 1 - вираз 2	віднімання
вираз 1 * вираз 2	множення
вираз1 / вираз 2	ділення
вираз 1%| вираз 2	остача від ділення

printf

Команда *printf* є тільки в сучасних командних оболонках, її слід застосовувати замість команди *echo* для генерації форматованого виведення. У команди наступний синтаксичний запис.

printf "рядок формату" параметр1 параметр2. . .

Рядок формату дуже схожий з деякими обмеженнями на застосовувану на мові програмування С і С ++. Головним чином не підтримуються числа з плаваючою точкою, оскільки всі арифметичні операції в командній оболонці виконуються над цілими числами. Рядок формату складається з довільної комбінації літеральних символів, escape-послідовностей і специфікаторів

перетворення. Всі символи рядка формату, які відрізняються від \ і %, відображаються на екрані при виведенні.

У табл. 2.8 наведені escape-послідовності, які підтримуються командою.

Таблиця 2.8

Підтримувані escape-послідовності

Escape-послідовність	**Опис**
\”	Подвійні лапки
\\	Зворотній слеш
\a	Звуковий сигнал тривоги
\b	Символ backspace
\c	Відкидання наступного виводу
\f	Символ from feed (подавання паперу)
\n	Символ переходу на новий рядок
\r	Повернення каретки
\t	Символ табуляції
\v	Символ вертикальної табуляції
\ooo	Один символ з вісімковим значенням ooo
\xHH	Один символ з шістнадцятковим значенням HH

Специфікатори перетворень досить складні, тому ми наведемо найбільш поширені варіанти їх застосування. Більш детальну інформацію можна знайти в інтерактивному довідковому керівництві командної оболонки *bash*.

Специфікатор перетворення складається з символу %, за яким йде символ перетворення. Основні варіанти перетворень перераховані в табл. 2.9.

Таблиця 2.9

Символ перетворення	**Опис**
D	Виведення десяткового числа
C	Виведення символу
S	Виведення рядку
%	Виведення знаку %

Рядок формату використовується для інтерпретації інших параметрів команди і виведення результату, як показано в прикладі:

#printf "% s \ n" hello

hello

#printf "% s% d \ t% s" "Hi There" 15 people

Hi There 15 people

Звернемо увагу на те, що для захисту рядка '*Hi There*' і перетворення його в єдиний параметр, рядок потрібно заключити в лапки ("").

Команда *return* служить для повернення значень з функцій. Команда приймає один числовий параметр, який стає доступним в сценарії, що викликає функцію. Якщо параметр не заданий, команда *return* за замовчуванням повертає код завершення останньої команди.

set

Команда *set* задає змінні-параметри командної оболонки. Вона корисна при використанні полів в командах, які виводять значення, розділені пробілами. Наприклад, необхідно в сценарії використовувати назву поточного місяця. В системі є команда *date*, яка містить назву місяця у вигляді рядка, але потрібно відокремити його від інших полів. Це можна зробити за допомогою комбінації команди *set* і конструкції *$(...)*, які забезпечать виконання команди *date* і повернення результату. У виведені команди *date* рядок з назвою місяця - другий параметр.

```
#!/bin/sh
echo the date is $(date)
set $(date)
echo The month is $2
exit 0
```

Програма задає список параметрів для виведення команди *date*, потім використовує позиційний параметр $2 для отримання назви місяця.

Використання команди *date* наведено для демонстрації застосування позиційних параметрів. Оскільки команда *date* залежить від мовних параметрів або локалізації, отримати назву місяця можна командою *date+%B*. У команди *date* багато інших варіантів форматування,

Команду *set* можна також застосовувати для передачі параметрів командній оболонці і тим самим управляти режимом її роботи. Найбільш часто використовується варіант команди *set-x*, який змушує сценарій виводити на екран трасування команди, яка виконується в даний час.

shift

Команда *shift* просуває усі змінні-параметри на одну позицію назад, так що параметр $2 стає параметром $1, параметр $3 − $2 і так далі. Попереднє значення параметра $1 відкидається, а значення параметра $0 залишається незмінним. Якщо у виклику команди *shift* заданий числовий параметр, параметри зсуваються на вказану кількість позицій. Решта змінних $*, $@ і $# також змінюються в зв'язку з новим розміщенням змінних-параметрів.

Команда *shift* корисна при почерговому перегляді параметрів, переданих в сценарій, і якщо сценарію потрібно 10 і більше параметрів, знадобиться команда *shift* для звернення до 10-го параметру і наступного за ним.

Наприклад, можна переглянути всі позиційні параметри:

```
#!/bin/bash
while [ "$1" != ]; do
echo "$1"
shift
done
exit 0
```

stat

Команда **stat** виводе детальну інформацію щодо файлу або файлової системи, включаючи розмір, кількість блоків, дату останнього доступу, модифікації і тощо.

stat [OPTION] FILE...

Виведення вмісту полів дескриптора файлу або статус файлової системи

#stat example.file
File: `example.file'
Size: 5614 Blocks: 16 IO Block: 4096 regular file
Device: 30bh/779d Inode: 786506 Links: 1
Access: (0644/-rw-r--r--) Uid: (500/max) Gid: (500/max)
Access: 2018-04-26 20:13:06.000000000 +0000
Modify: 2018-04-26 20:17:39.000000000 +0000
Change: 2018-04-26 20:18:09.000000000 +0000

#stat -f example.file File: example.file
ID: 0 Namelen: 255 Type: ext4
Blocks: Total: 4125127 Free: 3855924 Available: 3646377 Size: 4096
Inodes: Total: 2097152 Free: 2086052

Опції

-c --format=FORMAT застосовується вказаний формат виведення FORMAT, замість використовуваного за замовчуванням

-f, --filesystem виведення статусу файлової системи (на якій розташований FILE) замість статусу файлу

-L, --dereference виведення інформації щодо статусу оригінального файлу, з яким пов'язане посилання FILE

-t, --terse виведення інформації в скороченій (стислій) формі

--version виводить інформацію про версію програми і завершує її роботу

У якості формату виведення **FORMAT** для файлів (але не для опції -f, --filesystem) припустимі наступні прапорці та їх комбінації послідовностей:

%A – права доступа у зручній для сприйняття людиною формі (rwx)

#stat -c%A example.file
-rw-r--r--

%a – права доступа у вісімковому форматі (0..7)

#stat -c%a example.file
644

%F – тип файлу (наприклад, звичайний файл, каталог, посилання, сокет, спец.блочний файл, спец.сімвольний файл, файл іменованого каналу)

#stat -c%F example.file
regular file (тобто звичайний файл)

%G – чинний (іменний) ідентифікатор групи власника файлу (наприклад, назва групи)

#stat -c%G example.file
max

%g – числовий ідентифікатор групи (GID) власника файлу

#stat -c%g example.file
500

%h – число жорстких посилань

#stat -c%h example.file
1

%i – номер індексного дескриптора

#stat -c%i example.file
786506

%N – файл, а для символічних посилань − ім'я посилання і оригінального файлу з повним шляхом його розташування

#stat -c%N link.example.file
`link.example.file' -> `/home/max/example.file'

%n – ім'я файлу

```
#stat -c%n example.file
example.file
```

%s – загальний розмір файлу в байтах

```
#stat -c%s example.file
5614
```

%U – іменний ідентифікатор власника файлу

```
#stat -c%U example.file
max
```

%u – числовий ідентифікатор власника файлу (UID)

```
#stat -c%u example.file
500
```

%X – час останнього доступу до файлу в секундах з початку комп'ютерної ери ("ключового моменту"), тобто моменту, з якого відраховує час Unix 00:00:00 1 січня 1970

```
#stat -c%X example.file
1114546386
```

%x – час останнього доступу до файлу в звичному (дата-час) вигляді

```
#stat -c%x example.file
2018-04-26 20:13:06.000000000 +0000
```

%y – час останньої модифікації файлу в звичному (дата-час) вигляді

```
#stat -c%y example.file
2018-04-26 20:17:39.000000000 +0000
```

%z – час останніх змін файлу в звичному (дата-час) вигляді

```
#stat -c%z example.file
2018-04-26 20:18:09.000000000 +0000
```

У якості формату виведення FORMAT для файлових систем (для опції -f, --filesystem) припустимі наступні прапорці та їх комбінації послідовностей:

%a – число блоків доступних не тільки суперкористувачеві (*root*)

```
#stat -f -c%a example.file
3646377
```

%b – загальна кількість блоків в файлової системі

```
#stat -f -c%b example.file
4125127
```

%c – загальна кількість файлових дескрипторів в файлової системі

```
#stat -f -c%c example.file
2097152
```

%n – ім'я файлу

```
#stat -f -c%n example.file
example.file
```

trap

Команда *trap* застосовується для завдання дій при отриманні сигналів. Звичайна дія − видалити сценарій, коли він переривається. Історично командні оболонки завжди використовували числа для позначення сигналів, але в сучасних сценаріях слід застосовувати імена, які беруться з файлу *signal.h* директиви #include з опущеним префіксом *sig*. Для того щоб подивитися номери сигналів і відповідні їм імена, можна ввести в командному рядку команду *trap* -l. Стандартно сигнали викликають припинення виконання програми.

За допомогою команди *trap* передається відповідна дія, за якою слідує ім'я (імена) сигналу для перехоплення:

trap команда сигнал

Нагадуємо, що зазвичай сценарії обробляються інтерпретатором зверху вниз, тому треба поставити команду *trap* перед тією частиною сценарію, яку необхідно захистити.

Для повернення до стандартної реакції на сигнал треба задати замість параметра ‘команда’ − ‘-‘. Для ігнорування сигналу задайте в ‘команда’

порожній рядок ' '. Команда *trap* без параметрів виводить поточний список перехоплень і дій.

У табл. 2.10 перераховані найважливіші сигнали, які можна відстежити (зі стандартними номерами в дужках).

Таблиця 2.10

Найважливіші сигнали

Сигнал	**Опис**
HUP(1)	Несподівана зупинка. Зазвичай посилається, коли вимикається термінал, або користувач виходить із системи
INT(2)	Переривання. Зазвичай посилається натисканням комбінації клавіш <Ctrl>+<C>
QUIT(3)	Завершення виконання. Зазвичай посилається натисканням комбінації клавіш <Ctrl>+<\>
ABRT(6)	Аварійне завершення. Зазвичай посилається при виникненні серйозної помилки виконання
ALRM(14)	Аварійний сигнал. Зазвичай посилається для обробки перевищень ліміту часу
Term(15)	Завершення. Зазвичай посилається системою, коли вона завершує роботу

Приклад. ***Сигнали переривань***

У наступному сценарії показана проста обробка сигналу.

```
#!/bin/bash
trap 'rm -f /tmp/my_tmp_file_$$' INT
echo creating file /tmp/my_tmp_file_$$
date > /tmp/my_tmp_file_$$
echo "press interrupt (CTRL-C) to interrupt..."
while [ -f /tmp/my_tmp_file_$$ ] ; do
echo File exists
sleep 1
done
```

```
echo The file no longer exists
trap INT
echo creating file /tmp/my_tmp_file_$$
date > /tmp/my_tmp_fi1e_$$
echo "press interrupt (CTRL-C) to interrupt..."
while [ -f /tmp/my_tmp_file_$$ ] ; do
echo File exists
sleep 1
done
echo we never get here
exit 0
```

При виконанні сценарію, натискаючи та утримуючи клавішу <Ctrl> і потім за допомогою клавіші <C> (або будь-яку іншу комбінацію клавіш для переривання процесу) в кожному з циклів, то отримаєте наступне виведення:

creating file /tmp/my_tmp_file_141

press interrupt (CTRL+C) to interrupt ...

File exists

File exists

File exists

File exists

The file no longer exists

creating file /tmp/my_tmp_file_141

press interrupt (CTRL+C) to interrupt ...

File exists

File exists

File exists

File exists

Сценарій використовує команду *trap* для організації виконання команди *rm -f /tmp /my_tmp_file_$$* при виникненні сигналу *INT* (переривання). Потім сценарій виконує цикл *while* до тих пір, поки існує файл. Коли користувач натискає комбінацію клавіш <Ctrl> + <C>, виконується команда *rm -f / tmp / my_tmp_ file_$$*, а потім поновлюється

виконання циклу *while*. Оскільки тепер файл видалений, перший цикл *while* завершується стандартним чином.

Далі сценарій знову застосовує команду *trap*, на цей раз для того, щоб повідомити, що при виникненні сигналу *INT* ніяка команда не виконується. Потім сценарій створює заново файл і виконує другий цикл *while*. Коли користувач знову натискає комбінацію клавіш <Ctrl> + <C>, не задана команда для виконання, тому реалізується стандартна поведінка: негайне припинення виконання сценарію. Оскільки сценарій завершується негайно, заключні команди *echo* і *exit* ніколи не виконуються.

find

Команда *find* застосовується для пошуку файлів.

Перш ніж розглядати та вивчати опції, критерії і аргументи команди, наведемо простий приклад пошуку файлу *test*. Виконайте наступну команду під ім'ям суперкористувача *root*, щоб мати достатньо прав доступу для обстеження всього комп'ютера.

```
# find / -name test -print
/usr/bin/test
#
```

Можна знайти і інші файли, також з назвою *test*. Тобто команда звучить так: "шукати, починаючи з каталогу /, файл з ім'ям *test* і потім вивести на екран ім'я файлу".

Виконання команди займає деякий час, вона буде шукати на комп'ютері і в мережі. У цьому випадку на допомогу приходить перша опція. Якщо вказати опцію *-mount*, то зможемо повідомити команді *find* про те, що змонтовані каталоги перевіряти не потрібно.

```
#find / -mount -name test -print
/usr/bin/test
#
```

Знаходимо той же файл, але набагато швидше і без пошуку в змонтованих файлових системах.

Синтаксично запис команди *find* виглядає наступним чином:

find [шлях] [опції] [критерії] [дії]

Частина записи *[шлях]* зрозуміла і проста: можна вказати абсолютний шлях пошуку, наприклад, */bin*, або відносний, наприклад .. При необхідності можна задати кілька шляхів − наприклад, *find / var / home*.

У табл. 2.11 перераховані основні опції команди *find*.

Таблиця 2.11

Основні опції команди *find*

Опція	**Опис**
-depth	пошук в підкаталогах перед пошуком у самому каталозі
-follow	прямувати за символьними посиланнями
-maxdepth n	при пошуку перевіряти не більше N вкладених рівнів каталогу
-mount(або –xdev)	не шукати в каталогах інших файлових систем

У команді *find* можна задати велику кількість критеріїв, і кожен з них повертає або true, або false. В процесі роботи команда *find* розглядає по черзі кожен файл і застосовує до нього усі критерії в порядку їх визначення. Якщо черговий критерій повертає значення false, команда *find* припиняє аналіз поточного файлу і переходить до наступного; якщо критерій повертає значення true, команда застосовує наступний критерій до поточного файлу або здійснює задану дію над ним.

grep

Команда *grep* означає загальний синтаксичний аналізатор регулярних виразів (general regular expression parser). Тоді як, команда *find* застосовується для пошуку файлів у системі, команда − *grep* для пошуку рядків у файлах.

Дійсно, дуже часто при використанні команди *find* команда *grep* передається після аргументу *-exec*.

Команда *grep* приймає опції, шаблон відповідності та файли для пошуку:

grep [опції] шаблон [файли]

Якщо імена файлів не задані, команда аналізує стандартне введення.

Розглянемо основні опції команди *grep* (табл. 2.12).

Таблиця 2.12

Основні опції команди *grep*

Опція	**Опис**
-c	замість виведення на екран рядків, які співпали з шаблоном, виводить їх кількість
-E	вмикає розширені регулярні вирази
-h	стискає звичайний початок кожного рядка виведення за рахунок видалення імені файла, у якому рядок було знайдено
-i	не враховувати регістр букв
-l	Перелічує імена файлів з рядками, які співпали з шаблоном, не виводить знайдені рядки
-v	змінює шаблон відповідності для вибору замість рядків, які відповідають шаблону, рядків, які не співпадають з ним

Приклад. Розглянемо команду *grep* на прикладах простих шаблонів.

#grep in words.txt

When shall we three meet again. In thunder, lightning, or in rain?

I come, Graymalkin!

#grep -c in words.txt words2.txt

words.txt:2

words2.txt:14

#grep -c -v in words. txt words2. txt

words.txt:9

words2.txt:16

Перший приклад без опцій; в ньому просто шукається рядок ‘*in’* в файлі *words.txt,* і виводяться на екран будь-які рядки, котрі відповідають умові пошуку. Файл не відображається, оскільки пошук проводиться у єдиному файлі.

У другому прикладі в двох різних файлах підраховується кількість рядків, які відповідають шаблону. В цьому випадку імена файлів виводяться на екран.

На завершення застосовується опція *-v* для інвертування критерію пошуку і підрахунку рядків, котрі не збігаються з шаблоном.

Регулярні вирази дозволять виконувати більш складний пошук. Регулярні вирази застосовуються в системі Linux і багатьох мовах програмування з відкритим вихідним кодом. Можна використовувати їх в редакторі *vi* і в скриптах, застосовуючи одні й ті ж принципи, загальні для регулярних виразів.

При обробці регулярних виразів певні символи інтерпретуються особливим чином. У табл. 2.13 наведені найбільш часто використовувані в регулярних виразах символи.

Таблиця 2.13

Символи, які частіше використовуються у регулярних виразах

Символ	**Опис**
^	прив’язка до початку рядка
$	прив’язка до кінця рядка
.	будь-який поодинокий символ
[]	у квадратних дужках міститься діапазон символів, з будь-яким з них можливо співпадіння, наприклад, діапазон символів *a-e* або інвертований діапазон, перед яким стоїть відкритий символ ^

Все це виглядає трохи заплутано, але якщо засвоювати усі можливості поступово, то все не так складно. Розглянемо приклади.

1. Почнемо з пошуку рядків, котрі закінчуються буквою "e". Можна здогадатися, що потрібно використовувати спеціальний символ $:

```
#grep e$ words2. txt
```

Art thou not, fatal vision, sensible

I see thee yet, in form as palpable

Nature seems dead, and wicked dreams abuse

#

знайдені рядки, закінчуються буквою "e".

2. Проведемо пошук трилітерних слів, які починаються з символів *"Th"*. В даному випадку знадобиться шаблон *[[: space:]]* для обмеження довжини слова і для єдиного додаткового символу.

```
#grep Th.[[:space:]] words2.txt
```

The handle toward my hand? Come, let me clutch thee.

The curtain'd sleep; witchcraft celebrates

Thy very stones prate of my whereabout,

#

3. На закінчення застосуємо розширений режим пошуку в команді *grep* для виявлення слів, які починаються з малих літер, довжиною 10 символів. Для цього задамо діапазон співпадаючих символів від a до z і 10 повторюваних збігів.

```
# grep -E [a-z]\{10\} words2.txt
```

Proceeding from the heat-oppressed brain?

And such an instrument I was to use.

The curtain'd sleep; witchcraft celebrates

Thy very stones prate of my whereabout,

Наведені приклади лише торкнулися найбільш важливих компонентів регулярних виразів, для більш докладної інформації треба звертатися до додаткової документації, але кращий спосіб - експериментувати з ними.

Виконання команд

Часто потрібно перехоплювати результат виконання команди для використання її потім в сценарії командної оболонки; тобто необхідно виконати команду і помістити її вивід в змінну. Зробити це можна за допомогою синтаксичної конструкції *$(команда).*

Результат виконання конструкції *$(команда)* − просто виведення команди. Маємо на увазі, що це не статус повернення команди, а просто строкове виведення, яке демонструється далі.

```
#!/bin/sh
echo The current directory is $PWD
echo The current users are $(who)
exit 0
```

Оскільки поточний каталог − це змінна оточення командної оболонки, перший рядок не потребує застосування підстановки команди. Результат виконання програми *who*, навпаки, потребує підстановки команди, якщо він повинен стати змінною в сценарії.

У випадку розміщення результату в змінну можна просто привласнити його таким чином:

```
whoisthere=$(who)
echo $whoisthere
```

Можливість помістити результат виконання команди в змінну сценарію − дуже потужний засіб, оскільки він полегшує використання існуючих команд в сценаріях і перехоплення результату їх виконання.

Підстановки в арифметичних виразах

Команда *expr* дозволяє виконувати прості арифметичні операції, але робить це дуже повільно, тому що для її виконання запускається нова командна оболонка.

Сучасна і найкраща альтернатива − синтаксична конструкція *$((...)).* Вираз, який треба обчислити, поміщаємо в цю конструкцію та виконуємо прості арифметичні операції набагато ефективніше.

```
#!/bin/sh
x=0
```

```
while [ "$x" -ne 10 ]; do
echo $x
x=$(($x+1))
done
exit 0
```

Зверніть увагу на відмінність наведеної підстановки від команди *x=$(...)*. Подвійні дужки застосовуються для підстановки значень в арифметичні вирази. Варіант з поодинокими дужками, показаний раніше, використовується для виконання команд і перехоплення їх виведення.

Підстановка значень параметрів

Розглядався найпростіший варіант присвоювання параметра і підстановки значення параметра:

foo = fred

echo $foo

Проблема виникає при розміщені додаткових символів у кінець значення змінної. Припустимо, треба написати короткий сценарій обробки файлів *l_tmp* і *2_tmp*. Можна написати:

```
#!/bin/sh
for i in 1 2
do
   my_secret_process $i_tmp
done
```

Але при кожному проході циклу отримаємо наступне повідомлення:

my_secret_process: too few arguments

Проблема полягає в тому, що командна оболонка спробувала підставити значення змінної *$i_tmp*, яка не існує. Оболонка не вважає це помилкою; вона просто не робить ніякої підстановки, тому в сценарій *my_secret_process* не передаються ніякі параметри. Для забезпечення підстановки в змінну частини її значення *$i* необхідно *i* укласти в фігурні дужки наступним чином:

```
#!/bin/sh
for i in 1 2
do
my_secret_process ${i}_tmp
```

done

При кожному проході циклу замість *${i}* підставляється значення *i* та отримуються реальні імена файлів, виконується підстановка значення параметра в рядок.

У командній оболонці можна виконувати різноманітні види підстановок. Часто вони допомагають знайти гарне рішення задач, котрі вимагають обробки багатьох параметрів. Найпоширеніші види підстановок значень параметрів наведені в табл. 2.14.

Таблиця 2.14

Шаблони

Шаблон підстановки параметру	**Опис**
$(парам: - значення за стандартом)	якщо у *парам* немає значення, йому присвоюється значення за стандартом
$(#парам)	задається довжина *парам*
$(парам%рядок)	з кінця значення *парам* відкидається найменша порція, що співпадає з *рядком*, та повертається інша частина значення
$(парам%% рядок)	з кінця значення *парам* відкидається найбільша порція, що співпадає з *рядком*, та повертається інша частина значення
$(парам# рядок)	з початку значення *парам* відкидається найменша порція, що співпадає з *рядком*, та повертається інша частина значення
$(парам## рядок)	з початку значення *парам* відкидається найбільша порція, що співпадає з *рядком*, та повертається інша частина значення

Ці підстановки дуже корисні при роботі з рядками. Останні чотири варіанти, які видаляють частини рядків, особливо знадобляться при обробці імен файлів і шляхів до них, як показано у прикладі.

Приклад. Обробка параметрів

У сценарії показано застосування шаблонів при підстановках значень параметрів.

```
#!/bin/bash
unset foo
echo ${foo:-bar}
foo=fud
echo ${foo:-bar}
foo=/usr/bin/Xll/startx
echo ${foo#*/}
echo ${foo##*/}
bar=/usr/local/etc/local/networks
echo ${bar%local*}
echo ${bar%%local*}
exit 0
```

У цього сценарію такий вивід:

bar

fud

usr/bin/Xll/startx

startx

/usr/local/etc

/usr

Перша підстановка *${foo: -bar}* дає значення *bar*, оскільки у *foo* немає значення в момент виконання команди. Змінна *foo* залишається незмінною, тобто вона залишається незаданою.

Підстановка *${foo: = bar}* встановила б значення змінної *$foo*. Цей строковий шаблон встановлює, що змінна *foo* існує і не дорівнює null. Якщо значення змінної не дорівнює null, оператор повертає її значення, в іншому випадку замість цієї змінної *foo* присвоюється значення *bar*.

Підстановка *${foo:? bar}* виведе на екран *foo: bar* і аварійно завершить команду, якщо змінної *foo* не існує або її значення не визначено. І нарешті, *${foo: + bar}* поверне *bar*, якщо *foo* існує і не дорівнює null.

Шаблон *{foo #*/}* задає пошук і видалення тільки лівого символу / (символ * відповідає будь-якому рядку, у тому числі і порожньому). Шаблон *{foo ##*/}* задає пошук максимального підрядка, який збігається з ним, і, таким чином, видаляє найправіший символ / і всі попередні для нього символи.

Шаблон *${bar%local*}* визначає перегляд символів в значенні параметра, починаючи від крайнього правого, до першої появи підрядка *local*, за яким слідує будь-яка кількість символів, а в разі шаблону *${bar%%local*}* шукається максимально можлива кількість символів, починаючи від крайнього правого символу значення і закінчуючи крайньою лівою появою підрядка *local*.

Оскільки в системах Linux багато чого засноване на ідеї фільтрів, результат будь-якої операції часто повинен перенаправлятися вручну. Припустимо, треба перетворити файли GIF в файли JPEG за допомогою програми *cjpeg*:

```
#cjpeg image.gif> image.jpeg
```

Може виникнути потреба виконати цю операцію над великою кількістю файлів. Можна це автоматизувати:

```
#!/bin/bash
for image in *.gif
do
cjpeg $image > ${image%%gif}jpeg
done
```

Цей сценарій створює в поточному каталозі для кожного файлу формату GIF файл формату JPEG.

2.7 Налагодження сценаріїв

Зазвичай налагоджувати сценарії командної оболонки досить легко. Розробник запускає сценарії на виконання, переглядає результати його роботи і на основі їх аналізу приймає рішення щодо роботи сценарію. Якщо помилок відразу не видно, можна додати декілька додаткових команд *echo* , які відображатимуть вміст змінних, і протестувати фрагменти програмного коду, просто вводячи їх в командній оболонці в інтерактивному режимі.

Оскільки сценарії обробляються інтерпретатором, немає витрат на компіляцію при коригуванні і повторному виконанні сценарію.

Однак в ряді випадків, наприклад для великих сценаріїв, або таких, які змінюють конфігурацію системи, спроба знайти джерело проблеми стає не таким простим. Налагодження виходить більш ефективним при використанні спеціальних засобів. Оболонка забезпечує кілька вбудованих команд для вирішення різних режимів підтримки налагодження. Можна виділити два режими налагодження:

• перевірка синтаксису;

• трасування оболонки.

Для проведення налагодження сценарій повинен бути запущений спеціальним чином. Для цього перший рядок сценарію повинен мати вигляд: #!/bin/bash опція

Для налагодження не всього сценарію, а тільки деякого фрагмента, цей фрагмент позначається двома викликами команди *set* із зазначенням потрібних опції з таблиці. 2.15. При цьому для включення режиму налагодження перед буквою опції вказується знак «-», а для виключення знак «+».

Основний спосіб відстеження помилок, які важко виявляються, − завдання різних опцій командної оболонки. Для цього можна застосовувати опції командного рядка після запуску командної оболонки або використовувати команду *set*. У табл. 2.15 перераховані ці опції.

Таблиця 2.15

Опцій командної оболонки

Опції командного рядка	**Опції команди *set***	**Опис**
-n <сценарій >	set –o noexec set –n	тільки перевіряє синтаксичні помилки, не виконує команди
-v <сценарій >	set –o verbose set –v	виводить на екран команди перед їх виконанням
-x <сценарій >	set –o xtrace set –x	виводить на екран команди після обробки командного рядка
-u <сценарій >	set –o nounset set –u	видає повідомлення щодо помилки при використанні невизначеної змінної

```
#!/bin/sh
#приклад налагодження сценарія
set –x
if [ –z "$1" ] ; then
echo "Помилка: мало аргументів."
exit 1
fi
set +x
```

Загальна методика налагодження сценаріїв

Загальна методика налагодження сценарію полягає в тому, що перш, ніж запустити його на виконання, необхідно перевірити його синтаксис за допомогою опції *-n*. А для більшої деталізації рекомендується використати сукупність ключів *-nv*. На наступному етапі, після усунення синтаксичних помилок, проводиться налагодження з трасуванням за допомогою опції *–x*.

Можна задати параметри за допомогою прапорців *-o* і скинути їх за допомогою прапорців *+o* подібним же чином в скорочених версіях. Отримати просте відстеження виконання можна, використовуючи опцію

xtrace. Спочатку можна застосувати опцію командного рядка, але для більш ретельного налагодження слід помістити опції *xtrace* (задаючи виконання і скидання відстеження виконання) всередину сценарію, в той фрагмент коду, який створює проблему. Відстеження виконання змушує командну оболонку перед виконанням кожного рядка сценарію виводити на екран цей рядок і підставляти в нього значення використовуваних змінних. Для установки опції *xtrace* використовується наступна команда:

```
set -o xtrace
```

Для того щоб відключити цю опцію, застосовується команда:

```
set +o xtrace
```

Рівень виконуваних підстановок задається (за замовчуванням) кількістю знаків + на початку кожного рядка. Можна замінити знак + на щось більш осмислене, визначивши змінну командної оболонки *PS4* в файлі конфігурації оболонки.

РОЗДІЛ 3

Лабораторний практикум

Початок роботи в ОС LINUX.

Робота в операційній системі LINUX виконується на віддаленому робочому столі. Після підключення треба перевірити IP-адресу 192.168.14.56. У вікні Параметри встановити Екран →24 біти. Після цього виконується підключення до LINUX MATE. Система запропонує ввести:

ім'я користувача *login*:

та пароль *password*:

Користувачі : ***student1 ÷ student35***

Паролі: ***st01) ÷ st35)***

Лабораторна робота 1

Основи роботи з ОС LINUX. Створення файлів

Мета роботи: набути практичних навичок роботи в операційній системі LINUX, ознайомитися з основними командами для роботи з файлами та каталогами.

Для початку роботи в операційній системі LINUX треба ввести ім'я користувача та пароль. За іменем система пізнає студента як одного з користувачів, які можуть працювати в системі. Для кожного користувача визначається каталог за замовчуванням, який називається робочим або домашнім каталогом (*home directory*). Користувачі мають доступ до обмеженої кількості каталогів і команд, окрім користувача *root*.

За замовчуванням у сучасних дистрибутивах при вході до системи запускається графічний менеджер реєстрації. Для переходу із графічного до консольного режиму треба перейти у меню **Terminal** або натиснути клавіатурну комбінацію Ctrl+Alt+F_n , де n – номер консолі від 1 до 6. Дуже важливо навчитися працювати в командному рядку.

Операційна система Linux чутлива до регістру, тому важливо контролювати правильність регістру. Як правило, команда виконується після натискання клавіші <Enter>. Для перемикання між консолями натискаємо комбінації клавіш відкриття терміналу Alt+F1, …, Alt+F6.

При використанні складних команд застосовують такі засоби:
> − запис до файлу.

Наприклад, існує файл *gem* з деяким вмістом, набираємо

#date>gem

вміст файлу стирається і в нього записується поточна дата.
>> − дозапис в кінець існуючого файлу.

Наприклад, існує файл *gem* з деяким вмістом, набираємо

#date>>gem

вміст файлу залишається + в кінець файлу *gem* додається рядок з поточною датою.

| − програмний канал - результат однієї команди передається на вхід іншій команді.

& − процес виконується у фоновому режимі, не чекаючи закінчення попередніх процесів.

***** − будь-яка кількість будь-яких символів.

; − кілька команд в одному рядку, команди виконуються одна за одною.

&& − при об'єднанні команд: подальша команда виконується тільки при нормальному завершенні попередньої.

|| − при об'єднанні команд: подальша команда виконується тільки, якщо не виконалася попередня команда.

( ) − групування команд в дужки в арифметичних виразах.

{ } − групування команд з об'єднаним виводом.

[ ] − вказання діапазону даних або перерахування даних (без коми).

У ОС Linux є засіб повторного звернення до вже виконаних команд, який не переривається навіть при вимиканні комп'ютера. Попередня команда викликається після натиснення клавіші <Up>, а для її виконання треба натиснути <Enter>. Для виводу списку вже набраних команд використовують команду: *history*:

```
#history
#1 clear
#2 adduser
#3 history
```

Щоб виконати команду з хронологічного списку, викликайте за допомогою клавіші <Up> попередню команду доти, поки у командному рядку не з'явиться потрібна, або ж натисніть <!> і введіть номер потрібної команди. Наприклад, щоб повторно виконати команду *adduser* з представленого вище списку, введіть:

```
#!2
```

Максимальна кількість команд в хронологічному списку задається в користувальницькому конфігураційному файлі **.***profile***.**

Основні команди ОС Linux

Для отримання довідки щодо будь-якої команди Linux використовується команда *man*. Якщо не пам'ятаєте точно імені потрібної команди, введіть команду *man* з параметром **k**, потім ключове слово для пошуку потрібної команди. Наприклад, якщо ввести команду *man ls*, Linux виведе на екран довідку щодо команди **ls** з усіма її параметрами. Якщо ввести *man -k cls* виводиться список усіх команд, у яких є сполучення **cls.**

Команди для роботи з каталогами

Для ОС Linux існує поняття робочого каталогу користувача. Робочий каталог зазвичай позначається символом **~ .**

Для переміщення по дереву каталогів Linux застосовується команда *cd***.**

Синтаксис команди:

#cd new-directory

де *new-directory* — новий каталог, в який необхідно перейти. Крім того, в Linux поточний каталог позначається однією крапкою (.), каталог-батько — двома(..).

***ls** (list)* — на екран виводиться інформація щодо файлів і каталогів.

Синтаксис команди:

#ls [прапорці] [ім'я]

Приклад: #ls –la

Команда *ls* для кожного імені каталогу роздруковує список файлів, які входять до цього каталогу; для файлів - повторюється ім'я файлу і виводиться додаткова інформація відповідно до наведених прапорців:

-l – виведення в довгому форматі: перед іменами файлів видається режим доступу, кількість посилань на файл, імена власника і групи, розмір в байтах і час останньої модифікації;

-R – рекурсивно обійти підкаталоги;

-a – вивести список всіх файлів (включаючи приховані файли);

-F – якщо файл є каталогом, то видавати після імені символ /; якщо файл є виконуваним, то видавати після імені символ ***.**

pwd – виводиться ім'я поточного каталогу.

Синтаксис команди:

#pwd

mkdir – створення нового каталогу**.**

Синтаксис команди:

#mkdir имя нового каталогу

rmdir – видаляє каталог. Ця команда видаляє тільки пустий каталог.

Синтаксис команди:

#rmdir имя каталогу, який треба видалити

Команди для роботи з файлами в ОС Linux

В операційній системі Linux існує три засоби створення файлів:

touch — команда призначена для установки часу останньої зміни файла або доступу в поточний час. Також використовується для створення пустих файлів.

Створимо файл *myfile.txt* і встановимо час останньої зміни і доступу в поточний час у системі; якщо файл існує — оновлює час останньої зміни та доступу не змінюючи при цьому вмісту файлу:

```
#touch myfile.txt
```

Встановлює час останньої зміни і доступу в 10:26:38 22 січня 2016 р.:

```
# touch -t 201601220846.26 index.html
# touch -d '2006-01-22 10:26:38' index.html
# touch -d 'Jan 22 2016 10:26:38' index.html
```

cat — перегляд текстового файлу. Можна виводити послідовно вказані файли (або пристрої), об'єднуючи їх в єдиний потік. Якщо замість імені файлу вказується «-», то читається стандартний ввід.

```
#cat a.txt - b.txt > abc.txt
#cat /dev/null > file_to_clear.txt
```

echo - виводить текст або значення змінних на стандартний вивід (за замовчуванням — монітор):

```
#echo "Вивід текста на екран"
```

Якщо використовувати перенаправлення виводу, застосувавши символ >, то набраний текст буде записаний в новий файл:

```
#echo > test "Текст записується в файл test"
```

У випадку, коли файл з таким ім'ям вже існує, він перезапишеться, а вся стара інформація видаляється. Для додавання інформації у кінець файлу можна використовувати символ >>:

```
#echo >> test "Цей текст додається в файл test"
```

Виведення тексту можна перенаправити для друку на принтері, використовуючи символ ‘|’:

#echo "Цей текс відправляється на принтер" | lp

cp (copy) — копіювання одного або декількох файлів з одного каталогу до іншого.

Синтаксис команди:

#cp [первинний файл] [файл, в який копіюємо],

Для виконання команди треба вказати обидва параметри. Для копіювання файлу с тим же ім’ям в якості другого параметру ставиться крапка.

mv (move) — файли переміщуються з одного каталогу в інший.

Дія цієї команди аналогічна дії команди копіювання з подальшим видаленням вихідних файлів. Команда *mv* не створює копій файлів. Синтаксис команди *mv*:

#mv [початковий файл] [новий файл]**,**

rm – видалення файлів.

***Видалений файл відновити неможливо*.**

Для безпечної роботи слід користуватися наступним форматом цієї команди:

#rm -i filename,

де *filename* — ім'я файлу, що видаляється; *-i* — параметр, який вказує на необхідність підтвердити видалення файла.

Наприклад, по команді *rm File25* файл *File25* буде просто знищено, по команді ж *rm -i File25* він буде видалений тільки після підтвердження користувачем необхідності видалення.

Синтаксис команди:

#rm [-i] файл

rm [-f] [-i] файл ...

rm -r [-f] [-i] каталог ... [файл ...]

Допускаються наступні три опції:

-f команда не видає повідомлень, коли файл, що видаляється, не існує, не запитує підтвердження при видаленні файлів, запис в які немає прав. Якщо немає права і на запис у каталог, файли не видаляються;

-r відбувається рекурсивне видалення усіх каталогів і підкаталогів, перелічених у списку аргументів;

-i перед видаленням кожного файлу запитується підтвердження. Опція *-i* усуває дію опції *-f*.

more - перегляд вмісту текстового файлу.

Наприклад, для виведення на екран вмісту конфігураційного файлу *emacs* вводиться команда:

```
#more .emacs
```

less – команда перегляду файлу з можливістю скролінга (посторінковий перегляд), удосконалена команда *more*.

Синтаксис команди *less*:

```
#less файл
```

find – утиліта пошуку файлів:

```
#find шлях –опции
```

де *шлях* — це каталог, в якому потрібно здійснити пошук. В якості шляху можна вказувати наступні значення:

. — пошук в поточному каталозі;

/ — пошук в кореневому каталогу;

~ — пошук в домашньому каталозі.

Використання команди *find* включає три етапи, які в свою чергу можуть складатися із одного або декількох етапів: де шукати, що шукати, що робити, коли файл знайдено.

Якщо відомо ім’я файлу, але не відомо, де він знаходиться в структурі каталогів Linux, то самим простим способом використання команди *find* для пошуку буде команда:

#find / -name *filename* -print

Треба бути уважним при пошуку від кореня − у великих системах такий пошук може тривати багато часу.

Можливо, більш прийнятним буде пошук в декількох каталогах. Наприклад, якщо відомо, що файл, імовірно, знаходиться в каталогах /*usr* або /*usr2*, використовуйте наступну команду:

#find /usr /usr2 -name *filename* -print

В команді *find* можна використовувати множину різних параметрів. Список параметрів команди наведено в таблиці 3.1.

Таблиця 3.1

Параметри команди *find*

Команда	**Опис**
-name *file*	Параметр *file* може бути ім'ям або шаблоном, що містить символи підстановки. Якщо це шаблон, то для обробки вибирається кожен файл, чиє ім'я задовольняє цьому шаблону.
-links *n*	Для обробки вибираються всі файли, на кожен з яких є *n* або більше посилань.
-size n [c]	Для обробки вибираються всі файли, розмір яких дорівнює або більше *n* 512-байтних блоків. Якщо до розміру доданий символ *c*, то вибираються файли, які складаються з *n* або більше символів.
-atime *n*	Для обробки вибираються всі файли, до яких Ви отримували доступ за останні n-днів. Зверніть увагу, що сама команда *find* здійснює доступ до файлів, тому змінює час останнього доступу до файлу
~print	Ця найбільш часто використовувана команда просто відображає імена всіх знайдених файлів

free – інформація щодо використання оперативної та віртуальної пам’яті

#free [-b | -k | -m] [-o] [-s delay] [-t] [-V]

Використовується для перевірки зайнятого і вільного місця фізичної пам'яті та підкачування пам'яті.

Опції:

-b показує кількість пам'яті в байтах;

-k (за замовчуванням) показує кількість пам'яті в кілобайтах;

-m показує кількість пам'яті в мегабайтах.

-t показує рядки, які містять повну кількість пам'яті.

-o забороняє показувати рядки, які відносяться до «масиву буфера».

-s дозволяє безупинно виводити інформацію з проміжком в *delay* секунд.

df — визначає кількість вільного місця на пристрої.

#df [-t тип] [шлях]

Якщо ім'я пристрою не вказане, то виводиться інформація щодо вільного місця на кожній змонтованій файловій системі. Наприклад, якщо створено один загальний розділ для усієї ОС, то команда виведе наступну інформацію:

#df

Filesystem 1024-blocks Used Available Capacity Mounted /dev/hda5 225 420 203 346 10 434 152 /

Вільне місце підраховується блоками по 1024 байт, але можна вказати і 512.

У наведеному прикладі розділ /*dev/hda5* займає 225 Мбайт. На ньому зайнято 203 Мбайт, а вільного місця залишилося всього 10 Мбайт. Якщо скласти два останніх числа, то виявиться, що сума менша, ніж розмір розділу. Причина такого розходження у тому, що файлова система резервує місце для системного адміністратора, який може зберігати додаткову інформацію навіть у випадку, коли інші користувачі будуть отримувати повідомлення про відсутність вільного місця.

du – підраховує зайняте файлами місце:

#du [параметри] [файли] [каталоги]

Наприклад, для підрахунку обсягу конкретного каталогу можна використовувати наступний варіант команди *du*:

```
#du -s /usr 169 209K /usr
```

Завдання до лабораторної роботи:

1. Вивести вміст домашнього (робочого) каталогу.
2. Створити в поточному каталозі каталог з іменем *name*, де *name* - ваше прізвище маленькими латинськими літерами.
3. Перейти в створений каталог.
4. Створити у ньому новий каталог *MLKX*.
5. В каталозі *MLKX* створити три текстових файла різними способами.
6. Вивести зміст файлів на екран.
7. Створити у домашньому каталозі каталог з ім'ям *FFW*.
8. Скопіювати один зі своїх файлів у каталог *FFW* з ім'ям *xyz1.text*.
9. Перейти в каталог *FFW* і переглянути вміст файлу *xyz1.text*.
10. Перейменувати файл *xyz1.text* на файл *xyz2.text*.
11. Об'єднати файли *xyz.text* і *xyz2.text* та записати його у файл з іменем *xyz3.text* в каталозі *FFW*.
12. Переглянути файл *xyz3.text*.
13. Записати в нього поточний час і дату, переглянути вміст на екрані.
14. Провести пошук файлу *xyz.text* за допомогою команди *find*.
15. Видалити файли *xyz.text, xyz2.text, xyz3.text*.
16. Видалити каталог *FFW*.
17. Ввести команду *whoami*.
18. Вивести кількість вільної пам'яті.
19. Дізнатися скільки місця на диску займає ваш каталог.

Контрольні запитання.

1. Якими командами можна створити файл.
2. Які команди використовуються для копіювання та переміщення файлу.
3. Якою командою можна виділити файл, каталог.

Лабораторна робота 2

Створення посилань

Мета роботи: набути практичних навичок роботи в операційній системі LINUX, ознайомитися з сутністю індексного дескриптора та посиланнями.

Файл − це іменований блок інформації, який зберігається на диску і має такі ознаки: фіксоване ім’я (назва файлу) та певне логічне уявлення. Проте в операційних системах на базі ядра Linux вся інформація щодо файлу прив'язана не до імені, а до **індексного дескриптора**.

Оскільки індексні дескриптори є номерами, а файлів в операційній системі зазвичай дуже багато, то шукати файл по номеру його дескриптора незручно. Тому будь-якому файлу в системі присвоюється осмислене ім'я (словесне), яке не містить інформації про файл, а лише вказує (посилається) на його дескриптор.

Ім'я файлу, котре посилається на його індексний дескриптор, називається **жорстким посиланням**. Механізм жорстких посилань − це основний спосіб звертання до файлів за ім’ям в операційних системах, які ґрунтуються на ядрі Linux.

Видалення одного жорсткого посилання на файл не призводить до його видалення із системи, якщо існують інші жорсткі посилання. Усі жорсткі посилання рівноправні між собою, незалежно від часу створення, місцезнаходження в структурі каталогів та інше. Файл буде доступний системі, поки існує хоч одне жорстке посилання на нього. В разі видалення усіх посилань на файл, він видаляється з системи, оскільки стає недоступним їй.

Жорсткі посилання можна створювати тільки на файли, але не на каталоги. Також жорстке посилання не можна створити з одного диска на інший. Це означає, що не можна створити жорстке посилання на файл,

котрий знаходиться, наприклад, на зміному носії (флеш-пам'ять, CD-R та ін.) або іншому розділі жорсткого диска.

Для створення посилання на каталог використовують **м'які посилання.** Часто їх також називають **символьними посиланнями**. Вони являють собою файли, які вказують не на індексні дескриптори, а на імена файлів.

Індексний дескриптор файлу завжди один, а імен може бути декілька. Також може існувати необмежена кількість символьних посилань на кожне ім'я файлу. При видаленні жорсткого посилання, на яке було м'яке посилання, останне не успадковує зв'язок з дескриптором і втрачає свою "працездатність". Такі посилання часто називають "битими".

При копіюванні файла створюється новий файл, дані якого записуються на вільне місце на диску, і який має власний індексний дескриптор. У разі ж створення жорсткого посилання файл залишається в однині, з'являється лише додатковий покажчик на нього.

При внесенні змін у файл, звернення до якого було під одним ім’ям, ці зміни виявляться і тоді, коли звернення до файлу відбудеться під іншим ім'ям. При створенні копії файла і подальшому змінені даних цієї копії, дані первинного файлу не змінюються.

Для того, щоб додати файлу ще одне ім'я (створити ще одне жорстке посилання на файл) необхідно виконати команду *ln* (від link - посилання, зв'язувати). В якості першого параметра вказується існуюче ім'я файлу, другого – нове ім'я.

```
#ln file1 file12
```

В даному випадку в поточному каталозі створюється ще одне жорстке посилання на файл з ім'ям ***file1***. Створене посилання знаходиться у тому ж каталозі, що і перше. Можна перемістити посилання в інший каталог за допомогою команди ***mv.*** Можна відразу вказати місце знаходження посилання за допомогою адреси.

Кількість жорстких посилань на файл (тобто різних імен для файлу) можна дізнатися, виконавши команду ***ls*** з параметром ***l***, яка виводить докладні відомості щодо кожного об'єкту каталогу (рис. 3.1).

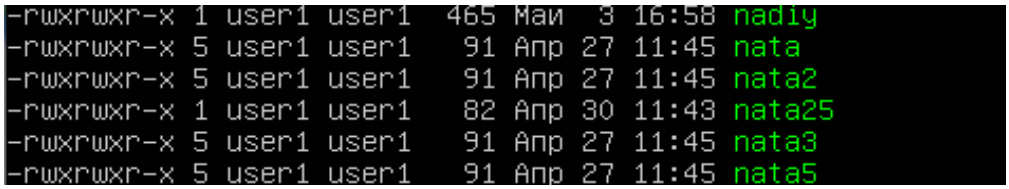


Рисунок 3.1. Приклад створення посилань

В даному прикладі файли ***nata*** і ***nata2*** мають по 5 жорских посилань (вона стоіть після вказівки прав доступу **(-rw-rw-r--**). Те, що усі посилання вказують на один і той же файл, говорить про ідентичність інформації у файлі.

Символьне посилання створюється за допомогою команди ***ln*** з ключем ***-s*** (від "symbolic"). В якості першого параметра пишеться абсолютна адреса та ім'я вихідного файлу, в якості другого – адреса та ім'я м'якого посилання.

Створюється символьне посилання ***nata55*** на файл ***nata*** (рис. 3.2).

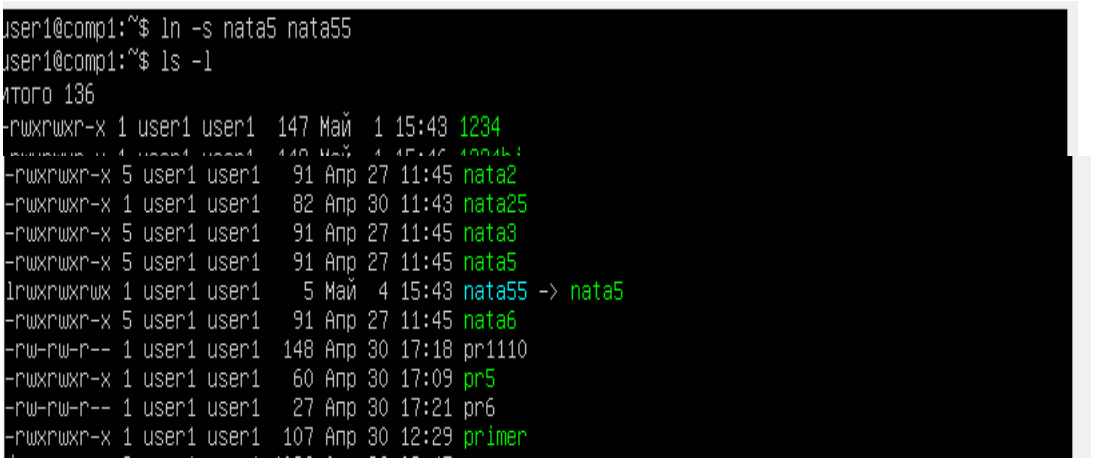


Рисунок 3.2 Приклад символьного посилання

Символьні посилання можна створювати й на каталоги. Наприклад, нижче створюється посилання на каталог (рис. 3.3). Тепер доступ до нього можна здійснювати безпосередньо з робочого столу (що набагато зручніше).

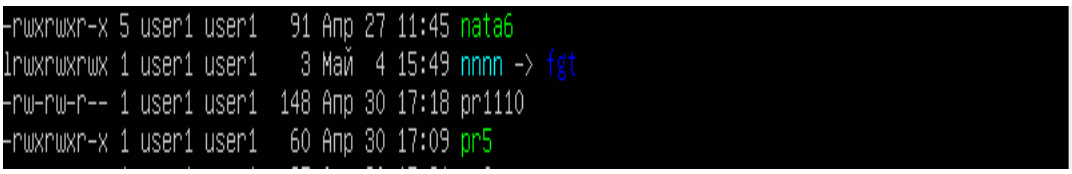


Рисунок 3.3 Створення посилання на каталог

script – команда записує у файл все, що виводиться на термінал. Щоб почати записувати, потрібно ввести команду **script**, провести сеанс роботи, а потім при натисканні **Ctrl+D** створюється файл і туди запишеться все, що виконувалося в терміналі.

who – виводить список користувачів, які працюють в даний момент з системою.

write - надсилає повідомлення користувачу, який знаходиться у системі, шляхом копіювання рядків з терміналу відправника на термінал одержувача.

tee — команда виводить на екран, або ж перенаправляє, вихідний матеріал команди і копіює його в файл в змінну**.**

Текстовий редактор *nano*

В операційній системі Linux часто виникає необхідність ручного корегування конфігураційних файлів. **Nano** – це єдиний редактор, котрий доступний даже на стадії інсталяції системи.

nano — немодальний редактор, і для вставки текста можна відразу починати набор. Якщо виникає необхідність в корегуванні конфігураційного файлу, наприклад, */etc/fstab*, треба зазначати параметр *-w*:

```
#nano -w /etc/fstab
```

Для збереження зроблених корегувань натискаємо **Ctrl+O**. Для виходу з редактору nano натискаємо **Ctrl+X**. При цьому пропонується зберегти проведені корегування у файлі. При відмові нитискаємо **N**, а для збереження — **Y**. Редактор запитає им’я файлу. Після введення імені натискаємо **Enter**.

Якщо випадково підтвердили необхідність збереження файлу, котрий не треба зберігати, то від зберігання можна відмовитися натискаючи **Ctrl+C** у момент запиту імені файлу.

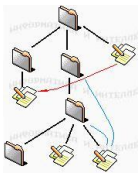
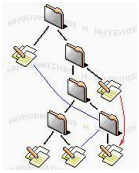
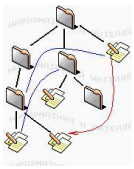
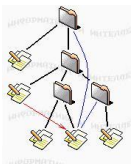
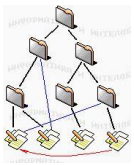
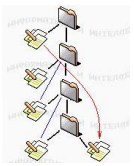
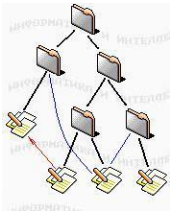
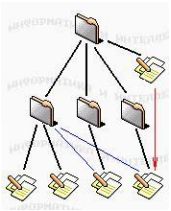
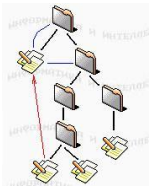
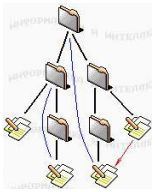
При запуску в терминалі **nano** знизу терминала з’являється підсказка, яка демонструє набір основних керуючих клавиш, котрі доступні у сполучені з **Ctrl +**. Клавиша (в даному випадку символ ^) заміняє клавишу **Ctrl**. Мета клавиша використовується і в інших сполученнях, наприклад, "**M**" означає клавишу **Alt**.

Завдання до лабораторної работи:

1. Ознайомитися з командою *script* і почати створення протоколу виконуваних дій.
2. Встановити зв'язок з сусідами, використовуючи команди *who, write*, запротоколювати листування.
3. Створити текстовий файл з довільним змістом за допомогою команди *tee***.** Переглянути вміст файлу за допомогою команди *cat* та виправити в ньому помилки за допомогою текстового редактора *nano*.
4. Подивитись довідку команди *unlink***.**
5. Зробити копію файла командою *cp*, видалити її командою *unlink***,** запротоколювати ці дії.
6. Створити структуру каталогів відповідно до варіантів (табл. 3.2). Чорними лініями представлена структура файлів/підкаталогів у каталозі. Червоними лініями позначені жорсткі посилання. Синіми лініями – символьні посилання. Файли створюються копіюванням раніше створеного файлу командою *cp* з внесенням у копії деяких змін.

Таблиця 3.2

Варіанти завдань

Варіант 1. 	Варіант 6. 
Варіант 2. 	Варіант 7. 
Варіант 3. 	Варіант 8. 
Варіант 4. 	Варіант 9. 
Варіант 5. 	Варіант 10. 

Виконати наступні дії:

- створити жорсткі посилання (Червоні лінії).
- створити символічні посилання (Сині лінії).
- провести ряд експериментів, що ілюструють доступ до файлів за основними іменами, за жорсткими та символьними посиланнями. Для доступу використовувати команду *cat* або редактор *nano*.

- провести ряд експериментів, котрі ілюструють реакцію системи на видалення файлу, на який є жорсткі посилання, та файлу, на який є символьні посилання. Перевіряти результати командою *ls -la*.
- знищити створені підкаталоги і файли, використовуючи команди *rmdir* і *unlink* (попередньо показати викладачеві виконану роботу).

Контрольні запитання.

1. Що означає жорстке посилання на файл**.**
2. Чим розрізняються символьні та жорсткі посилання**.**
3. Якою командою видаляються посилання.

Лабораторна робота 3

Робота з процесами

Мета роботи: набути практичних навичок роботи в операційній системі LINUX, ознайомитися з основними командами для керування роботою процесів.

Для отримання інформації щодо існуючих процесів у системі Linux використовуються команди **ps** і **pstree**. Ці команди виводять список процесів, які запущені в даний момент у системі, згідно з встановленим критерієм.

Синтаксис команд **ps**:

```
#ps [PID][options]
```

Основні опції *options* команди:

-A або *-e*: показувати усі процеси, включаючи системні;

-a: показувати процеси, запущені і іншими користувачами;

-x: процеси, що не мають контролюючого терміналу або запущені з іншого термінала;

-E: додатково виводити значення змінних оточення на момент запуску.

-u: виводить докладну інформацію для кожного з процесів.

-f: от сортувати за абеткою;

-l: розширений вивід.

Команда **pstree** виводить процеси у формі дерева. Перевагою є те, що можна відразу побачити, який процес є батьком якого.

Програма **top** призначена для виводу інформації щодо процесів реального часу. Процеси сортуються за максимально займаним процесорним часом, програма повідомляє про вільні системні ресурси.

Для знищення процесу (а також усіх його нащадків) використовуються команди **kill** та **killall**. Команда **kill** вимагає як аргумент номер процесу, а **killall** потребує ім'я процесу.

Синтаксис команд:

#killall [ім’я процесу]

#kill [-номер сигналу] PID

де *PID* - ідентифікатор процесу, який можна дізнатися за допомогою команди **ps.**

Режими роботи процесів

Завдання можуть виконуватися або на передньому плані (**foreground**), або у фоновому режимі (**background**). На передньому плані в будь-який момент часу може бути тільки одне завдання, з яким відбувається взаємодія користувача. Воно отримує введення з клавіатури та надсилає вивід на екран. Фонові завдання не отримують вводу з терміналу; як правило, такі завдання не потребують взаємодії з користувачем. Деякі завдання виконуються дуже довго і під час своєї роботи не потребують втручання користувача. Приклад – компілювання програм, а також стиснення великих файлів. Такі завдання слід пускати в фоновому режимі.

При перериванні (**interrupt**) завдання, процес «гине». Переривання завдань здійснюється натисканням відповідної комбінації клавіш, зазвичай це **Ctrl-C.** Відновити перерване завдання ніяким чином неможливо. Деякі програми, перехоплюють команду переривання, тому натискання комбінації

клавіш **Ctrl-C** може не перервати процес. Це зроблено для того, щоб програма могла знищити сліди своєї роботи перш, ніж вона буде завершена.

Приклад. Команда **yes** посилає нескінченний потік рядків, які складаються з символу «**y**», у стандартний вивід

```
#yes
y
y
...
```

Послідовність таких рядків буде тривати нескінченно. Щоб на екран не виводилася ця нескінченна послідовність, можна перенаправити стандартний вивід команди **yes** на пристрій ***/dev/null***, котрий діє як «чорна діра»: всі дані, послані в цей пристрій, зникають. За допомогою цього пристрою можна позбавлятися від занадто великого виведення деяких програм:

```
#yes > /dev/null
```

Для продовження роботи команди **yes**, але і щоб запрошення командної оболонки було на екрані «активним», потрібно команду **yes** перевести у фоновий режим.

Одним із способів переведення процесу у фоновий режим є приписати символ «**&**» до кінця команди. Наприклад

```
#yes > /dev/null&
\verb+[1] 155+
#
```

Повідомлення «[1]» являє собою номер завдання (job number) для процесу **yes**. Командна оболонка присвоює номер завдання кожному виконуваному завданню. Оскільки **yes** є єдиним виконуваним завданням, йому присвоюється номер 1. Число «155» є ідентифікаційним номером, яке відповідає даному процесу (*PID*), цей номер надано процесу системою. До процесу можна звертатися, вказуючи обидва ці номери.

Для того, щоб дізнатися статус процесу, потрібно виконати команду **jobs**, яка є внутрішньою командою оболонки:

```
#jobs
[1]+ Running yes >/dev/null &-
```

#

Для переривання роботи завдання використовується команда **kill.** Аргументом цієї команди може бути або *номер завдання*, або *PID*. У розглянутому вище випадку номер завдання *1*, тоді команда:

```
#kill %1
```

перерве роботу завдання. Коли до завдання звертаються за його номером (а не *PID*), тоді перед цим номером в командному рядку потрібно поставити символ відсотка (%).

Призупинення та продовження роботи завдань

В ОС Linux завдання можна призупинити (**suspend**, буквально – «підвісити»). «Підвішене» завдання не буде знищене; його виконання буде тимчасово призупинено до нового відновлення. Для припинення завдання треба натиснути відповідну комбінацію клавіш, зазвичай це **Ctrl-Z.**

Призупинений процес не виконується, на нього не витрачаються обчислювальні ресурси процесора. Призупинене завдання можна запустити знов виконуватися з точки призупинення. Для відновлення виконання завдання на передньому плані можна використовувати команду **fg** (від сл. «foreground» – передній план)

Командна оболонка ще раз виведе на екран назву команди, так що користувач буде знати, яке саме завдання він в даний момент запустив на передньому плані. Запустити завдання у фоновому режимі можна командою **bg** (від слова «background» – фон). Це аналогічно запуску команди з символом «&» на кінці.

Для призупинення завдання, яке працює у фоновому режимі, не можна користуватися комбінацією клавіш **Ctrl-Z**. Перш ніж зупиняти завдання, його потрібно перевести на передній план командою **fg** і лише потім призупинити. Таким чином, команду **fg** можна застосовувати або до припинених завдань, або до завдання, яке працює у фоновому режимі.

Командами **fg** і **bg** впливають на ті завдання, які були призупинені останніми (ці завдання позначаються символом «+» поруч з номером

завдання, якщо ввести команду *jobs*). Якщо в один і той же час працює одне або декілька завдань, завдання можна поміщати на передній план або у фоновий режим, задаючи в якості аргументів команди **fg** або команди **bg** їх ідентифікаційний номер (*job ID*).

Наприклад, команда:

```
#fg %2
```

поміщає завдання номер 2 на передній план, а команда:

```
#bg %3
```

поміщає завдання номер 3 у фоновий режим.

Використовувати *PID* в якості аргументів команд **fg** і **bg** не можна. Більш того, для переводу завдання на передній план можна просто вказати його номер. Так, команда:

```
# %2
```

Означає теж саме, що і

```
#fg %2
```

Функція управління завданням належить оболонці. Команди **fg**, **bg** і **jobs** є внутрішніми командами оболонки.

Стандартні засоби Linux дають змогу використовувати псевдоніми команд (аліаси) та помітно скоротити введення команд. Наприклад:

```
install   замість   sudo apt-get install
update    замість   sudo apt-get update
add       замість   sudo add-apt-repository
```

Це можна зробити за допомогою команди *alias*:

```
#alias install='sudo apt-get install'
#alias update='sudo apt-get update'
#alias add='sudo apt-get add-apt-repository'
#alias woman='man'
```

Щоб аліаси працювали і після перезапуску терміналу, необхідно додати вищенаведені рядки (або свої) в файл *~/.bashrc*:

1. #nano ~/.bashrc

додати аліаси в кінець документа

2. виконати: #source ~/.bashrc

Щоб видалити аліас, потрібно виконати команду: #unalias «ваш_аліас».

Переглянути список вже існуючих аліасів: #alias (без параметрів).

Дізнатися яка команда закріплена за аліасом: #alias «назва аліаса».

За допомогою команди **nice** можна змінювати пріоритети при запуску процесів. Коефіцієнт зниження/підвищення вказується в діапазоні 1..19 (за замовчуванням він дорівнює 10). Суперкористувач може підвищувати пріоритет команди, для цього потрібно вказати негативний коефіцієнт, наприклад − -10. Якщо вказати коефіцієнт більше 19, то він буде розглядатися як 19. Синтаксис команди:

#nice [-коефіцієнт зниження] команда [аргумент]

Якщо один або декілька процесів використовують занадто багато ресурсів системи, можна змінити їх пріоритети замість того, щоб вбивати. Для цього використовується команда **renice**. Її синтаксис:

#renice пріоритет [[-p] pid ...] [[-g] pgrp ...] [[-u] користувач ...]

де *пріоритет* - значення пріоритету,

pid - ідентифікатор процесу (використовуйте опцію *-p* для визначення кількох процесів)

pgrp - ідентифікатор групи процесу

user - ім'я користувача, що володіє процесом.

Завдання до лабораторної работи:

1. Ознайомитися з командою *ps*, виконати її з ключами *-a, -e, a, x, aux*.
2. Запустити кілька завдань (наприклад, команд перегляду файлів *less*), повертаючись в командний рядок комбінацією клавіш **Ctrl-Z** та розглянути дію команд *ps, jobs, fg, bg, kill, killall*.
3. Запустити процеси різними способами (у звичайному і фоновому режимі). Завершити («вбити») будь-який на свій вибір процес.

4. Відкрити другу і третю консолі з першого терміналу. Закрити першу і третю консолі з другого терміналу (не для всіх дистрибутивів можна виконувати це завдання).
5. Змінити пріоритет деяких процесів.
6. Привести 3 приклади призначення і використання аліасів.
7. Створити, заблокувати нового користувача *TEST_прізвище студента*.

Контрольні запитання

1. Що означає процес.
2. Як дізнатися про всі процеси, які працюють у даний час в системі.
3. Як завершити роботу процесу, якщо відома його назва.
4. Як визначити ідентифікатор процесу.
5. Як визначити розмір памяті, який займає процес.
6. Як визначити стан процесу.
7. Що означає аліас.
8. Як перевести процес у фоновий режим.
9. Як змінити пріорітет процесу.
10. Як дізнитися який процес займає більше всього процесорного часу.
11. Активні, фонові та відкладені процеси, як ними керувати.

Лабораторна робота 4

Встановлення прав доступу

Мета роботи: набути практичних навичок роботи в операційній системі LINUX, ознайомитися з командами, за допомогою яких можна міняти права доступу до файлу.

Кожному файлу відповідає дев'ять бітів дозволів. Дані біти формують режим доступу до файлу і визначають, які користувачі та групи мають права на даний файл.

Права доступу до файлів (дев'ять бітів) розділяються на три категорії: права власника файла, права групи пов'язаної з файлом, і права усіх інших користувачів. Кожна категорія має свій набір прав доступу до файлу, які забезпечують можливість читання (**r**) з файлу, запису (**w**) до файлу і його виконання (**x**) (або, навпаки, забороняють ці дії). Права доступу називаються також режимом доступу до файла.

В наступному прикладі створюється файл за допомогою команди **touch** і перевіряються права доступу до нього за допомогою команди **ls:**

```
#touch file
#ls –l file
#-rw-rw-r--  1 max max  0 jul 23 12:28 file
```

Права доступу до файлу це група символів: **-rw-rw-r--**. Перший символ (-) вказує на тип об'єкту. Дефіс показує на звичайний файл, **d** – означає каталог, **c** – символьний пристрій, **b** – вказує на блок-орієнтованний пристрій.

Права доступу зазначаються в такій послідовності – для користувача, групи і всіх інших. Відсутність права на будь-який вид доступу позначається знаком дефіса.

Права доступу до об'єкта можна задати двома способами: в цифровому або в символьному форматі. При використанні символьної форми ці категорії позначаються: *u* – користувач (власник), *g* – група, *o* – інші, які не входять у групу, й *a* – усі користувачі системи разом. Комбінації *r, w* і *x* для трьох категорій і є правами доступу до файлу:

User	Group	Others
rwx	rwx	rwx

В ОС Linux, як і в інших UNIX системах, каталоги також вважаються файлами. Наприклад, виконаємо наступні команди:

```
# mkdir foo
# ls –ld foo
drwxrwxr-x 2 max max 4096 jul 23 12:37 foo
```

Команда *mkdir* створює каталог. Команда *ls* з параметром *–ld* відображає дозвіл на доступ та іншу інформацію, яка стосується цього каталогу в цілому, але не його змісту. Перша буква *d* означає, що даний файл

є каталогом. Бачимо, що права доступу до каталогу мають значення *rwxrwxr-x* (власник має всі права, група також всі права, а всі інші не можуть нічого змінити в ньому).

Якщо за допомогою команди *ls* вивести інформацію щодо файлів пристрою для послідовного порту, то можна побачити наступне:

```
#ls -l /dev/ttyS0
crw-rw---- 1 root uucp 4,64 Mar 23 23:38 /dev/ttyS0
```

Файл */dev/ttyS0* представляє символьний пристрій (послідовний порт); володіє даним файлом користувач **root** і цей файл доступний також будь-якому члену групи **uucp**. Права доступу до файла мають значення (читання+запис, читання+запис, немає прав).

Змінити права доступу до файла можна за допомогою команди **chmod**. Для зміни параметрів використовують різні форми запису, включаючи вісімкову і мнемонічну. У мнемонічної формі параметри команди **chmod** позначають наступне (зі знаком плюс (+) вони використовуються для додавання права на доступ, зі знаком мінус (–) для їх видалення):

Синтаксис**:**

```
#chmod [права доступу] [файл або директорія]
```

u - додати (або видалити) право на яку-небудь операцію з файлом (каталогом) для користувача.

g - додати (або видалити) право для групи.

o - додати (або видалити) право для усіх інших.

a - додати (або видалити) право усім користувачам (*all*).

r - додати (або видалити) право на читання.

w - додати (або видалити) право на запис.

x - додати (або видалити) право на виконання.

Наприклад, створимо файл *readme.txt*, права доступу встановлюються за замовчуванням, вони визначаються маскою unmask у файлі */etc/bsdhrc*;

```
-rw-rw-r--1 max max  12 Oct 2 16:48  readme.txt
```

Якщо потрібно заборонити всім без винятку користувачам модифікувати цей файл, потрібно використовувати команду *chmod*:

```
#chmod –aw readme.txt
#ls –l readme.txt
-r--r--r-- 1 max max 12 Oct 2 16:48 readme.txt
```

Якщо прибрати права на виконання певного каталогу, файли які в ньому містяться, будуть приховані в межах каталогу, крім власника. Використовуючи комбінації різних прав доступу можна створити безпечне середовище.

Права доступу можна вказувати використовуючи числовий або символьний код:

```
#chmod 0755 foo.sh
#chmod +x  bar.sh
```

Для додаткової інформації можна скористатися командою *man chmod*.

chown – дозволяє змінити власника файла або каталога.

Синтаксис команди:

```
#chown [опції] <користувач [група]> <файл|каталог> [файл|каталог...]
```

Основні опції:

-*R*: Рекурсивна зміна власника каталогів та їх вмісту.

-*v*: Докладний опис дії (або відсутність дій) для кожного файлу.

-*c*: Докладно описувати дію для кожного файлу, власник якого дійсно змінюється.

Команда *chgrp* дозволяє змінити групу для власника файлу (або файлів); її синтаксис схожий на синтаксис команди *chown*:

```
#chgrp[опції] <група> <файл|каталог> [файл|каталог...]
```

Опції для цієї команди однакові з командою *chown*, і вона використовується дуже схожим способом:

```
#chgrp disk /dev/hd*
```

змінює групу власника всіх файлів в каталозі */dev/* з іменами, котрі починаються з *hd*, на групу *disk*.

Для визначення коду UID користувача і груп (GID), до яких він належить, використовується команда **id**:

```
#id
#uid=1000(max) gid=1000(max)
#группы=1000(max),4(adm),24(cdrom),27(sudo),30(dip),46(plugdev),108(lpadmin),124(sambashare)
```

Ключі:

-u — вивід тільки кодів *UID*

-g — вивід тільки кодів *GID*

-gn — вивід імені первинної групи користувача (замість *GID*)

Команда **groups** перераховує список груп, до яких належить (має доступ) даний користувач.

Приклад:

```
#groups
max adm cdrom sudo dip plugdev lpadmin sambashare
```

stat - виводить докладну інформацію щодо файлу або файлової системи, включаючи розмір, кількість блоків, дату останнього доступу, модифікації.

```
#stat [ім'я файлу]
```

usermod – змінює обліковий запис користувача.

Для зміни імені користувача спочатку зазначається нове ім'я, а потім старе:

```
#usermod -l first_name old_name
```

Для зміни UID без зміни імені користувача спочатку вказується новий код UID, а потім ім'я:

```
#usermod -u 1100 name
```

Визначення зміних оточення.

PATH - використовує системну змінну оточення для вказівки підмножини папок, в яких слід шукати при отриманні невідомої команди.

Можна виводити на екран наступною командою **echo** (символ $ обов'язково):

#echo $PATH

При кожному введенні невідомої команди Linux буде переглядати кожну з папок, вказаних у змінній оточення, в порядку їх завдання, намагаючись знайти програму з тим же ім'ям.

MANPATH − змінна містить список каталогів, розділених двокрапками, в яких команда **man** шукає сторінки довідки.

PAGER − змінна містить шлях до програми, що дозволяє посторінково переглядати вміст файлів, наприклад **less** або **more**.

grep знаходить потрібні рядки у текстових файлах, які містять вказаний користувачем текст

Наприклад:

#grep “max” /etc/passwd

umask є командою і функцією, яка встановлює режим створення файлу маски поточного процесу, який обмежує режими доступу до файлів і каталогів, створених у процесі.

Командою **umask** можна задати права доступу до нових файлів, які створює процес. **user mask** містить вісімкові значення дозволу, які потрібно встановити для всіх нових файлів, і підраховує значення, які потрібно отримати від 666 (для файлів) або 777 (для каталогу). Для зручності можна використовувати таблицю 3.3.

Таблиця 3.3

Вісімкове значення прав доступу.

umask	File Permissions	Directory Permissions
0	rw-	Rwx
1	rw-	rw-
2	r--	r-x
3	r--	r--
4	-w-	-wx
5	-w-	-w-
6	--x	--x
7	--- (none)	--- (none)

Створимо файл в каталозі */tmp*

```
#cd /tmp
#touch firstfile
#ls -l firstfile
-rw-r--r-- 1 root root 0 Mar  3 23:34 firstfile
```

Розглянемо права доступу: власник файлу може читати і записувати в файл; група і всі інші можуть тільки читати файл. Це стандартний *umask* зі значенням 0022. Щоб перевірити *umask*, який використовується в терміналі, можна виконати команду *umask* без будь-яких аргументів.

umask 0022 відповідає дозволу *-rw-r-r-*, але у багатьох випадках потрібно дати своїм колегам права на запис у каталог і файли, які створюють, тому, щоб встановити дозвіл *-rw-rw-r-* , згідно таблиці потрібно ввести :

```
#umask 0002
#cd /tmp
#touch secondfile
#ls -l secondfile
```

Результат команди:

```
-rw-rw-r-- 1 root root 0 Mar  4 23:16 secondfile
```

Завдання до лабораторної работи:

1. Ознайомитися з теоретичним матеріалом.

2. Визначити абсолютний шлях свого домашнього каталогу.

3. Визначити значення таких змінних оточення:

PATH, MANPATH, PAGER.

4. Визначити межі файлового простору, де система дозволяє створювати власні файли і каталоги (можливе використання автоматичного скрипта).

5. Перевірити, чи можливо втручання в особистий файловий простір іншого користувача.

6. Ознайомитися з командами визначення прав доступу до файлів і їх зміни (команди *id, groups, ls -l, stat, chmod, chown, chgrp, usermod*).

7. Змінити (призначити різні) права (доступ, власників, групи) на всі об'єкти в папці *ot4et* окремо на кожен файл.

8. Знайти запис у файлі */etc/passwd* відповідно вашому реєстраційному імені.

9. Визначити свій UID, дізнатися, до яких груп належить ваше реєстраційне ім'я, пояснити вивід команд *id, groups*.

10. Визначити список груп, в які входить користувач *root*.

11. Дізнатися, якими правами володіють новостворювані файли і каталоги (тобто створити новий файл і новий каталог, і переглянути для них права доступу).

12. Зробити свій домашній каталог видимим для всіх користувачів групи *users*.

13. Створити в домашньому каталозі підкаталог *tmp*, файли в якому зможуть створювати, видаляти і перейменовувати всі користувачі, які входять до групи *users*, при цьому вміст цього підкаталогу не видимий всім іншим користувачам.

Контрольні запитання

1. За допомогою якої команди можна визначити права доступу до файлу.
2. Що включають атибути доступу до файлу.
3. Що означає користувач *root***.**
4. Назвіть змінні оточення

Лабораторна робота 5

Утиліти архівування і стиснення

Мета роботи: освоєння навичок роботи з утілітами стиснення і архівування

Основна функція **gzip** — стиснути файл, зберегти стислу версію у вигляді *filename.gz* і видалити вихідний нестиснений файл. Вихідний файл видаляється тільки у випадку успішного виконання *gzip*. Випадково видалити

при цьому файл дуже важко. Утіліта **gzip** має велику кількість параметрів. Характер роботи цієї програми можна модифікувати за допомогою параметрів командного рядка.

Маємо великий файл, наприклад, з ім'ям *arch.txt*:

```
#ls -l arch.txt
-rw-r-r-- 1 user-005 max 2355 25 Aug 17:54:00 arch.txt
```

Для стиснення файлу за допомогою **gzip** потрібно виконати наступну команду:

```
#gzip arch.txt
```

В результаті *arch.txt* буде замінений стисненим файлом *arch.txt.gz***.** Кінцевий результат буде таким:

```
#gzip arch.txt
#ls -l arch.txt.gz
-rw-r-r-- 1 max max 321 14 апр. 09:54:00 arch.txt.gz
```

Маємо на увазі, що по завершенні роботи **gzip** файл *arch.txt* видаляється.

Можна передати **gzip** список імен файлів. Кожен файл списку буде стиснутий і отримає розширення *.gz*.

Ефективність стиснення файлу залежить від його формату і змісту. Наприклад, багато файлів графічних форматів, такі як *jpeg* і *png*, вже значною мірою стиснуті, і застосування **gzip** до таких файлів може не мати ефекту. Зазвичай гарно стискаються текстові файли і двійкові файли такі, як виконувані модулі та бібліотеки.

Відомості щодо стислих файлів можна отримати за допомогою команди **gzip -l**, наприклад:

```
#gzip -l arch.txt.gz
compressed        uncompr    ratio  uncompressed_name
103116            312996     67.0%        arch.txt
```

Для того щоб отримати назад оригінал стислої версії, використовується **gunzip**, наприклад:

```
#gunzip arch.txt.gz
```

Після виконання цієї команди отримаємо:

```
#gunzip arch.txt.gz
#ls -l arch.txt
-rw-r-r-- 1 max max 2355 14 апр. 17:54:00 arch.txt
```

Після декомпресії файлу стисла версія видаляється. Замість **gunzip** можна скористатися **gzip -d**.

gzip − записує ім'я вихідного файлу в його стислу версію. Тому якщо ім'я стисненого файлу (з розширенням *.gz*) виявляється занадто довгим для вибраного типу файлової системи, ім'я вихідного файлу може бути відновлено **gunzip**, навіть якщо при стисненні воно було стиснуто. Для декомпресії файлу з присвоєнням вихідного імені використовується параметр *-N*. Наприклад:

```
#gzip arch.txt
#mv arch.txt.gz garr.txt.gz
```

*Я*кщо тепер декомпрессувати *garr.txt.gz*, то в результаті файл вийде *garr.txt*, тобто з тим же ім'ям, що і стиснутий файл. Однак, задавши параметр *-N*, виходить:

```
#gunzip -N garr.txt.gz
#ls -l arch.txt
-rw-r-r-- 1 max max 2355 25 апр. 17:54:00 arch.txt
```

gzip і **gunzip** можуть також здійснювати компресію і декомпресію даних зі стандартного вводу і виводу. Якщо не повідомити **gzip** імен файлів, що підлягають компресії, то програма намагається стискати дані зі стандартного вводу. Аналогічно **gunzip** з параметром *-c* виводить декомпресвані дані на стандартний пристрій виводу. Можна, наприклад, передати по каналу вихідні дані деякої команди на **gzip**, щоб за один крок стиснути вихідний потік і записати його у файл:

```
#ls -laR $HOME gzip > filelist.gz
```

Ця команда виконує вивід вмісту вихідного файлу з підкаталогами і запише результат у стиснутий файл *filelist.gz.* Вміст файла можна вивести командою:

```
#gunzip -з filelist.gz more
```

Ця команда декомпрессує *filelist.gz* і передає результат по каналу команді *more*. Якщо використовується команда з параметром *gunzip -c*, на диску залишається стиснутий файл.

Команда **zcat** аналогічна **gunzip -c**. Можна розглядати її як версію **cat** для стислих файлів.

При стисненні файлів можна використовувати параметри *-1, -2, ..., -9*, щоб задати швидкість і ступінь стиснення. При цьому *-1 (або --fast)* задає найшвидший метод, але слабкіше стискає файли, а *-9 (або --best)* задає більш повільний метод, але сильніше всього стискає файли. За замовчуванням значення методу стиснення дорівнює *-6*. Ці параметри не впливають на використання команди **gunzip**, яка здатна здійснити декомпресію будь-якого файлу незалежно від використаного методу стиснення.

Існує ще одна програма компресії/декомпресії **bzip2.** Вона стискає в середньому на 10-20% сильніше, ніж **gzip**, за рахунок більшої тривалості процесу стиснення. Не можна використовувати **bunzip2** для декомпресії файлів, стислих **gzip**, і навпаки. Для стиснення файлів слід використовувати **gzip/gunzip** або **bzip2/bunzip2**. Якщо зустрічається файл з розширенням *.z*, то він, швидше за все, був створений за допомогою *compress* (застаріла програма), і **gunzip** може виконати декомпресію.

Використання tar

tar − є універсальною утилітою архівування, здатною упаковувати декілька файлів в один архівний, зберігаючи при цьому дані, необхідні для повного відновлення, такі як права доступу і володіння. Назва *tar* походить від *tar archive*, оскільки спочатку ця утиліта призначалася для архівування файлів у вигляді резервних копій на магнітних стрічках.

Формат команди **tar**:

#tar function options files...

де *function* є літерою, яка вказує виконувану операцію, *options* є списком однолітерних параметрів цієї функції, а *files* - списком файлів в архіві, які упаковуються чи розпаковуються.

Параметр *function* може приймати наступні значення:

c – створити новий архів;

x – витягнути файли з архіву;

t – перерахувати вміст архіву;

r – дописати файли в кінець архіву;

u – замінити файли в архіві більш новими;

d – порівняти архівовані файли з файлами файлової системи.

Найчастіше використовуються функції *c, x* і *t.*

Використовуються наступні значення *options*:

k – зберегти при разархівації існуючі файли, тобто не замінювати існуючі файли;

f filename – задати ім'я архівного файлу, який потрібно прочитати або записати;

z – вказати, що записані в архів файли або наявні в ньому стискаються за допомогою **gzip;**

v – змусити **tar** показати імена файлів, які розміщені в архів або вилучені з нього.

Синтаксис **tar** може спочатку здатися складним, але на практиці він дуже простий. Наприклад, є каталог *mt*, що містить наступні файли:

```
# ls -l mt
-rw-r--r-- 1 max max 2355 25 апр. 17:54:00 arch.txt.gz
-rw-r--r-- 1 max max 2355 25 апр. 10:04:00 READMI
-rwxr-xr-x 1 max max 9220 5 Sep 7:50:00 mt
-rw-r--r-- 1 max max 2775 13 Sep 18:34:00 mt.1
-rw-r--r-- 1 max max 6421 28 Sep 10:26:00 mt.c
-rw-r--r-- 1 max max 3948 2 Nov 22:04:00 mt.0
-rw-r--r-- 1 max max 11204 12 Nov 19:47:00 st_my.txt
…
```

Потрібно упакувати вміст цього каталогу в один архівний *tar*-файл, для цього використовується команда:

```
#tar cf mt.tar mt
```

В даному випадку першим аргументом **tar** є функція *c* (create), за якою слідують параметри. Використання одного параметра *f mt.tar* вказує, що результуючий архівний файл повинен мати ім'я *mt.tar*. Останнім аргументом є ім'я (або імена) файлу, котрий архівується. В даному випадку задається ім'я каталогу *mt* для того, щоб **tar** упакував в архів всі файли цього каталогу.

Першим аргументом повинна бути буква, яка позначає функцію, далі список її параметрів. Тому немає сенсу ставити дефіс (-) перед параметрами, як того вимагають багато команд Linux. **tar** дозволяє використовувати дефіс наступним чином:

```
#tar -cf mt.tar mt
```

В деяких версіях **tar** першою літерою повинна бути функція, тобто *c, t* або *x*. В інших версіях порядок букв не є суттєвим.

Важливо вказати ім'я файлу, якщо використовуються літери *cf*, інакше буде переписаний перший файл у списку упаковуючих файлів, оскільки він буде прийнятий за ім'я архіву!

Часто корисно використовувати **tar** з параметром *v*, що дозволяє вивести перелік архівованих файлів. Наприклад:

```
#tar cvf mt.tar mt
mt/
mt/st_my.txt
mt/READMI
mt/mt.1
mt/arch.txt.gz
mt/mt.c
mt/mt.0
mt/mt
```

Якщо задати *v* кілька разів, виводяться додаткові відомості, наприклад:

```
#tar cvvf mt.tar mt
drwxr-xr-x max/max 0 15 Nov 17:35:00 mt/
```

-rw-r--r--max/ max 11204 12 Nov 19:47:00 mt/st_my.txt

-rw-r--r-- max/ max 847 30 Aug 10:04:00 mt/README

-rw-r--r-- max/ max 2775 13 Sep 18:34:00 mt/mt.1

-rw-r--r-- max/ max 6421 28 Sep 10:26:00 mt/mt.c

-rwxr-xr-x 1 max max 9220 5 Sep 7:50:00 mt/mt

Тепер можна передати файл *mt.tar* іншим користувачам для розпакування на своїх машинах. Розпакувати можна командою:

#tar xvf mt.tar

В результаті виконання цієї команди створюється підкаталог *mt*, в який поміщаються всі вихідні файли з тими ж правами доступу, які були у вихідній системі. Власником нових файлів буде користувач, що виконав команду *tar xvf*, якщо тільки розархівацію не зробив *root*. В останньому випадку зберігається початковий власник файлу. Параметр *x* означає вилучення файлів, параметр *v* використовується для перерахування усіх назв файлів. Отримаємо:

```
#tar xvf mt.tar
mt/
mt/st_my.txt
mt/README
mt/mt.1
mt/arch.txt.gz
mt/mt.c
mt/mt.0
mt/mt
```

Команда **tar** зберігає ім'я шляху для кожного файлу щодо того розташування, де спочатку було створено архівний файл. Коли створювався архів *tar cf mt.tar mt,* єдиним заданим ім'ям вхідного файлу було *mt* – ім'я містить файли каталогу. Тому **tar** записує в архівний файл сам каталог і всі файли, які знаходяться в ньому. Коли вилучаються файли з архіву, створюється каталог *mt*, в який поміщаються всі файли, це є процедурою, прямо протилежною створенню архіву.

За замовчуванням **tar** витягує з архіву всі файли щодо поточного каталогу, в якому запускається **tar**. Наприклад, якщо упаковувати вміст каталогу */bin* командою:

```
#tar cvf bin.tar /bin
```

то отримаємо від системи попередження:

```
tar: Removing leading / from absolute path in the archive.
```

Це означає, що файли будуть збережені в архіві у папці *bin*. При розпакуванні каталог *bin* створюється в робочому каталозі *tar*, а не як */bin* в системі, в якій проводиться розпакування. Це має мету запобігання катастрофічних помилок при розпакуванні. В іншому випадку розпакування файлу, що містить упакований */bin*, призвела б до знищення поточного каталогу */bin*. Якщо ж дійсно потрібно розпакувати такий архів в */bin*, то це потрібно робити, перебуваючи в кореневому каталозі /.

Іншим способом створення архіву *mt.tar* був би перехід командою *cd* безпосередньо в каталог *mt* і використання такої команди, як

```
#tar cvf mt.tar *
```

При такому способі підкаталог *mt* не записується в архівний файл, і при розпакуванні файли поміщаються в поточний робочий каталог.

Подивитися на таблицю його вмісту і визначити, як він був упакований можна за допомогою команди:

```
#tar tvf tarfile
```

В результаті виведеться таблиця вмісту архіву з ім'ям *tarfile*. При використанні функції *t* потрібно тільки один параметр *v*, щоб отримати «довгу» роздруківку переліку файлів.

```
#tar tvf mt.tar
drwxr-xr-x max/ max 0 15 Nov 17:35:00 mt/
-rw-r--r-- max/ max 11204 12 Nov 19:47:00 mt/st_my.txt
-rw-r--r—max/ max 847 30 Aug 10:04:00 mt/README
-rw-r--r-- max/ max 2775 13 Sep 18:34:00 mt/mt.1
-rw-r--r-- max/ max 24 25 Aug 17:54:00 mt/arch.txt.gz
-rw-r--r-- max/ max 6421 28 Sep 10:26:00 mt/mt.c
-rw-r--r-- max/ max 3948 2 Nov 22:04:00 mt/mt.0
```

-rwxr-xr-x 1 max max 9220 5 Sep 7:50:00 mt/mt

Розпакування при цьому не проводиться, видно тільки таблицю вмісту архіву. Імена файлів говорять про те, що файл архіву був створений з усіх файлів підкаталогу *mt*, і при розпакуванні архіву буде створено підкаталог *mt*, в який будуть поміщені усі файли.

Можна витягувати з архіву і окремі файли. Для цього потрібно скористатися командою:

#tar xvf tarfile files,

де *files* є списком файлів, що витягаються з архіву. Якщо не вказати імен, **tar** розпаковує весь архів.

Слід вказувати повний шлях, як він зазначений в архіві. Наприклад, якщо потрібно отримати з попереднього архіву тільки файл *mt.c*, це можна зробити командою

#tar xvf mt.tar mt/mt.c

В результаті створюється підкаталог *mt*, в який поміщається *mt.c*.

Використання *tar* спільно з *gzip* і *bzip2*

tar не здійснює компресії даних, котрі зберігаються в його архівах. Якщо створити *tar*-архів з трьох файлів по 300 Кбайт, то вийде архів розміром 900 Кбайт. Часто *tar*-архіви стискаються за допомогою *gzip*. Стислий *tar*-архів можна створити, виконавши команди:

#tar cvf tarfile files...

#gzip -9 tarfile

Але це вимагає наявності достатнього обсягу пам'яті для збереження нестисненого архівного файлу, перш ніж піддати його обробці *gzip*.

Інший спосіб виконати ту ж задачу – використання функції *tar*, яка дозволяє вивести архів на стандартний пристрій виводу. Якщо задати архівний файл для читання або запису як «-», то дані будуть зчитуватись зі стандартного пристрою вводу та виводитися на стандартний пристрій виводу. Наприклад:

```
#tar cvf – files | gzip -9 > tarfile.tar.gz
```

Команда *tar* створює архів файлів з назвами імен і виводить його на стандартний пристрій виводу; потім *gzip* читає дані зі стандартного пристрою введення, стискає їх і виводить на свій стандартний пристрій виводу; і потім стислий архівний файл перенаправляється в *tarfile.tar.gz.*

Розпакувати такий архівний файл можна за допомогою команди:

```
#gunzip -9c tarfile.tar.gz | tar xvf
```

gunzip декомпресує вказаний архівний файл, виводить результат на стандартний пристрій виводу, звідки він зчитується утилітою *tar*, як зі стандартного вводу, і розпаковується.

Але, обидві ці команди довго друкувати. Тому *tar* має параметр *z*, завдання якого автоматично створювати або розпаковувати стислі архіви. Можна використовувати наступні команди:

```
#tar cvzf tarfile.tar.gz files...
```

та

```
#tar xvzf tarfile.tar.gz
```

Файлам, що створені таким чином, потрібно присвоювати розширення *.tar.gz* , для зрозуміння їхнього формату. Параметр *z* нормально працює і з іншими функціями *tar*, наприклад як *t*.

Можна також використовувати *tar* разом з *bzip2*. Щодо вибора програми стиснення потрібно повідомити наступним чином:

```
#tar cvf tarfile.tar.bz2 --use-compress-program=bzip2 files
```

або коротше:

```
#tar cvf tarfile.tar.bz2 --use=bzip2 files...
```

або ще коротше:

```
#tar cvjf tarfile.tar.bz2 files
```

Остання версія працює тільки з останніми версіями *GNU tar*, які підтримують параметр *j*.

Можна писати короткі сценарії оболонки або псевдоніми для створення архівних файлів і їх розпаковування. Використовуючи оболонку *bash*, можна включити у файл *.bashrc* наступні функції:

```
tarc () { tar czvf $1.tar.gz $1 }
tarx () { tar xzvf $1 }
tart () { tar tzvf $1 }
```

Якщо визначено такі функції, то створити стислий архівний файл з окремого каталогу можна за допомогою команди:

```
#tarc directory
```

Отриманий архівний файл буде називатися *directory.tar.gz*. Щоб вивести список файлів стисненого *tar*-архіву, потрібно виконати таку команду:

```
#tart file.tar.gz
```

Для розпакування такого архіву виконуємо команду:

```
#tarx file.tar.gz
```

Файли, створені за допомогою *gzip* і/або *tar*, можуть розпаковуватися відомою утилітою **winzip** в Windows. І навпаки, файл у форматі *.zip* можна розпакувати на комп'ютері під Linux за допомогою команди *unzip*.

Прийоми роботи з *tar*

Оскільки *tar* записує в архів права володіння і доступу до файлів, а також зберігає повну структуру каталогів, символічні і жорсткі посилання, з його допомогою дуже зручно копіювати або переміщувати ціле дерево каталогів з одного місця в інше в межах даної системи і, навіть, з однієї машини на іншу. Використовуючи описаний раніше синтаксис з мінусом, можна вивести *tar*-архів на стандартний пристрій виводу, звідки він читається і розпаковується в будь-яке місце через стандартний пристрій введення.

Наприклад. Маємо каталог з двома підкаталогами − *ms1* та *ms2*. Каталог *ms1* містить ціле дерево файлів, символічних посилань і тощо, які

важко точно відобразити, використовуючи команду *cp* з рекурсією. Для того щоб скопіювати все дерево з каталогу *ms1* в каталог *ms2*, можна виконати наступні команди:

```
#cd ms1
#tar cf - . | (cd ../ms2; tar xvf -)
```

Починається робота в каталозі *ms1* і створюється архів поточного каталогу, який виводиться на стандартний пристрій виводу. Цей архів читається підоболочкою (міститься в дужках команди); підоболочка здійснює перехід до цільового каталога *../ms2* (щодо *ms1*) і виконує *tar xvf*, здійснюючи читання зі стандартного пристрою вводу. Жодних архівних файлів на диск не пишеться; дані надсилаються через канал від одного процесу *tar* іншому. Другий процес *tar* запускається з параметром *v*, завдяки чому виводяться імена усіх файлів, які розпаковуються, це дозволяє контролювати правильність роботи команди.

Завдання до лабораторної работи:

1. Створити в домашньому каталозі нові файли і підкаталоги будь-яким відомим способом. Для підтвердження унікальності виконуваних дій рекомендується при створенні файлів і каталогів використовувати своє прізвище.
2. Відпрацювати навички створення і управління архівами за допомогою команд *tar, gzip,bzip* окремо і разом, використовуючи окремі файли та дерева підкаталогів.
3. Від імені іншого користувача розпакувати раніше створені архіви. Перевірити отримані права доступу і володіння.

Контрольні питання

1. Які особливості використання команди *gzip***.**
2. Для чого використовується команда *tar*
3. Порівняйте наведені команди стиснення
4. Порівняйте наведені команди архівування.

Лабораторна робота 6

Програмуванняд на мові BASH

Мета роботи: ознайомитися та набути практичних створення *shell*-скриптів на мові **bash**.

Для виконання індивідуальних завдань використовуйте ваш порядковий номер у журналі.

Розроблений *shell*-скрипт повинен називатися

<номер_группы>-<номер_студента>-bash-1.

BASH - Bourne-Again SHell (перекладається як "перероджений шел", або "Знову шел Борна(розробник sh)"), найбільш популярний командний інтерпретатор в юніксоподібних системах, в особливості, в **GNU/Linux**.

В найпростішому випадку, скрипт − простий список команд системи, записаний у файл. Створення скриптів допомагає зберегти час і сили, які витрачаються на введення послідовності команд всякий раз, коли необхідно їх виконати.

Запустити сценарій можна командою *bash scriptname*. Більш зручний варіант - зробити файл скрипта виконуваним за допомогою команди *chmod*.

Наприклад:

chmod 555 scriptname (видача прав на читання/виконання будь-якого користувача в системі)

chmod +rx scriptname (видача прав на читання/виконання будь-якого користувача в системі)

chmod u+rx scriptname (видача прав на читання/виконання тільки власнику скрипта)

Після того, як сценарій став виконуваним, його можна запустити командою *./scriptname*. Якщо, при цьому, текст сценарію починається з коректної сигнатури *("sha-bang"=#bash),* то для його виконання буде викликано відповідний інтерпретатор.

Завершивши налагодження сценарію, можна помістити його в каталог */usr/local/bin* (звісно, якщо володієте правами *root*) для того, щоб зробити його доступним для себе та інших користувачів системи. Після цього сценарій можна викликати, просто надрукувавши назву файлу в командному рядку і натиснути клавішу [ENTER].

Загальні вимоги:

Усі завдання необхідно виконати за такими правилами:

- скрипт повинен при запуску виводити: прізвище та ім'я автора, назву програми, короткий опис;
- скрипт повинен виводити підказки для користувача, які допомагають користувачу коректно працювати з програмою;
- скрипт повинен працювати в режимі діалогу з користувачем (у вигляді "запитання"-"відповідь");
- скрипт повинен обробляти аварійні ситуації, помилки і некоректно введені дані;
- після виконання операцій скрипт повинен пропонувати користувачеві почати виконання або вийти з циклу;
- якщо на останній ітерації виводиться повідомлення про помилку і користувач закінчив роботу скрипта, скрипт повинен завершитися з кодом повернення 250;
- у більшості сценаріїв можна використовувати команду *stat*.

Приклад

Приклад діалогу користувача з програмою:

```
$ ./program.sh
Программа пошуку користувачів.
За допомогою цієї программи можна шукати користувачів за його UID.
Розробник: Іванов Іван
Введіть UID користувача:
```

```
100000
Помилка! Користувача з таким UID не знайдено.
Хочете продовжити? (y/n)
y
Введіть UID користувача:
-1
Помилка! UID Не може бути від’ємним числом.
Хочете продовжити? (y/n)
y
Введіть UID користувача:
1000
Користувач: user
Хочете продовжити? (y/n)
n
...вихід з програми...
```

Варіанти завдань

Завдання 1

Розробити скрипт, який:

- виводить ім'я поточного каталогу;
- запитує ім'я користувача;
    - якщо зазначений користувач не зареєстрований в системі, то виводить повідомлення про це та повторно запитує ім'я користувача;
- запитує ім'я каталогу;
- виводить кількість звичайних файлів, власником яких є користувач;

Для виконання завдання використовуйте команди *grep, find*.

Завдання 2

Розробити скрипт, який:

- запитує ім'я користувача;

- якщо зазначений користувач не зареєстрований в системі, то виводить повідомлення про це та повторно запитує ім'я користувача;
- якщо зазначений користувач зареєстрований у системі, то виводить його унікальний ідентифікатор (UID) і імена груп, до яких входить цей користувач, розділені пробілом.

Завдання 3

Розробити скрипт, який:

- виводить ім'я поточного каталогу;
- запитує шлях до першого файлу;
- підраховує кількість слів у першому файлі (команда *wc*);
- запитує шлях до іншого файлу;
- записує у другий файл обчислену кількість слів;
- якщо другий файл існує і містить дані, то запитує у користувача підтвердження на запис у файл.

Завдання 4

Розробити скрипт, який:

- виводить ім'я поточного каталогу;
- запитує ім'я файлу;
- якщо файл не існує, виводить повідомлення про помилку і знову запитує ім'я файлу;
- запитує ім'я користувача;
- виводить права даного користувача до файлу в форматі: ЧИТАТИ/ПИСАТИ/ВИКОНУВАТИ.

Завдання 5

Розробити скрипт, який:

- виводить ім'я поточного каталогу;

- запитує ім'я файлу;
- якщо файл не існує, виводить повідомлення про помилку і знову запитує ім'я файлу;
- запитує ім'я користувача;
- якщо користувач не є власником файлу, то виводить ім'я власника файлу і ім'я файлу групи.

Завдання 6

Розробити скрипт, який:

- виводить ім'я поточного каталогу;
- запитує перше розширення файлу;
- запитує друге розширення файлу;
- знаходить всі файли у поточному каталозі з першим розширенням і перейменовує їх так, що змінює їх розширення на друге розширення;
- якщо таких файлів не існує, виводить повідомлення про помилку і починає спочатку.

Завдання 7

Розробити скрипт, який:

- виводить ім'я поточного каталогу;
- запитує ім'я файлу;
- якщо файл не існує, виводить повідомлення про помилку і знову запитує ім'я файлу;
- запитує дату;
- визначає, чи здійснювався доступ до файлу після зазначеної дати, і виводить повідомлення про це;
- якщо доступ до файлу не здійснювався, завершується з кодом 120.

Завдання 8

Розробити скрипт, який:

- виводить ім'я поточного каталогу;
- запитує ім'я файлу;
- якщо файл не існує, виводить повідомлення про помилку і знову запитує ім'я файлу;
- запитує дату;
- визначає, змінювалося вміст файлу після зазначеної дати, і виводить повідомлення про це;
- якщо вміст файлу змінювалося, завершується з кодом 120.

Завдання 9

Розробити скрипт, який:

- запитує тип дії: пошук по імені файлу або пошук за розміром;
- запитує каталог, в якому потрібно знайти файл;
- запитує ім'я файлу або розмір в залежності від необхідного дії;
- виводить всі файли з заданим ім'ям або всі файли більше зазначеного розміру в залежності від необхідного дії;

Для виконання завдання використовуйте команду *find*.

Завдання 10

Розробити скрипт, який:

- виводить ім'я поточного каталогу;
- запитує ім'я файлу;
- якщо файл не існує, виводить повідомлення про помилку і знову запитує ім'я файлу;
- запитує дату;
- визначає, чи змінювався індексний дескриптор файлу після зазначеної дати, і виводить повідомлення про це;

- якщо індексний дескриптор файлу змінювався, завершується з кодом 120.

Завдання 11

Розробити скрипт, який:

- запитує тип дії: показати поточний каталог, піднятися на каталог вище, перейти в каталог;
- якщо обрано дію перейти в каталог, вивести список варіантів і дозволити користувачеві вибрати один з них;
- якщо обрано дію піднятися на каталог вище, то створити в ньому файл і розглянути його права доступу, модифікувати їх.

Завдання 12

Розробити скрипт, який:

- запитує ім'я файлу;
- якщо файл не існує, виводить повідомлення про помилку;
- якщо файл існує, виводить інформацію щодо файлу у форматі:
    - ім'я файлу (не включаючи шлях до файлу);
    - тип файлу;
    - розмір файлу;
    - власник файлу;
    - права доступу;
    - дата створення файлу;

Для виконання завдання використовуйте команди *ls* і *stat*.

Завдання 13

Розробити скрипт, який:

- запитує тип операції над файлом: створити, видалити, перемістити;
- для створення файлу запитує ім'я нового файлу;

- якщо файл з таким ім'ям існує, виводить повідомлення про помилку;
- для видалення або переміщення файлу запитує ім'я файлу;
- якщо файл із таким ім'ям не існує, виводить повідомлення про помилку;
- для переміщення додатково запитує шлях до каталогу, в який потрібно перемістити файл;
- якщо каталог не існує, виводить повідомлення про помилку.

Завдання 14

Розробити скрипт, який:

- запитує шлях до каталогу;
- запитує розмір файлу;
- знаходить всі файли більше заданого розміру і видаляє їх, попередньо питаючи підтвердження користувача;

Для виконання завдання використовуйте програму *find*.

Завдання 15

Розробити скрипт, який:

- запитує шлях до файлу;
- для даного файлу виводить порядково тимчасові мітки у форматі:
- час останнього доступу;
- час останньої зміни;
- час зміни індексного дескриптора.

Для виконання завдання використовуйте команди *ls* или *stat*.

Завдання 16

Розробити скрипт, який:

- виводить ім'я поточного каталогу;
- запитує ім'я файлу;

- якщо файл не існує, виводить повідомлення про помилку і знову запитує ім'я файлу;
- знаходить у файловій системі жорсткі посилання на даний файл та виводить їх імена.

Завдання 17

Розробити скрипт, який:

- запитує шлях до першого файлу;
- запитує шлях до другого файлу;
- встановлює права доступу до другого файлу за зразком першого файлу.

Завдання 18

Розробити скрипт, який:

- запитує шлях до першого файлу;
- запитує шлях до другого файлу;
- визначає, чи введені імена файлів жорсткими посиланнями на один і той же файл;
- якщо не є, виводить повідомлення про помилку в потік *stderr;*
- якщо на останній ітерації виводилося повідомлення про помилку і користувач закінчив роботу скрипта, завершується з кодом 250.

Контрольні питання

1.Що означає сценарій оболонки?

2. Яке призначення рядка *#!/bin/sh* у сценаріях оболонки?

3.Перерахуйтн керуючи конструкції мови програмування оболонки.

4.Каково назначение команды *test*?

5. Які засоби налагодження сценаріїв надаються розробнику оболонки?

СПИСОК РЕКОМЕНДОВАНОЇ ЛІТЕРАТУРИ

1. Alapati Sam R. Modern Linux Administration» / Sam R Alapati , O'Reilly Media , 2016 — 500 p.
2. Barrett D. Linux. Pocket Guide. Essential Commands / D. Barrett 3rd edition, O'Reilly. 2016, – 272 p.
3. Cannon J. Linux for Beginners: An Introduction to the Linux Operating System and Command Line, /J. Cannon, Copyrighted material, 2014, -189p.
4. Граннеман С. Linux: Необходимый код и команды, Карманный справочник, /С. Граннеман – Вильямс, 2015, - 416 с.
5. Григорьев В.М. Компьютерные сети. Виртуализация в обучении и проектировании / В.М. Григорьев, В.С. Хандецкий . – Днепропетровск: Акцент ПП, 2012, - 224 с.
6. Колисниченко Д.Н. LINUX: полное руководство / Д.Н.Колисниченко, Аллен Питер В. - СПб: Наука и Техника, 2006. - 784 с.
7. Колисниченко Д.Н. Командная строка Linux и автоматизация рутинних работ / Д.Н. Колисниченко - СПб: БХВ-Петербург, 2012, -352 с.
8. Колисниченко Д.Н. Linux: От новичка к професионалу / Д.Н. Колисниченко - СПб: BHV, 2016, – 608 с.
9. Кофлер М. Linux. Установка, настройка, администрирование. : Пер. с нем. /М. Кофлер - СПб: Питер, 2014. – 768 с.
10. Лав Р. Linux. Системное программирование /Р. Лав - СПб: Питер, 2016, – 448 с.
11. Лав Р. Ядро Linux. Описание процесса разработки /Р. Лав 3-е изд., Изд. «Вильямс», 2014, – 496 с.
12. Мэтью Н. Основы программирования в Linux: Пер. с англ. / Н. Мэтью, Р. Стоуне. - 4-е изд., перераб. и доп. — СПб.: БХВ-Петербург, 2009. - 896 с.
13. Негус К. Ubuntu и Debian Linux для продвинутых: более 1000 незаменимых команд. - СПб: Питер, 2011. – 352 с.

14. Никсон Р. Up and Running / Ubuntu для всех, / Р. Никсон БХВ-Петербург, Русская Редакция, 2011, – 459 с.
15. Робинс А. Bash. Карманный справочник для системного администратора /А. Робинс, Изд. «Вильямс» , 2010, – 152 с.
16. Уорд Б. Внутреннее устройство Linux : Пер. с англ. /Уорд Брайн. - СПб: Питер, 2016. – 384 с.
17. Херцог Р., Настольная книга администратора Debian/ Р. Херцог, Р. Ма изд-во Freexian SARL, 2016, - 522 с.
18. Процессы и управление заданиями [Електронний ресурс] – Режим роботи: http://heap.altlinux.org/modules/linux_users.prev/index.html
19. Командная оболочка Bourne-Again Shell [Електронний ресурс] – Режим доступу: http://rus-linux.net/MyLDP/BOOKS/Architecture-Open-Source-Applications/Vol-1/bash.html
20. Команды Linux [Електронний ресурс] – Режим доступу: http://shkola-linux.ru/articles/305-commands-linux-rjvfyls-linux.html
21. Создание ссылок в Linux [Електронний ресурс] – Режим доступу: http://younglinux.info/bash/link.php
22. Журнал Linux Format, все номера за 2014, Издательство: Future Publishing Режим доступу https://studylinux.ru/linux-format-vse-nomera-za-2014-arhivami-skachat-besplatno-bez-registratsii.html
23. Журнал Linux Format все номера за 2015, Издательство: Future Publishing. Режим доступу https://studylinux.ru/linux-format-vse-nomera-za-2015-arhivami-skachat-besplatno-bez-registratsii.html
24. Журнал Linux Format все номера за 2016, Издательство: Future Publishing. Режим доступу https://studylinux.ru/linux-format-vse-nomera-za-2016-arhivami-skachat-besplatno-bez-registratsii.html
25. Журнал Linux Format все номера за 2017, Издательство: Future Publishing. Режим доступу https://studylinux.ru/linux-format-vse-nomera-za-2017-arhivami-skachat-besplatno-bez-registratsii.html

Навчальне видання

Матвєєва Наталія Олександрівна
Хандецький Володимир Сергійович
Спірінцева Ольга Володимірівна

ОСНОВИ РОБОТИ ТА ПРОГРАМУВАННЯ В ОПЕРАЦІЙНІЙ СИСТЕМІ LINUX

Навчальний посібник

Підписано до друку 15.08.18.
Формат 60х84/16. Папір офсетний. Друк офсетний.
Ум. друк. арк. 3,23. Наклад 50 пр. Зам. №

Видавництво та друкарня ПП «Ліра ЛТД».
49107, м. Дніпро,вул.. Наукова, 5.
Свідоцтво про внесення до Держреєстру
ДК № 6042 від 26.02.18.